Excerpted from Chapter 1 of

Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization,

Robert Metzger and Zhaofang Wen, MIT Press, 2000.

Introduction

Optimizing compilers have a fundamental problem. No matter how powerful their optimizations are, they are no substitute for good application algorithms. Consider the case of sorting. For sufficiently large data sets, a merge sort algorithm compiled with a less powerful optimizer will always out-perform a selection sort algorithm compiled with the most powerful optimizer. Or consider the case of solving systems of equations. For sufficiently large data sets, a Gaussian Elimination algorithm compiled with a less powerful optimizer will always out-perform a Cramer's rule algorithm compiled with the most powerful optimizer.

Developers of optimizing compilers also have an opportunity to leverage an under-used asset. There are many high-quality numerical libraries that are publicly available, such as the BLAS and LAPACK, that provide broadly applicable algorithms for scientific and engineering computing. Vendors of high performance computers often provide versions of these libraries that have been highly tuned to their particular system architecture. Users of high performance computers sometimes employ these libraries. Unfortunately, many users are unaware of their existence, and don't use them even when they are available.

What if compilers could recognize a poor algorithm written by a user, and replace it with the best implementation of a better algorithm that solves the same problem? Given reasonable implementations of both algorithms, such a replacement would result in as significant performance improvement. This book explains an approach that makes this possible.

The scope over which compilers perform optimizations has steadily increased in the past three decades. Initially, they performed optimizations on a sequence of statements that would be executed as a unit, e.g. a basic block. During the 1970's, researchers developed **control flow analysis, data flow analysis** and **name association** algorithms. This made it possible for compilers to do optimizations across an entire procedure. During the 1980's, researchers extended these analyses across procedure boundaries, as well as adding **side effect analysis**. This made it possible for compilers to do optimizations across an entire program.

We can characterize the analyses behind these optimizations from another perspective besides scope. They all add data structures that represent information synthesized from the program source. Some add new abstract entities, like a basic block. Others add new relationships, like **control flow dominance.** Some add both. The more abstract entities and relationships a compiler has available to characterize the semantics of a program, the more opportunities exist for optimization.

Now that commercial compilers can analyze a program across its entire scope, optimization researchers need to ask where the next layer of synthesized information will come from. This book explains an approach that adds a whole new layer of entities and relationships to the semantic analysis that compilers can perform on application codes.

Parallel computation will be the norm in the twenty-first century. Parallel hardware has gone from use in super-computers to departmental servers. Recently, we have seen multiple processors available even in high-end workstations and PC's. Unfortunately, the parallel potential of hardware has raced far ahead of the parallel applications of software.

There are currently two approaches to applying parallelism to applications. The first is to write completely new applications in new languages. Abandoning applications that work is simply unacceptable to most non-academic users of high performance computers.

The second approach is to convert existing applications written in existing languages to a parallel form. This can be done manually or automatically. The labor required to rewrite applications to make use of current parallel systems is great. Even partial success in automatically parallelizing existing codes has obvious economic advantages. This book explains an approach to automatically parallelizing applications that is complementary to current automatic parallelization methods.

Prehistory of the Solution

The past often explains the present. In our case, our professional experience prior to the project that resulted in this book provides clues to the motivation and direction that the project took. We have identified the following activities as motivations for our research:

- interaction with people doing competitive benchmarking,
- experiences implementing vectorizing compilers,
- experiences implementing interprocedural compilers, and
- interest in compiling APL programs.

Competitive Benchmarking

The peculiar nature of the high-performance computer business provided a major motivation for investigating automatic algorithm recognition. At some point in the lengthy (six months or more) sales cycle, the customer will narrow down the prospective vendors to a short list. Some customers only use third-party applications. They will ask the vendors to run these applications on the bid configuration with data sets that represent their workload. If they have written their own applications, they give the vendors the most important of these for porting and execution on the vendors' hardware.

The task of porting and running the application falls to a pre-sales software engineer. The time that this engineer has to get the application running and showing the best performance on the target system is limited. Typically, one to four weeks are allowed. Sometimes just a few days are available.

Once the engineer has the application ported and generating correct answers, he or she turns to the problem of optimizing performance. Customers provide a variety of constraints on what can be done at this point. At one extreme, a customer may require that no changes be made to the source code of the application. This makes the benchmark activity as much a test of the compiler's automatic optimization capabilities as the performance of the hardware system. All the engineer can do is specify compiler command-line options, typically stored in a **makefile**.

At the other extreme, the customer may allow the vendors to rewrite any portion of the system they want in assembly language to achieve a performance improvement. This makes the benchmark

activity as much a test of the expertise of the benchmarkers as the performance of the hardware system. The sophisticated customers ask for both approaches, in order to evaluate the difference between typical and peak performance.

During application tuning, the engineer often sees unrealized opportunities for performance improvement. The engineer may observe procedures that do the same computation as highly optimized procedures in the vendor's mathematical subroutine library. If the customer's constraint is ``no source changes,'' he or she can't insert a call to the library subroutine. The engineer may observe assembly code generated by the compiler that is slower than what could be written manually. If the customer's constraint is ``high-level language source changes only,'' he or she can't insert a call to an alternative procedure written in hand-polished assembly code.

These circumstances led to regular requests from pre-sales engineers for compilers that make handling benchmarks simpler. One request was to provide a feature to match user code and replace it with calls to special procedures. These requests led us to see the potential commercial value of an algorithm recognition project.

Such a feature must be usable by a programmer who knows nothing about the internal workings of a compiler. The patterns must be generated directly from high-level language source. The code replacement actions must be a simple specification of a name and a mapping between expressions in the original program and arguments of the special procedure. Adding a new pattern to the knowledge base must be no more complicated than running the compiler with some special command line options.

Vectorizing Compilers

In 1988, Convex Computer Corporation began shipping its C-2 series of vector-parallel processors. The Fortran compiler (version 5.0) for that system included a powerful optimizer that performed automatic **vectorization** and **parallelization.**

At that time, the Livermore Loops suite was one of the key benchmarks used to evaluate high performance systems and their compilers. Two of the loops were not vectorizable by standard algorithmic means. The loop computing the partial **prefix sum** of a vector has an actual **recurrence**. The loop searching for the first minimum value of a vector and returning the index has a premature loop exit. Recurrences and loop exits hinder standard vectorization algorithms.

To get the maximum performance on the Livermore loops, the Fortran project leader implemented a special pass after normal vectorization. This pass found two loops that could be partially realized with vector instructions. He described the technique as follows: ``The only pattern matching performed by the compiler currently is to recognize recurrences that can in fact be vectorized but not by straightforward methods.'' Most compilers for high-performance computers contain such techniques for improving their benchmark standing, but vendors are rarely willing to admit it publicly.

The compiler internal representation used **directed graphs** to represent expressions **data flow**, and **control flow**. The pattern matching for the two loops was a cascading series of tests on the arcs and nodes that represented a singly-nested loop. It took 40 tests to identify the vector prefix sum loop and 73 tests to identify the index of minimum search. These counts treat a *switch* statement as one test. They also include support for both Fortran and C, since this was a language-independent optimizer.

Once the candidate loop was identified, the contents of the loop were removed, the trip count was set to one, and a call to a vectorized runtime library procedure was inserted. Writing the identification and replacement code was time-consuming and error-prone.

Once the pre-sales engineers found out that the compiler could pattern match these two loops, they asked for similar support vectorizing additional similar loops. One of us (Metzger) took over responsibility for the Fortran compiler project after the initial pattern matching work was completed. He added support for matching and replacing two more loop types that had not previously been vectorizable because of premature loop exits:

- searching a vector for the first element of a vector that matches a scalar quantity according to a relation (e.g., equality), and
- searching for the first minimum magnitude value of a vector and returning the index.

It took 80 tests to identify generalized search loops and 71 tests to identify the index of minimum magnitude search. The same caveats apply as with the test counts for the original patterns.

This effort was not purely an exercise in **benchmarking**. Loops that search an array for maximum or minimum values or for an element that meets a certain criterion occur frequently in scientific and engineering applications. Subroutines that implement these loops are found in the standard version of the BLAS (Basic Linear Algebra Subroutine library).

A vendor-supplied version of the BLAS typically runs substantially faster than the version one created from compiling the public domain sources. If a programmer replaces these loops with calls to the vendor's version of the BLAS, the application will speed up. The procedures called when the loops were pattern matched were special versions of the vendor-supplied BLAS subroutines. So when the compiler matched the loops, the application ran faster without the programmer having to make any changes.

The lack of generality and the labor-intensive nature of the work, however, made it untenable for further development. We wanted an approach to recognizing and replacing patterns that was general and easier to maintain. We believed such an approach required an external **pattern database**. This database would consist of patterns and actions that could be maintained and enhanced without having to change the source code of the compiler itself.

Interprocedural Compilers

In 1991, Convex Computer Corporation began shipping a product, called the *Application Compiler*, which performed **interprocedural optimization** on programs written in Fortran and C. It was the logical conclusion to a series of increasingly powerful optimizing compilers that became available during the previous two decades. First, compilers performed optimization over a **basic block**, which is a group of sequentially executed statements with a single exit point. Next came compilers that performed optimization over an entire procedure. Finally, compiler technology reached the scope of an entire application.

There were two main motivations for developing this product. The first was to develop new sources of information that would improve **scalar optimization** and vectorization. The second was to automatically parallelize loops that contained procedure calls. With the release of version 2.0 of the Application Compiler at the beginning of 1995, both of these goals were reached.

The *Application Compiler* performed the following analyses:

- Call Analysis -- Which procedures are invoked by each call?
- Alias Analysis -- Which names refer to the same location?
- Pointer Tracking -- Which pointers point to which locations?
- Scalar Analysis -- Which procedures (and subordinates) use and assign which scalars?
- Array Analysis -- Which procedures (and subordinates) use and assign which sections of arrays?

Having a compiler that could automatically parallelize loops that contained procedure calls did expose more high-level parallelism. Unfortunately, it also exposed the inefficiencies in the sequential code that was being parallelized. Some questions naturally arose when this phenomena was observed.

- Could a given sequential algorithm be replaced by another more efficient sequential algorithm?
- Could a given sequential algorithm be replaced by a parallel version of a different algorithm that computes the same result?
- Could a compiler do this automatically?

As the *Application Compiler* became a mature product, there were few new optimizations that remained to be implemented. The department that produced this compiler specialized in compiler optimization. It was clear that we needed to undertake research that would open new doors for optimization. Algorithm recognition seemed to be the most likely candidate.

A programmer can frustrate any algorithm recognition system that works only on a procedural level by hiding some of the details in called procedures. Such modular structure is considered good programming practice. The Application Compiler presented us with a platform for investigating algorithm recognition when dealing with applications written in a modular style.

The results of interprocedural analysis can be used to determine whether a called procedure is relevant to an algorithm to be recognized. If the call is relevant, the procedure can be substituted inline, so that the code executed by the called procedure can be completely analyzed. The Application Compiler had a complete facility for doing this substitution. It also had the infrastructure necessary to use profile information to identify the computational kernels of the application. It became the logical basis for an algorithm recognition project.

APL Compilers

One of us (Metzger) had a long-standing interest in the compilation of APL programs. APL is typically interpreted, rather than compiled. This is because, at any point during execution, names can be bound to an object that has a different data type or dimensionality than the previously bound object.

APL interpreters normally operate by preparing operands and executing an operation by dispatching the appropriate runtime subroutine. Several researchers have worked on hybrid interpreter-compilers, or even ``pure'' compilers for APL since the late 1970's. These compilers typically generate code by composing data structures representing the access patterns of the APL operations into a demand-driven execution model.

These APL compiler efforts have largely focused on optimizing individual statements. This makes sense since APL is a very high level language. One line of APL can be the equivalent of pages of C or Fortran.

There are several kinds of interpretive overhead within a statement that such compilers can reduce:

- checking operand types before each operation,
- converting operands to the correct type,
- dispatching the correct runtime procedure for a given operation,
- allocating temporary storage for results, and
- copying values to and from temporary variables holding results.

What is not optimized are the actual operations themselves.

APL interpreters are typically coded so that individual operations on an array are just as efficient as any compiled code. An APL interpreter makes a series of calls to runtime subroutines, even for a single statement. An APL compiler replaces those calls with a single call to a procedure generated to execute the equivalent of that line of APL code.

One of the well-known characteristics of APL is the use of **idioms:** ``An idiom is a construction used by programmers for a logically primitive operation for which no language primitive exists.''

Most APL interpreters provide very limited support for idiom recognition. The APL grammar is simple enough that it is often analyzed with a finite state machine. Those idioms that can be recognized during the syntax analysis are replaced with calls to runtime procedures that are not directly accessible to the user.

Snyder's paper on “Recognition and Selection of Idioms for Code Optimization” suggested an alternative approach to compiling APL. He describes an algorithm to find idioms in an expression by tree matching. If his approach is used to implement a line of APL with a call to a single runtime procedure, it provides the same order of execution speedup as the other compilation model.

The advantage of Snyder's approach is that it is extensible by the user. If a line of APL code is a performance bottleneck, the user can add a pattern for the idiom in the pattern database, and a procedure that implements it in the runtime library. Snyder's paper inspired us to reconsider trees as a representation for algorithm recognition, when the consensus approach favored graphs.

Our prior experience in developing compilers for Convex and our personal research interests led us to the following conclusions.

1. Experience with competitive benchmarking showed us that algorithm recognition could be commercially valuable.
2. Vectorizing by ad hoc pattern matching showed us that pattern matching was useful for optimization. It must, however, be driven by a database of patterns and actions external to the compiler in order to be maintainable.
3. Developing an interprocedural optimizing compiler showed us that automatic algorithmic parallelization would expose inferior sequential algorithms. It also showed us that it was possible to deal with obstacles to algorithm recognition through program transformations.
4. APL compiler research, and Snyder’s approach in particular, inspired us to reconsider trees as a representation for algorithm recognition, when the consensus approach favored graphs.