

ITR: Building Practical Compilers Based on Adaptive Search
— A Proposal to the National Science Foundation —

Keith D. Cooper
Devika Subramanian
Linda Torczon

Department of Computer Science
Rice University
6100 Main Street, MS 132
Houston, Texas 77005

Technical Contact: Keith D. Cooper
keith@rice.edu

Administrative Contact: Darnell Price
darnell@cs.rice.edu

A. Project Summary

In the past decade, we have seen an explosion in the number of computers in use and in their diversity. Even commodity microprocessors come in low-power versions, embedded versions, and a variety of high-performance versions; the performance parameters of these versions can differ widely. Embedded processors, such as microcontrollers and DSP chips, show even wider variation; different models of the same processor have different numbers and kinds of functional units. At the same time, processor designs have become more complex, with multiple functional units, partitioned register sets, exposed pipelines, and multiple distinct latencies that must be managed.

As computers have found new application, developers and users have new priorities for their programs. Traditionally, users wanted programs to run fast. Now, issues such as code size, page faults, and power consumption have become important. Since most computers run compiled programs, a large part of the responsibility for addressing these issues falls to the compiler. While we understand how to build high-quality compilers that produce efficient code for most modern processors, actually building such a compiler is an expensive, labor-intensive effort that requires experts—compiler writers who are in short supply.

To address these problems, we have developed a framework for experimenting with adaptive optimizing compilers—compilers that reconfigure themselves to better handle specific programs, objective functions, and target machines. These adaptive compilers use search to discover an appropriate set of optimizations and an order in which to apply them. By explicitly considering the program, the target machine, and the objective function, the adaptive framework finds configurations that produce better results than a traditional, fixed-sequence compiler. Changing its objective function changes its behavior. This gives an end-user the ability to focus optimization on different program properties, such as speed, space, power, or page faults. We have used our prototype to optimize for speed, for space, and for some properties related to power consumption—as well as combinations of these.

Our work to date has focused on understanding the potential and the problems with this approach. In this project, we propose a sustained five-year program of experimentation and research to bring these techniques to the point where they can be routinely used to compile real applications. To focus our effort and to demonstrate the credibility of this approach, we will attack a challenge problem: automating the performance tuning of library code. Today, numerical libraries are hand-tuned for a variety of performance environments. Our adaptive compiler should automatically produce comparable results. One goal of our implementation efforts will be to automate the kind of tuning that the ATLAS team has done on the level 2 and level 3 BLAS libraries [42]. Success in this challenge will demonstrate that adaptive compilation can replace significant human effort and can automatically bring the kind of hand-tailored performance found in ATLAS to a broader range of codes.

To succeed on this problem, the project must focus on new issues in adaptive compilation. On the research front, we must improve our techniques for searching the space of compiler configurations and decrease the cost of each probe into that space. On the implementation front, we must build a set of iteration-space transformers and memory-related optimizations that can be run in arbitrary order. Finally, we must develop a set of guidelines and design principles for building and debugging order-independent optimization passes. Our testbed already includes twenty reorderable passes; current practice in both industry and academia does not produce such reorderable optimizations.

Each of these activities leads to many challenges. We must understand the nature of the search space, such as the frequency, distribution, and depth of both local minima and global minima. We must learn to recognize both convergence (a good global solution) and futility (a deep local minimum from which escape is hard), and develop strategies for handling both. We must develop tools and techniques for building order-independent compiler passes and for automatic fault isolation in the resulting adaptive system. Finally, we must develop ways to apply the knowledge gained in our work to improve the behavior of routine compilation.

Adaptive compilation applies the increased speed of our computers directly to the problem of producing better code. It offers the opportunity, for the first time, to optimize for arbitrary, user-selected criteria. It will change the way that we build, tune, and use optimizing compilers. With this technology, we can begin to tame the performance-variability that dominates modern computing. Over time, this work should produce practical compilers that can routinely and consistently deliver a large fraction of available performance.

The time has come to couple the quality of our compilers to the speed of our computing engines. This project will develop the knowledge, the algorithms, and the implementation techniques required to make adaptive compilation both practical and routine.

1 Introduction

Each year, microprocessors, microcontrollers, and other computing engines find application in new areas. The number of computers in use is growing rapidly, as is the diversity of those systems. The variety among processors—different instruction sets, different performance parameters, different memory hierarchies—is increasing. Even commodity microprocessors come in low-power versions, embedded versions, and a variety of high-performance versions; the performance parameters of these versions can differ widely. Processors have also grown more complex, with multiple functional units, exposed pipelines, and myriad latencies that must be managed. For example, some versions of the PowerPC and the Pentium have units capable of performing short vector operations. In most cases, these computers execute code produced by a compiler—a translator that consumes source code and produces equivalent code for some target machine.

At the same time, new processor designs and implementations have shifted more of the responsibility for speed onto the compiler. Processor performance has become increasingly sensitive to specific properties of executing code; properties such as locality, available instruction-level parallelism, and the ability to combine operations such as multiply and add into a single operation. This means that the fraction of programs that routinely execute at a large fraction of peak performance—those that optimize well—has decreased. For example, scientists at Los Alamos report that their codes routinely attain only 10% of peak performance [38].

The application of computing to new problems has also created demand for compilers that optimize programs for new criteria. In the 1980s and the early 1990s, the speed of compiled code was the dominant concern of users. In the late 1990s, the size of compiled code became an issue, driven by limited memory in embedded systems and by the importance of compiled applications transmitted over the Internet. Recently, we have seen a new interest in the impact that compiled code can have on the power consumption of battery-powered devices [6, 19, 24].

To be fair, after forty years of research and of practice, we know how to build optimizing compilers that produce efficient code for a single uniprocessor target. We can do this for most modern processors. Unfortunately, building high-quality compilers is expensive, primarily because it requires years of effort by experts, who are usually in short supply. We have also learned to build compilers that are easier to retarget, but produce less optimized code. These “retargetable” compilers are used in many situations where the economics cannot justify a large standalone compiler effort. What we have not developed is an economical way to produce high-quality compilers for a wide variety of target machines.

The overriding goal of this project is to bring computing power to bear on the problem of producing high-quality compilers for a variety of processors, applications, performance environments, and end-user optimization criteria. This requires that we rethink, in fundamental ways, the organization of our compilers. For several years, we have experimented with a flexible new approach for structuring an optimizing compiler. Our *adaptive compilers* reconfigure themselves to generate the best code that they can with respect to an explicit external objective function. These compilers discover program-specific configurations for the optimizer to meet the end-users’ goals (expressed in an objective function). We have used our experimental system to optimize for speed, for space, for properties related to power consumption, and for combinations of these three criteria. The results demonstrate that an adaptive compiler can produce consistently better code than a traditional fixed-sequence compiler [12, 15].

Our current prototype has several shortcomings that make it impractical for routine use. First, the process of finding a configuration that minimizes the objective function is impractically slow. We must discover better ways to find these configurations. Second, the prototype includes only classic scalar optimizations. While this has sufficed for our demonstration of concept, we need to extend the work to include high-payoff optimizations that target memory latency, blocking, and parallelism. Finally, our experience with the prototype has shown us the difficulty of building optimization passes that can run correctly in arbitrary orders. We will codify our experience in building such passes into a collection of design rules and tools to help others with the design and debugging of order-independent optimization passes.

To attack these problems, we need expertise in both compilation and in search-based strategies for problem solving. Our research team brings together expertise in several subareas of computer science to attack these problems. Cooper has lengthy experience in optimization and code generation. Subramanian is an expert in the application of techniques from artificial intelligence to real-world problems. Torczon has

worked for years in the areas of whole program analysis and optimization. All of these distinct skills and perspectives are needed to attack this problem.

This project will develop the knowledge, the algorithms, and the experience needed to move adaptive compilation from the realm of expensive prototypes to the point where it can be used for routine compilation. It has the potential to change the economics of producing high-quality compilers in a fundamental way—by automating much of the work required to tune a compiler for new circumstances.

The time has come to couple the quality of our compilers to the speed of our computing engines. This project will develop the knowledge, the algorithms, and the implementation techniques required to make adaptive compilation both practical and routine. To demonstrate the power of these techniques, we will attack a challenge problem where adaptive compilation can replace expensive human effort.

2 Adaptive Compilation

Since the 1960s, compilers have consisted of a fixed sequence of passes, applied in some preselected order, as shown in Figure 1. For example, the original Fortran translator had six passes [3, 4], the classic Fortran H compiler had ten passes [30], and the Bliss-11 compiler had seven passes [46]. IBM’s PL.8 compiler [2] and HP’s compiler for the PA-RISC [25] had the same basic organization. Modern systems retain this basic structure, with more passes. For example, the SUIF compiler has eighteen or more passes for its optimizer [28], and the recently released Pro64 compiler from Silicon Graphics has over twenty passes [39]. These compilers differ in the number of passes, the selection of specific passes, and the order of application for those passes, but their basic structure resembles that in Figure 1.

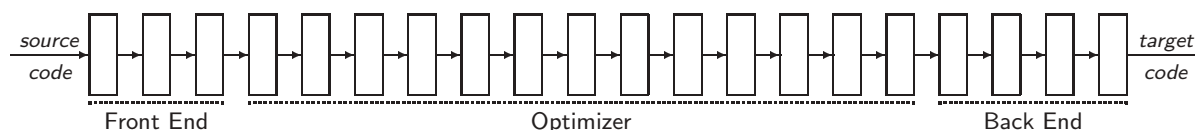


Figure 1: Classic Compiler Structure

A critical step in the design of an optimizing compiler is the selection of specific transformations and an order in which to apply those passes. (We call an ordered list of transformations a compilation sequence.) Since the 1960s, compiler writers and researchers have answered these twin questions in an *ad hoc* fashion, guided by experience and limited benchmarking. Efforts to find the “best” sequence have foundered on the complexity of the problem. Transformations both create and foreclose opportunities for other transformations. [35, 44]. Different techniques for the same problem catch different subsets of the available opportunities [10]. Finally, combinations of two or more techniques for different problems achieve the same result as some single techniques.

From our experiments, it is clear that the best compilation sequence depends on many factors, including 1) the specific details of the code being compiled, 2) the pool of available transformations, 3) the target machine and its performance parameters (which vary model to model), and 4) the particular aspect of the code that the user desires to improve (*speed, space, power, page faults, etc.*). To further complicate matters, the transformations used in optimization have subtle interactions and overlapping effects. This makes it difficult to predict the impact of changes in the compilation sequence. Today, the community lacks the knowledge to analytically predict the results of a particular sequence in a particular set of circumstances; this prevents a purely analytical process from deriving good code sequences. One major result of this project will be the knowledge required to understand these issues. Gathering this knowledge will require extensive experimentation.

The Promise of Adaptive Compilation To understand the potential of adaptive compilation, consider three arenas where performance is a critical issue:

Benchmarking: Because purchasing decisions depend on the ability to obtain reasonable performance, many vendors have in-house benchmarking groups. These experts use a compiler that has many flags to control specific details of optimization and code generation. Through experience, experiments, and extensive profiling, they find the appropriate settings for the code being tested.

High-performance computing: The authors of the ATLAS project have taken the approach used for benchmarking one step further [42]. They have hand-transformed the BLAS level 2 and level 3 library codes to optimize performance across a range of input problem sizes and machine types. Through extensive testing and experimentation, they have created a collection of variant implementations, and a decision procedure to select the best variant in a given situation. The user simply calls the high-level routine, and the ATLAS library does the rest.

Another alternative, proposed by Demmel *et al.*, is to code high-performance applications in a stylized dialect of C [5]. They identified a style of C code from which most compilers can generate high-quality code. Their experiments show that compilers for today’s machines have rather narrow ranges of code where they achieve good performance. Our adaptive compilers should widen that window where codes achieve good performance.

Embedded systems: In embedded computing, economic and physical constraints often make performance—measured in speed, in space, or in power—a critical issue. While the quality of embedded compilers is improving, it is often the case that the users resort to assembly code because they cannot achieve their objectives with a compiler. Thus, the limitations of current compilers force these users to step outside the framework of compiled code to meet their needs.

In each of these areas, performance is a serious issue. In benchmarking and high-performance computing, that performance is obtained by having an expert determine how the compiler should optimize the program. In embedded systems, where economic incentives have not produced the flexible, high-quality compilers of the benchmarking community and where the code reuse does not exist to sustain highly-optimized community codes like ATLAS, programmers resort to assembly code—bypassing the compiler and hand-optimizing the executable code. As these embedded processor grow more complex, our ability to program them in assembly code will diminish. For example, the TI TMS320C6x00 processors have an assembler that schedules the code; the task is too hard for most assembly-level programmers.

The promise of adaptive compilation is to bring the benefits of customized expert tuning—so useful in benchmarking and high-performance numerical libraries—to a much broader audience. By automating the discovery of program-specific compilation sequences, adaptive compilers can improve the quality of code produced for virtually all applications. With explicit, user-selected objective functions, these adaptive compilers can apply the same ideas and techniques to optimizing for other criteria—space, power, page faults, *etc.*

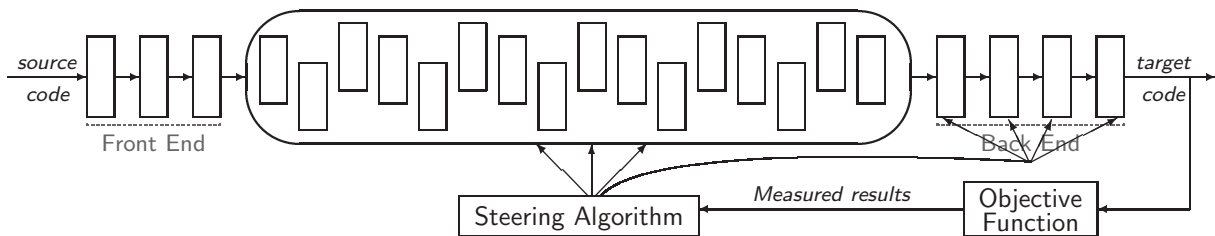


Figure 2: Structure of the Prototype Adaptive Compiler

The Structure of an Adaptive Compiler We have built a prototype adaptive compiler, shown in Figure 2. The front end remains intact, structured along the same lines as a classic compiler. The back end must still perform the final steps in translation: instruction selection, instruction scheduling, and register allocation. These processes, however, have parameters that the adaptive compiler can manipulate. The optimizer has been completely restructured. In place of a fixed sequence of passes executing in a predetermined order, it now consists of a pool of transformations, a steering algorithm, and an explicit, external objective function. The steering mechanism selects a compilation sequence and uses it to compile the input program. It evaluates the objective function on the resulting target-machine program, and uses the measured results to refine its choices and explore the space of possible compilation sequences. Through repeated experiments, the steering algorithm will discover a compilation sequence that minimizes the objective function, given the source code, the available transformations, and the target machine.

While our work, to date, has only addressed the selection of passes and the order of application, the work in this project will extend the steering algorithm’s control to address parameters to both back-end passes and to individual transformations. For example, as we add blocking optimizations to the adaptive compiler, the steering algorithm will need to discover appropriate settings for tile sizes, loop unrolling factors, and other critical parameters. These can have a significant impact on performance [9, 45, 5, 42].

Adaptive compilers address one of the fundamental challenges in the design and implementation of an optimizing compiler—what to do and in what order—by computing answers to these questions. They rely on the speed of modern computers to replace the 1960s compiler structure shown in Figure 1 with a structure that can adapt to new performance parameters, can change for new input programs, can incorporate new transformations, and can optimize for new objective functions (like the energy-time product suggested by Irwin *et al.* [26]). They apply inexpensive cycles to solve a problem in compiler design that has defied both theory and practice for forty years. They make the compiler’s objective function explicit and changeable, rather than implicit and fixed.

Early Results Our initial experiments with adaptive compilation have shown promising results. Using a simple genetic algorithm that targeted code size, with run-time speed as a tie-breaker, our prototype was able to reduce code size by an average of 13% over our fixed sequence compiler [13, 15]. This more than doubled what we could achieve with a direct compression technique via procedure abstraction [11]. Furthermore, the genetic algorithm produced code that was faster than the original code, where the compression technique always produced code that was marginally slower. Turning the objective function around and making speed the primary criterion with space as a tie breaker produced codes that were up to 20% faster than those produced by our fixed-sequence compiler, and marginally smaller on average.

3 Research Issues

To make adaptive compilation practical and to speed its adoption as a best-practice technique, we must perform basic research on the nature of the problem, on improved steering algorithms, and on practical ways to apply the knowledge gained in adaptive search to routine compilation. We must also make it easier for others to build order-independent passes for adaptive compilers. This work will combine extensive experimentation with ideas and insights from compilation, from artificial intelligence, and from statistics.

At each point in the project, we intend to push the limits of what is computationally feasible. The FMIN experiment described below will have consumed almost one year of computer time when it finishes. It will provide the first actual characterization of a compiler configuration space. The insights that we can gain from the analyzing the results will let us design more effective search techniques. They will also provide the base knowledge that will let us use smaller samples to characterize other spaces.

The Challenge of ATLAS To demonstrate the impact of adaptive compilation, we will apply the evolving prototype to the problem of tuning library codes, which Dongarra *et al.* have described:

The traditional method of handling this problem has been to produce hand-optimized routines for a given machine. This is a painstaking process, typically requiring many man-months of highly trained (both in linear algebra and computational optimization) personnel. The incredible pace of hardware evolution makes this technique untenable in the long run, particularly so when considering that there are many software layers (e.g., operating systems, compilers, etc) which also effect these kinds of optimizations, that are changing at a similar, but independent rate [42].

As we improve the adaptive compiler, we will compare the results of adaptive compilation against those of the hand-tuned implementations of the level 2 and level 3 BLAS routines from the ATLAS library. This activity will provide both a benchmark of our effectiveness and a convincing demonstration of the practical value of adaptive compilation. It will also guide our efforts to extend the current prototype compiler, stretching the limits of both our implementation and our understanding.

Implementation issues: We must extend our prototype with a number of transformations that it currently lacks—mostly loop-based transformations that block for memory-hierarchy performance. To use these transformations effectively, the adaptive compiler must find the appropriate parameter settings—choosing blocking factors, unrolling factors, and loop permutations. We will need to extend the steering algorithm to choose such settings.

Research issues: Adding these new transformations and parameters will expand the search space exponentially. This will place a premium on effective search strategies and pruning heuristics. Testing the impact of new configurations on scientific library codes is also expensive. This will reinforce the need for fast estimators and proxies that enable rapid evaluation of the objective function.

We believe that the prototype compiler can reach the point where it automatically tunes ATLAS-like codes to produce results similar to those achieved by human experts. (It will not, however, discover fundamental algorithmic improvements that the human expert might find.) If we succeed, this will make automatic tuning of this kind a reality for other kinds of application.

Indeed, Dongarra *et al.* explicitly recognize the role of search and experimentation in their work with ATLAS. They identify four critical requirements for “Automated Empirical Optimization of Software” (AEOS): 1) isolation of performance-critical routines; 2) a method of adapting software to differing environments; 3) robust, context-sensitive timers; and 4) an appropriate search heuristic [5, § 2]. Our adaptive compiler can automate the process of adapting software to differing environments. Our work on extracting and applying knowledge addresses the issue of deriving appropriate search heuristics; it should have direct application in their context as well as in our own.

3.1 Understanding the Problem Space

Our preliminary work on adaptive compilation has used search as a paradigm for finding compilation sequences rather than pursuing an analytical technique to derive appropriate sequences.¹ If the compiler could predict, with reasonable accuracy, the impact of a specific compilation sequence on a specific program in a given execution context, it would open up other avenues of research for us to pursue. Today, however, we know too little about the nature of the problem to devise such predictors.

Several factors make such predictions difficult and inaccurate.

- The improvement from a given transformation, in isolation, varies widely as a function of detailed properties of the input program. An optimization targets specific opportunities; it can only improve the code if those opportunities exist.² To estimate its impact with any accuracy, the estimator would need to discover those sites where the transformation applies and estimate their impact on execution time. Finding those sites is a large part of the transformation’s work; in most cases, the cost of an accurate estimator would be roughly the same as the cost of performing the transformation.
- The interactions between transformations, and their overlapping effects, make isolated predictions inaccurate. Consider trying to predict the effect of performing transformation *A* at some point in the compilation sequence. The predictor could determine the number of occurrences of code sequences that *A* can improve and it can estimate how often they execute. However, earlier transformations might either rewrite the code to reduce or eliminate these opportunities, or move them to points in the code where they execute a different number of times—either more often or less often. Thus, the predictor must operate on the code that is input to the transformation—the result of all previous transformations in the sequence. This fact, alone, makes accurate standalone prediction as expensive as performing the actual compile.
- The alternative appears to be deriving statistical models that capture, with less accuracy at less cost, the overall behavior of subsequences. This approach may prove fruitful; however, we lack the knowledge today to attack this problem. We will conduct experiments aimed at gaining enough knowledge and understanding about the behavior of transformations and the interactions between them to make more abstract estimators practical and sufficiently accurate. We will use ideas from machine learning in an attempt to correlate gross program properties with the results of adaptive compilation.

¹The analytical approach has had limited success. For example, Lam *et al.* recast loop transformations to improve memory behavior into a framework of unimodular transformations [29, 45]. They succeeded in deriving good optimization sequences for the unimodular transformations; however, their model included a limited set of transformations that attacked a single problem (cache locality) with a single objective function (reducing cache misses). The work does not generalize to arbitrary (non-unimodular) transformations.

²For example, Briggs and Cooper report results from algebraic reassociation that range from 49% fewer operations executed through 12% more operations executed [7]. It is clear that low-level details in the code being compiled determine the profitability of this technique. These details depend not only on the source code, but also on the impact of earlier optimization passes and on code-shape decisions made in the compiler’s front end.

Until we understand much more about the behavior of optimizations in their compile-time context, prediction will remain impractical.

The FMIN Experiment To begin our exploration of the problem space, we are conducting a major experiment to enumerate completely a subspace of the problem for one program. In other experiments, we noticed that one of our test programs, FMIN, displayed interesting and complex behavior, even though it is just 150 lines of Fortran [15]. Despite its size, FMIN has a rather complex control-flow graph—forty-four basic blocks. To explore the impact of optimization choice and optimization order, we ran the adaptive compiler with a hill-climbing search from a large set of randomly-generated starting points. We took the five transformations that occurred most frequently in those “winning” sequences and set up an experiment to enumerate the results of all sequences of ten passes drawn from these five (a set of 5^{10} , or 9,765,625 sequences).

To date, we have enumerated 5,597,161 individual configurations, at roughly 30,000 experiments per dedicated CPU day. (At 30,000 per CPU day, the entire experiment will take 326 CPU days.) The results, summarized in the following table, show that outstanding solutions are rare.

Score	Sequences	% of Solutions
Best	1	0.000036%
1%	14,696	0.26%
2%	91,437	1.6%
5%	480,421	8.6%
10%	576,193	10.3%
15%	640,222	11.4%
20%	640,223	11.4%
25%	708,034	12.6%
Worst	8,212	0.15%

The best sequence that the experiment has uncovered produced code that took 1,002 cycles to execute. So far, only one sequence achieves this result. The worst sequence takes 3,316 cycles to execute. We have found 8,212 sequences that achieve this result. With no optimization at all, the code takes 1,716 cycles to execute. With the fixed optimization sequence that our compiler used for years, it took 1,122 cycles to execute. (Note that the fixed sequence uses transformations not included in the FMIN experiment.)

The five transformations in the FMIN experiment can improve the code by 42% from the unoptimized code or make it 93% slower. The difference is a matter of transformation selection and transformation ordering. Even with this limited transformation repertoire, the adaptive compiler can improve the code by almost 11% over the fixed sequence compiler. If we add dead code elimination to the sequence that produced 1,002 cycles, we can produce code that runs in 822 cycles—a 27% improvement over the fixed sequence compiler and a 52% improvement over no optimization. Adding another transformation to the search space, however, increases the number of possibilities to over sixty million sequences.

With the full set of transformations, the adaptive compiler routinely finds solutions that are better than 1,002 cycles. Using a genetic algorithm for steering, it has discovered sequences at 822 cycles (in 184 generations of population size 100) and at 825 cycles (in 152 generations of population size 100). Using a hill-climber for the steering algorithm, it has found sequences at 830 (in 750 rounds) and 833 (in 2232 rounds). The potential improvement exists; the adaptive compiler discovers it.

The FMIN experiment should complete by the end of December, 2001. At that point, we will take the results (roughly ten million sequences and their fitness scores) and analyze them. Our goal is to gain fundamental insight into the nature of the search space. In particular, we are interested in the distribution of local minima, in the steepness of those depressions, and in the impact of those characteristics on the design of effective search algorithms. We also intend to try several machine learning techniques to look for significant patterns in the sequences. If we can discover for example, that certain replacements produce consistent results (when bc follows an a , rewriting bc with bd usually helps while be usually hurts), the compiler can use these insights to prune the search space or to let the search algorithms take larger steps.

3.2 Improved Search Algorithms

In the prototype system, we have implemented three distinct approaches to the steering algorithm:

Hill-climber: The hill-climbing search starts with a single random configuration and systematically improves it. At each step, the search tries to improve the current configuration. We have experimented with a steepest descent strategy and a randomized strategy. The former finds the largest improvement from the current configuration by systematically testing the options. It halts when no single replacement improves the current configuration. The latter randomly picks a pass to change and a replacement for it. Because it is harder to detect a minimum in this scheme, it halts after a specified number of trials.

Genetic algorithms: The genetic algorithm in the prototype is parameterized to allow experimentation. Our best results, in terms of both final outcome and trials to reach that outcome, have used pools of 100 to 300 variable-length chromosomes,³ a two-point randomized crossover, and scaled fitness values as weights in making reproductive choice. For efficiency, it memoizes the fitness values using the configuration string as its key.

One strategy incorporated into the genetic algorithm has proven particularly important. When it generates a new chromosome from crossover, it checks for that sequence in the existing pool. If the new sequence is a duplicate, it mutates the new sequence until it is unique. (Since we are using the genetic algorithm to search the configuration space, duplicates are wasted effort.) This “mutate until unique” heuristic helps the genetic algorithm avoid stagnation and duplicated effort; it appears to help the algorithm reach its final answer more quickly.

Exhaustive enumeration: To explore specific configuration spaces, we have built a steering algorithm that exhaustively enumerates a subspace. This simple algorithm turns the adaptive system into a data collection device. It returns configurations paired with their fitness values from the objective function. We have used this tool in our study with FMIN. Data of this sort is necessary to understand the characteristics of the configuration space and to design more effective search algorithms and pruning heuristics.

The current prototype is too slow for routine use. Improving the performance of the search process is critical. Today, we see several approaches for improving the searches. As we gain experience, other improvements will emerge; we will pursue them.

Pruning Heuristics Using experimental data, we will search for heuristics that can prune the search space. This work may produce absolute rules (“never replace B with D , unless an A precedes the B ”). It may produce probabilities to use as weights in selecting replacements (“replacing B with C succeeds more often than replacing it with D ”). Because even simple heuristics can shrink the space exponentially, they can have a major impact on reducing search times. Such heuristics can improve both the genetic algorithm and the hill-climbing approach.

Some pruning heuristics will come from correctness constraints, as ways of enforcing inter-optimization dependences. For example, the prototype’s implementation of partial redundancy elimination needs the code to have a particular names space [32, 20]. The global renaming pass builds that name space as a side-effect of its work [7]. Other pruning heuristics may encode useful knowledge about the interactions between optimizations. For example, the prototype’s strength reduction pass assumes that constant propagation and lazy code motion precede it [14]. It leaves behind dead induction variables, so that following it with dead code elimination is necessary for good results. Pruning heuristics that encode such insights (*i.e.*, ensure that dead code elimination eventually follows strength reduction) can rule out many potential configurations and shrink the search space.

³With fixed-length chromosomes, the genetic algorithm “discovers” NOPS to pad the strings. This necessitates a regimen of “knock-out” testing to find the substring that actually causes the improvement. Because of overlap in optimization coverage, the knock-out test must consider the effects of removing single passes and combinations of passes. It tries all the tests in both forward and backward order over the chromosome. With variable-length chromosomes and tie-breaking in favor of the shorter sequence, these NOPS don’t appear in the winning sequences.

Starting Points We will use our experimental experience to devise better ways of picking starting points. The current system chooses starting points randomly. It is clear that the starting points affect the behavior of both the hill climber and the genetic algorithm. By gaining experience, logging results, and analyzing those results, we hope to discover better strategies than random initialization. One outcome of our knowledge extraction work might be an analyzer that looks at the input program and suggests good starting points for the search—not necessarily points that, themselves, have good results, but points from which the search can rapidly find good solutions.

Adaptively Choosing Step Sizes By analyzing the data from empirical characterizations like the FMIN experiment, we expect to discover the “depth” of local minima—*i.e.*, how many individual replacements are needed to escape them. (By definition, this must be > 1 .) This kind of data will let us design searches that adaptively change their step sizes—the number of passes changed between two experiments. Ideally, this would let the hill-climber take bigger steps, across small valleys, to find a global minimum, rather than getting stuck in a shallow valley. Alternatively, it might let the hill-climber differentiate local minima from a valley that is likely to be a global minimum, and give it a mechanism for escaping the former.

Incremental Search To reduce the apparent cost of search, we will explore incremental strategies that distribute the search across multiple compilations. In such a scheme, the compiler would perform several experiments on each compile and accumulate the results. This would trade a minor increase in compile time for a long-term improvement in code quality. Open questions that must be investigated include the extent to which minor perturbations in the code being compiled affect configuration choice, and how to prevent stagnation. For example, an incremental search might require periodic upset—a randomized disturbance of its model—to help it avoid local minima based on the historical evolution of the input program.

Faster Experiments Each configuration that the search algorithm considers generates a small experiment. In the current prototype, each experiment entails compiling the program and evaluating the objective function on the result—a process that usually requires running the program. We will work on estimators and proxies to allow objective function evaluation without running the program. For example, memory operations or floating-point operations might serve as a proxy for the overall performance of some applications. Similarly, static estimates of operation counts may be accurate enough for the early stages of a search. Finding effective proxies and estimators will require extensive experimentation.

3.3 Extracting and Applying Knowledge

Our preliminary work has shown the need for better understanding of the search space. For example, the FMIN experiment will yield a complete mapping of one subspace of the adaptive compiler’s configuration, for one program. By analyzing that data, we can learn about the impact of various transformations and combinations of transformations on FMIN. It is an open question whether or not that insight generalizes to other programs. To begin answering that question, we need to relate results from our experiments back to fundamental program properties.

The relationship between the best configuration and the detailed structure of an application is complex. To discover which program properties are significant and how they relate to choices in configuration space, we will use both statistical analysis and techniques from machine learning.

We can treat the 10 million (*sequence, fitness value*) pairs produced by the FMIN experiment as samples of an unknown function f that maps optimization sequences to their fitness values. The learning problem is to find that f from these examples. The function f is a model of the impact of compiler transformations on the FMIN code.

There are two classes of functions from which f may be drawn: deterministic and stochastic. An example of a deterministic function class is boolean expressions in disjunctive normal form (also known as decision trees). Rules of the form “ BC followed by an A or D followed by EFG produces high fitness values” can be generated by a standard decision tree learning algorithm [37]. One of the key issues in the choice of learning algorithm is how to strike the right balance between generalization power and accuracy on the example data. We know that no finite set of examples is sufficient to uniquely identify a function f that discriminates sequences with high fitness from ones with low fitness values, for all programs. If the set of functions from which f is drawn is too large, then f will likely be heavily specialized to the given examples.

We would not expect such a function to perform well on new, unseen examples. This is called the overfitting problem, or the bias-variance tradeoff. On the other hand, if the set of the functions is too small, the learner is forced to represent an overly general approximation of the true function. Sophisticated learning algorithms like *support-vector machines* make principled trade-offs between bias and variance [17]. We will use support-vector machines to learn what distinguishes good sequences from bad sequences for the FMIN problem.

Stochastic learning algorithms infer conditional probabilities of the form $P(a|bc\dots f)$, which is the probability that optimization a yields a high fitness value when used in the context of optimizations $bc\dots f$. These techniques learn relationships between subsequences of optimizations and represent them as a directed acyclic graph called a belief network [22]. Associated with each node of the graph is a subsequence of optimizations as well as the conditional probabilities of the effectiveness of that subsequence given all subsequences in nodes that are its immediate parents in the network. This context-dependent model of the effectiveness of optimization subsequences can be used to predict the fitness value of any sequence in the space of possible optimization sequences.

While having either deterministic or stochastic models of how different optimization sequences perform for a particular benchmark can be useful, our ultimate goal is to understand the relationship between program properties and sequences that are particularly useful for them. The inductive learning methodology described above gives us algorithms to learn such models from examples of the form:

(program property 1, . . . , program property N, sequence, fitness value).

By studying more benchmark codes, we can put together a large database of such examples. Program properties will include properties that can be cheaply inferred from control and data-flow graphs as well as executions of the program. The models can be deterministic or stochastic functions that map program properties to sequences with high fitness values. Such predictive models will have much wider utility since they can be used to narrow the space of possible sequences for further tuning by a search algorithm.

Our goal for this portion of the project is to learn enough to relate program characteristics to outcomes. In the near term, we will use the results of this analysis to identify pruning heuristics. In the long run, we hope to understand the problem at a level where we can build tools that statically predict good configurations for use in a single sequence compilation or that predict good starting points for adaptive search.

Applications to Fixed-sequence Compilers Extracting knowledge from our experiments will be critical to our ability to derive better search techniques and to speed up adaptive compilation. At the same time, it should create the opportunity to apply new knowledge directly in more traditional, fixed-sequence compilers. We will explore approaches that include:

- *Training sets:* The user would invoke the adaptive compiler on a set of representative codes, with a specific objective function. The compiler would identify and keep the k best compilation sequences. Routine compilations with the production compiler would return the best result from using these k sequences.
- *Subsequence extraction:* By analyzing winning sequences, we may be able to construct a single sequence that obtains most of the benefit of full adaptive compilation for a single program or a collection of programs. Automating this analysis would create a complete tool for “tuning” a traditional fixed sequence compiler.

Our earliest experiments with adaptive compilation used a simple genetic algorithm to discover configurations that produced compact code [13]. In those experiments, we analyzed the best configurations by hand and were able to identify and extract common subsequences. Using this knowledge, we built a “good” sequence that obtained about 85% of the improvement, on average, with a single compilation. Finding such sequences can improve the overall behavior of a traditional compiler.

3.4 Extending the Prototype & Engineering for Reconfiguration

To extend the configuration spaces that our experiments encounter, we will add new transformations to the prototype adaptive compiler. For the ATLAS challenge, we must add a number of loop-based transformations

Assertion insertion	Logical peephole optimization
SCC-based value numbering [40]	Loop peeling
Global constant propagation [41]	Algebraic reassociation [7]
Dead code elimination [18]	Copy coalescing [8]
Partition-based value numbering [1]	Strength reduction [14]
Partial redundancy elimination [32]	Local value numbering
Global renaming	Lazy code motion [27]
Useless control-flow elimination	

Figure 3: Transformations in the Current Prototype

to achieve cache and register blocking, to reorder the iteration space for better locality, and to expose more instruction-level parallelism for scheduling.⁴

The difficult issue in extending the prototype is not choosing specific transformations, but rather engineering those implementations in ways that make them suitable for the adaptive compiler. The heart of the adaptive compiler is a collection of transformations that can produce correct results when run in arbitrary order. Few compilers work when used this way.

Our prototype includes a set of fifteen optimizations (see Figure 3). Extensive testing, particularly in adaptive search, has given us confidence that they can be run in arbitrary order. However, generating and debugging new passes for an adaptive compiler remains a challenging exercise. Our experience has illuminated a number of engineering issues that must be addressed to make building order-independent passes easier—a necessity before this technology can become widely usable.

Inter-optimization dependences Some optimizations depend, explicitly or implicitly, on the results of earlier transformations, or on their absence. Such a dependence can simplify the implementation of a transformation. For example, partial redundancy elimination [32, 20] relies implicitly on the code following a particular naming discipline. In the prototype, the “global renaming” pass constructs that particular name space, among other effects [7]. Running partial redundancy elimination without global renaming produces incorrect code. Thus, any configuration must respect and enforce the dependence between these two transformations.

We see two reasonable approaches to this problem. It will require experimentation to choose between them.

- All transformations can be made self-sufficient, for some penalty in compile-time. In the worst case, if B depends on A , we can package them as A and AB . (The prototype packages global renaming with partial redundancy elimination for correctness.) This frees the steering algorithm to pursue any optimization plan, but limits the set of valid combinations. For example, the search engine cannot generate ACB . It can generate $ACAB$, which might or might not have equivalent effects.
- The dependences can be expressed as constraints that a valid configuration must satisfy, and the search engine can be taught to respect those constraints. This approach would ensure that A executed before B , and allow ACB as well as $ACAB$. It forces the search engine to deal with a more complex set of facts. For example, D might destroy the properties that A creates and B needs, creating a need for rules of the form “ A must precede B without an intervening D .”

The best answer is probably a combination of these two approaches. The compiler writer can eliminate some dependences easily and efficiently. These should be removed. When that is not practical, the system should understand the dependences and enforce them.

Codifying Our Experience We have built a large collection of order-independent passes. Based on our experience, we will develop a set of design and implementation rules intended to minimize inter-optimization dependences. Such a guide can help a compiler writer avoid many of the problems. It will not, however,

⁴We already collaborate with members of the ATLAS team on other projects and they are interested in the results this particular experiment. We will consult with them to fill in the specific list of needed transformations.

prevent subtle order-dependent bugs from arising. For example, an implementor building partial redundancy elimination from the original paper might well miss the subtle implications of the brief discussion on expression shape.

To debug the prototype, we have developed a simple set of fault-isolation tools. These tools take a configuration that produces erroneous results and find the smallest subset of the configuration that contains the error. In our experience, even simple tools have proved invaluable. As we build and debug new passes, we will improve these tools and include them in the code base for the adaptive compiler.

A quiet revolution of modularity has crept across the various subfields of systems design. To achieve our goals, we must bring a truly modular style of implementation to the optimizing compiler. This will require a significant amount of programming in the demonstration system, along with a concerted effort to transfer the results of our experience to other compiler writers.

4 Related Prior Work

Previous attempts at building adaptive compilers [33] have focused on feeding dynamic profile information from program execution back into the compiler to guide optimization. Other attempts to use search in optimization include Nisbet's system, which used genetic algorithms in an attempt to parallelize loop nests [34, 21], and Massalin's Superoptimizer, which used exhaustive search in an attempt to perform optimal instruction selection [31]. Nisbet's system was ineffective, probably because search was not a good fit to his problem. Massalin's technique produced good results, but was too expensive for routine use. Granlund and Kenner adapted Massalin's ideas to produce a design-time tool that generates assembly sequences for use in GCC's code generator [23]. Our preliminary work using a genetic algorithm to find compilation sequences suggests that search is a good fit to the problem and that adaptive randomized sampling can yield good results in a reasonable amount of time.

A few authors have written on the interactions between optimizations in a specific way. In the context of performing incremental global optimization, Pollock and Soffa encountered such interaction; they characterized the interactions among the set of transformations in their system [35, 36]. Whitfield and Soffa characterized the interactions between a set of optimizations, including their ability to enable and disable opportunities for each other [43]. Later, they designed a systematic way of describing interactions between optimizations and for analyzing and working with those interactions [44].

5 Plan of Work

The proposed course of research requires a balance between experimentation, implementation, and algorithmic development. The experiments required to characterize the configuration spaces and to test different steering algorithms require large amounts of computer time—indeed, the wall time for the experiments is one limiting factor on our progress. At the same time, we propose to implement a number of complex transformations for inclusion in the prototype system. Again, this activity will take a significant amount of time. Finally, we must put together the results of our experiments—direct results as well as insights derived from extensive analysis of the data—to derive better algorithms, approaches, and systems.

This leads to a five year program of research, development, and experimentation. At each point in the project, we will be conducting one or two large-scale experiments to refine our understanding of the spaces where these adaptive compilers search.

In the first year, we will continue experimentation, while launching an effort to implement the appropriate transformations for the ATLAS challenge problem. The experiments will involve using larger search spaces on large programs, analyzing the data, using the knowledge to refine our search techniques, and testing the performance of the new search strategies. The implementation effort will require us to build tools for dependence analysis and to build the new transformations on top of our existing infrastructure. This effort will continue through the first and second years.

By the start of the project's third year, we expect to have the tools in place to start transforming the level 2 and level 3 BLAS routines. At this point, we will shift some of our experimental focus onto evaluating the performance of our adaptive system against codes like the ATLAS library. This work will expose new challenges; in particular, choosing discrete parameter values may require different strategies and pruning heuristics than selecting transformations and ordering them.

These new challenges will motivate the research portion of the project in the final two years. On the implementation front, we will shift some effort toward applying the knowledge extracted in the first three years of experimentation directly toward improving the behavior and performance of more traditional compilers. At the same time, we will continue to experiment with techniques to handle larger search spaces effectively.

We would like to include other compilers in our experiments. For example, the Texas Instruments compiler for the TMS320C6000 chips has some passes that can be reordered. Their current interface, inspired by our preliminary work, allows the user control over those passes. To date, however, our attempts to reorder passes in other research systems have been disappointing. For example, GCC contains myriad inter-optimization dependences that cause it to malfunction with even simple reorderings. We will continue to investigate other compilers that might work in our experiments; the SUIF 1 system, the SGI PRO64 compiler, and Intel's new open-source Itanium compiler are all possibilities.

This project will be characterized by iterative refinement—we will repeatedly refine our search strategies and our strategies for building production-style compilers. Our preliminary experience with a genetic algorithm that found compilation sequences for compact code gave us insight into the problems of selecting and ordering transformations. It led, directly, to many of the ideas presented in this proposal. As we explore these issues more deeply in this project, we expect to gain new insights and new knowledge that will spark further avenues for investigation.

6 Technology Transfer and Outreach

The principal investigators have a long history of designing algorithms that are used in commercial compilers. Techniques developed in the scalar compiler group at Rice are used in compilers from BOPS, Compaq, Ericsson, Hewlett-Packard, IBM, Intel, Motorola, Sequent, Silicon Graphics, SUN Microsystems, and Texas Instruments. (For example, our new strength reduction algorithm has already been implemented in compilers at Compaq, HP, Intel, and BOPS, even though the paper will not appear until later this year [14].) We understand the technical and political constraints that often prevent academic developments from appearing in commercial products. Our experience in directly addressing these problems and encouraging commercial groups to adopt best-practice techniques are critical to the success of our proposed project, since the results of this project will lie far outside common industrial practice.

We will take advantage of existing outreach efforts at Rice, including the Computer Science Affiliates Program and the annual meeting of the Los Alamos Computer Science Institute (LACSI), to publicize the results of this work. These meetings provide exposure to communities that often do not attend the major computer science research meetings—implementors from many companies in the case of the Affiliates Meeting and scientists from other disciplines at the national laboratories at the LACSI meeting.

We believe strongly in distributing the software produced in this project as widely as possible. For over a decade, we have made the implementations developed in our group available to other groups in both industry and academia. We will continue this practice with the code developed as part of this project.

The project will produce another set of artifacts: the results of our large scale experiments. These data sets may be, in practice, too large to place directly on a web site. For example, the FMIN experiment will produce roughly 160 megabytes of raw data. We will advertise the availability of this data on our project web site and will make it available to researchers who request it.

All three investigators have participated as speakers in Richard Tapia's GirlTech/MCSA program, which trains teachers in technology issues, retention issues, and gender issues in science and mathematics education. In our preliminary work, we have recruited female undergraduate students to program parts of the system and to run the experiments. As part of this project, we will aggressively seek out students from underrepresented groups for these undergraduate research positions.

Finally, Cooper and Torczon are writing a new textbook for an introductory compiler-construction course [16]. The first edition includes some material on program-specific optimization sequences. If this project succeeds, we will incorporate results from this research into the senior-level compiler course at Rice and provide deeper coverage of this material in the revised edition of the textbook and the accompanying lecture notes.

7 Significance

If this project succeeds, it will create the first compiler that systematically applies the increasing speed of our computers to the problem of building better compilers. This can change the economics of building compilers. It will create the technology to build a portable compiler that optimizes well for different applications, different systems, and different objective functions.

This project will develop the tools and techniques needed to build the next generation of optimizing compilers—both research systems and commercial products. These tools will directly address the economic problem that confronts compiler-builders: how to handle the rapid proliferation of processors, applications, performance goals, and environmental constraints without abandoning the notion of high-quality optimization. It is not economically feasible to produce distinct compilers for all these circumstances using current practices. Without a new way to organize, build, and tune optimizing compilers, we will force ourselves to accept poor quality code that fails to meet the users' real needs.

Our strategy—building adaptive compilers and the tools to automate the process of configuring them—ties the strength of our compilers to the speed of our machines—a resource that compilers have not exploited well in the past. These compilers will broaden the range of input programs that attain good performance. These compilers will be responsive to new performance goals, expressed in the form of new external objective functions. These compilers will easily accommodate new results from the research of others; new transformations can simply be added to the pool. These compilers may well automate a large part of the hand-tuning effort currently required to produce a high-performance numerical library. The compiler framework produced by this project will be an excellent platform from which to pursue development of specialized compilers and to pursue further research in optimization.

8 Management and Budget Issues

To provide the project with overall direction, the principal investigators will meet bi-weekly to assess the progress of experiments, development work, and new research. Dr. Torczon will work with the lead programmer to coordinate the development, experimental, and tech-transfer efforts. She will handle the day-to-day management tasks that arise in running an effort of this size.

The proposed budget includes one month of summer support for Dr. Cooper and for Dr. Subramanian. This covers a portion of the research time that they intend to devote to the project; the remainder of their research time falls under their academic-year responsibilities. Dr. Torczon is scheduled for three months of support; as a Research Scientist, this represents one quarter of her time. We expect that her effort will divide (roughly) into two-thirds research and one-third administration and supervision of development and technology transfer activities.

While all members of the research team will be involved in implementing tools, the lead programmer will supervise all development work. This includes new transformations, new steering algorithms, and the various data-analysis tools and debugging tools. The lead programmer will be responsible for ensuring the clarity of the implementations that we distribute. The lead programmer will post code to the web site, along with notices when new experimental data is available. The budget includes full-time support for the lead programmer.

The experiments that we intend to run take a long time to complete—days, weeks, and even months. In our preliminary experiments, we recruited a female undergraduate student to set up experiments, to gather and filter the data, and to perform preliminary data analysis. This exposed her to the research process and to the ideas involved in the project. It involved her directly in the publication process, as a co-author. We will aggressively recruit students from underrepresented groups to fill the undergraduate slots on this project. We expect that this will lead to deeper involvement in the long-term research agenda of this project.

This project will provide support and focus for three graduate students. As part of their thesis projects, they will take the lead in pursuing the major research issues, with appropriate advice and supervision from the principal investigators. They will need to work with the lead programmer, with the principal investigators, and with the undergraduate students on software development, on experimental design, and on data collection and analysis.

Project duration This proposal calls for a five-year program of research, development, and education. Five years is a realistic period to accomplish the proposed activities.

- We will implement a new set of transformations to block for memory and register performance, to manipulate iteration spaces of loops, and to improve computationally intensive loop nests in general. These transformations require dependence analysis, currently not implemented in our system. Thus, they represent an ambitious implementation effort.
- We will perform a series of ambitious, computationally-intensive experiments to demonstrate the technology and to push the boundaries of our knowledge. Our first such experiment, with FMIN, will consume a CPU-year and four wall-time months by the time it completes. The preliminary results from FMIN have led us to formulate two follow-on experiments: one with a stochastic hill-climbing strategy on FMIN and another that will try to characterize a bigger search space on a larger code.

At each point in the project, we intend to push the limits of what is computationally feasible. These experiments improve our understanding of compilation and should lead to direct improvements in our search strategies. Each of these experiments will take a significant period of time.

- We will produce, from our implementations, a set of reference implementations that we can distribute via our web site. From our experience over the last decade, we believe that this almost doubles the total development time. From the time that we have a working implementation to the time that we are ready to distribute it, we invest an effort that is roughly equal to the development effort. This time, invested in documentation, in creation of test data and examples, and in explaining the inner workings of the algorithm, pays off when other compiler writers use our reference implementations as guides to their own implementations.

To summarize, the research plan that we have proposed involves extensive development, aggressive long-term experimentation, and active production of reference implementations. We expect these activities to fill the five-year time period.

Results of Prior NSF Support

Keith D. Cooper Dr. Cooper is one of twelve principal investigators on the *Grid Application Development Software (GrADS)* project (grant EIA-9975020) funded by the NSF Next Generation Software program. NSF support for this project is currently expected to be \$5,599,994 over five years (10/99–9/04). Dr. Cooper's subproject is budgeted at \$400,000 over three years (10/99–9/02). The GrADS project is developing tools and techniques to simplify development of applications that run on distributed, heterogeneous networks. The NSF has also awarded a group of the GrADS investigators additional funding to continue the work on program-preparation issues raised by the GrADS project. That funding will be treated as a continuation of the GrADS contract, at a sharply-reduced funding level for two years. Dr. Torczon is also a principal investigator on the GrADS project and its NGS continuation. For further information, see <http://hipersoft.rice.edu/grads>.

Devika Subramanian Dr. Subramanian's research on automatic reformulation was supported by grant IRI-89027121 from 1989 to 1993. It resulted in algorithms for designing new representations by detection and elimination of irrelevant computation. 3 Ph.D. students and over 15 undergraduates were trained under this grant. IRI-9319409 awarded to Dr. Subramanian from 1995 to the present funds her research on automating conceptual synthesis of opto-electromechanical devices from specifications of behavior. It has resulted in real-time algorithms for conceptual design, and has produced several novel designs for imaging systems of copiers as well as multi-legged walkers.

Linda Torczon *Center for Research on Parallel Computation:* From 1990 to 2000, Dr. Torczon served as the executive director of the Center for Research on Parallel Computation (CRPC), an NSF Science and Technology Center established with the goal of making scalable parallel computer systems truly usable. To achieve this goal, the CRPC developed and distributed software systems for programming scalable machines in an architecture-independent fashion. The CRPC worked closely with scientific application groups to devise parallel algorithms for the solution of problems in linear algebra, numerical optimization, and numerical simulation of physical phenomena. The CRPC also pursued an active program of education and outreach to distribute widely parallel computing technology and research results; to transfer technology to industry; and to expand the pool of computational scientists and engineers, with a particular emphasis on minorities and women.

The CRPC was funded at approximately \$5 million/year (cooperative agreement CCR-9120008) from February 1989 through July 2000. Projects were shared among seven CRPC sites: Argonne National Laboratory, the California Institute of Technology, Los Alamos National Laboratory, Syracuse University, Rice University, the University of Tennessee, and the University of Texas. Ken Kennedy was the director of the CRPC. See <http://www.crpc.rice.edu/CRPC> for more detailed information on CRPC activities.

Girl Games Prototype CD-ROM Project: Girls Designing Games for Girls: Dr. Torczon was a principal investigator on this project (supplement to CCR-9120008), which investigated the computer game preferences of teenage girls and tried to translate those preferences into prototypes. The goal was to provide the entertainment industry with examples of games and game concepts that appeal to girls. The availability of such games should foster interest in computers among girls and encourage girls to pursue technological careers by breaking down their perceptions that “computers are not for girls.” To achieve the project goal, CRPC researchers at Rice University worked with Girl Games, Inc. to design interactive software that appeals to girls.

Group Travel Grant for Faculty at Minority/Female Institutions to Attend PLDI '97: Dr. Torczon served as tutorial chair for the 1997 Conference on Programming Language Design and Implementation (PLDI) of the ACM Special Interest Group on Programming Languages. In connection with that activity, she sought and obtained NSF funding (CCR-9702079) to help twenty faculty members from colleges with large minority and/or female enrollments attend both PLDI 97 and the teaching-track tutorials that accompanied it. The goal of the effort was to expose the faculty members to state-of-the-art research in programming languages, to provide them with materials and training that would help them improve their existing curricula and/or introduce new curricula, and to establish links among participants that would lead to a long-term network.

References

- [1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, Jan. 1988.
- [2] M. A. Auslander and M. E. Hopkins. An overview of the PL.8 compiler. *SIGPLAN Notices*, 17(6):22–31, June 1982. *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction*.
- [3] J. Backus. The history of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.
- [4] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference*, pages 188–198, Feb. 1957.
- [5] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ANSI c coding methodology. In *International Conference on Supercomputing*, pages 340–347, 1997.
- [6] J. G. Bradley and G. A. Frantz. DSP microprocessor power management: A novel approach. Texas Instruments Technical White Paper, 1998.
- [7] 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*.
- [8] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–456, May 1994.
- [9] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. *SIGPLAN Notices*, 25(6):53–65, June 1990. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*.
- [10] K. D. Cooper. Why is redundancy elimination hard? Excerpt from talk at Rice Computer Science Affiliates Meeting. Available at <http://www.cs.rice.edu/~keith/1960s>, Oct. 2000.
- [11] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proceedings of the ACM SIGPLAN 99 Conference on Language Design and Implementation*, May 1999.
- [12] K. D. Cooper and P. J. Schielke. Non-local instruction scheduling with limited code growth. In *Proceedings of the 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, June 1998.
- [13] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, May 1999.
- [14] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Transactions on Programming Languages and Systems*, to appear 2001.
- [15] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. In *Proceedings of the 2001 LACSI Symposium*. Los Alamos Computer Science Institute, Oct. 2001. Available at <http://www.cs.rice.edu/~keith/LACSI>.
- [16] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan–Kaufmann Publishers, 2001.

- [17] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, Cambridge, 2000.
- [18] 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.
- [19] DARPA PAC. Proceedings of the May 2000 PAC/C PI meeting. *Many of the talks are online on the meetings agenda page*. See <http://darpa.mil/ito/research/pacc/meetings.html> .
- [20] K.-H. Drechsler and M. P. Stadel. A solution to a problem with morel and renvoise’s “global optimization by suppression of partial redundancies”. *ACM Trans. Prog. Lang. Syst.*, 10(4):635–640, Oct. 1988.
- [21] N. G. Fournier. Enhancement of an evolutionary optimising compiler. Master’s thesis, Department of Computer Science, University of Manchester, Sept. 1999.
- [22] N. Friedman, D. Geiger, M. Goldszmidt, G. Provan, P. Langley, , and P. Smyth. Bayesian network classifiers. *Machine Learning*, 29:131, 1997.
- [23] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the GNU C compiler. *SIGPLAN Notices*, 27(7):341–352, July 1992. *Proceedings of the ACM SIGPLAN ’92 Conference on Programming Language Design and Implementation*.
- [24] Intel, Microsoft, and Toshiba. Advanced configuration and power interface specification. *Manual for joint Intel-Microsoft-Toshiba interface*, Feb. 1999.
- [25] M. S. Johnson and T. C. Miller. Effectiveness of a machine-level global optimizer. *SIGPLAN Notices*, 21(7):99–108, July 1986. *Proceedings of the ACM SIGPLAN ’86 Symposium on Compiler Construction*.
- [26] M. Kandemir, N. Vijaykrishnan, M. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *Proceedings of the International Symposium on Computer Architecture*, June 2000.
- [27] 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*.
- [28] M. Lam and the SUIF group. Documentation with the SUIF-2 compiler release. *Available from the SUIF web site*, <http://suif.cs.stanford.edu> .
- [29] M. S. Lam. Locality optimizations for parallel machines. *Lecture Notes in Computer Science*, 1994.
- [30] E. Lowry and C. Medlock. Object code optimization. *Commun. ACM*, pages 13–22, Jan. 1969.
- [31] H. Massalin. Superoptimizer – A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, Palo Alto, CA., 1987.
- [32] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Commun. ACM*, 22(2):96–103, Feb. 1979.
- [33] T. Night. <http://www.ai.mit.edu/projects/transit/tn101/tn101.html>. Web site for the Transit Project.
- [34] A. P. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *Poster Session at the International Conference and Exhibition on High-Performance Computing and Networking, HPCN Europe ’98*, 1998.
- [35] L. L. Pollock. *An Approach to Incremental Compilation of Optimized Code*. PhD thesis, University of Pittsburgh, Department of Computer Science, 1986.

- [36] L. L. Pollock and M. L. Soffa. Incremental global reoptimization of programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(2):173–200, Apr. 1992.
- [37] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
- [38] J. Reynders. Components and scalability. *Colloquium talk at Rice University. Reynders discussed the performance of large-scale applications in various environments at LANL*, Feb. 2000.
- [39] Silicon Graphics, Inc. Documentation with the SGI PRO64 compiler release. *SGI released the compiler in open-source form during the summer of 2000. Code is available from the company.*, 2000.
- [40] L. T. Simpson. *Value-driven Redundancy Elimination*. PhD thesis, Rice University, Department of Computer Science, Apr. 1996.
- [41] M. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.
- [42] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. Technical Report UT-CS-00-448, Department of Computer Science, University of Tennessee, Knoxville, Sept. 2000.
- [43] D. L. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. *SIGPLAN Notices*, 25(3):137–146, Mar. 1990. Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP).
- [44] D. L. Whitfield and M. L. Soffa. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):1053–1084, Nov. 1997.
- [45] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.
- [46] W. Wulf, R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke. *The Design of an Optimizing Compiler*. Programming Language Series. American Elsevier Publishing Company, 1975.