A Portable Global Optimizer and Linker

Manuel E. Benitez

Jack W. Davidson

Computer Science Technical Report TR-87-22
December 1, 1987

A Portable Global Optimizer and Linker

MANUEL E. BENITEZ
JACK W. DAVIDSON

University of Virginia

To reduce complexity and simplify their implementation, most compilers are organized as a set of passes or phases. Each phase performs a particular piece of the compilation process. In an optimizing compiler, the assignment of function and order of application of the phases is a critical part of the design. A particularly difficult problem is the arrangement of the code generation and optimization phases so as to avoid phase ordering problems caused by the interaction of the phases. In this paper, we discuss the implementation of a compiler/linker that has been designed to avoid these problems. The key aspect of this design is that the synthesis phases of the compiler and the system linker share the same intermediate program representation. This results in two benefits. It permits the synthesis phases of the compiler to be performed in any order and repeatedly, thus eliminating potential phase ordering problems. Second, it permits code selection to be invoked at any point during the synthesis phases as well as at link time. The ability to perform code selection at link time presents many opportunities for additional optimizations. Measurements about the effectiveness of using this approach in a C compiler on two different machines are presented.

1. INTRODUCTION

An important aspect of the design of a compiler is the overall structure and organization of the compiler. To reduce complexity and simplify their implementation, most compilers are organized as a set of passes or phases where each phase performs a particular job or function. For example, most compilers include a lexical analysis phase that converts the input source text into tokens that are passed to the syntax analysis phase. The assignment of function and order of application of the phases is a critical part of compiler design. This is particularly true for the phases of the compiler that are responsible for code generation and optimization. It is extremely difficult to design these phases so that they can operate in isolation thereby simplifying their implementation, yet are able to interact to produce good code. The problem of designing and ordering the phases so that they work together and do not interfere with each other is called the *phase ordering problem.*

A classic example of a phase ordering problem involves the interaction of the register allocation and instruction selection phases of the code generator. Register allocation determines which program variables will reside in the machine's registers, while instruction selection determines which target machine instructions should be used to implement a source language construct. In many compilers, register allocation is performed before instruction selection. The reason, of course, is that to choose the most efficient instruction(s) to operate on a value, it is necessary to know where that value is located—in a register or in memory. Unfortunately, instruction selection can affect the number of registers available for allocation. For example, code for some source language constructs may require the use of registers to hold temporary values, or certain machine instructions may require the use of particular registers. An approach used in some compilers is to assume an infinite supply of virtual or pseudo-registers. Instruction selection is performed, after which the virtual registers are mapped to hardware registers. If there are not enough hardware registers, code to spill and load registers is introduced. Had it been known at instruction

selection time that these values would be spilled and loaded, better code could have been generated.

In this paper, we describe the implementation of a compiler that uses a single representation for the synthesis phases of the compiler. The major advantage is that these phases of the compiler can be invoked in any order and repeatedly if necessary. This largely eliminates code inefficiencies often caused by the ordering of the phases. The system linker also operates on the intermediate representation used by the compiler. This gives the linker the ability to perform optimizations that could not be done by the compiler whose view is limited to a single function.

This paper has the following organization. Section 2 presents the intermediate representation and the organization of the compiler. We use a number of examples from real machines to illustrate how phase ordering problems are eliminated. The system linker and the optimizations it performs are described in Section 3. Section 4 describes the implementation and presents some measurements of the effectiveness of the compiler/linker. A comparison with related work is presented in Section 5. Section 6 contains a summary.

2. COMPILATION

We have implemented a C compiler that uses the same intermediate representation throughout all the synthesis phases. This compiler has been ported to a number of machines using a variety of approaches to building the back end. In this paper, we describe a C compiler which includes a back end that performs a number of optimizations. Using the diagrammatic notation of Wulf [9], Figure 1 shows the overall structure of this compiler.

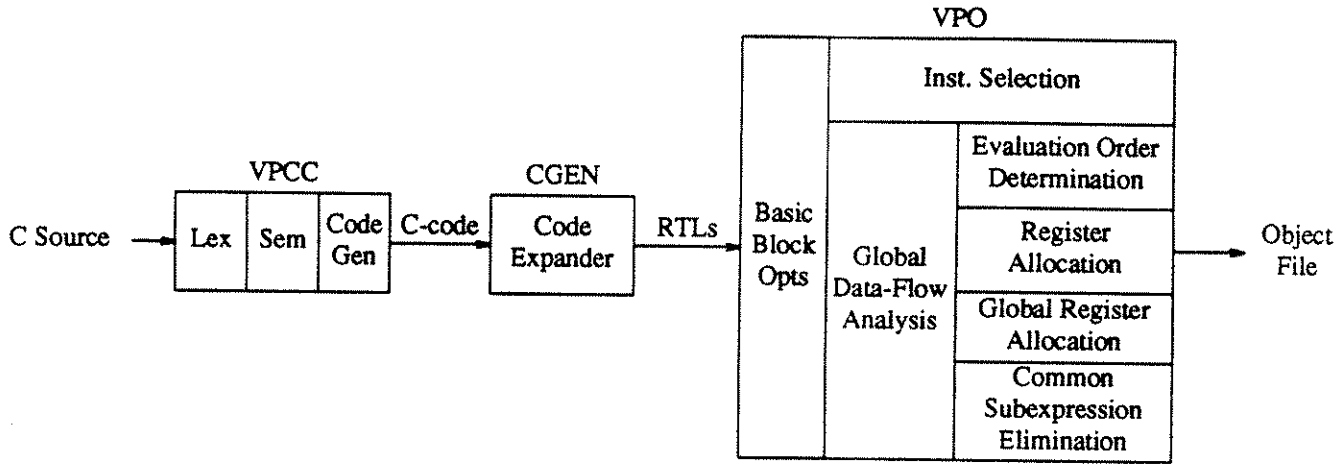

Figure 1. Schematic of C Compiler

Vertical columns represent logical phases which operate serially. Columns that are divided horizontally into rows indicate that the subphases of the column may be executed in an arbitrary order. In this compiler, block reorganization occurs before all other phases. Instruction selection may then be performed at any time during the optimization process. Global data-flow analysis is performed before register allocation, register assignment, global register assignment, and common subexpression elimination. These phases can be invoked in any order.

2.1 Intermediate Representation

The intermediate representation emitted by the code expander and processed by all the optimization phases consists of three types of records. One type of record contains linker directives. A second type of record contains information of use to the global data-flow analysis and common subexpression elimination phases. The third type of record contains a machine record in the form of a register transfer list ('RTL'). An RTL represents, in a machine-independent fashion, a machine dependent operation. 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 RTL is machine specific, its form is not. All phases of the compiler as well as the linker can manipulate RTLs.

There are a number of advantages to using RTLs as the basis of the intermediate representation. Because the form is machine-independent, programs can be constructed that manipulate RTLs in machine-independent ways. For example, the phase that performs data-flow analysis on RTLs is largely machine independent. Because RTLs represent actual machine instructions, specifics of the target machine are exposed to the various optimization phases which results in more complete and thorough optimization. Finally, because RTLs are well-defined, it is possible to construct recognizers that can determine whether an RTL represents a legal target machine instruction. The ability to determine, at any time, whether an RTL represents a legal target machine instruction is key to our optimization strategy. The following sections describe the operation of the C compiler. We focus on *vpo*, the optimizer that implements the code generation and optimization strategy.

2.2 Front End/Code Expander

The front end of the compiler is called *vpcc* (Very Portable C Compiler). It emits a stack-oriented intermediate code called C-code which is similar in spirit to P-code [1], except that it models a RISC machine. There are only 43 C-code operations. C-code is sufficiently general that it can be interpreted [5], or translated directly into machine code [8]. The code expander's job is to translate each C-code statement into machine-specific code.

Constructing compilers using the above approach provides a number of benefits. First, the front end's code generator is easy to build. The simplicity of C-code permits the complexity of the code generator at the level of a project in a compiler construction class. This permits the implementation effort to be focused on the more difficult semantic analysis phases. Finally, writing a code expander for an intermediate language with only 43 simple operations is quick and easy. Indeed, full C compilers (including bit fields, floating point, union, and enumeration types) have been constructed in a couple of days by writing a code expander that translates C-code operations to naive assembly language sequences.

2.3 VPO

The first phase of *vpo* (Very Portable Optimizer) reads the input produced by the code expander and partitions the program into basic blocks. The first optimization performed is the rearrangement of the basic blocks to minimize the number of jumps. At this point, instruction selection via peephole optimization [3] is performed on each basic block. This reduces by a factor two to three the volume of RTLs that must be processed by the data-flow analysis phase.

The next step in the compilation process is global data-flow analysis. This is performed early in order to determine lives and deads and reaching information that is necessary for register assignment. In order to keep the optimizer relatively language-independent, we use an iterative approach to solving the data-flow equations that permits arbitrary flow graphs. After performing data-flow analysis, instruction selection is performed again to detect optimizations that data-flow analysis indicated could safely be performed across basic blocks.

After data-flow analysis and any subsequent instruction selection nececessary, evaluation order determination and register assignment are performed. Register assignment maps the virtual registers used by the code expander to the available hardware registers. Using the data-flow information, the register assigner is able to use temporary registers (such as r0 and r1 on a VAX). For a machine with a small register set, it is possible that there will not be enough hardware registers. In this case, spills and loads are introduced as necessary. If register spills do occur, instruction selection is performed to see if the spill code can be optimized.

Consider the example in Figure 2. At line 9, register 2 must be spilled to a temporary location. The old contents of register 2 is reloaded at line 11 before the value is needed. By invoking instruction selection after the spill code is introduced, the final code uses a memory-to-register add instruction instead of a load and a register-to-register add instruction.

	Unmapped RTLS		*After Mapping*		*After Inst. Selection*
1.	`r[23] = L[i]`	1.	`r[2] = L[i]`	1.	`r[2] = L[i]`
2.	`r[25] = r[23]`	2.	`r[3] = r[2]`	2.	`r[3] = r[2]`
	.		.		.
	.		.		.
	.	9.	`L[tmp] = r[2]`	9.	`L[tmp] = r[2]`
10.	`r[28] = L[j]`	10.	`r[2] = L[j]`	10.	`r[2] = L[j]`
	.		.		.
	.		.		.
	.	15.	`r[3] = L[tmp]`		.
16.	`r[28] = r[28] + r[23]`	16.	`r[2] = r[2] + r[3]`	16.	`r[2] = r[2] + L[tmp]`

Figure 2. Register Assignment

After register assignment and any subsequent instruction selection that is needed, global register allocation is performed. We note that the global register allocator now knows how many registers are available for allocation. Using information provided by the front end and the global data flow analysis information, variables that can be allocated to registers are selected. A number of cost criteria are used to rank the variables that are candidates for allocation to a register. After variables are assigned to registers, instruction selection is performed again. We have found this to be particularly useful as many machines have instructions that require certain operands to be held in registers.

Consider the example in Figure 3. The register allocator determined that the loop index should be held in a register. The instruction selection phase then determined that `d[4]` could be used directly and eliminated the unnecessary register-to-register move. Note that in the final instruction sequence one fewer register is being used. This register (`d[5]`) is now available for allocation.

If a local variable is allocated to a register, there are several other optimizations that may occur. For functions with only a few local variables, often all the locals will be allocated to registers. In this case, the space on the stack to hold these values is no longer needed and the instructions that allocated this space are removed. In some cases,

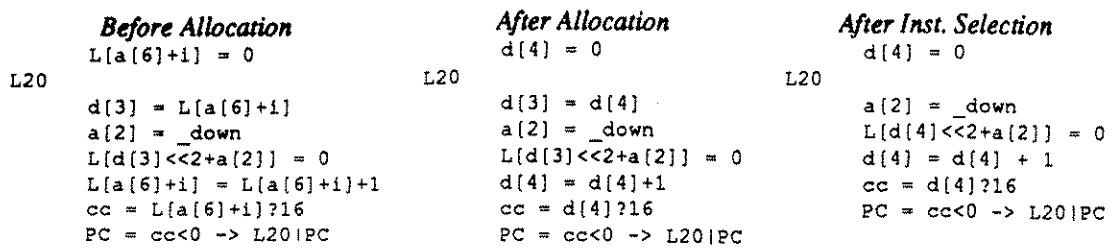

Figure 3. Global Register Allocation

not all locals are allocated to registers, but enough are to reduce the space needed for locals on the stack. In these cases, a smaller or more efficient instruction may be used to allocate the smaller amount of space. Finally, the code that initialized the local variables is examined. It is often the case that more efficient code sequences can be used to initialize local variables now held in a registers.

The last phase of *vpo* is the common subexpression eliminator. Because *vpo* operates at the machine level, common subexpressions that higher-level approaches may miss are detected and eliminated [4]. Previous implementations of this algorithm operated early, before the virtual registers had been mapped to hardware registers. Because the number of registers that were available for holding common subexpressions was not known, inferior code was sometimes produced. A number of pathological cases were discovered where the code produced with the common subexpression optimization enabled was worse than the code produced when it was disabled [2].

In a manner similar to the register allocator, if the common subexpression eliminator makes any changes to the code, instruction selection is reinvoked. Actually, the instruction selector is invoked during common subexpression elimination to determine if the change made by the common subexpression eliminator is legal and cost-effective. If registers become available after instruction selection, global register allocation is also reinvoked.

The code generation strategy outlined in the preceding paragraphs has proven to be flexible and effective. The ability to perform instruction selection at any time and repeatedly if necessary largely eliminates phase ordering problems. Section 4 contains comparisons of the code generated by this compiler and the code generated by other compilers.

3. LINKING

The object files produced by the compiler are linear representations of data structures operated on by the synthesis phases of the compiler. The system linker reads and processes these files performing the normal functions of a linker. In addition to the normal processing a linker must perform, this linker performs a number of optimizations. After reading the input files, the linker builds an extended call graph that shows the flow from procedure to procedure. It then attempts to perform optimizations across procedure calls and returns. A common optimization is the removal of unnecessary test instructions after a procedure call. Consider the following program fragment taken from the Unix utility *grep*.

```
 ...
if (advance(p1, p2)) {
 ...
 }
```

The code generated for this fragment is

```
 ...
L[--a[7]] = a[3]
L[--a[7]] = a[2]
pc = jbsr.(_advance)
a[7] = a[7] + 8
cc = d[0] ? 0
pc = cc != 0 -> L2 | pc
 ...
```

Using the extended flow graph, the linker is able to determine that all possible paths back to test instruction set the condition codes correctly and were not reset by any intervening instructions. It is able to delete the test instruction. In *grep*, there are nine such occurrences that are deleted.

In addition to optimizing the return code, the linker can modify the calling sequence so that parameters are passed in registers. Here we also consider a fragment from *grep*. For this example, we examine the entry to the routine `advance`.

```
_advance:  L[a[7]] = link.(a[6],-40)
           a[7] = moveml.(3c3c)
           a[4] = L[a[6] + ep]
           a[5] = L[a[6] + lp]
            ...
```

Using the extended flow graph, the linker determines that the parameters to advance could be passed in registers. At the site of the each call to `advance` the code is edited to pass the parameters in `a[0]` and `a[1]`. The routine itself is edited to take these parameters from the registers instead of the stack. In addition, the instruction that adjusts the stack pointer after the function call is no longer required and is deleted. For the previous examples, the final code is

```
            ...
           a[0] = a[3]
           a[1] = a[2]
           pc = jbsr.(_advance)
           pc = cc != 0 -> L2 | pc
            ...
            ...
            ...
_advance:  L[a[7]] = link.(a[6],-40)
           a[7] = moveml.(3c3c)
           a[4] = a[1]
           a[5] = a[0]
           ...
```

For some procedures, the resulting transforms remove the need for an activation record. This opens the possibility to use a simpler, faster calling sequence on machines that have an expensive one (like the VAX). The final step is to produce an executable file by resolving all addresses and converting the RTLs to machine instructions.

4. IMPLEMENTATION

Both *vpcc* and *vpo* are implemented in C. Using various types of back ends, *vpcc* has been ported to six machines including an IBM PC. It is retargeted by supplying information about the sizes of the basic data types, and the order to evaluate arguments to functions.

At this time, *vpo* has been ported to two machines, the VAX-11 and the SUN-3. Retargeting *vpo* requires considerably more effort than *vpcc*. The instruction selection phase of *vpo* is implemented by a machine-independent peephole optimizer. This optimizer is retargeted by supplying a description of the target machine's instruction set. A machine description consists of a grammar and semantic actions. The grammar is used to produce a parser that checks the syntax of an RTL. The semantic actions check context-sensitive constraints imposed by a particular architecture. Currently, the RTL parsers are constructed using the Unix parser generator *yacc*. There is a certain appeal to the symmetry of using the tool that was used to construct the front end to help construct the back end.

Machine description grammars are relatively easy to write. The goal is to compose a grammar and semantic actions that produce a parser that accepts all legal RTLs (instructions) and rejects all illegal RTLs. Our experience is that it is easier to write a machine description for an instruction set than it is to write a grammar for a programming language. The task is further simplified by the similarity of RTLs across machines. This permits a grammar for one machine to be used as the model for a description of another machine. We have used this technique to describe the instruction sets of the following machines: VAX-11, Motorola 68020, National Semiconductor 32016, Ridge 32, Concurrent Computer Corporation 3230, Western Electric 32100, and the Prime 9950.

In addition to providing a description of the target machine's instruction set, information about the registers must be provided to the various optimization phases. For example, the number of allocable registers and the relationships of the registers must be provided to the optimizer. For efficiency reasons, this information is supplied by writing machine-specific subroutines. The register allocator is sufficiently general that it can handle machines with different register classes (e.g. Motorola 68020), machines that require multiplication and division to be performed in an even-odd register pair, and machines where the general register set is also used for floating point operations. Approximately 250 lines of the allocator is machine-dependent.

The bulk of *vpo* is machine- and language-independent. It consists of 5741 lines of C code. Overall, no more than 15 percent of *vpo* may require modification to handle a new machine or language. The degree of modification required depends on the peculiarities of the target machine's register set.

The prototype linker is also largely machine independent, but less so than *vpo*. It uses *vpo*'s instruction recognizer to perform instruction selection. Currently, the linker produces a large assembly language file with all external references resolved. A production version would use the machine description to translate RTLs directly to a binary object file. The linker must be able to identify procedure calls, the entry and exit point of procedures, the code that pushes arguments on the stack, as well as code that references arguments. Conceptually these are simple to identify, but each machine requires different patterns. Approximately 200 lines of code in the linker must be retargeted by hand.

The goal of *vpo* and the linker is to produce high-quality code. Table I compares execution times for several programs compiled using *vpcc/vpo* with the execution times of the same programs compiled using the standard C compilers found on a VAX-11/780 running 4.3BSD Unix and on a SUN-3/75 running SunOS 3.0. *grep* and *nroff* are well known Unix utilities. We used the source that is distributed with Berkeley Unix 4.3. *mincost* is a VLSI

wire-router that uses simulated annealing, while *cache* is a trace-driven cache simulator. *queens*, *ackerman*, and *puzzle* are well-known benchmark programs. Linker optimizations were not enabled for these measurements.

Program	VAX-11/780			SUN-3/75		
	vpo	cc	vpo/cc	vpo	cc	vpo/cc
grep	6.99	7.80	.90	4.46	4.63	.96
mincost	12.16	12.66	.96	5.13	5.49	.93
cache	83.98	90.20	.93	39.35	39.59	.99
queens	5.71	6.51	.87	2.62	3.60	.72
nroff	9.66	9.23	1.04	4.80	4.00	1.20
ackerman	5.06	6.34	.80	3.87	4.14	.93
puzzle	9.81	12.00	.81	3.15	3.80	.82

Table I. Comparison of Execution Times for Two Machines.

The measurements show that *vpcc/vpo* was very effective when compared to the standard C compiler on the VAX-11. On the SUN, *vpcc/vpo* produce code of essentially the same quality as the standard compiler. We believe the difference is not because of the ineffectiveness of *vpo* on the SUN, but rather that the SUN C compiler produces very good code as well.

We also measured the execution times of these same programs with the linker optimizations enabled. Table II contains these measurements. We see that performing optimizations at link time can be very effective. Again, the VAX-11/780 code was improved the most by the linker optimizations. We believe that is is due to the linker's ability to sometimes use a more efficient calling sequence on the VAX. The 68020's instructions for function call and return can not be simplified.

Currently, the compilation speed of *vpcc/vpo* is about that of the standard C compilers (measured without the assembly step). The linking phase, however, is three to six times slower than that of the Unix system linker *ld*. Most of the time is taken by assembling the large assembly file currently produced by the linker. Our measurements indicate that this time could be reduced by a factor of two if we produced an executable file directly.

Program	VAX-11/780			SUN-3/75		
	vpo (link)	vpo (no link)	link/no link	vpo (link)	vpo (no link)	link/no link
grep	6.59	6.99	.94	4.33	4.46	.97
mincost	11.80	12.16	.97	5.02	5.13	.97
cache	79.20	83.98	.94	38.38	39.35	.98
queens	5.50	5.70	.96	2.50	2.60	1.00
nroff	9.07	9.66	.94	4.79	4.80	1.00
ackerman	5.06	5.06	1.00	3.88	3.87	1.00
puzzle	9.20	9.81	.93	3.14	3.15	1.00

Table II. Comparison of Execution Times with Linker Optimizations Enabled.

5. RELATED WORK

Johnson and Miller [6] describe the design and effectiveness of a machine-level global optimizer. This optimizer is designed for use with several languages, but is targeted for only one machine, the HP Precision Architecture family of machines. They also use a single representation or data structure for all optimization processing. In the compiler they describe, most optimizations are done before register allocation (all except peephole optimization, branch optimization, and instruction scheduling). This ordering eliminates many of the situations that can cause phase ordering problems. If registers must be spilled by the during register allocation, presumably the peephole optimizer would collapse the load instruction with any instructions that follow if possible. If, however, the peephole optimizer is able to free any registers, the register allocator would not be able to put them to use. Because the target machine in this case is a RISC, such a situation probably occurred infrequently, if at all.

Wall [7] describes the effectiveness of performing global register allocation at link time. This linker decides which variables are used most frequently, assigns them to registers, and rewrites the code to reflect that the variables are in registers rather than memory. Our linker only attempts to optimize the call and return sequences based on information gathered in the call graph. With some effort we could extend the linker to rewrite RTLs to use different registers than were assigned during "local" register allocation. However, because our linker is intended to be portable and applicable to a wide range of architectures, global register allocation on the scale described by Wall appears more difficult. Wall's linker relied on memory being accessible through simple load or store instructions. For machines that allow memory to be accessed by instructions other than loads and stores, global register allocation would require instruction selection to be reapplied to all affected instructions. While our framework could accommodate this, the time to link a program would become unacceptably long.

6. SUMMARY

This paper has described a portable global optimizer and linker that uses the same intermediate representation. The advantage is that the synthesis phases of the compiler can be invoked in any order and repeatedly if necessary. This capability largely eliminates phase ordering problems that face compilers that use different representations for code generation and optimization. In addition, using the same intermediate representation for the object file format allows the linker to perform a number of optimizations that could not be done by the compiler because of its limited view. Our experiments show that even for complex machines with a "few" registers, link time optimizations can be particularly effective.

7. ACKNOWLEDGEMENTS

Elizabeth Dall perfected the machine description technique. David Whalley wrote the machine descriptions for the VAX and the Motorola 68020.

8. REFERENCES

1. Berry, R. E., Experience with the Pascal P-Compiler, *Software—Practice and Experience 8*, 5 (September 1978), 617-627.
2. Crowley, T. R., *Combining Table-Driven Effect Selection and Description-Driven Peephole Optimization for Automatic Code Generation*, M.S. Thesis, Massachusetts Institute of Technology, Boston, MA, September 1982.
3. Davidson, J. W. and Fraser, C. W., Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems 6*, 4 (October 1984), 7-32.
4. Davidson, J. W. and Fraser, C. W., Register Allocation and Exhaustive Peephole Optimization, *Software—Practice and Experience 14*, 9 (September 1984), 857-866.
5. Davidson, J. W. and Gresh, J. V., Cint: A RISC Interpreter for the C Programming Language, *Proceedings of the ACM SIGPLAN Notices '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, MN, June 1987, 189-198.
6. Johnson, M. S. and Miller, T. C., Effectiveness of a Machine-Level, Global Optimizer, *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, 99-108.
7. Wall, D. W., Global Register Allocation at Link Time, *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, 264-275.
8. Watts, J. S., *Construction of a Retargetable C Language Front-end*, Masters Thesis, University of Virginia, Charlottesville, VA, 1986.
9. Wulf, W., Johnsson, R. K., Weinstock, C. B., Hobbs, S. O. and Geschke, C. M., *The Design of an Optimizing Compiler*, American Elsevier, New York, NY, 1975.