# Building Adaptive Compilers

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

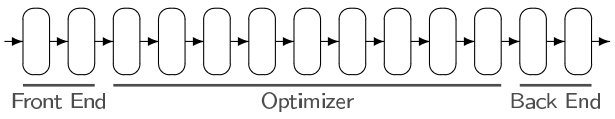
## 1. INTRODUCTION

After more than forty years of research, compiler writers have at their disposal a huge collection of transformations and analyses. Unfortunately, modern compilers are not designed to use these techniques to their greatest effect. These compilers apply the same algorithms, analyses, and transformations on each program, modulo some minor variations introduced by command-line options (e.g., `-g`, `-O1`, `-O2`, ...). Today's compilers are built to produce good code for a wide array of programs. Unfortunately, that same structure limits the set of programs for which they can produce excellent code.

Our experiments indicate that a compiler capable of adapting its behavior to specific inputs and target machines can produce better code than a fixed-behavior compiler. These improvements come not from the introduction of new transformations, but, instead, from making more effective use of existing transformations. The results presented in Sections 3 and 4 of this paper argue strongly that adaptive control of optimization is an important and missing piece of modern compilers. This paper discusses some of the practical issues that arise in the design and construction of adaptive compilers.

### 1.1 Compilation Order as an Example

As an example of the kind of adaptive behavior that a compiler can use to improve code quality, consider the problem of deciding which optimizations to apply to a specific input program and in which order to apply them. A typical optimizing compiler has a single answer to this question, or a small set of answers. Most modern compilers are partitioned into a front end, an optimizer, and a back end. Each of these components consists of a series of passes that run in a predetermined order.






The specific sequence of passes applied to a program, which we call a *compilation order*, determines how it is optimized. A mismatch between a specific compilation order and a particular program can reduce the compiler's effectiveness and reduce code quality.

Unfortunately, no single compilation order works well on all input programs. The interactions between individual optimizations are sufficiently complex that we cannot adequately describe them. Given two transformations, $a$ and $b$, $a$ may create opportunities for $b$ or it may foreclose opportunities for $b$. To make matters more complex, the opportunities for optimization change dramatically from one input program to another; a transformation can only improve the code if the inefficiency that it targets occurs in the code to which it is applied. Selecting good compilation orders is hard, in part because of the complexity of these interactions and in part because the space of compilation orders is huge. (Section 3 explores this issue in greater depth.)

Given a fixed pool of transformations, an adaptive compiler that selects program-specific compilation orders can produce better results than any single program-independent order chosen from the same set of transformations [7, 9, 32]. To build a practical compiler that chooses good program-specific compilation orders is hard. Such a compiler needs algorithms that find good sequences while examining just a small portion of the space of potential compilation sequences.

### 1.2 Adaptation in Compilation

Finding compilation orders is one instance of a general problem: building compilers that the their behavior to improve the quality of compiled code. Adaptive behavior has been creeping into the literature on compilation and high-performance software for a decade. Adaptive schemes have been used to find good blocking factors for numerical codes [22, 21, 11], to derive good heuristics for scheduling and register allocation [27, 29], and to choose command-line parameter settings [17]. Dynamic reoptimization techniques (in JITs and other systems) [1] and self-tuning libraries [32, 20] also use feedback from program execution to adapt program behavior in an attempt to improve performance.

This paper focuses on two particular kinds of adaptive behavior in compilation. Section 3 explores the problem of finding good compilation orders and presents results that offer insight into the nature of the search spaces in which such sequences lie. Section 4 examines the problem of adapting the behavior of an existing compiler by using command-line settings; it presents results from an experiment that tries to duplicate the performance of an ATLAS code [32] using only

automatic techniques. In both instances, our approach is to perform extensive experiments that expose properties of the mathematical spaces in which the adaptive systems function and to use those properties to develop more effective adaptive systems. Section 5 explains some of the lessons that we have learned about designing and engineering compilers that are suitable for adaptation. Section 6 lays out some of the challenges that must be solved before the technologies of adaptation are ready for deployment in widely used commercial systems. Before delving into our experiments and results, Section 2 surveys the related literature.

## 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, algorithmic adaptation in single transformations, and attempts to adaptively control the compilation process.

Several groups have produced adaptively self-tuning numerical libraries. The ATLAS library chooses blocksizes adaptively at runtime; the installation runs a series of machine-specific performance tests that compute parameters for the runtime blocking [32]. Both FFTW and UHFFT generate custom-optimized codes for performing fast Fourier transforms [16, 15, 25]. The numerical libraries for the Thinking Machines CM-2 and CM-5 machines used metrics based on problem size to make algorithm-choice decisions [20].

Adaptation has been applied inside the individual passes of an optimizing compiler. Motwani *et al.* developed an algorithm to combine register allocation and scheduling; they used *alpha-beta tuning* to find a priority function that optimized the algorithm's performance [27]. Amarasinghe *et al.* used genetic programming to derive good priority heuristics for list scheduling; they suggest that their approach should help with any priority-driven optimization [29]. Profile-driven techniques, including trace scheduling [13], superblock scheduling [19], and code positioning [28] also change the compiler's behavior in response to observed runtime behavior.

Granston and Holler developed an algorithm that picked program-specific or file-specific command-line options for the HP PA-RISC compiler [17]. They interviewed experts, performed experiments, and distilled the results into a deterministic algorithm.[1] Knijnenburg *et al.* used a parameter sweep to find the best blocking factors for loops [22, 21]; we have shown similar results with search-based techniques that use fewer probes of the search space [11] (see also § 4).

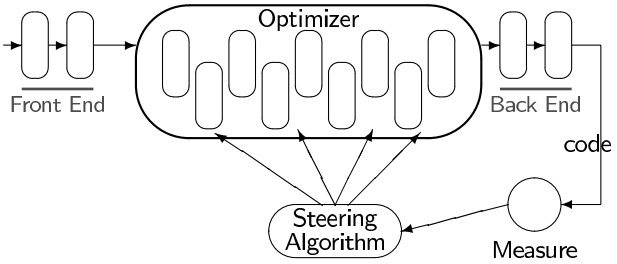
Other authors have looked at the compilation order problem. Kulkarni *et al.* used a genetic algorithm (GA), performance information, and user input to select local optimizations in the VISTA system [24]; their GA is modeled after Schielke's GA [7]. Zhao *et al.* are developing analytical models that can be used to steer the selection of a compilation order and, potentially, other kinds of adaptation [33]. August *et al.* recognize the potential benefits of finding good compilation orders; to limit the compilation costs, their system explores a fixed set of sequences and retains the best results [30]. (This approach resembles the *best-of-three* spilling heuristic proposed by Bernstein *et al* [2].)

[1]We are aware of several other industrial groups that have conducted similar experiments; the results are both anecdotal and unpublished.

## 3. SELECTING COMPILATION ORDERS

Selecting a compilation order is one of the fundamental design challenges that a compiler writer must face. The traditional solution to this problem has the compiler writer select a single compilation order (or perhaps one each for `-O1`, `-O2`, ...). Our preliminary work demonstrated that automatically-chosen, program-specific compilation orders can produce significantly different results, measured in execution time and size of the compiled code [7]. In the prototype system, program-specific, automatically derived orders consistently beat any fixed sequence that we have derived.

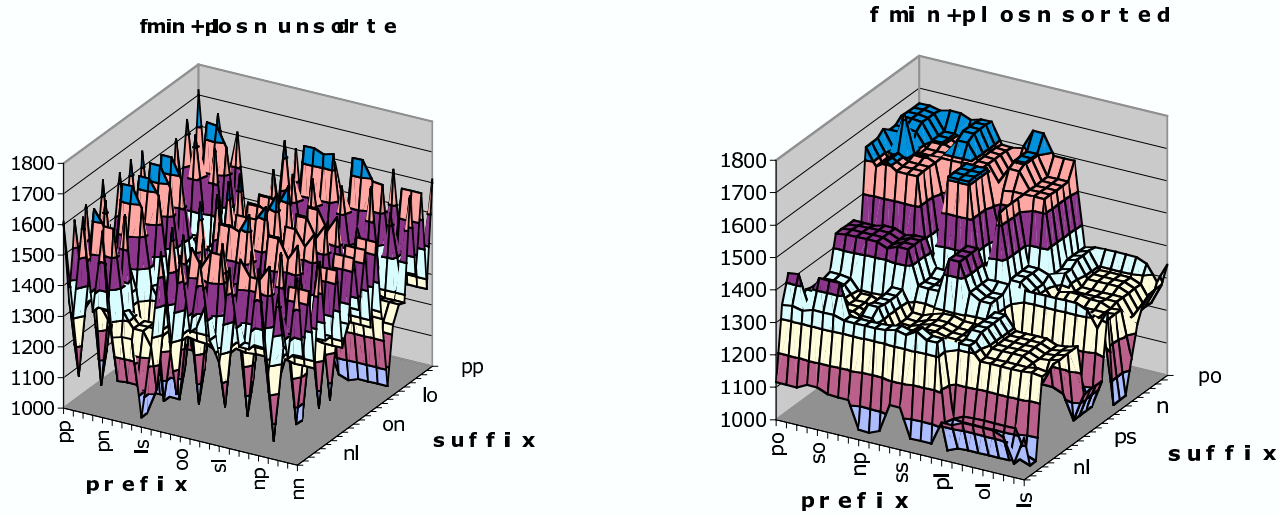
Our subsequent work on choosing compilation orders has focused on two issues: understanding the search space in which these algorithms operate, and developing better techniques to find good program-specific compilation orders. The experiments that we report in this paper have been performed in our prototype adaptive compiler, shown below.




The prototype has front ends for Fortran and C, a collection of thirteen distinct optimizations, and back ends that generate code for a simulator and for the SPARC. The optimizations can be run in arbitrary order. The adaptive compiler 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 with a variety of objective functions, including one that executes the code on a representative input and counts instructions executed, one that simply counts bytes in the compiled code, and one that measures a property believed to correlate with power consumption. We have used GAs, hill climbers, greedy constructive algorithms, and enumeration engines to steer the prototype. In each case, program-specific compilation orders produced better results than the fixed sequence compiler.

Why is choosing a compilation order hard? As a community, we have developed hundreds of algorithms for analysis and for transformation; the compiler writer must choose a small set (typically between five and twenty) to implement. Each of these techniques attacks some inefficiency in the code being compiled; for each problem in the code, there are multiple algorithms that improve it. Since most of the techniques capitalize on multiple effects, the techniques have significant overlap, too. (A Venn diagram of effects would show large areas of commonality and small regions of difference.) Finally, each transformation changes the code seen by its successors. Thus, a transformation may create or destroy opportunities for other transformations; these interactions are both complex and input dependent.[2]

[2]Whether or not transformation $a$ can create an opportunity for transformation $b$ depends on characteristics of both $a$ and $b$. However, $a$ only creates that opportunity if the targeted code appears in the input program; if it never occurs in $a$'s input, then $a$ cannot create the opportunity for $b$. Of course, the symmetric effect happens with $a$ destroying opportunities for $b$.




(a) Unsorted (b) Best order

**Figure 1: Effects of Order on Strings of Length Four**

## 3.1 Understanding the Search Space

The prototype adaptive compiler tries to find a minimizer for some objective function in a large and complex discrete space. Before we can design algorithms that operate effectively in this space, we need to understand some of its properties. Ideally, we would like the space to be convex and smooth, so that a downhill algorithm, such as steepest descent, would reach a minimizer. Unfortunately, too little is known about the interactions between optimizations and relationship between source-code properties and the effectiveness of optimizations to allow us to reason about the space analytically.

To discover properties of the space, we have performed a series of large-scale experiments. The experiments fall into two categories: enumerations and explorations. In an enumeration, we select a subspace of the full search space and enumerate all of the points in that space for a single program and a sample input. Enumerations are computationally expensive. In an exploration, we use a search algorithm to find good compilation orders for a variety of programs under some objective function. Explorations are orders of magnitude faster than enumerations. Both enumeration and exploration are critical to our work. Enumerations help us understand the properties of the search space. Explorations let us evaluate the effectiveness of specific search techniques. The pattern of our work is to use enumerations to gain insight into the behavior of the adaptive compiler and to use explorations to test the application of those insights on searching the space.

*Enumeration Results* The search space available to the prototype compiler is huge. If we allow it to use strings of length nine (a 9-of-13 search space), the space contains $13^9$, or 10,604,499,373 points. (Schielke's experiments operated in the smaller 10-of-10 space.) To keep the enumerations manageable, we have examined 10-of-5 subspaces that contain 9,765,625 distinct compilation orders. The enumeration engine compiles the program with each sequence and records the fitness value produced by applying the objective function to the compiled code. This process creates large but manageable data sets that we can use in offline experiments to assess the properties of the space and the behavior of various search techniques.

For our initial enumeration experiment, we selected a small Fortran program, `fmin`,[3] that had proven problematic for a variety of GAs and hill climbers. The objective function tried to minimize instructions executed. To select a set of five optimizations, we started the hill-climber at 100 randomly-chosen starting spaces in a 10-of-13 space. Each hill-climber run produced a final compilation order. We chose the five transformations that occurred most often in this set of final compilation orders. They were loop peeling (`p`), partial redundancy elimination (`l`), peephole optimization over logical windows (`o`), register coalescing via graph coloring (`s`), and useless control-flow elimination (`n`). We refer to this subspace as `plosn` and the `fmin` enumeration data for the space as `fmin+plosn`. For reference, the full set of transformations is shown in Figure 2.

| Symbol | Transformation |
|---|---|
| `c` | sparse conditional constant propagation [31] |
| `d` | dead code elimination [12] |
| `g` | optimistic global value numbering |
| `l` | partial redundancy elimination [26] |
| `m` | renaming for PRE and `lcm` |
| `n` | useless control-flow elimination [10] |
| `o` | peephole optimization over logical windows [10] |
| `p` | peeling one iteration of a loop |
| `r` | algebraic reassociation [4] |
| `s` | register coalescing via graph coloring [6] |
| `t` | operator strength reduction [8] |
| `v` | dominator-based value numbering [5] |
| `z` | lazy code motion [23] |

**Figure 2: Full Set of Transformations**

[3] `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.

The enumeration of `fmin+plosn` took 14 CPU-months and six months of wall time. We have subsequently made engineering improvements in the compiler that have radically decreased the overall time for such enumerations and increased the scalability of the process. As a result, we can enumerate a 10-of-5 space in less than two weeks; adding more hardware could easily reduce that time.

In `fmin+plosn`, the best compilation order produces code that executes 1,002 instructions, a 42% improvement over the 1,716 instructions executed by the unoptimized code. The worst sequence in the space also executes 1,716 instructions; that sequence consists of ten instances of `p` – loop peeling. Comparing these results to the full search space, we note that the best sequence found by the GA operating in a 10-of-13 space executes 822 instructions.

This enumeration shows that the search space is neither smooth nor convex. (All other enumerations have confirmed 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. 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.

Understanding the data from these enumerations is, in itself, a difficult task. We cannot easily visualize surfaces in a ten-dimensional space. To complicate matters, any visualization requires that we impose an order on `p`, `l`, `o`, `s`, and `n`. Different orders produce strikingly different results, as shown in Figure 1. The two plots show `fmin` fitness values (executed instructions) for sequences of length four drawn from `plosn`. Figure 1.a shows the fitness value as a function of compilation string using the original order `plosn`. It resembles the surface of a heavily-cracked glacier and conveys little or no intuition about the relationships between sequences and fitness values.

Figure 1.b shows a reordered view of the same data. This view of the data is much smoother. It was produced by averaging the fitness values across rows and sorting by row, then averaging fitness values by column and sorting by column. In this view, the space appears to have structure. Unfortunately, no assignment of ordinals to the letters `p`, `l`, `o`, `s`, and `n` produces the data shown in Figure 1.b. Notice that the prefix and suffix strings are not presented in the same order; while the picture is pretty, this view does not correspond to any consistent ordering of the data. Examination of the 120 potential consistent orders for the data shows that the resemble Figure 1.a much more closely than Figure 1.b.

*Measured Properties of the Search Space* To date, we have enumerated four 10-of-5 subspaces, `fmin+plosn`, `fmin+pdvnt`, `zeroin+plosn`, and `zeroin+pdvnt`.

| | Best | | Worst | |
|---|---|---|---|---|
| | Value | # | Value | # |
| `fmin+plosn` | 1,002 | 1 | 1,716 | 1 |
| `fmin+pdvnt` | 1,216 | 8 | 1,716 | 1024 |
| `zeroin+plosn` | 832 | 3,099 | 1,446 | 1 |
| `zeroin+pdvnt` | 1,020 | 8 | 1,446 | 1024 |

Columns labeled Best and Worst indicate the number of instructions executed for the best or worst sequence in the space; the column labeled # shows the number of sequences with that value.

These enumerations have exposed several properties of the search space that can help in the design of search algorithms to help the compiler find minimizers:

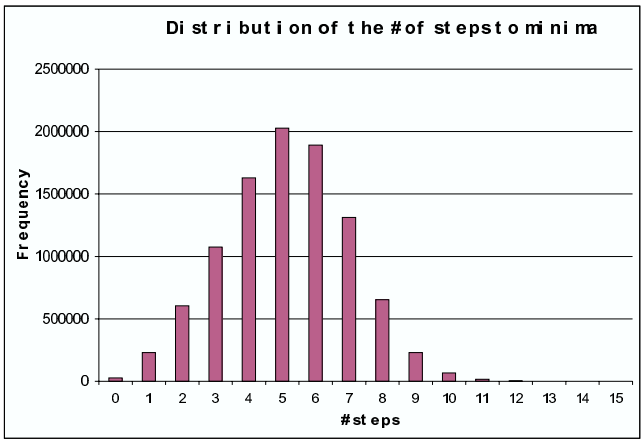



**Figure 3: Hamming-1 Distance to Local Minimum**

- The subspaces have many local minima, both good and bad. In `fmin+plosn`, most of the local minima are good—lying within 5% of the global minimum. However, another group of local minima lie between 20% and 30% above the global minimum. Thus, a single hill-climber can reach a "bad" local minima—one that is far from the best fitness value. However, retaining the best result from multiple hill-climber runs, starting from randomly-chosen sequences, usually finds a local minimum that is close to the global minimum.

- The distance from a randomly-chosen sequence to a local minimum, measured in Hamming-1 steps, is small relative to the size of the space, as shown in Figure 3 In the 10-of-5 subspaces, a hill-climber finds a minimum in twelve or fewer steps. This property appears to carry over to the larger spaces. In the larger 9-of-13 space, a hill-climber almost never takes more than fifteen steps. The most steps that we have seen a hill climber take in the 9-of-13 space is 24.

- 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. (Of course, building the model also provides a concrete upper bound on the results—the best probe found building the model.)

One further aspect of the subspace enumerations bears mention. The second subspace that we enumerated, `fmin+pdvnt`, was chosen by the authors to address the perceived weaknesses of the `plosn` subspace. (For example, the best sequence in `fmin+plosn` is 18% slower than the best 10 pass sequence found by a GA.) In fact, the hand-picked set of transformations did worse than `plosn`. This fact highlights, again, the fact that sequences found by adaptive search are usually more effective than those chosen by humans.

## 3.2 Techniques for Searching the Space

While enumeration studies can provide insight into the structure of the search space, enumeration is not a practical technique for finding program-specific compilation sequences. A major thrust of our work has focused on exploration studies to evaluate the suitability of specific search techniques for use in a sequence-finding adaptive compiler. We perform exploration studies in a variety of spaces; for the purposes of this paper, the 10-of-5 spaces and the 9-of-13 space are interesting.

Studying a search algorithm's behavior in one of the 10-of-5 spaces is attractive because we have complete maps for the space; this fact lets us accurately evaluate the quality of a proposed final sequence. However, for a search technique to be useful, it also must operate productively in the adaptive compiler's full search space (for this paper, the 9-of-13 space). Thus, we test search algorithms with a variety of programs in the 9-of-13 space. To evaluate a given search algorithm in more detail, we return to the enumerated subspaces (such as `fmin+plosn`) where we can run experiments offline and compare the results against full knowledge of the search space.

*Improving the Genetic Algorithm* Our initial work used a GA to steer the adaptive compiler [7]. That GA used a population of 20 sequences in a 10-of-10 search.[4] Sequences were ranked by fitness value. At each generation, the GA removed the worst string along with three 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. The GA found its best sequences in 200 to 300 generations.

Experiments with variations on the GA improved its behavior. Our best results, for speed and code size, use a population of 50 to 100 sequences, a single-point random crossover, and fitness-proportionate selection to exaggerate small late improvements. At each generation, the best 10% of the sequences survive without change. The rest of the strings in the generation are created by choosing a pair of sequences, performing a single-point, random crossover operation, and subjecting the resulting sequences to a low probability, character-by-character mutation. The random selection prior to crossover is biased by fitness value. If this process generates a sequence that has already been tested, it is mutated until the new sequence is an untried sequence.[5] With these modifications, the new GA usually finds its best solution within 30 to 50 generations.

*Hill Climbers* An alternate approach to finding good compilation orders is to choose an initial point in the search space, 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 in local minima, they are often invoked multiple times from distinct starting points. (The enumeration experiments show that the adaptive compiler's search spaces are littered with local minima.)

In designing a hill climber, we must define the notion of "neighbor." Our hill climbers have defined the neighbors of a string $x$ as all strings of the same length that differ in one position from $x$—that is, strings at Hamming distance one from $x$ (Hamming-1 strings). To find the hill-climbing step that produces the greatest change in fitness value requires evaluating each Hamming-1 neighbor; for the 10-of-5 subspaces, each string has 40 Hamming-1 neighbors. The clustering results from § 3.1 suggest that the hill climber should typically halt, in `fmin+plosn`, in eight or fewer steps, with a worst case upper bound of twelve steps.

We have experimented with a number of variations on the hill climber, both in `fmin+plosn` and with a variety of programs in their larger 9-of-13 spaces. In practice, a random descent algorithm that takes the first downhill step often outperforms the steepest descent algorithm that examines all neighbors and takes the largest downhill step. We have experimented with an algorithm that trades exploration for randomization: an *impatient* random descent algorithm. The impatient algorithm places an upper bound on the number of neighbors it will evaluate at each step. To compensate for its shallower exploration of the local neighborhood, we have the impatient random descent algorithm perform more trials. As the results in Figure 4 show, an impatient random descent algorithm can compete well against more expensive techniques such as the GAs.

*Constructive Greedy Algorithm* Greedy algorithms produce good results for a variety of complex problems (e.g., list scheduling). If a greedy algorithm works well in the search spaces of the adaptive compiler, it might be an attractive alternative. To assess the potential for greedy techniques, we implemented and evaluated a constructive greedy algorithm. It views the search space as a DAG where the root node represents the empty sequence. The root node has $n$ children, representing the sequences of length one. 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.

Complications arise when a node has multiple children that each have the best fitness value. 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). When ties occur, they can drastically increase the cost of GC-EXH; for example, on the program `adpcm-d`, GC-EXH explores 91,633 sequences while GC-50 examines less than 2,200 and GC-BRE examines only 325. (As expected, the more expensive searches produce better results, but GC-50 does as well as GC-EXH – within 0.003%. GC-BRE produces code that executes 2% more instructions.)

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

[4] The experiments excluded several passes that displayed order sensitivity. Some of those have been subsequently fixed, enlarging the universe of transformations.

[5] 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. If a proposed new sequence has already been evaluated, it is mutated until an unevaluated sequence is obtained. Since the best sequence always survives, this never discards a winning sequence.

| | GC-50 | HC-50 | GA-50 | Best Sequence |
|---|---|---|---|---|
| `fmin` | 88.3% | 74.3% | 73.9% | `opvcdlsnd` |
| *# probes* | 529 | 1,355 | 4,550 | |
| `zeroin` | 71.4% | 70.6% | 69.9% | `oppvdscnd` |
| *# probes* | 2,498 | 1,265 | 4,550 | |
| `nsieve` | 53.8% | 53.8% | 53.8% | `pogvpcsdn` |
| *# probes* | 5,262 | 1,222 | 4,550 | |
| `adpcm-c` | 68.0% | 68.0% | 67.6% | `ropcvspnd` |
| *# probes* | 1,102 | 1,326 | 4,550 | |
| `adpcm-d` | 70.0% | 67.7% | 65.0% | `cdnlosncs` |
| *# probes* | 2,113 | 1,287 | 4,550 | |
| `g721-e` | 84.5% | 84.7% | 84.2% | `npcpdznls` |
| *# probes* | 220 | 1,827 | 4,550 | |
| `fpppp` | 80.6% | 76.7% | 76.8% | `npoclvnsd` |
| *# probes* | 345 | 1,980 | 4,550 | |
| `tomcatv` | 120.4% | 87.6% | 85.1% | `pppcorvsn` |
| *# probes* | 220 | 1,776 | 4,550 | |

**Figure 4: Improvement over Fixed Sequence**

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 sequences. Our efforts have focused on evaluating the range of improvement available with adaptation, on understanding the search spaces in which the adaptive compiler operates, and on developing algorithms that operate well in that space.

*Comparing the Algorithms* The table in Figure 4 presents results of typical runs of one instance of each of the search algorithms. GC-50 is the greedy constructive algorithm with random tie breaking, run 50 times. HC-50 is an impatient random descent hill climber; it tests up to 10% of the current point's neighbors and returns the best result from 50 starting points. GA-50 is the genetic algorithm, run with a population of 50 sequences for 100 generations.

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 [14]. `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, we show the best result from three runs of the search algorithm, choosing sequences of length 9 from the 13 transformations in Figure 2. The runs in Figure 4 all target a simulated RISC architecture.[6] The first line shows the percentage of instructions executed versus the original fixed sequence compiler. The second line shows the number of probes that the algorithm made of the search space—the number of sequences that it evaluated. The right column shows the sequence that produced the best result in the nine trials (three each of GC-50, HC-50, and GA-50).

All three search techniques find consistent improvements. (The sole negative result is GC-50 on `tomcatv`.) In general, the GA finds better sequences than GC-50 or HC-50, but it also evaluates more sequences. (GC-50 evaluated more sequences for `nsieve`.) We also evaluated GA-100, which uses a population of 100 for 100 generations (9,100 evaluations). The results from GA-50 are consistently within a fraction of a percent of those from GA-100; doubling the number of evaluated sequences in the GA produces no gain.

These larger experiments target a simple simulated RISC architecture. For `fmin` and `zeroin` using our SPARC backend, search algorithms produce the following results, in the same format as Figure 4:

| | GC-50 | HC-50 | GA-50 | Best Sequence |
|---|---|---|---|---|
| `fmin` | 90.8% | 91.2% | 90.7% | `ppppppodl` |
| *# probes* | 400 | 1,247 | 4,550 | |
| `zeroin` | 94.0% | 93.8% | 94.0% | `cpnovdglc` |
| *# probes* | 375 | 1,257 | 4,550 | |

Changing the architecture makes different sequences important. In the SPARC experiments, we found many more ties than on the simulated RISC architecture. The sequences shown are taken from the best results produced by GA-50.

# 4. COMMAND-LINE SETTINGS

Adaptive compilation has the potential to custom-tailor program performance to a specific machine and a specific set of input characteristics. In concept, an adaptive compiler should be able to produce results similar to those obtained with adaptive numerical libraries, such as ATLAS. Success with this kind of custom optimization should make the results of hand-tailored code available to a much wider audience—replacing the programmer-intensive hand-tuning that produced ATLAS with automatic tailoring by an adaptive compiler.[7]

To test the feasibility of generating ATLAS-like performance with the techniques of adaptive compilation, we conducted an experiment using a commercial FORTRAN compiler, the MIPSPro compiler for the MIPS R10000, with an adaptive steering algorithm. We used this combination to compile one of the workhorse codes in ATLAS: `dgemm`, a general matrix-matrix multiply. The ATLAS version of `dgemm` is widely accepted as offering superior performance to the standard BLAS routine and to many hand-coded versions.

(In one sense, this experiment attempts to automate the practice of industrial benchmarking groups. Benchmarking teams typically use human experts to discover the best set of compiler options and parameters for a specific program or file within a program. Often, these benchmark teams use a version of the compiler with many more switches than are documented in the end-user version.)

We evaluated several compilers for use in these experiments; the MIPSPro compiler was the only one we found that was capable of producing ATLAS-like results on small problems without any adaptive control. As Figure 5 shows, MIPSPro produces results comparable to ATLAS for arrays up to roughly 800×800. (Times are normalized to the ATLAS running time.) For larger matrices, the performance of MIPSPro code without our adaptive methods degrades markedly relative to ATLAS.

[6]Using a simulated architecture allows us to run the experiments on a variety of underlying machines. The data in Figure 4 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.

[7]It will be harder to match custom code generators, such as FFTW and UHFFT, since they generate what are essentially different algorithms for different sizes of inputs.

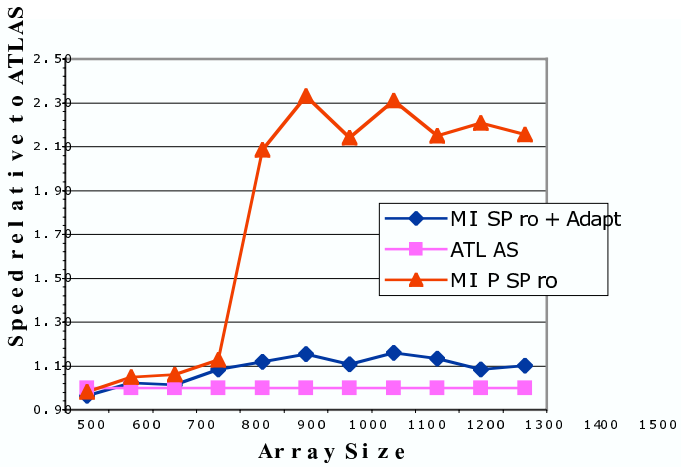



**Figure 5: Normalized `dgemm` Time Versus Array Size**

ATLAS gains its advantage by adapting its behavior to the size of the input datasets. Extensive experimentation with MIPSPro found only one command-line option that had a significant and variable impact on the performance of `dgemm` across a variety of datasets: the blocking factor. (Other options had consistent and size-independent settings. For example, all the compilations were performed with the highest level of optimization (`-O3`) and included interprocedural analysis (`-IPA`). We performed a series of experiments to determine the sensitivity of MIPSPro to externally-specified blocksizes. After looking at the results of these experiments, we derived a simple search-based algorithm that finds the best MIPSPro blocksize parameter for `dgemm` invoked on matrices of a given size [11].

With the external adaptive system choosing blocksizes, the MIPSPro compiler is able to produce code that keeps performance within roughly 10% of ATLAS' performance (see the line marked with diamonds in Figure 5). The remaining difference between ATLAS and the code produced by the adaptively-controlled MIPSPro compiler might be reduced or eliminated with better adaptive control of other optimizations. The current command-line interface to MIPSPro did not let us close this gap.

The experiments with ATLAS and `dgemm` demonstrate two important points. First, even limited adaptive control via the command line can improve the behavior of a good compiler such as MIPSPro. Simple, feedback-driven adaptation of this kind can improve the results of compilation. Second, the available command-line parameters offered little opportunity for effectively tailoring the compiler's behavior to the input program. To build compilers that are more amenable to adaptive control, we need to reexamine the kinds of control over optimization that we expose to the end user; this notion includes both the way that we parameterize optimizations and the ways that we expose those parameters to the end user.

As an example of how parameterization can constrain the ability of an adaptive steering mechanism, consider the heuristic used to make inline substitution decisions in the ORC 1.0 compiler for the IA-64. (The ORC 1.0 compiler that was originally derived from parts of the MIPSPro code base.) In ORC 1.0, inlining decisions are based on estimates of the speed and space impact of inlining each specific call site. The compiler estimates a temperature for each call-graph edge as the ratio $\frac{cycle\_ratio}{size\_ratio}$. Here, *cycle_ratio* approximates the fraction of total execution cycles attributable to the callee from invocations along that edge and *size_ratio* estimates the fraction of total codes attributable to the callee [34].

ORC 1.0 inlines a call site if the corresponding edge in the call graph has a temperature that exceeds some fixed threshold value. Adjusting the threshold value can change the compiler's inlining behavior; however, the relationship between threshold value and choice of call sites to inline is indirect, at best. Zhao and Amaral experimented with a mechanism for adaptively choosing threshold values [34]; they report improvements across a variety of programs.

Adaptive control of the threshold can choose better thresholds. However, many desirable patterns of inlining cannot be selected using this threshold mechanism. The MIPSPro heuristic exposes one dimension of the space of inlining decisions. Other ways of parameterizing the decision might yield a finer-grain control that would be more amenable to adaptive control. For example, in a space constrained environment, the user might want to inline procedures that are called from just one call site, independent of their execution frequency.[8] Similarly, inlining call sites where the body of the callee is smaller than the procedure linkage code often yields a reduction in space. Neither of these heuristics fits easily into the temperature-versus-threshold tradeoff.

# 5. DESIGNING FOR ADAPTATION

Building compilers that can adaptively change their behavior requires careful attention to a set of engineering issues that do not arise in static compilers. Some issues are simply the application of sound software-engineering principles to compiler design and implementation. Others require reexamination of fundamental techniques and algorithms. It is clear, however, that we cannot simply add an adaptive control loop to most existing systems.

## 5.1 Reordering Passes

The ability to run optimization passes in arbitrary order is a critical property of our prototype adaptive compiler. The prototype has this property by design; each pass was intended as a standalone transformation that could be invoked and studied in isolation. (Our interest in reordering transformations arose after most of them had been built.) It follows a few simple design rules.

- *Passes operate as standalone programs.* Each optimizer pass reads in the IR, analyzes and transforms it, and produces a valid IR file. It leaves the IR program in a consistent and correct state, on external media (either a Unix pipe or a file). The passes rely on a set of standard tools, including a parser, prettyprinter, and SSA-builder. Each pass allocates and frees all of its storage; the passes use an arena-style allocator.

- *Avoid global state, other than the IR.* Global state is a major impediment to pass-reordering in some compilers. Analysis results computed in one pass are reused in a later pass. An instantiated IR version of the program, perhaps with a global symbol table, survives across all passes. Heap-allocated data structures persist across multiple passes. In practice, these shared

[8]The temperature heuristic implicitly assumes that each inlined edge increases total code size, where inlining procedures with a single call site often decreases total code size.

structures prevent some orderings from working correctly.

- *Make interactions between passes explicit.* When pass $a$ must precede pass $b$, this restriction must be both well documented and encoded into the tools. In our prototype, both partial redundancy elimination (PRE) and lazy code motion (LCM) depend on specific properties of the program's name space. Thus, the steering algorithms automatically insert the necessary pass before either PRE or LCM.

These design rules have a minor compile-time cost. The prototype parses and prettyprints the IR in each pass; it repeats some analyses that could be shared. The resulting modularity has paid off in easier debugging and in the reconfigurable nature of the optimizer. To recoup some of the lost compile time, we have focused some effort on designing efficient common components, such as the parser, prettyprinter, and SSA-builder. The parser, for example, builds a control-flow graph, and constructs both reverse postorder and reverse preorder numberings. The prettyprinter eliminates unreachable blocks.

Our experiments have revealed interpass dependences and order-sensitive bugs. The design rules prevent many bugs that might arise from shared data structures, such as moving an allocation, deallocation, or initialization; or inserting a pass that invalidates some stored analysis result. Most of the order-sensitive bugs that testing has exposed involve implicit assumptions about the IR program. The problems tend to fall into two categories: a pass relies on properties of the IR program's name space that are destroyed by other transformations,[9] or a transformation produces code with unexpected properties that cause problems for a later pass.[10]

## 5.2 Managing Adaptation

Creating mechanisms to manage adaptation and to let the user control adaptation in a useful way remains a significant challenge. The search algorithms described in § 3.2 can determine good compilation orders. Before they can be deployed in production systems, however, a number of management issues must be addressed.

*Stopping Criteria* The steering algorithm must have mechanisms that decide when to cease searching and declare that the current solution is "good." In practice, the user should control the stopping criteria. Such control might take the form of an arithmetic specification—within $x\%$ of the expected global minimizer or at least $x\%$ faster than the unoptimized code. It might take the form of a resource constraint—return the best solution found in $x$ minutes of CPU time. It might include a hard measurement—find a solution that produces an executable with fewer than $x$ bytes of code.

The steering algorithm should track its own progress, to recognize both good solutions and circumstances when it is not making adequate progress. Tracking the rate of improvement in fitness values can let the steering algorithm recognize convergence, or lack of progress. For example, the algorithm might declare convergence when a GA has not improved the solution in several hundred trials or a hill-climber is producing tiny or nonexistent improvements.

The open question, of course, is how the steering algorithm recognizes a "good" solution. A local minimum may have a fitness value close to the global minimum or it may not. With sufficient probing of the search space, the steering algorithm can build a model of the distribution of solutions; such a model would let the steering algorithm determine that a solution was in the neighborhood of the good solutions. Kullback-Liebler divergence testing on `fmin+plosn` suggests that 1,000 probes are enough to build a good model of the distribution of solutions in the space.

*Controlling Compilation Time* If compilation time is scarce, other schemes for using the knowledge gained in search experiments may be necessary. The compiler might try $k$ sequences, either a fixed set resulting from earlier searches or $k$ sequences drawn at random from a database of known "good" sequences, and retain the best result.[11]

A steering algorithm that performs its search over multiple compilations can spread the cost of a detailed search over the development cycle of an application. It is easy to see how the greedy constructive algorithm can be incrementally evaluated. Each time the compiler is invoked, it could perform several steps of the GC evaluation and save the results. Incremental versions of the other algorithms would be equally desirable.

While our work to date has focused on search algorithms, we may yet discover improvements on the greedy constructive search that consistently build good sequences while using a bounded number of probes. Such techniques might provide better results than a static compiler, with a bounded increase in compile time.

*Engineering for Exploration* The prototype adaptive compiler that we have built necessarily uses more compile time than a traditional fixed sequence compiler. However, the process of conducting our subspace explorations has made us sensitive to the overhead time needed per sequence.

The initial enumeration of `fmin+plosn` required 14 CPU-months and a little over 6 calendar months to complete. We began on a single Pentium processor and added other processors (SPARCs and PowerPCs) to the problem as it became apparent that progress was excruciatingly slow. Across the four processors eventually used in the experiment, we averaged between 34,000 to 39,000 experiments per day – compiling and running `fmin`.

Since a 10-of-5 subspace enumeration involves 9,765,625 distinct sequences, the processes used to generate the sequences, to invoke the individual standalone passes, to gather the data, and to manage the resulting files all deserve attention. The original fixed-sequence compiler used a shell script to invoke the passes and connect them appropriately; the initial enumeration engine used that same scheme. A simple redesign using C code that performs explicit `fork`

[9] The implementation of PRE relies on carefully constructed properties of the name space.

[10] For example, our strength reduction pass can create an excessively long sequence of register-to-register copy operations at the end of a loop. A sequence of more than $k$ consecutive copy operations can create a clique of registers with infinite spill costs that cannot be $k$ colored. A coloring allocator that strictly follows Briggs' description [3] will loop indefinitely on such a graph. Harvey fixes this bug in his thesis [18].

[11] This strategy has long been applied in register allocation [2]. August *et al.* have built a system that uses this approach to try a small number of compilation orders [30].

and `exec` calls removed most of the overhead from that process. The new test harness operates at 150,000 to 250,000 `fmin` experiments per processor day, depending on the processor. Our later studies have been performed on a cluster of dual-processor MacIntosh OS-X server machines, where they take 40 to 50 CPU days for a subspace enumeration. On the cluster, each subspace takes less than two weeks. These same improvements, speed up the individual trials when the compiler is using one of the search algorithms.

# 6. 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 with compilation order has demonstrated that we can find good orders with algorithms that examine only a tiny fraction of the possible compilation orders. Our work with adaptive control of command-line parameters suggests that we can build adaptive controllers that use commercial systems as black-box compilers. Many open problems must be solved before adaptive compilers become practical tools for routine compilations.

*Search Algorithms* The experiments described in Section 3 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.

*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. 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.

*Predictive Algorithms* As we gain experience with adaptive compilers and as we derive better models, we hope to discover predictive algorithms. Such an algorithm would examine source code properties and use models of optimization impact and behavior. They might simply predict better starting points for the search techniques; alternatively, they might predict good sequences that can be tested directly.

*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. One way to encourage construction of modular compilers would be to develop reusable algorithm libraries, modeled on the successful libraries found in the linear algebra community. These libraries should encourage reuse with a philosophy of representation independence and strong, well-defined interfaces. Further research is needed to discover how optimizations should be structured to improve the opportunities for command-line adaptive control.

*Infrastructure* Many of the issues that arose in designing programming environments and compilers to support interprocedural analysis will also arise in the context of building adaptive compilers. The adaptive compiler will need a mechanism to record concisely the results of program-specific searches. This knowledge must be stored where it can easily be found by later compilations.


## Acknowledgements

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 these people go our heartfelt thanks.


# 7. REFERENCES


[1] V. Bala, E. Deusterwald, and S. Banerjia. Dynamo: a transparent dynamic optimizing system. *SIGPLAN Notices*, 35(5):1–12, May 2000. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[2] 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)*.

[3] P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Apr. 1992.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

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

[15] 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)*.

[16] 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.

[17] 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.

[18] T. J. Harvey. *An Experimental Analysis of a Set of Compiler Algorithms*. PhD thesis, Rice University, May 2003.

[19] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscaler compilation. *Journal of Supercomputing – Special Issue*, 7:229–248, July 1993.

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

[21] 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.

[22] 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.

[23] 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*.

[24] 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.

[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] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report TR 698, Courant Institute, July 1995.

[28] K. Pettis and R. C. Hansen. Profile guided code positioning. *SIGPLAN Notices*, 25(6):16–27, June 1990. In *Proceedings of the 1990 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[29] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. *SIGPLAN Notices*, 38(5):77–90, May 2003. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[30] 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.

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

[32] 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.

[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.

[34] P. Zhao and J. N. Amaral. To inline or not to inline? enhanced inlining decisions. In *Proceedings of the 2003 Workshop on Languages and Compilers for Parallel Computing (LCPC '03)*, Oct. 2003. *(to appear from* Springer*)*.