Automatic Isolation of Compiler Errors

DAVID B. WHALLEY
Florida State University

This paper describes a tool called *vpoiso* that was developed to automatically isolate errors in the *vpo* compiler system. The two general types of compiler errors isolated by this tool are optimization and nonoptimization errors. When isolating optimization errors, *vpoiso* relies on the *vpo* optimizer to identify sequences of changes, referred to as transformations, that result in semantically equivalent code and to provide the ability to stop performing *improving* (or unnecessary) transformations after a specified number have been performed. A compilation of a typical program by *vpo* often results in thousands of *improving* transformations being performed. The *vpoiso* tool can automatically isolate the first *improving* transformation that causes incorrect output of the execution of the compiled program by using a binary search that varies the number of *improving* transformations performed. Not only is the illegal transformation automatically isolated, but *vpoiso* also identifies the location and instant the transformation is performed in *vpo*. Nonoptimization errors occur from problems in the front end, code generator, and *necessary* transformations in the optimizer. If another compiler is available that can produce correct (but perhaps more inefficient) code, then *vpoiso* can isolate nonoptimization errors to a single function. Automatic isolation of compiler errors facilitates retargeting a compiler to a new machine, maintenance of the compiler, and supporting experimentation with new optimizations.

General Terms: Compilers, Testing

Additional Key Words and Phrases: Diagnosis procedures, nonoptimization errors, optimization errors

1. INTRODUCTION

To increase portability compilers are often split into two parts, a front end and a back end. The front end processes a high-level language program and emits intermediate code. The back end processes the intermediate code and generates instructions for a target machine. Thus, the front end is dependent on the source language and the back end is dependent on the instruction set for the target machine. Retargeting such a compiler for a new machine requires creating a new back end.

Much of the effort required to retarget a back end occurs during testing. Often much time is spent determining why code generated by a compiler for a program does not execute correctly. Determining the

A preliminary version of the error isolator was described in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation* under the title "Isolation and Analysis of Optimization Errors."
Author's address: Department of Computer Science B-173, Florida State University, Tallahassee, FL 32306, U.S.A.; e-mail: whalley@cs.fsu.edu; phone: (904) 644-3506

reason has been typically accomplished in two steps. First, the compiler writer attempts to isolate the instructions generated by the compiler that cause incorrect execution. The next step is to determine why the compiler generated these incorrect instructions. Both steps can require much time and effort. The resolution of a compiler error may easily require hours or even days.

This paper describes a tool that automatically isolates compiler errors. For optimization errors, the tool can automatically determine the first transformation during the optimization of a program that causes the output of the execution to be incorrect. Nonoptimization errors occur from problems in the front end, code generator, and *necessary* transformations in the optimizer. If another compiler is available that can produce correct (but perhaps more inefficient) code, then the first nonoptimization error can be isolated to a single function.

2. OVERVIEW OF THE COMPILER

The tool described in this paper supports automatic isolation of errors in the *vpo* compiler system [1]. The optimizer, *vpo*, replaces the traditional code generator used in many compilers and has been used to build C, Pascal, and Ada compilers. The back end is retargeted by supplying a description of the target machine. Using the diagrammatic notation of Wulf [2], Figure 1 shows the overall structure of a set of compilers constructed using *vpo*. Vertical columns within a box represent logical phases which operate serially. Columns divided horizontally into rows indicate that the subphases of the column may be executed in an arbitrary order. IL is the Intermediate Language generated by a front end. Register transfers or register transfer lists (RTLs) describe the effects of legal machine instructions and have the form of conventional expressions and assignments over the hardware's storage cells. For example, the RTL

```
r[1] = r[1] + r[2]; cc = r[1] + r[2] ? 0;
```

represents a register-to-register integer add on many machines. While any particular RTL is machine-specific, the *form* of the RTL is machine-independent.

All phases of the optimizer manipulate RTLs. One advantage of using RTLS is that optimizations can be performed on machine-specific instructions in a machine-independent manner. Another advantage is

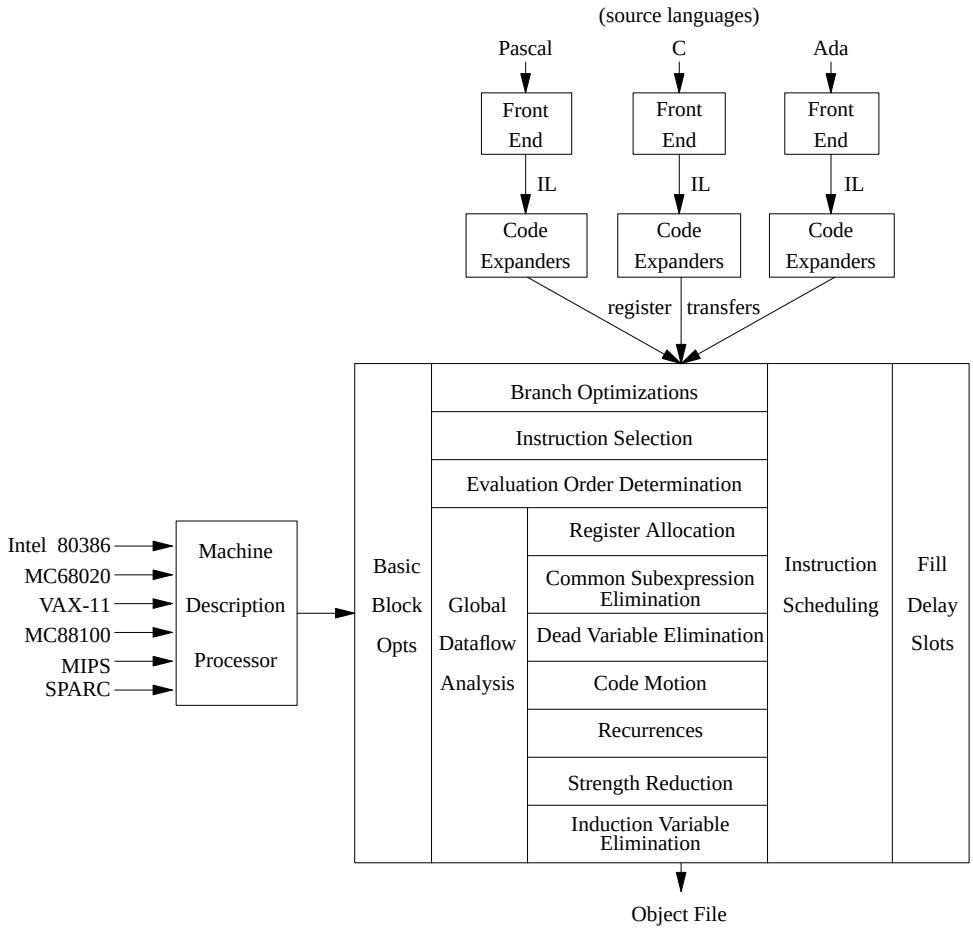

Figure 1: Compiler Structure

that many phase ordering problems are eliminated since optimizations are only performed on RTLs. Most optimizations can be invoked in any order and are allowed to iterate until no more improvements can be found.

The RTLs are stored in a data structure in *vpo* that also contains information about the order and controlflow of the RTLs within a function. The *vpo* optimizer was modified to identify each *change* to this data structure and to denote each serial sequence of changes that preserves the meaning of the compiled program. In this paper these sequences of changes are referred to as *transformations*.

3. ISOLATION OF OPTIMIZATION ERRORS

Testing is often the most time-consuming component of retargeting a back end of an optimizing compiler to a new machine. Much of the time spent during testing involves isolating errors in an optimizer to determine why specific programs do not execute correctly. One must not only determine what was produced incorrectly in the erroneous program, but also at what point it was produced within the compiler.

3.1. Traditional Isolation of Optimization Errors

Traditionally, the compiler writer initially attempts to determine the specific instruction (or instructions) generated by the compiler that causes the compiled program to execute incorrectly. One could first isolate a function that contains incorrect instructions. This is accomplished by compiling some functions with optimizations and other functions without optimizations and executing the program. If the program executes correctly, then the compiler writer knows the problem is in the set of functions that were not compiled with optimizations. Otherwise, the compiler writer assumes the problem is in the set of functions that were compiled with optimizations. The compiler writer continues to narrow down the set of functions that could contain an error until the function with incorrect code is isolated.

The compiler writer can then compile the isolated function with and without various optimizations until finding the additional optimization being applied to the function that causes the compiled program to execute incorrectly. At this point the compiler writer can visually inspect the differences between the two assembly versions of the functions in an attempt to determine the instruction or instructions that appear to cause incorrect behavior.

Given that the compiler writer is able to conclude that a specific instruction within a function causes the compiled program to produce incorrect results, finding the reason why the compiler produced this instruction is the next task. One approach is to successively turn off compiler optimizations until the offending instruction is no longer produced in an effort to identify the specific optimization that has caused the problem.

While these techniques may sometimes be effective, they are also quite tedious. Furthermore, some compiler optimizations that reduce execution time while increasing code size are becoming more popular. These optimizations include subprogram inlining [3], loop unrolling [4], and replicating code to avoid unconditional jumps [5]. When these types of optimizations are applied, a single function may expand into several thousands lines of assembly code. Visual inspection of such functions to discover incorrect instructions is impractical. Using traditional methods to identify the point in the compiler that causes an invalid instruction to be produced in these functions may also be unrealistic. Identifying the optimization that produces the problem may be difficult since the instruction may only be produced when a specific combination of optimizations are performed. Even if the compiler writer happens to correctly identify the optimization that produces the problem, the point in the compiler when the incorrect transformation occurs still has to be found. A specific optimization in *vpo* may be applied in hundreds of transformations on RTLs when compiling a single function.

3.2. Automatic Isolation of Optimization Errors

A tool, called *vpoiso*, has been developed to automatically isolate errors in the *vpo* compiler system. This tool isolates optimization errors by determining the first transformation that causes incorrect output from the execution of the compiled program. First, the optimization phases applied by *vpo* were classified as one of two types, *necessary* or *improving*. A *necessary* phase is required to produce code that can be compiled and executed. These phases, which are usually regarded as code generation activities, include assigning pseudo registers to hardware registers and fixing the entry and exit points of a function to manage the run-time stack. All phases within the optimizer that are not required are referred to as *improving*. Only *improving* transformations that cause incorrect output can be isolated by *vpoiso*.

The *vpoiso* tool performs a binary search that relies on the ability to limit the number of *improving* transformations applied to a specified function. Preceding and following each transformation, *vpo* invokes functions called `starttrans` and `endtrans` respectively. In the `endtrans` function, which is invoked when the end of a transformation is identified, *vpo* checks a counter to determine if the specified limit to the number of *improving* transformations has been reached. Unfortunately, *vpo* can be in quite

deeply nested routines and logic at a point when a transformation has been completed. To check a status flag at each of the points after returning from the `endtrans` function to prevent further *improving* transformations would have required significant modifications to *vpo*. To minimize the updates to the optimizer, the UNIX `setjmp` and `longjmp` functions were used to back out of code within *vpo* when the last transformation was performed. Execution then resumes within a high level routine and only the remaining *necessary* transformations are applied.

The *vpoiso* tool is a C program which uses the C `system` function to invoke various UNIX shell commands. First, *vpoiso* reads in a file of information indicating how to isolate an error within a program. This information includes the basenames of the files that are output from the code expander (or input to *vpo*), link and execute commands, maximum cpu time in seconds allowed for execution (i.e. in case an error causes the program to not terminate), desired and actual output filenames, compilation flags (the user can specify any combination of optimizations to be performed), and strings indicating lines to disregard (i.e. the output contains information dependent on time). For instance, a manufactured error was inserted during the compilation of the program *yacc*. To isolate the error, the following information was input to *vpoiso*.

```
cexfiles: y1 y2 y3 y4 #
link command: cc -o yacc y1.o y2.o y3.o y4.o
execute command: yacc cgram.y
maximum time: 15
desired output file: yacc.out
actual output file: y.tab.c
compilation flags: LVGOCMSFA
disregard strings:
```

After reading this information *vpoiso* has to determine if an incorrect transformation can be isolated. Thus, *vpoiso* invokes *vpo* for each file to be compiled with an option set to record for each function the basename of the file in which the function resides, the function name, and the number of *improving* transformations required. The *vpoiso* tool then links and executes the program using the specified commands. If the actual output is the same as the desired output, then *vpoiso* quits after informing the user that it could find no error when all optimizations were applied to each function in the program. Otherwise, *vpoiso* reads the information generated during the previous compilation and invokes *vpo* for each file to be compiled

indicating that no *improving* transformations are to be performed. Again, *vpoiso* issues commands to link and execute the program. If the actual output is the same as the desired output, then *vpoiso* has determined that the problem is an optimization error and it performs a binary search to isolate the first incorrect *improving* transformation. The binary search is depicted in the following pseudocode.

```
lastmin = 0;
lastmax = total number of improving transformations
while (lastmax - lastmin > 0) {
   midnum = (lastmin + lastmax)/2;
   recompile program with only the first midnum transformations performed
   remove actual output file
   link and execute program
   if (actual output file == desired output file)
      lastmin = midnum+1;
   else
      lastmax = midnum;
   }
if (last result was incorrect)
   badtrans = midnum;
else
   badtrans = midnum+1;
```

At this point *vpoiso* prints the name of the function containing the first incorrect transformation and the incorrect transformation number within that function.[1] The user can then set a breakpoint in a source-level debugger executing *vpo* that will stop when the transformation with that number is encountered. The `starttrans` function in *vpo* that is invoked when the start of a transformation is identified contains the following portion of code.

```
...
if (opttransnum == breakopttransnum)
   fprintf(stderr, "improving transformation breakpoint encountered\n");
...
```

The user assigns the displayed transformation number to the `breakopttransnum` variable, sets a breakpoint at the line where the message is printed, and executes *vpo*. Thus, using this feature, the compiler writer can quickly access the point during the compilation that precedes the incorrect *improving* transformation.

---

[1] The *vpoiso* tool is only guaranteed to find the first *improving* transformation that causes incorrect output. It is possible that a previous transformation was invalid and the isolated transformation was the first transformation that moves invalid instructions into a path that was executed. This situation has not occurred when testing *vpoiso* with manufactured or actual errors.

3.3. Decreasing the Isolation Time

The potentially most time-consuming component of the execution of *vpoiso* when isolating an optimization error is the recompilation of each instance of the compiled program during the binary search performed. A naive implementation of *vpoiso* would recompile the entire program before each execution. In a previous implementation of *vpoiso* [6], recompilation was limited to the files that were within the current search range. If the transformations on functions in a file were not within the current search range that could contain the first incorrect transformation, then the file was not recompiled. Recompilation of a file was also unnecessary when all the functions in the file would be compiled with the same number of transformations as in the previous compilation. In addition, if a function was in a file that needed to be compiled and it was not within the current search range, then the function was compiled with no optimizations to decrease the compilation time.

The current implementation of *vpoiso* further decreases isolation time by substituting file merging for recompilation when possible. Even recompiling a function with no optimizations requires much more time than simply copying the function from a file. Some *necessary* transformations, such as assigning pseudo registers to hardware registers and parsing each RTL using the machine description to translate it to an assembly code instruction, are expensive operations. As mentioned previously, *vpoiso* first compiles the program with all *improving* transformations applied and then no *improving* transformations applied. This determines if there is an error and if it is the result of an *improving* transformation. The *vpoiso* tool saves the assembly and object code files from both of these compilations. An assembly comment was also inserted between functions to facilitate the identification of the start and end of a function. The algorithm for performing the binary search was slightly modified. The `midnum` value, which represents the middle of the current search range of *improving* transformations that could contain the error, gets adjusted to the closest function boundary. Instead of having *vpo* process the code expander files, the assembly file that is to contain both optimized and unoptimized functions is created by merging these functions from the corresponding assembly files produced by the two initial compilations. At the point that the error is isolated to a single function, the portion of the code expander file containing information for that function is extracted.

The binary search that is performed on the transformations within that function only requires recompilation of this single function. The preceding functions in the file are merged in from the corresponding optimized assembly file and the subsequent functions are merged from the unoptimized assembly file.

To illustrate the performance of *vpoiso* for finding optimization errors, the results for finding a manufactured optimization error inserted into the compilation of the *yacc* program is described. There were a total of 13,955 *improving* transformations applied with the complete optimization of *yacc*. Three different versions of *vpoiso* were executed to illustrate the effect of attempting to avoid unnecessary recompilation. All three versions perform a binary search to isolate the first invalid *improving* transformation. However, the level of recompilation at each point during the binary search varied with each version. The first was the naive approach that processes all the files for each recompilation. The second version was the previous implementation of *vpoiso* [6] that avoided unnecessary recompilation of files that were outside the current search range or were processed the same as during the previous compilation. The third version was the current implementation that avoids recompilation by using file merging when possible. The *vpoiso* tool required 16 compilations/executions of *yacc* for each version to correctly isolate the erroneous transformation on a Sun SPARC IPC.[2] The time required for each version is shown in Table 1.

Version	Minutes:Seconds
1	17:41
2	9:56
3	6:08

Table 1: Time Required to Isolate an Optimization Error in Yacc

4. ISOLATION OF NONOPTIMIZATION ERRORS

If the execution of a program that was compiled with no optimizations by *vpo* does not produce correct output, then the error must have been introduced by the front end, code expander, or a *necessary* transformation in the optimizer. Unlike *improving* transformations in the optimizer, actions performed by the

[2] Note that the first 2 executions were only performed to verify that an incorrect transformation could be isolated.

front end, code expander, or *necessary* transformations by the optimizer cannot be selectively disabled to isolate an error. Yet mistakes in constructing the code expander, which expands each intermediate operation to instructions for the machine, are typically encountered more frequently than problems in the coding of the optimizer when the *vpo* compiler system is retargeted to a new machine.[3] Invalid code expander operations often result in nonoptimization errors. If another compiler is available that can produce correct (but perhaps more inefficient) code, then *vpoiso* can isolate nonoptimization errors to a single function.

The isolation of nonoptimization errors by *vpoiso* is accomplished in the following manner. It is assumed that for each specified code expander file, there exists a corresponding assembly file generated by a native compiler for the machine. For this paper the native compiler was *pcc* (Portable C Compiler) [7]. After determining that incorrect output is still produced when no *improving* transformations are applied, *vpoiso* first modifies the labels in the native assembly files to ensure they are unique from the labels in the nonoptimized assembly files generated by *vpo*. The *vpoiso* tool then performs a binary search on the functions in the program. For instance, in the *yacc* program there are 48 functions and they were associated with numbers from 0-47. Functions associated with a number less than or equal to the `midnum` value, which now represents the middle of the current search range of functions that could contain the error, are obtained from the nonoptimized assembly files generated by *vpo*. Functions associated with a number greater than the `midnum` value are obtained from the native assembly files.[4] If a file is to contain both nonoptimized functions generated by *vpo* and functions generated by the native compiler, then the file is created by merging the appropriate functions from the *vpo* and native compiler generated files.[5]

Ideally, a compiler writer would like to obtain a finer level of isolation of nonoptimization errors comparable to that when *vpoiso* isolates optimization errors. One possibility is to isolate the error to a C

---

[3] Most optimizations are performed in machine-independent code within *vpo* since the general form of an RTL is machine-independent. Therefore, errors in the retargeting of the optimizer occur relatively infrequently compared to non-optimization errors since most of the errors have already been diagnosed and corrected when *vpo* was retargeted to other machines. It may be that optimization errors are more frequently encountered during the maintenance of a compiler since most nonoptimization errors can be detected during the initial testing of the compiler. It has been the experience of this author that optimization errors are typically more difficult to isolate by hand than nonoptimization errors.

[4] Note that both *vpo* and the native compiler have to use similar calling sequences since functions compiled by each of the compilers will be intermixed.

[5] Isolation of nonoptimization errors introduced some machine-dependencies in *vpoiso* since the native assembly files were used. This included machine-dependent functions to modify labels and to identify the start of a function.

statement since most front ends, including the C front end for *vpo* called *vpcc* [8], can identify the intermediate operations associated with each C statement. One approach would be to attempt to merge the assembly code generated by the *vpo* and the native compiler within the function that was identified as containing the nonoptimization error. While no user-allocable registers are typically live across C statements when no optimizations are performed, other problems may arise. For instance, the offset of local variables on the run-time stack or the amount of space allocated on the run-time stack could differ. In addition, labels that are the target of goto statements will have to be consistent with the labels referenced in the jump instructions used to implement the gotos. While such an approach may be feasible, the implementation would be very dependent on the code generation strategies used by the native compiler.

Another approach is to use a tool that separates each C statement within a function into separate functions (i.e. each C statement would be replaced with a function call). This tool could be applied to the function with the nonoptimization error. A binary search could then occur on the newly generated functions to identify the first C statement with the error. Unfortunately, access to local variables and parameters must somehow be permitted. Introduction of new C statements to allow this access may introduce new nonoptimization errors, which would subvert the isolation process.

5. APPLYING THE TECHNIQUES TO OTHER OPTIMIZERS

There are certain features of *vpo* that simplified the development of the tool to isolate optimization errors. Performing code generation before all optimizations allows *vpoiso* to accurately determine that a code generation error was not caused by the optimizer. If code generation was performed after optimizations, then a code generation error may only occur when the intermediate representation is in a specific form (e.g. a particular instance of a dag). When the number of optimizations performed is reduced, this specific form may not appear. In this situation it would be difficult to have a tool automatically determine that the error was not caused by the optimizer. The structure of the *vpo* optimizer also made it easy to stop performing *improving* transformations at any point during a compilation. This ability may not be as straightforward to implement in other optimizers.

6. COMPARISON WITH RELATED WORK

A tool known as *bugfind* [9] was developed to assist in the debugging of optimizing compilers. The *bugfind* tool attempts to determine the highest optimization level at which each file within a program can be compiled and produce correct output. To isolate a function that was not optimized correctly, one has to place each function within the program in a separate file. The *bugfind* tool uses the *make* facility in Unix and is generalized enough to work with different compilers.

While *bugfind* and *vpoiso* share some similar ideas, there are also considerable differences. Both *bugfind* and *vpoiso* use a binary search technique to isolate optimization errors. The *vpoiso* tool finds not only the failing module, but also the first transformation within a function that causes incorrect results. The transformation number can be used to access the point in *vpo* when the transformation is about to be applied. This finer level of isolating errors is important when optimization errors occur in large functions or code size increasing transformations are performed. The *vpoiso* tool also isolates nonoptimization errors to the first function that causes incorrect output. Unlike *bugfind*, *vpoiso* can only isolate errors within the *vpo* compiler system. However, the techniques *vpoiso* uses can be applied with many other compilers.

7. CONCLUSIONS

The tool described in this paper provides several important benefits. Both the isolation of optimization and nonoptimization errors are important when retargeting the back end of a compiler. A tool with the ability to automatically isolate the first function containing incorrect instructions will be very valuable for finding many nonoptimization errors. For instance, a common code generation error when retargeting a compiler to a new machine is to incorrectly implement the calling sequence. An oversite when constructing the code expander, such as an inappropriate indication of a dead register, may result in an invalid *improving* transformation. Thus, isolation of optimization errors may also be useful for finding problems not in the optimizer itself.

The *vpoiso* tool will also be useful when experimenting with new optimizations. While it may be obvious that the newly introduced optimization was responsible for causing an error, manual isolation of the actual error can still be quite challenging, particularly when employing code size increasing

optimizations. For instance, a new loop optimization may be applied to 50 different loops in a given program. The *vpoiso* tool not only isolates the illegal *improving* transformation, but also identifies the location and instant the transformation is performed in *vpo*.

There are other benefits of using a tool that can automatically isolate compiler errors. If a compiler is used as a commercial product, then it typically has to be maintained for several years after its initial release. This maintenance includes responding to bug reports from users. Automatic isolation of compiler errors will ease this task. Compilers can also be used to guide instruction set design to determine if proposed architectural features can be exploited [10]. Decreasing the time to retarget a compiler to a proposed architecture would also decrease the time required to design and develop a new machine.

The techniques described in this paper could perhaps also be used with other applications besides compilers. The techniques of isolating optimization errors could be applied to any application that performs a series of optional transformations on its input. A technique similar to isolating nonoptimization errors may be used with applications that are developed with a configuration management tool. One could develop a system to automatically determine the first set of changes to an application that causes incorrect results. Testing and maintenance are expensive phases in the software product life cycle. Tools that can automatically isolate programming errors would have enormous potential benefits.

ACKNOWLEDGEMENTS

The authors thank Jack Davidson for allowing *vpo* to be used for this research. The identification of each change and the sequences of changes that comprised the transformations in *vpo* was simplified by the high quality of coding of *vpo*, which in a very large part is due to the efforts of Manuel Benitez. Indrakshi Ray tested *vpoiso* and made several suggestions that resulted in an improved tool.

REFERENCES

1. M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).
2. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, NY (1975).

3. J. Davidson and A. Holler, “A Study of a C Function Inliner,” *Software—Practice & Experience* **18**(8) pp. 775-790 (August 1988).

4. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA (1990).

5. F. Mueller and D. B. Whalley, “Avoiding Unconditional Jumps by Code Replication,” *Proceedings of the SIGPLAN ’92 Conference on Programming Language Design and Implementation*, pp. 322-330 (June 1992).

6. M. R. Boyd and D. B. Whalley, “Isolation and Analysis of Optimization Errors,” *Proceedings of the SIGPLAN ’93 Conference on Programming Language Design and Implementation*, pp. 26-35 (June 1993).

7. S. C. Johnson, “A Tour Through the Portable C Compiler,” *Unix Programmer’s Manual, 7th Edition* **2B** p. Section 33 (January 1979).

8. J. W. Davidson and D. B. Whalley, “Quick Compilers Using Peephole Optimizations,” *Software—Practice & Experience* **19**(1) pp. 195-203 (January 1989).

9. J. M. Caron and P. A. Darnell, “Bugfind: A Tool for Debugging Optimizing Compilers,” *Sigplan Notices* **25**(1) pp. 17-22 (January 1990).

10. J. W. Davidson and D. B. Whalley, “A Design Environment for Addressing Architecture and Compiler Interactions,” *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).