A Global Object Code Optimizer

Manuel E. Benitez

Computer Science Research Memorandum RM-89-01
January 1, 1989

A GLOBAL OBJECT CODE OPTIMIZER

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science (Computer Science)

by

Manuel E. Benitez

January 1989

ABSTRACT

A global object code optimizer performs optimizations at the machine level. It uses information at the procedure level as well as knowledge about the specific capabilities of the target machine to produce high quality code. It takes many months, even years, of development effort to produce these optimizers. Performing optimizations at the machine level has traditionally been accomplished by machine-dependent algorithms. Retargeting to a new machine requires large portions of the optimizer to be re-written to handle the idiosyncrasies of the new architecture.

This thesis presents an optimizing compiler that performs global optimizations using a machine-independent notation to describe machine-dependent instructions at the machine level. The machine-independent notation that we call Register Transfer Lists is presented and its ability to describe machine-dependent instructions is discussed. Machine-independent algorithms that manipulate the Register Transfer Lists to perform common code-improvement transformations are explained.

Measurements of the effectiveness of the optimizing compiler on three different machines are given. The effectiveness of the some of the global optimizations is discussed. The effort required to retarget the optimizing compiler to new machines is estimated to be in the order of weeks.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to all those who contributed to this research. Jack W. Davidson, my advisor, was my primary source of guidance and encouragement. Robert P. Cook and James H. Aylor provided some very helpful suggestions. I would also like to thank Joseph Orost and S. Shastry for listening to some of my ideas and encouraging me to implement them. Finally, I would like to thank my wife, Eva, for her patience.

The research reported in this thesis was supported in part by Concurrent Computer Corporation and the Center for Innovative Technology under grant INF-87-003.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

INTRODUCTION

Traditional compilers are developed with one source language and one target machine in mind. This philosophy leads to $l \times m$ compilers. Early in the development of compilers, this fact was recognized [STEE61], and a solution was proposed: partition compilers into a front end capable of translating a source language into a more manageable intermediate language and a back end capable of translating the intermediate language to machine instructions. In this manner, only one front end is required per high-level language and only one back end is needed for each different machine. As simple as this strategy is, it is not until the last decade that it has seriously been applied. The reasons include:

(1) Designing an intermediate language capable of supporting a number of high-level languages is difficult.

(2) Assumptions made by the intermediate language may not result in an efficient translation to machine code on some computer architectures.

(3) The optimal placement of a code optimizer in the front end/back end approach is not obvious.

Despite these difficulties the ability to write a simple back end and immediately open up a huge library of software to a new computer architecture, or to write a front end for a new language and instantly have the ability to execute on any existing machine has it advantages.

This thesis describes an optimizing compiler that uses a front end/back end approach to portability. The front end transforms a *C* source file [KERN78] into a intermediate language file. The intermediate language makes few assumptions about the target machine. Each inter-

mediate language statement is expanded into a low-level representation that describes machine-specific instructions in a machine-independent notation. Machine independent optimizations are performed on the low-level representation. Machine-dependent peephole optimization is also employed.

1.1. Compilers and High-Level Language Evolution

From the earliest days of high-level languages, the quality of the code generated by a compiler has been important. Development of FORTRAN [BACK81] had as one of its primary goals the ability to generate code that was not significantly slower than code written by hand in assembly language. For all but academic settings, where programs are developed, executed a few times and then discarded, the quality of the code that a high-level language compiler generates determines the usefulness of the high-level language as a labor-saving tool. Although many of the problems that initially gave compilers a reputation for being difficult to develop have been overcome by compiler generating tools such as *lex* [LESK79] and *yacc* [JOHN78], writing a good optimizing compiler remains a difficult and time-consuming task.

Another important factor affecting the usefulness of a high-level language is the speed at which a compiler can translate it into machine code. Although this factor is controlled to some extent by the actual implementation of the compiler, it is also determined by the language itself. For example, languages that support separate compilation of modules are preferred for development systems over languages that force the recompilation of the entire program source even when only a single procedure has been changed. The amount of run-time checking required by a language may also affect compiler speed by requiring extra code to be generated to verify that the proper constraints are satisfied after operations are performed on objects. Although compiler users are generally willing to wait a little longer for a compiler to generate better quality code, the ability to trade code quality in exchange for increased execu-

tion speed is desirable.

1.2. Previous Work

Retargetable optimizing compilers and optimizers capable of working with many different source languages are not new developments. A brief survey of the recent work on these optimizers suggests a number of useful strategies towards the goal of developing a highly portable, high-quality, multiple-source language optimizer.

1.2.1. The PL.8 Compiler

The PL.8 compiler's optimizer [AUSL82] is designed to accept multiple source languages and produce high-quality code for a number of different machines. The PL.8 compiler works on a low-level intermediate language representation that is essentially assembly language for an abstract machine. All operations in the intermediate language are performed on registers and, in order to delay hardware register binding until late in the optimization process, an unlimited number of symbolic registers is assumed. In order to avoid phase ordering problems caused by the interdependency of some phases, the intermediate language representation is maintained by all phases of the optimizer, thus allowing the iteration of phases to resolve these interdependencies.

The PL.8 compiler is cleanly separated into many phases. Each phase performs one well-defined optimization on the intermediate language. The most important phases include: dead code elimination, common subexpression elimination, code motion, constant expression evaluation, strength reduction, dead store elimination, instruction scheduling and unnecessary range checking elimination. Register allocation is performed by a graph coloring algorithm with an additional mechanism to insert register spills as required to color the register interference graph.

Final assembly of the intermediate language to machine code is performed by a table-driven algorithm. The final assembly phase also deals with the condition codes, because the intermediate language has no suitable mechanism for handling them. A final peephole optimization pass is required to remove useless condition code recalculations.

The PL.8 compiler generates high-quality code for two's complement, byte-addressable 32 bit machines. Because of the nature of the intermediate language, portability is subject to some architectural restrictions. For machines having 32 or more general-purpose registers, 95 percent of the routines compiled generate no spill code [AUSL82]. This value goes down to less than 50 percent when only sixteen registers are available. These figures represent the drawbacks of performing optimizations that tend to increase the lifespan of temporary expressions before performing register allocation when there is a limited number of registers available to hold temporary expressions. The PL.8 compiler would produce unacceptable amounts of spill code on machines with fewer than sixteen registers.

1.2.2. UOPT

Chow [CHOW83] describes a machine-independent optimizer, called UOPT, that works at the intermediate language level. UOPT performs a large number of machine-independent transformations on U-Code [PERK79], which is more machine and source-language independent than the P-Code [PERK79] that it is based on. With UOPT, there is only one front-end per source language and one code generator per target machine. U-Code is low level to permit optimization of array references calculations and global register allocation, yet high level enough to allow machine-independent optimizations to be made with only the most basic information about the size of the different types of memory references required. U-Code is stack based and uses Pascal-like semantics that can easily be extended to support most algebraic languages (e.g., FORTRAN, Algol, *C*, PL/1).

UOPT performs the following optimizations: stack height reduction in expression evaluation, constant propagation, constant expression evaluation, address collapsing in array expressions, dead code elimination, copy propagation, common subexpression elimination, loop-invariant expression optimization, code motion, strength reduction, induction variable elimination, redundant store elimination, dead variable elimination, evaluation order determination, and global register allocation. All of these optimizations, with the notable exception of global register allocation, are machine independent. Global register allocation is included in UOPT because of its high payoff in terms of code improvement and because the data-flow information required is available to UOPT. The global register allocator is given information regarding the number and types of registers available on the target machine, as well as the relative access times for the different registers and memory types in the memory hierarchy of the target machine.

Final translation of U-Code to machine instructions is performed by a machine specific code generator. Since the nuances of the target machine are not known to UOPT, the code generator is expected to perform some machine-dependent optimizations. A drawback to this strategy is that a peephole optimizer must be included in the code generator. Any changes in the code made by the code generator cannot fed back to UOPT for further optimization.

UOPT is intended to be an automated optimizer tool. Once a front end has been constructed that generates U-Code, UOPT will provide the optimizations required to produce high quality code. Code generators have been developed for the DECsystem-10 [DIGI76], Motorola 68000, VAX-11 [DIGI81], MIPS, FOM and S-1 [CHOW83] machines. The amount of time and effort required to implement a code generator and its associated peephole optimizer is not mentioned, but the benefits of performing machine-independent optimizations were evident on all of these machines.

1.2.3. HP Precision Architecture Optimizer

Johnson and Miller [JOHN86] determined the benefits of performing global optimizations at the machine level. The optimizer works with a number of high-level languages, including *C* [KERN78], Pascal [JENS75], FORTRAN and COBOL. The optimizer is invoked after code generation and is source-language independent. Portability is not an issue and the target machine is the Hewlett-Packard Precision Architecture, which is a 32 register RISC machine with a simple instruction pipeline.

The optimizations performed by the optimizer include: dead code elimination, common subexpression elimination, unnecessary memory reference elimination, code motion, induction variable elimination, register allocation, peephole optimization, branch optimization and instruction scheduling. Phases were ordered so that global data-flow information could be updated rather than recalculated as each phase was invoked. In order to reduce the amount of work required to generate the data-flow information, early peephole optimization was performed to reduce the number of instructions.

Although modeled after the PL.8 compiler, the optimizer is conservative enough to avoid excessive generation of spill code. Two register allocators are available, one is a fast but simple allocator that performs no optimizations other than eliminating redundant register-to-register transfers. The second is a graph coloring allocator capable of computing the pay-offs involved in spilling registers in order to allocate them to frequently referenced local variables. The optimizer generates significantly fewer register spills than the PL.8 compiler, but its inability to be ported to machines with less than 32 registers prevents closer inspection of the overall effectiveness the optimizer's register allocation strategy.

1.2.4. Retargetable Code Generators

Glanville and Graham [GRAH82] have made significant a breakthrough in the area of code generation with an approach that uses a context-free grammar describing the instruction set of the target machine to generate a parsing table to drive a machine-independent code generator. Using Graham-Glanville techniques, code generator generators have been developed [AIGR84]. The instruction set descriptions contain the syntactic and semantic information required to transform a low-level intermediate representation based on Polish prefix expressions into assembly code for the target machine.

Graham-Glanville code generators are very promising despite their limitations: the context-free grammar required to describe the VAX-11 [DIGI81] has over 1000 productions [GANA82] and sometimes fails to make correct decisions when choosing between two-address and three-address variants of instructions. Ganapathi, Fischer and Hennessy consider table-driven code generation methods such as the Graham-Glanville method to be the most flexible and least ad-hoc technology for creating retargetable code generators.

Ganapathi and Fischer added extended semantic attributes, disambiguating predicates and a semantic evaluation framework to the basic Graham-Glanville code generation algorithm and described the target machines using attribute [KNUT68] instead of context-free grammars. These enhancements permit the machine description to fit better into the framework of an optimizing compiler and more completely address the problems of real machine architectures. The Ganapathi-Fischer approach allows multiple instruction results, such as condition code results to be tracked and provides a method of describing complex transfer productions that entail data-type conversion and register-to-memory transfers.

1.2.5. PO

Davidson [DAVI81] describes a retargetable peephole optimizer, PO, that uses a machine description and a machine-independent low-level intermediate representation, called register transfer lists (*RTLs*) to accomplish instruction selection in the process of performing peephole optimization. Common subexpression elimination, branch chain elimination and register assignment are added to the peephole optimizer to generate code comparable to existing production compilers. PO's machine descriptions are transformed by *lex* [LESK79] into subroutines that implement the recognizer and transducer. Addressing modes are factored out of the instruction definition allowing even complex machines such as the VAX-11 [DIGI81] to be described in a few hundred lines of grammar. PO's machine description allows multiple instruction results to be tracked, so that condition code results are explicitly described by *RTLs* instead of being implicitly handled by the semantic actions as is the case in the Ganapathi-Fischer approach.

1.3. Retargetable Optimizing Compilers

Despite the recent advances in retargetable and multiple source language optimizers, there is still room for improvement. The previous work suggests possible strategies towards an improved optimizer:

(1) Use a very simple front end that performs no optimizations. It generates a simple, naive intermediate language code and concentrates primarily on ensuring a correct translation from the source language to the intermediate language.

(2) Use a simple code generator to translate each intermediate language operation into a sequence of target machine instructions. The code generator should not perform any optimizations. It must be simple and easy to modify, since it must be written or modified to suit each target machine.

(3) Do all optimizations at the machine level, where all calculations are exposed and all possible addressing modes can be utilized.

(4) Effectively use the registers available on the target machine and avoid register spills.

(5) Maintain a consistent representation of the code so that phase ordering problems can be eliminated through iteration.

(6) Perform optimizations that yield good code quality improvements for the amount of time invested in making them.

In the next chapter, we present a machine-independent representation of machine instructions that permits optimizations at the machine level and explain the machine description and semantic actions required to build a recognizer for a machine. Chapter three will discuss the general layout of a retargetable optimizing compiler. Chapter four presents the results obtained by comparing the optimizing compiler to existing compilers on the VAX-11/780 [DIGI81], the Concurrent 3230 [PERK82] and the Motorola 68020 [PREN85]. The final section concludes with general observations on the effectiveness of our code generation strategies and the areas that require further research.

CHAPTER 2

REGISTER TRANSFER LISTS

Register transfer lists (*RTLs*) are machine-independent representations of machine-dependent operations. *RTLs* have their origins in the *ISP* notation developed by Bell and Newell [BELL71]. The initial notation was altered slightly to facilitate its use as a low-level intermediate language. The *RTLs* described here are identical to those described by Davidson for PO [DAVI81] which was the logical starting point for the work presented in this thesis. The advantages of using *RTLs* as a powerful, low-level intermediate representation were evident from Davidson's work with PO; what PO lacked was the global optimizations necessary to show that the advantages of the *RTL* notation could be exploited by an optimizing compiler.

For pedagogical reasons, *RTLs* will be presented here in a readable notation. In actual implementations, a more compact *RTL* notation is used to reduce the amount of memory required to store *RTLs* and reduce the amount of time required to parse *RTLs*. The examples used in this chapter are not intended to pertain to any one particular target machine. The purpose is to describe the *RTL* notation in general and the reader should be aware that the *RTL* notation can be augmented to suit the requirements of the machine instruction set of any particular target machine.

An *RTL* describes the effect of a machine instruction. Most machine operations can be defined in terms of the data movement that they effect. For this reason, the *RTL* notation is particularly suited to describing data transfers between registers and memory locations.

2.1. Register References

The most basic instruction available on most machines is a register-to-register transfer. Such an operation is represented in *RTL* notation in the following way:

```
r[3] = r[5];
```

In this instruction, the contents of general-purpose register five are moved to general-purpose register three. On the VAX-11, this instruction may be represented by the assembly language statement:

```
movl   r5,r3
```

On the Motorola 68020, it would be:

```
movl   d5,d3
```

And on the Concurrent 3230:

```
lr 3,5
```

The assembly language instruction or machine instruction notation actually required to perform the register-to-register transfer is not important, what is important is that general-purpose register three contains the same value that is stored in general-purpose register five after the operation is performed.

Since most machines have more than one type of register, the *RTL* notation permits register type discrimination. For example, the Motorola 68020 has address registers and data registers, and the *RTL* for describing a data register to address register transfer is:

```
a[3] = d[5];
```

Many machines allow a register to contain data objects that are smaller than the register. Both the VAX-11 and the Motorola 68020 have 32-bit general-purpose registers and addressing modes that refer to the least-significant 16 bits (word) and least-significant 8 bits (byte) of

a register. In *RTL* notation, size is described by the same mechanism that is used to discriminate register types. Moving the least-significant 8 bits of general-purpose register five to the least-significant 8 bits of general-purpose register three is represented in *RTL* notation in the following way:

```
b[3] = b[5];
```

And moving 10000 into the least-significant 16 bits of general-purpose register four:

```
w[4] = 10000;
```

2.2. Operations

RTLs permit logical and arithmetic operations. These operations generally use operators as defined by the *C* programming language:

Operation	*RTL* notation
Addition	`r[5] = r[3] + r[2];`
Subtraction	`r[5] = r[3] - r[2];`
Multiplication	`r[5] = r[3] * r[2];`
Division	`r[5] = r[3] / r[2];`
Modulus	`r[5] = r[3] % r[2];`
Unary minus	`r[5] = -r[3];`
2's complement	`r[5] = ~r[3];`
Bitwise OR	`r[5] = r[3] | r[2];`
Bitwise AND	`r[5] = r[3] & r[2];`
Bitwise exclusive OR	`r[5] = r[3] ^ r[2];`
Left shift	`r[5] = r[3] << r[2];`
Right shift	`r[5] = r[3] >> r[2];`

Complex combinations of these operations are permitted. Parentheses can be added to indicate the required operator precedence.

2.3. Memory References

A memory reference can be denoted in *RTL* notation in the following way:

```
L[_base] = r[2];
```

In this instruction, the contents of general-purpose register two are moved to the longword (32 bits on most machines) memory location assigned to `_base`. The size of the memory reference or the expected format of the data in a memory location can be specified by the memory reference type. For example:

Reference size	*RTL* notation
Byte (8-bits)	`B[_base]`
Word (16-bits)	`W[_base]`
Longword (32-bits)	`L[_base]`
Floating-point	`F[_base]`
Double-precision	`D[_base]`

The addressing modes available from machine to machine vary, but *RTLs* provide a consistent and highly descriptive method of describing the semantics of an addressing mode. Here are a few examples:

Addressing Mode	*RTL* notation
Direct	`L[_base]`
Indirect	`L[L[_base]]`
Register Deferred	`L[r[5]]`
Displacement	`L[r[5] + offset.]`
Displacement Deferred	`L[L[r[5]] + offset.]`
Index	`L[r[4] << 2 + r[5]]`
Index Displacement	`L[r[4] << 2 + r[5] + offset.]`

RTLs provide a notation that is not only capable of describing every addressing mode possible, it also does so in a well-defined, intuitive way.

On machines with special *autoincrement* and *autodecrement* modes, `++` and `--` operators are used:

Addressing Mode	*RTL* notation
Autoincrement	`L[r[5]++]`
Autodecrement	`L[--r[5]]`

This syntax is borrowed from *C*. In the case of the autoincrement mode, the position of the ++ after the register indicates that the value in the register is referenced first and then the register is incremented–this is more generally known as a *postincrement*. In the autodecrement mode, the −− before the register indicates that the register is decremented first and then referenced–more generally known as a *predecrement*. The complementary *preincrement* and *postdecrement* addressing modes would be indicated, if the target machine supported them, as:

Addressing Mode	*RTL* notation
Preincrement	`L[++r[5]]`
Postdecrement	`L[r[5]--]`

2.4. Compare Instructions

Most machines have comparison operators that set proper bits in a condition code register according to the outcome of a comparison. In order to describe these, a special condition code register is defined. For example, comparing two general-purpose registers can be described in *RTL* notation as:

```
CC = r[3] ? r[5];
```

In this instruction, the contents of general-purpose register three is compared against the value of general-purpose register five. The value of the condition code register, *CC* is updated accordingly. Neither of the general-purpose registers are changed.

2.5. Branch Instructions

In order to describe a branch instruction, a special program counter (PC) register is utilized. An unconditional branch in *RTL* notation is:

```
PC = LABEL;
```

This *RTL* describes an unconditional branch to the program location denoted by LABEL.

For conditional branches, the condition code register is examined and, depending on its contents, a branch is executed. In *RTL* notation, a conditional branch is represented as:

```
PC = CC >= 0 -> LABEL | PC;
```

2.6. Type Conversions

Many machines have instructions that convert from one format to another. A common conversion is from floating-point format to integer format and vice-versa. In the *RTL* notation, proper register typing makes describing type conversion instructions simple. For example, if an integer value stored in a general-purpose register needs to be converted into a floating-point value and stored in a floating-point registers, the appropriate *RTL* representation of the instruction is:

```
f[2] = r[8];
```

This *RTL* indicates that the integer value in general-purpose register eight should be converted to the equivalent value in floating-point representation and moved to floating-point register two. On machines that do not have separate floating-point registers and use the general-purpose registers to hold floating-point values, the register typing should still be used in the *RTL*s. The optimizer is capable of mapping any number of register types to a single set of registers on the target machine.

2.7. Complex Instructions

Some machines have a number of complex, but useful instructions whose semantics are difficult to capture using *RTL* notation. The general rule for handling these cases is to use a macro-like syntax that shows which registers values are needed by the instruction. An example of this is the *link* instruction on the Motorola 68020 that sets up an activation record for a procedure. This instruction places the value of the frame pointer on the stack and allocates enough space on the stack to store local variables. It also remembers how much stack space was allocated so that the complementary *unlink* instruction can remove the activation record at the end of the procedure. The *RTL* form of the *link* instruction is:

```
L[a[7]] = LINK(a[6], 40);
```

where address register seven is the stack pointer, address register six is the frame pointer and 40 is the number of bytes to allocate for local variables. The macro notation requires additional knowledge about the target machine to select the proper machine instruction. This information is present in the form of semantic actions.

Another example of a complex instruction is the *call* instruction available on many machines to transfer control to a subroutine. A *call* instruction in *RTL* notation is:

```
STK = CALL(foo);
```

This *RTL* denotes an instruction that transfers control to the subroutine `foo`.

2.8. Multiple Effects

Many machine instructions perform more than one register transfer operation. One common example is setting the condition code register in a register-to-register transfer. A naive approach to describing this operation would be to leave out the effect of setting the condition codes in favor of the register to register transfer. The disadvantage of the naive approach is

that it does not give a machine-independent optimizer enough information to recognize sequences of *RTLs* that can be combined to form a single target machine instruction.

The following sequence of *RTLs* can be performed with a single target machine instruction:

```
r[3] = r[4];
CC = r[3] ? 0;
```

In order to allow the representation of multiple effects, the *RTL* notation permits *RTLs* to be concatenated. The semicolon is used to separate the effects. The full effect of a register-to-register transfer instruction can be represented with the following *RTL*:

```
r[3] = r[4]; CC = r[4] ? 0;
```

RTL semantic rules state that all uses are evaluated before any updates. This eliminates any ambiguity on *RTLs* that use and update the same register or memory location.

2.9. Advantages

This brief introduction to the *RTL* notation has demonstrated the expressive power of *RTLs* to denote complex machine instructions and addressing modes. Additional reasons why the *RTL* notation is a powerful instruction representation form include:

(1) Since the *RTL* form is machine independent, the algorithms that manipulate *RTLs* are also machine independent.

(2) Because *RTLs* represent machine specific instructions, they allow optimizations to be performed at the machine level.

(3) Because *RTLs* are well-defined, it is possible to construct recognizers that can determine whether an *RTL* represents a legal instruction on a target machine.

(4) *RTLs* are powerful enough to represent machine instructions in all optimization phases.

CHAPTER 3

IMPLEMENTATION

The optimizing compiler consists of three main sections: The front end, the code expander and the optimizer (see Figure I). The code expander and the optimizer comprise the back end of the compiler.

3.1. The Front End

The front end translates a program written in a high-level language into a corresponding set of intermediate language statements. Although the front end is capable of performing optimizations, our strategy is to just perform a naive translation of the source code. There are several reasons why we believe this to be a good strategy:

(1) A naive front end is easier to write and debug than an optimizing front end.

(2) Few optimizations, with the possible exception of procedure inlining, are as effective when applied to source or intermediate-language code as they are at the machine level. The reason for this is that the capabilities of the target machine (the addressing modes, number of general purpose registers, number of addresses per instruction, etc.), have a

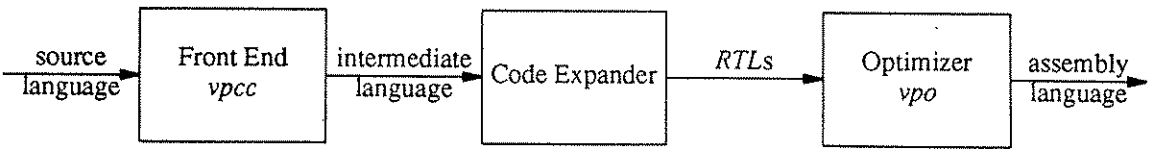

Figure I.

significant impact on the optimizations that can be or should be performed. The source or intermediate-language code are too far removed from the target machine to ensure that all of the optimizations can be performed are performed.

(3) Any optimizations performed by the front end would have to be duplicated for every high-level language front end that expects to produce the best possible code.

The front end used in this thesis is *vpcc* (Very Portable *C* Compiler) [WATT86]. *Vpcc* is a complete *C* compiler [KERN78] that includes bit fields, floating-point and enumeration types. Source code is translated into a stack-oriented intermediate language consisting of 46 simple operations. Consider the following *C* statement:

```
r = x * x + y << 3;
```

This statement will be translated by *vpcc* into the following sequence of intermediate-language operations:

Operation	Type	Argument	Description
ADDR	int	x	push address of local variable x
@	int		load an integer using address on top of stack
ADDR	int	x	push address of local variable x
@	int		load an integer using address on top of stack
*	int		multiply two integers on top of stack, push result
ADDR	int	y	push address of local variable y
@	int		load an integer using address on top of stack
+	int		add two integers on top of stack, push result
CON	int	3	push the integer constant 3
<<	int		left shift using two integers on stack, push result
ADDR	int	r	push address of local variable r
=	int		store into address on top of stack the integer below it

Note the simplistic approach that is taken to describe memory references. For example, a local variable reference is described as two separate operations: the address calculation and the dereferencing of the address. In general, the intermediate-language operations are simple, thereby ensuring a straightforward mapping into *RTL*s on the target machine, which is the task of the code expander.

Other front ends exist that are capable of working within the framework of our optimizing compiler, and they are all essentially similar in their approach to code generation.

3.2. The Code Expander

The code expander accepts a sequence of intermediate-language statements and translates them into a corresponding sequence of *RTLs*. The code expander is machine dependent because the *RTLs* that it emits must denote valid machine instructions on the target machine. The strategy of having the front end emit simple intermediate-language operations ensures a simple mapping to *RTLs* that represent valid target-machine instructions.

The code expander has knowledge of the types of register and memory references available on the target machine. In order to postpone register allocation, the code expander uses pseudo registers. This allows the code expander to assume that a large number of registers are available on the target machine. Pseudo registers have the same form as hardware registers in *RTL* notation. Any register with a number greater than the actual number of registers available on the target machine is a pseudo register. For example, on the Motorola 68020, `d[0]` through `d[7]` represent the data registers available and `d[8]` through `d[255]` are pseudo registers that will be bound later in the compilation phase to a data register. The number of pseudo registers is limited for the sake efficiency. Compiling a suite of non-trivial programs shows that 200 pseudo registers are sufficient. The method used to allocate pseudo registers is simple:

(1) A counter is set at the beginning of each source language statement to the first available pseudo register number.

(2) The value of the counter is used whenever a new pseudo register is needed, then the value of the counter is incremented to reflect the next available pseudo register.

Multiple effects increase the descriptive power of *RTLs*, but raise a serious issue: should the code expander be forced to describe the full effect of each instruction, or should it emit only the effect or effects that it immediately requires? Having to emit the full effect of an *RTL* increases the amount of storage required to keep *RTLs*. Not describing the full effect of each *RTL* undermines a machine-independent optimizer's ability to determine which values are stored in every register. A solution to both of these problems is to provide a simple method of indicating that the value of a register is altered by an instruction without having to describe the details of the change. This is done by placing a register whose contents are potentially destroyed on the *dead register list* of the *RTL*.

A hardware register that is altered by an instruction whose new value is not considered important is placed on the dead register list. In the case of a register-to-register transfer that also sets the value of the condition code register, the correct *RTL* representation would be:

```
r[3] = r[4];     (CC)
```

This is the standard notation for denoting that the condition code register is on the dead register list. The contents of the dead register list may be the determining factor in deciding whether an *RTL* represents a valid instruction on the target machine. In the case shown above, removing the condition code register could result in an illegal instruction if the target machine has no way of performing a register-to-register transfer without affecting the condition codes.

The dead register list performs another important task: if a pseudo register is placed on the dead register list, it does not mean that its value is altered by the instruction, but rather that the pseudo register will never be referenced again. This mechanism permits the hardware register allocation phase of the optimizer to free the hardware register bound to a pseudo register, thus allowing it to be bound to a new pseudo register.

The code expander also determines the calling conventions for procedure and function calls. Issues such as whether arguments are to be passed in registers or on the machine stack affect the code emitted by the code expander. The mechanism used to return the results of a function call is also determined by the code expander. Since pseudo registers do not overlap hardware registers, the code expander can reference hardware registers and thus generate code that returns function values in specific hardware registers.

Like the front end, the code expander does not attempt any optimizations and the resulting *RTL*s will usually present many opportunities for optimization. This naive approach to code expansion makes the code expander easy to write and modify.

Returning to the example given for the front end, the following is an example of the *RTL*s that the code expander for the Motorola 68020 would generate for the sequence of intermediate-language operations:

Operation	Type	Argument	*RTL*	
ADDR	int	x	`a[8] = a[6] + x.;`	`(CC)`
@	int		`d[9] = L[a[8]];`	`(CC,a[8])`
ADDR	int	x	`a[10] = a[6] + x.;`	`(CC)`
@	int		`d[11] = L[a[10]];`	`(CC,a[10])`
*	int		`d[9] = d[9] * d[11];`	`(CC,d[11])`
ADDR	int	y	`a[12] = a[6] + y.;`	`(CC)`
@	int		`d[13] = L[a[12]];`	`(CC,a[12])`
+	int		`d[9] = d[9] + d[13];`	`(CC,d[13])`
CON	int	3	`d[14] = 3;`	`(CC)`
<<	int		`d[9] = d[9] << d[14];`	`(CC,d[14])`
NAME	int	r	`a[15] = a[6] + r.;`	`(CC)`
=	int		`L[a[15]] = d[9];`	`(CC,a[15],d[9])`

Note how often the condition code register (CC) appears on the dead register list. The reason for this is that many instructions on the Motorola 68020 set the condition code register depending on the value calculated or transferred by the instruction. Even this simple example shows how much space is saved by not describing the full effect of every instruction. This example also shows pseudo registers on the dead register list of the last *RTL* in which they are

referenced. The code expander can determine when a pseudo register should be placed on the dead register list because it keeps every live pseudo register on the intermediate stack. When a pseudo register is taken off the stack, referenced, and not pushed back on the stack, the code expander places that pseudo register on the dead register list. This strategy is correct because, if the pseudo register is not on the intermediate stack, it cannot be referenced in any future operation.

Code expanders are easy to implement because a significant portion of the effort required to re-target the optimizing compiler to a new target machine entails writing or modifying a code expander. All of the code expanders that have been implemented consist of a loop that reads each operation in the intermediate file and a `switch` statement that transfers control to the statements or the procedure that emit the appropriate *RTL*(s). For example, the following code handles the + operation on the Motorola 68020's code expander:

```
type = getw(fp);                /* Operation type */
arg1 = *--sp;                   /* Take registers from stack */
arg2 = *--sp;

printf("+%c[%d] = %c[%d] + %c[%d];  (CC,%c[%d])\n",
         REGCHR(type), arg2, REGCHR(type), arg2,
         REGCHR(type), arg1, REGCHR(type), arg1);

*sp++ = arg2;                   /* Push register with result */
```

This example shows how the intermediate stack is managed. The stack contains only the number of the pseudo register that contains the value needed in an operation. The intermediate language provides the type, so there is no need to keep it on the stack along with the pseudo register number. The function `REGCHR` maps operation types to the corresponding register types used in the *RTL* notation for the target machine. Since the code explicitly takes two pseudo registers from the stack and only returns one back to the stack, the pseudo register that is not returned to the stack is placed on the dead register list. Since the addition operation

on the Motorola 68020 affects the condition codes, the condition code register is placed on the dead register list as well. Also note that the first character emitted is a ''+''. This indicates that the line contains an *RTL*. There are instances where the code expander must pass information other than an *RTL* to the optimizer, and the first character of a line indicates the nature of the information on the remainder of the line. For example, the code expander can pass text directly to the assembler (to allocate space for global variables, for instance) by placing a ''–'' at the beginning of the line.

3.3. The Optimizer

The optimizer is called *vpo* (Very Portable Optimizer) and is composed of optimizing phases that operate on *RTL*s. The optimizations currently implemented are: useless branch elimination, dead code elimination, instruction selection, global register allocation, common subexpression elimination, constant propagation and dead store elimination.

Breaking the difficult task of code optimization into a series of phases or passes in order to reduce complexity and simplify the implementation is standard practice. The order in which the different phases are invoked is a critical part of the optimizer's design. Designing the phases so that they are capable of operating in isolation simplifies the implementation of each phase; however, some interaction must take place between the phases in order to produce good code. Designing and ordering phases to interact without interfering with each other is called a *phase ordering problem* [BENI88]. Most optimizers suffer phase ordering problems because the ordering of the phases is *statically* determined in either a carefully planned or ad-hoc fashion. Interaction between the separate optimizing phases generally results in a specific ordering that cannot be changed easily after the optimizer has been designed. Carefully planned phase ordering attempts to ensure that the work performed at each phase does not create new situations that a previously executed phase is responsible for optimizing. Even the

most careful phase ordering cannot prevent these situations from occurring in general. *Vpo* uses an iterative approach to phase ordering. By maintaining the *RTL* notation through all optimization phases, *vpo* is capable of re-invoking previously executed phases when new possibilities have been introduced for further optimizations.

3.3.1. Basic Blocks

Vpo operates on a program a procedure at a time. As *vpo* reads in the *RTLs* that make up a procedure, it partitions them into basic blocks. A basic block is a sequence of *RTLs* having only one entry and exit point. *Vpo* recognizes *RTLs* that might have more than one predecessor because the code expander emits a label before every such *RTL* . Exit points are found by looking for branches: both conditional and unconditional. Since a branch ends a basic block, any *RTL* following a branch begins a new basic block.

The data structure used by *vpo* to represent basic blocks defines not only the boundaries of the basic blocks, but also the possible flow of control between the basic blocks. Every basic block has a left and right child pointer. This left and right pointer approach models the traditional method of transferring control of the program: the conditional and unconditional branch. The left child is considered to be the ''normal'' successor, while the right child is reached only when the basic block ends with a conditional branch. Only blocks that end with an unconditional ***return*** from the procedure have no children. Every basic block also keeps a list of the predecessor to the block, and every basic block except for the entry block to a procedure has one or more predecessors. For example, the following sequence of *RTLs*:

```
        CC = r[20] ? r[22];
        PC = CC != 0 -> L5 | PC;
        r[23] = 0;
        PC = L6;
L5:     r[23] = 1;
L6:     ...
```

Would be stored in a data structure roughly described in the following figure:

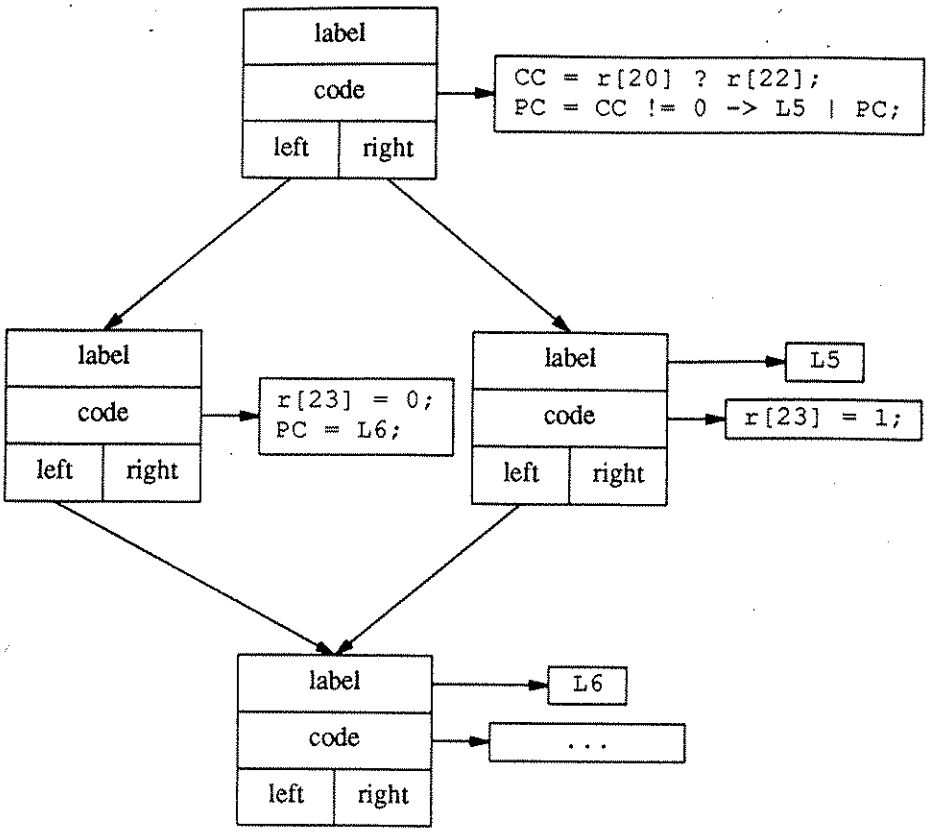

Figure II.

Basic blocks capture the essential program flow within a procedure. *Vpo* needs to know the flow of control for the following reasons:

(1) Knowing the flow of control allows *vpo* to eliminate dead code–that is, code that cannot be reached from the entry point of a procedure.

(2) Calculating data-flow information requires knowing the flow of control in a procedure.

(3) Locating loops in a procedure in order to determine how frequently local variables are likely to be referenced requires knowing the flow of control.

3.3.2. Useless Branch Elimination

Useless branch elimination removes branches by rearranging basic blocks. It also removes branch chains–instances where a branch branches to a branch instruction, by changing the target label of the first branch to reflect the final destination of the chain.

Useless branch elimination locates instances of a basic block having no predecessor that *falls through* to it. In such a case, the basic block is placed directly after one of its predecessors and the unconditional jump to the basic block in the predecessor is removed so that the predecessor now *falls through* to the basic block.

Another type of useless branch elimination is performed by finding instances of a conditional branch followed by an unconditional branch where the target of the conditional branch follows the unconditional branch:

```
        PC = CC > 0 -> L33 | PC;
        PC = L56;
L33:    ...
```

When such an instance is located, *vpo* removes the unconditional branch and negates the condition of the conditional branch. The sequence of *RTL*s shown above become the following equivalent sequence:

```
        PC = CC <= 0 -> L56 | PC;
L33:    ...
```

Branch chain elimination is accomplished by locating instances of branches to blocks that immediately branch to other targets:

```
        PC = L45;
        ...
L45:    PC = L82;
        ...
L82:    ...
```

Vpo optimizes branch chains by replacing the target label of the first branch with the final

destination of the chain. In the case shown above, the optimized sequence would be:

```
        PC = L82;
        ...
        PC = L82;
        ...
L82:    ...
```

Note that the label `L45` is no longer used, since all branches to `L45` have been replaced with branches to `L82`. The unconditional jump will be removed unless the predecessor of the basic block containing the jump *falls through* to the basic block, thus relying on the unconditional branch to reach `L82`.

3.3.3. Dead Code Elimination

Dead code elimination consists of determining which basic blocks cannot be reached from the entry point in a procedure and eliminating such blocks. *Vpo* determines which blocks are unreachable by first marking every basic block in the procedure unreachable and calling the following simple procedure with the entry block of the procedure:

```
Mark_Reachable (Block)
{
     while (Block != NULL && Block->Reached != TRUE) {
          Block->Reached = TRUE;
          if (Block->Right != NULL) {
               Mark_Reachable (Block->Left);
               Block = Block->Right;
               }
          else
               Block = Block->Left;
          }
}
```

After this algorithm is applied, *vpo* examines each basic block and disposes of any block for which `Block->Reached` is not equal to `TRUE`.

3.3.4. Window Definition

RTLs provide the ability to combine the effects of two instructions and quickly determine if the resulting *RTL* defines a valid machine instruction on the target machine. Consider the following *RTLs*:

```
r[17] = 5;
r[16] = r[16] + r[17];    (r[17])
```

Combining the effects of these two *RTLs* can be done with a simple machine-independent algorithm to yield:

```
r[17] = 5;r[16] = r[16] + 5;   (r[17])
```

Note, however, that the pseudo register `r[17]` has been placed on the dead register list (the combining algorithm concatenates the dead lists of the original *RTLs*) to indicate that `r[17]` will not be used again, thereby removing the need to store the constant 5 into `r[17]`. The actual effect of combining the two original *RTLs* is:

```
r[16] = r[16] + 5;
```

This new *RTL* will be passed to the machine description for the target machine and, if accepted, will replace the two original *RTLs* that combined to create it.

Combining *RTLs* is peephole optimization. Peephole optimization is commonly performed by using a small window into the code and replacing pairs or triples with single instructions according to a predefined template of patterns. Peephole optimizers have traditionally considered lexically adjacent instructions as candidates for optimization. *RTL* combining also uses the concept of a window, but instead of lexical adjacency determining the window, global data-flow analysis information is used to define windows based on the where a register is set and where it is first used. Consider the following example:

```
1     r[17] = 5;
2     r[16] = L[_base];
3     r[16] = r[16] + r[17];    (r[17])
```

An *RTL* combiner using lexical adjacency would attempt to combine *RTLs* 1 and 2 to obtain:

```
r[17] = 5;r[16] = L[_base];
```

which is an unlikely instruction on most target machines. It would then try 2 and 3 to obtain:

```
r[16] = L[_base] + r[17];      (r[17])
```

which is also an unlikely instruction, especially on RISC architectures. If the window size permits, it may consider all three instructions:

```
r[16] = L[_base] + 5;
```

and still fail to find a valid instruction on the target machine. Had it considered combining 1 and 3, it might have found a valid target instruction and reduced the three *RTLs* to the pair:

```
2     r[16] = L[_base];
3     r[16] = r[16] + 5;
```

Vpo addresses this problem by defining a window between the *RTL* that sets a register and the *RTL* that first uses the register. In the example above, *vpo* would define the following windows:

```
1          r[17] = 5;
2          r[16] = L[_base];
3 {1, 2}   r[16] = r[16] + r[17];    (r[17])
```

The numbers inside the braces indicate the windows. *RTL* three and *RTL* one are a window because `r[17]` is set by the first *RTL* and first used by the third *RTL*. *RTL* three and *RTL* two define a window because `r[16]` is set in *RTL* two and first used in *RTL* three.

Using data flow information to define combination windows increases the likelyhood that attempted combinations will result in valid target machine instructions, allows instruc-

tions separated by many other instructions to combine and because the algorithm used to set up windows is conservative, less work is required to consider just the windows defined than would be required to consider all lexically adjacent instructions. Davidson's experiments with PO and windows showed a 20% reduction in code size over the traditional lexical adjacency window by an optimizer employing the set and use windows [DAVI81].

3.3.5. Instruction Selection

The instruction selection phase of the optimizer combines *RTLs* using the set and use windows. In addition to combining *RTLs*, instruction selection also invokes an *RTL* simplifier that performs some of the following simplifications:

(1) Perform arithmetic operations where constants are available. For example:

```
r[16] = r[16] * (5 + 6);
```

would simplify to:

```
r[16] = r[16] * 11;
```

(2) Remove useless operations, such as adding by zero, subtracting by zero, multiplying by one, dividing by one and bitwise ORing or exclusive ORing by zero.

(3) Remove double unary minuses or one's complement instructions. For example:

```
r[18] = ~~r[15];
```

would become:

```
r[18] = r[15];
```

(4) Remove instructions whose outcome can be determined, like multiplying by zero, subtracting two identical values and bitwise ANDing by zero.

(5) On machines with autoincrement or autodecrement instructions, the simplify algorithm is responsible for noting combinations that can be simplified with an appropriate indexing mode. The following *RTL* combination:

```
r[21] = L[r[19]];r[19] = r[19] + 4;
```

would be changed to:

```
r[21] = L[r[19]++];
```

Combinations using the windows are attempted in pairs and in triples. Although there is no reason why four or more *RTLs* cannot be combined and tested for validity, efficiency considerations and the low probability of finding valid combinations of four or five *RTLs* that cannot be obtained by first collapsing a pair or a triple are responsible for the three *RTL* limit.

In order to determine whether a combination of two or three *RTL* s denote a valid target machine instruction, the machine description is consulted. The machine description consists of *yacc* [JOHN78] generated parser and an associated group of semantic actions that determine the validity of an instruction. In cases where the *RTL* describes a valid instruction, the machine description also returns the cost, in microseconds or some other arbitrary measure, of executing the instruction. The instruction cost is used by the instruction selection phase to prevent the combination of a pair or triple whose resultant instruction actually takes longer to execute than it would to execute the component *RTLs* separately. These cases are rare, but they do exist because of poor machine design or because combining instructions introduces a new pipeline delay.

Phase iteration is used by *vpo* to prevent phase ordering anomalies. Instruction selection is a commonly re-invoked phase because even small changes introduce the potential for new instructions to be used. In order to limit the amount of work that each iteration of instruction selection must do, each *RTL* has a flag that is set by any phase that modifies an *RTL* so that

the next iteration of the instruction selection phase will only operate on newly altered *RTLs*. Some of the examples in the rest of this chapter will show the importance of phase iteration.

3.3.6. Global Data-flow Analysis

The primary purpose of data-flow analysis in *vpo* is to define windows for instruction selection across basic blocks. Defining windows within basic blocks entails remembering where each register is set and defining the window when the first use of the register is encountered. Defining windows across basic blocks is not as simple because *vpo* must determine that a window is "safe" before defining it. Consider the following sequence of *RTLs*:

```
1           CC = r[16] ? 5;                 (r[16])
2           PC = CC == 0 -> L5 | PC;
3           r[17] = 0;
4           PC = L6;
      L5
5           r[17] = 1;
      L6
6           L[_base] = r[17];               (r[17])
```

The algorithm that we described above for defining windows inside basic blocks might attempt to define a window from *RTL* six to *RTL* five. If it did so, then instruction selection phase, depending on the capabilities of the target machine, might make the following optimization:

```
1           CC = r[16] ? 5;                 (r[16])
2           PC = CC == 0 -> L5 | PC;
3           r[17] = 0;
4           PC = L6;
      L5
      L6
6           L[_base] = 1;
```

This sequence of *RTLs* does not have the same semantics as the original sequence, therefore a window defined from *RTL* six to *RTL* five is not safe. A safe window will never cause the instruction selection phase to alter the semantics of the original *RTLs*.

There are two rules that *vpo* must follow to ensure that the windows defined across basic blocks are safe. The first rule prevents *vpo* from defining the unsafe window described above: define a window from the use of a register only if one definition of the register reaches the use. Looking at *RTL* six in the example shown above, it is easy to see that two definitions of `r[17]` reach *RTL* six: one at *RTL* three and a second at *RTL* five. In order to abide by this first rule, *vpo* must determine the reaching definitions for each basic block. Before stating the second rule, consider the following *RTLs*:

```
1           r[19] = 0;
2           CC = r[17] ? r[18];          (r[17],r[18])
3           PC = CC != 0 -> L19;
4           r[19] = r[19] + 5;
5           PC = L20;
      L19
6           r[19] = r[19] + 8;
      L20
7           L[_base] = r[19];            (r[19])
```

In this example we have a single definition of `r[19]`, so the first rule of window definitions across basic blocks is satisfied. Consider what might happen if *vpo* defines a window between the first use of `r[19]` at *RTL* four and its definition at *RTL* one:

```
2           CC = r[17] ? r[18];          (r[17],r[18])
3           PC = CC != 0 -> L19;
4           r[19] = 5;
5           PC = L20;
      L19
6           r[19] = r[19] + 8;
      L20
7           L[_base] = r[19];            (r[19])
```

The instruction selection phase combined *RTL* one and *RTL* four removing *RTL* one in the process. If RTL six is reached, then there is no longer any guarantee that `r[19]` will contain a zero, therefore the semantics of the original *RTLs* have been changed. The second rule of window definitions across basic blocks prevents this problem: a window can be defined from the first use if it is the only first use of the set. In order to enforce the second rule, *vpo* uses

live variable information.

Aho, Sethi and Ullman [AHO86] present two different algorithms for calculating reaching definitions and live variable information. One set of algorithms use the interval approach which works only on a class of flow graphs called *reducible*. The second set of algorithms use the iterative approach and work on any arbitrary flow graph. Although the iterative algorithms have a greater worst-case execution time, they are quite efficient for the flow graphs created from most user programs. *Vpo* uses the iterative algorithms because their ability to work with arbitrary flow graphs ensures that they are source-language independent.

Even thought the data-flow algorithms are efficient, a reduction in the number of *RTLs* that have to be considered yields a desirable decrease in the time it takes to calculate the data-flow information. *Vpo* takes advantage of this by performing dead code elimination and one instruction selection phase with windows defined only within basic blocks. These two phases combined reduce the number of original *RTLs* by an average of fifty percent. After data-flow analysis is performed, windows are defined across basic blocks and instruction selection is performed only for the *RTLs* affected by the new window definitions.

3.3.7. Register Allocation

There are two distinct register allocation phases in *vpo*. The first one deals with the problem of assigning hardware registers to temporary pseudo registers. This register allocation phase is executed only once, since *vpo*'s phases do not introduce any new pseudo registers. The second is a global register allocation phase that deals with allocating local variables to registers. This section discusses the first register allocator.

The code generator assumes that a number of registers are available on the target machine. Even though the code generator uses an average of ten pseudo registers per statement, after the initial code selection phase, the number of pseudo registers remaining is

roughly half of the original amount. Of these, only two or three of them are simultaneously live at any point in a procedure. The register allocator usually succeeds in allocating only as many hardware registers as the maximum number of instantaneously live pseudo registers. The need for special register pairs on some machines sometimes causes the register allocator to use one more hardware register than the simultaneously live number.

Register spills are introduced when the number of live pseudo registers becomes greater than the number of allocable registers available on the target machine. Registers are spilled to memory. When *vpo* is forced to spill a register, it defines a window between the spill location and the last set or next use of the register spilled so that the next iteration of the instruction selection phase can choose better instructions to perform the spill on the target machine. Consider the following sequence of *RTLs* as they would appear before the register allocation phase:

```
1           r[23] = L[_i];
2    {1}    r[25] = r[23];
            .
            .
            .
10          r[28] = L[_j];
            .
            .
            .
16          r[28] = r[28] + r[23];          (r[23])
```

After register assignment, the following *RTLs* would be left:

```
1           r[2] = L[_i];
2    {1}    r[3] = r[2];
            .
            .
            .
9           L[tmp1] = r[2];                    (r[2])
10          r[2] = L[_j];
            .
            .
            .
15          r[3] = L[tmp1];
16   {15}   r[2] = r[2] + r[3];                (r[3])
```

Note that the register allocator was forced to spill a register to a temporary memory location, but it also defined windows for the next instruction selection phase. After the next instruction selection phase, the resulting *RTLs* are:

```
1           r[2] = L[_i];
2    {1}    r[3] = r[2];
            .
            .
            .
9           L[tmp1] = r[2];                    (r[2])
10          r[2] = L[_j];
            .
            .
            .
16          r[2] = r[2] + L[tmp1];
```

RTL 15 and *RTL* 16 have combined to reference the spill location directly in the addition operation.

3.3.8. Global Register Allocation

Global register allocation assigns registers to local variables over the life of a procedure. First, the global register allocator estimates how often each local variable in a procedure is referenced. This estimation is made by using a simple heuristic that assumes that local variable references inside loops are executed more often than references outside loops. After estimating how often local variables are referenced, the global register allocator attempts to

replace the most frequently used local variable references with a register of the appropriate type, provided that there are no indirect references to the local variable. *Vpo* can determine which local variables are indirectly referenced because the address of these variables is calculated outside of a memory reference. For example, the *C* statement:

```
foo(x)
```

Where `x` is a local variable would result in the following *RTLs*:

```
1       r[2] = L[r[12] + x.];
2       L[--r[14]] = r[2];          (r[2])
3       STK = CALL(_foo);
```

An indirect reference such as:

```
foo(&x)
```

Results in the following *RTLs*:

```
1       r[2] = r[12] + x.;
2       L[--r[14]] = r[2];          (r[2])
3       STK = CALL(_foo);
```

The only difference in these examples is in the first *RTL*. In the first example, the address of local variable `x` is calculated inside a memory reference. In the second example, the address is not calculated inside a memory reference.

The process of allocating registers to local variables continues until there are no local variables remaining that are used often enough to overcome the cost of using a new register in the procedure. The cost of using a new register depends on the convention used to maintain the value stored in a register when a procedure calls another procedure. There are two popular conventions: *caller-saves* and *callee-saves*. The caller-saves convention requires the procedure that is making the call to save any live registers before making the call and restore them upon return from the called procedure. The callee-saves convention requires the called

procedure to save the registers that it will use and to restore them before returning to the caller.

When using the caller-saves convention, the cost of using a register in a procedure is determined by the number of calls made from the procedure, since each call requires a register save and restore for each register used. If a call is made from inside a loop, then the cost of saving and restoring must be multiplied by the number of times that the loop is expected to iterate. When using the callee-saves convention, the cost of using a new register is the effort required to save the register on entry to the procedure and restore it on exit.

Parameters passed to the procedure are also considered for register allocation by the global register allocator. If a parameter is allocated to a register, an initial load of the parameter from its initial location is inserted at the entry point of the procedure.

The global register allocation phase never creates spill code. If a register is not available over the whole procedure, then it is never allocated to a local variable. Since register allocation is performed before global register allocation, the number of registers allocated to temporary expressions is known, and all of the allocable registers remaining on the target machine can be used for local variable replacement. This strategy prevents *vpo* from having to limit the number registers available to each allocation phase.

3.3.9. Common Subexpression Elimination

Common subexpression elimination is performed by maintaining an equivalence list of available expressions and deleting code that recalculates an expression that is currently available. The common subexpression phase in *vpo* is similar to PO's Cacher, but has significant differences. PO's Cacher operated on *RTLs* before register assignment and was capable of detecting all common subexpressions present in a sequence of *RTLs* without labels. A label forced Cacher to forget all equivalences because PO did not have any mechanism that allowed

it to determine which equivalences were still valid at a point where multiple control paths merged. In addition, Cacher's ability to detect all common subexpressions, coupled with its lack of knowledge about how many allocable registers were available on the target machine, sometimes caused PO's register allocator to generate register spills.

Common subexpression elimination in *vpo* is performed after register allocation. The life of a register will never be extended to the point where a register spill is required. This limitation sometimes causes the common subexpression phase to miss some common subexpressions. Whereas PO's Cacher could claim to detect all common subexpressions, *vpo*'s common subexpression eliminator can claim to never make the code any worse than it was before common subexpression elimination.

PO's Cacher operated on the *RTLs* generated by the code expander. *Vpo*'s common subexpression phase does not operate until at least one instruction selection phase has executed. For an equivalent source language procedure, *vpo*'s common subexpression eliminator examines an average of half of the *RTLs* that Cacher would have to examine.

Common subexpression elimination works at the basic block level. Before the common subexpression eliminator operates on a basic block, it operates on all of the predecessors of that block. Having obtained a final equivalence list for each predecessor block, the common subexpression eliminator merges the lists to create a new equivalence list containing only equivalences that are true regardless of which path control is transferred from. Loops prevents the common subexpression eliminator from operating on all of the predecessors of a block. When a loop is detected, the common subexpression eliminator starts with an empty equivalence list, making no assumptions about which equivalences hold for the loop. We call this initial pass through a loop a *dryrun*. Once a dryrun is complete, the common subexpression eliminator operates on the loop once more, this time being able to merge the equivalence

lists of all of the predecessors of the loop. Consider the following sequence of *RTLs*:

```
1           r[3] = L[i.];
2           r[3] = r[3] * 4;
3           r[2] = L[a. + r[3]];        (r[3])
4           CC = r[2] == 0;             (r[2])
5           PC = CC == 0 -> L4 | PC;
6   L5:     r[3] = L[i.];
7           r[3] = r[3] * 4;
8           r[2] = L[a. + r[3]];        (r[3])
9           r[3] = L[i.];
10          r[3] = r[3] + 1;
11          L[i.] = r[3];               (r[3])
12          r[3] = L[i.];
13          r[3] = r[3] * 4;
14          L[a. + r[3]] = r[2];        (r[2])
15          r[3] = L[i.];
16          r[3] = r[3] * 4;
17          r[2] = L[a. + r[3]];        (r[3])
18          r[3] = L[i.];
19          CC = r[3] > 63;             (r[3])
20          PC = CC == 0 -> L5 | PC;
21  L4:     ...
```

The first iteration through the loop beginning at `L5` starts with an empty equivalence list, since it is impossible to merge the equivalence lists of all of the predecessors of `L5` until the loop is examined. This first pass yields the following *RTLs*:

```
1           r[3] = L[i.];
2           r[3] = r[3] * 4;
3           r[2] = L[a. + r[3]];        (r[3])
4           CC = r[2] > 0;              (r[2])
5           PC = CC == 0 -> L4 | PC;
6   L5:     r[3] = L[i.];
7           r[3] = r[3] * 4;
8           r[2] = L[a. + r[3]];        (r[3])
9           r[3] = L[i.];
10          r[3] = r[3] + 1;
11          L[i.] = r[3];               (r[3])
13          r[3] = r[3] * 4;
14          L[a. + r[3]] = r[2];        (r[2])
18          r[3] = L[i.];
19          CC = r[3] > 63;             (r[3])
20          PC = CC == 0 -> L5 | PC;
21  L4:     ...
```

After this first pass through the loop, the equivalence tables for the basic block ending with *RTL* 5 and the equivalence table for the loop indicate that `r[2]` and `L[a. + L[i.] * 4]` are equivalent. Assuming that these are the only predecessors of `L5`, when these two equivalence tables are merged the equivalence will remain. On the second pass through the loop, this equivalence will permit the remaining common subexpression to be eliminated. The final sequence of *RTLs* is:

```
1          r[3] = L[i.];
2          r[3] = r[3] * 4;
3          r[2] = L[a. + r[3]];        (r[3])
4          CC = r[2] > 0;              (r[2])
5          PC = CC == 0 -> L4 | PC;
6  L5:
9          r[3] = L[i.];
10         r[3] = r[3] + 1;
11         L[i.] = r[3];
13         r[3] = r[3] * 4;
14         L[a. + r[3]] = r[2];        (r[2],r[3])
18         r[3] = L[i.];
19         CC = r[3] > 63;             (r[3])
20         PC = CC == 0 -> L5 | PC;
21 L4:     ...
```

The common subexpression eliminator uses the instruction costs returned by the machine description to determine which of the items in an equivalence list provides the cheapest form of an expression. An additional advantage of using the machine description is that it prevents the common subexpression elimination phase from making any changes that result in an *RTL* that does not translate to an instruction on the target machine. This is especially useful on machines that require register pairs to perform certain operations.

3.3.10. Constant Propagation

Constant propagation is particularly effective for optimizing array references. Consider the following *C* statement:

```
a[i] = a[i + 1] + a[i + 2];
```

where `a` is an integer array and `i` is an integer. The *RTLs* for this statement would look like this:

```
1          r[1] = L[i.];
2  {1}     r[1] = r[1] * 4;
3          r[2] = L[i.];
4  {3}     r[2] = r[2] + 1;
5  {4}     r[2] = r[2] * 4;
6  {5}     r[2] = L[a. + r[2]];
7          r[3] = L[i.];
8  {7}     r[3] = r[3] + 2;
9  {8}     r[3] = r[3] * 4;
10 {9}     r[3] = L[a. + r[3]];
11 {6,10}  r[3] = r[3] + r[2];       (r[2])
12 {2,11}  L[a. + r[1]] = r[3];      (r[1],r[3])
```

The constant propagation phase attempts to locate instances where a constant can be propagated through an operation. In the *RTLs* shown above, constant propagation would perform the following transformations:

```
1          r[1] = L[i.];
2  {1}     r[1] = r[1] * 4;
3          r[2] = L[i.];
5  {3}     r[2] = r[2] * 4;
6  {5}     r[2] = L[a. + r[2] + 4];
7          r[3] = L[i.];
9  {7}     r[3] = r[3] * 4;
10 {8}     r[3] = L[a. + r[3] + 8];
11 {6,10}  r[3] = r[3] + r[2];       (r[2])
12 {1,11}  L[a. + r[1]] = r[3];      (r[1],r[3])
```

The fact that adding a constant to an expression before a multiplication is the same as adding the constant times the multiplicand to the result is exploited on machines having a displacement addressing mode. The biggest payoff in constant propagation is obtained when it is used in conjunction with common subexpression elimination. *vpo*'s common subexpression eliminator would not have been able to perform any further optimizations on the original set of *RTLs*: with constant propagation, common subexpression elimination can make the

following optimizations to the code:

```
1          r[1] = L[i.];
2  {1}     r[1] = r[1] * 4;
6          r[2] = L[a. + r[1] + 4];
10         r[3] = L[a. + r[1] + 8];
11 {6,10}  r[3] = r[3] + r[2];        (r[2])
12 {1,11}  L[a. + r[1]] = r[3];       (r[1],r[3])
```

3.3.11. Dead Store Elimination

Dead store elimination removes useless stores by keeping track of the number of times that a register or a memory location is used before it is updated. Dead store elimination is built into the common subexpression phase, since all the information required to eliminate dead stores safely is available to the common subexpression eliminator.

Although most useless stores are avoidable by writing more efficient code, there are instances where a front end will generate useless stores even when the user has written what appears to be efficient code. Consider the following *C* code:

```
struct {
   unsigned f1 : 1;
   unsigned f2 : 1;
} flags;

flags.f1 = 1;
flags.f2 = 1;
```

The *RTL*s generated for this example are:

```
1          r[1] = L[flags.];
2  {1}     r[1] = r[1] | 1;
3  {2}     L[flags.] = r[1];          (r[1])
4          r[1] = L[flags.];
5  {4}     r[1] = r[1] | 2;
6  {5}     L[flags.] = r[1];          (r[1])
```

Dead store elimination removes the useless store at *RTL* three. The combined effect of dead store elimination and common subexpression elimination will result in the following sequence

of *RTLs*:

```
1          r[1] = L[flags.];
2  {1}     r[1] = r[1] | 1;
5  {2}     r[1] = r[1] | 2;
6  {5}     L[flags.] = r[1];          (r[1])
```

A final iteration of the instruction selection phase will improve the code even further:

```
1          r[1] = L[flags.];
5  {1}     r[1] = r[1] | 3;
6  {5}     L[flags.] = r[1];          (r[1])
```

There are rare instances where eliminating what appears to be a useless store results in incorrect code. Some machines map device control registers to special memory locations, for instance, and stores to such locations must not be eliminated. There is a mechanism which can be used to prevent *vpo* from removing what appears to be a useless store. The code expander can emit a *trash* command, which is a line starting with a "`t`" and followed by the memory location whose previous history should be forgotten. Without the previous history of a memory location, *vpo* cannot determine if the next store to that memory location is useless, therefore, *vpo* will not eliminate any subsequent store. If a machine has device registers, then the code expander should be made capable of recognizing them so that it can emit a trash command whenever a store is made to one of these registers. If it is impossible to determine when a device register is referenced, then the user must supply a special option to *vpo* that prevents dead store elimination.

CHAPTER 4

RESULTS

The real test of a retargetable optimizing compiler is to port it to a number of machines and gauge not only the quality of the code that it generates, but also the effort required to retarget it. Towards this end, we have ported *vpo* to three machines: the VAX-11/780, the SUN-3/75 and the Concurrent 3230. This section presents the results of our efforts and some of the lessons learned along the way.

4.1. Machine Variations

The three machines used to measure *vpo*'s effectiveness represent a reasonable cross-section of the problems facing retargetable optimizing compilers. Some of the potential stumbling blocks that we have identified while retargeting *vpo* are:

(1) Complex addressing modes

(2) Non-regular use of addressing modes or registers

(3) Pairing adjacent registers

(4) Procedure calling conventions

Complex addressing modes present a potential problem by permitting a large number of combinations of addressing modes and instructions. A rule-based system of code generation, for example, would require a large number of rules to handle all of these combinations. A table-driven system might suffer inefficiencies because the size of the table is exponentially related to the number of addressing modes and instructions available. Fortunately, *vpo* factors out the addressing modes from the instructions, so that the problem is not how many different addressing modes are available on the target machine, but rather how many exceptions there

are as to which modes cannot be used in particular situations. For example, despite the fact that the VAX-11/780 provides twice as many addressing modes as the MC68020-based SUN-3/75, the machine description for the VAX-11 is only 95 lines longer than the description for the MC68020 (389 vs. 294).

It is difficult to find machines that do not have at least one exception to otherwise regular addressing modes and instruction sets. On the Concurrent 3230, for example, all of the general-purpose registers with the exception of register zero can be used in indexed addressing modes. Exceptions are difficult to express in a machine description. In order to handle them, semantic actions can be invoked to perform special case tests. In the case of the 3230, the description of the indexed addressing mode syntax allows any general-purpose register to form an indexed expression, but a semantic action is always invoked that checks the register and sets an error condition if the register used is register zero.

Some machines have instructions and data formats that implicitly use adjacent register pairs. On the VAX-11, double-precision floating point values are stored in general-purpose register pairs. The addressing mode states only the lowest numbered register of the pair. On the 3230, integer multiplications and divisions require odd/even register pairs to be available. The burden of providing appropriate register pairs, triples and so forth falls on the register allocator. The current register allocator in *vpo* is a product of two major revisions and four upgrades intented to enhance its ability to be quickly adapted to the particular needs of a large number of architectures.

Machines vary widely on the mechanisms that they provide to support the procedure model. On the VAX-11, an instruction is provided that not only transfers control to a procedure, but also saves and restores registers automatically through the use of a *mask* indicating the registers that should be saved on the call and restored on the corresponding return. The

MC68020 provides special instruction that set up and tear down an activation record. The 3230 is the most primitive of the three machines, providing only a simple branch-and-link instruction that saves the return address in a specified register. Multiple register save and load instructions are also provided on the 3230, but their usefulness is somewhat limited in that only the starting register number can be given, and the multiple store and load continues until every register from the starting register number to the highest available register of the given type is saved or loaded.

There are two popular schemes used to manage register usage across procedure calls: *caller-saves* and *callee-saves*. The caller-saves scheme in its purest form allows a procedure to modify the value of any register, thus forcing the caller to save and restore the value of any register whose life must reach across the call. The callee-saves scheme ensures that a procedure call will not alter any of the registers, thus placing the burden of saving and restoring any register that are needed on the called procedure. In practice, the schemes used tend to be mostly caller-saves or callee-saves with the exception of a small number of registers, known as *scratch* registers, that may or may not be altered across a procedure call. A smart compiler will use scratch registers to hold temporary values whose lives do not cross a procedure call because no overhead is required to save and restore these registers.

Sometimes, the instructions provided by a machine to support procedures suggest the best scheme for register management across procedure calls. For this reason, *vpo* is capable of supporting both the caller-saves and the callee-saves schemes. Our method for determining which scheme to use on a given machine is to adopt the same calling conventions used by the existing *C* compiler on the target machine. The ability to use the existing *C* library functions and to use the ability to compile one procedure or file at a time with *vpo* and link it to procedures or files compiled by a fully-tested compiler to aid debugging and reduce the amount of time required to retarget to a new machine. After the machine description is fully tested, it

is easy to choose or develop an alternate calling scheme and make the required changes to employ it. Then, the source of the *C* library can be compiled with the new compiler to bootstrap the system. Currently, the VAX-11 and the MC68020 versions of *vpo* implement a callee-saves scheme and the 3230 version uses a caller-saves approach.

4.2. Code Quality

Since portability is a desirable property of a compiler, it would not be unthinkable to trade-off code quality and compilation speed in favor of portability. Our experience, however, suggest that code quality does not have to a take back seat to portability. We cannot make the same claim regarding compilation speed, since we have not given that aspect of the compiler as much attention as we have given code quality, and because the subject has already been researched with success [DAVI88].

We have tested the quality of the code generated by *vpo* by compiling a test suite with both *vpo* and the existing vendor-supplied compilers on the VAX-11/780, the SUN-3/75 and the Concurrent 3230. When compiling the test suite, the compilers were instructed to apply all optimizations at their disposal. The test suite was then executed during light-load conditions and the amount of user and system time required to execute each program was recorded. The results are shown in Table I.

The test suite was chosen to provide a fairly broad range of program types. *Ackerman*, *sieve*, *queens*, and *puzzle* are well known benchmark programs. *Mincost* and *tsp* were chosen for their intensive floating-point calculations. They employ simulated annealing to solve VLSI wire-routing and the classic traveling salesman problem respectively. *Cache* is a cache simulator program. *Wc*, *grep*, *sort* and *nroff* are the standard Unix* utilities included to round

Program	VAX-11/780			SUN-3/75			Concurrent 3230		
	cc -O	*vpo*	*vpo*/cc	cc -O	*vpo*	*vpo*/cc	cc -O	*vpo*	*vpo*/cc
ackerman	4.76	4.30	0.90	1.10	1.06	0.96	3.59	3.46	0.96
sieve	1.69	1.53	0.91	0.65	0.61	0.94	2.34	2.44	1.04
queens	0.61	0.56	0.92	0.33	0.26	0.79	0.68	0.58	0.85
puzzle	10.71	6.98	0.65	3.76	3.07	0.82	10.36	9.71	0.94
mincost	12.52	11.73	0.94	5.76	4.56	0.79	18.34	18.47	1.01
tsp	281.09	218.09	0.76	119.01	112.80	0.95	232.70	228.70	0.98
cache	14.81	14.53	0.98	39.74	39.41	0.99	22.31	21.97	0.98
wc	1.58	1.21	0.77	0.78	0.51	0.65	1.06	1.10	1.04
grep	4.34	4.07	0.94	2.14	2.01	0.94	6.20	6.90	1.11
sort	69.63	73.60	1.06	34.66	35.81	1.03	91.48	109.49	1.20
nroff	20.31	18.92	0.93	8.59	8.44	0.98	23.39	24.86	1.06

Table I. Comparison of Execution Times (in seconds).

out the suite with some "real" programs.

The table contains three columns for each machine. The first column shows the time in seconds required to execute the benchmark compiled with the existing compiler on the target machine. The second column shows the time in seconds required to execute the benchmark compiled with *vpcc* and *vpo*. The third column is the result of dividing column two by column one. In cases where *vpo* generates higher-quality code than the existing compiler, the dimensionless number in the third column will be less-than one. When *vpo* fails to generate higher-quality code than the existing compiler, the number in the third column will be greater-than one. All numbers are rounded to the nearest hundredth.

The execution times show that *vpo* generates better quality code than the existing compiler on the VAX-11/780 and the SUN-3/75 for all but one program. The inability to generate better quality code for *sort* is the failure of the global register allocator to fully appreciate the

*Unix is a Trademark of Bell Laboratories

use of local variables in the sorting algorithm, consequently requiring registers to be saved and restored more often than their actual use warrants. The heuristic used by *vpo* assumes that a local reference inside a loop is executed more often than a reference outside of a loop. This is not always true. Consider the following loop:

```
if (cond)
  for (i = 0; i < 100; i++)
    a[i] = 0;
```

The variable `i` may be referenced either a few hundred times or zero times, depending on the value of `cond`. *Vpo* assumes that any local variable referenced inside a loop is used often enough to offset the cost of allocating it to a register. Using more complicated static (and perhaps even dynamic) procedure analysis, it is possible to improve on *vpo*'s global register allocation heuristic. It is encouraging, however, that even a simple heuristic can make reasonable decisions in the majority of cases.

On the Concurrent 3230, *vpo* was not able to produce significantly better code than the existing compiler. There are two reasons for *vpo*'s failure. First, the 3230 executes instructions faster when they begin on a longword boundary, a fact that we were not aware of while we where retargeting *vpo* to the 3230. The existing compiler on the 3230 avoids emitting instruction that occupy more or less than a longword. *Vpo*'s machine description and associated semantic actions have no knowledge of the longword boundary factor, and sometimes make code "improvements" that actually increase execution time. Second, in keeping with our policy of adopting the calling conventions of the existing compiler on the target machine, we used the caller-saves convention on the 3230. The caller-saves convention changes the heuristics that estimate the payoff of allocating a register to a local variable. We feel that the heuristic used to estimate payoffs for the caller-saves convention can, and should be, improved in the future.

4.3. Effectiveness

Since *vpo* maintains the *RTL* notation at every phase of the optimizer, it is possible to refrain from invoking any particular phase without affecting the correctness of any of the other phases. Additionally, by omitting an optimization, we can calculate the effectiveness of any particular optimization by compiling with and without the optimization in question and comparing the effect on the quality of the code generated. We are, of course, neglecting the fact that optimizations overlap so that the full effects of one optimization can not be fully measured in the presence of another. The next few sections present the results that we obtained by disabling certain optimizations. The results are presented only to show how *vpo* can be used to question the effectiveness of a given optimization on a particular machine, and not to attempt to prove the effectiveness of any one optimization over another.

4.3.1. Windows Across Basic Blocks

We have already discussed *vpo*'s ability to perform instruction selection across basic blocks by utilizing data-flow information to determine when it is safe to do so. Table II presents the results obtained by comparing the execution times of our test suite compiled with and without this optimization. The results suggest that performing instruction selection across basic blocks does not yield a high payoff. The reason is that the code generated by the *C* front end does not extend the life of temporary registers across a basic block with the notable exception of the ? operator. Experimentation with a number of other languages might prove this optimization to be more worthwhile than our current findings indicate.

4.3.2. Register Allocation

The effectiveness of the register allocator can be summed up with the following statement: no register spills are generated by *vpo* for any of the programs in the test suite. Some spills are needed to handle live registers across function calls on the 3230 version because the

Program	VAX-11/780			SUN-3/75			Concurrent 3230		
	none	links	lnk/no	none	links	lnk/no	none	links	lnk/no
ackerman	4.30	4.30	1.00	1.06	1.06	1.00	3.46	3.46	1.00
sieve	1.53	1.53	1.00	0.61	0.61	1.00	2.44	2.44	1.00
queens	0.56	0.56	1.00	0.26	0.26	1.00	0.58	0.58	1.00
puzzle	6.98	6.98	1.00	3.07	3.07	1.00	9.71	9.71	1.00
mincost	11.73	11.73	1.00	4.56	4.56	1.00	18.47	18.47	1.00
tsp	218.09	218.09	1.00	112.80	112.80	1.00	228.70	228.70	1.00
cache	14.53	14.53	1.00	39.50	39.41	1.00	22.00	21.97	1.00
wc	1.25	1.21	0.97	0.52	0.51	0.98	1.10	1.10	1.00
grep	4.19	4.07	0.97	2.01	2.01	1.00	6.90	6.90	1.00
sort	73.41	73.60	1.00	35.90	35.81	1.00	109.79	109.47	1.00
nroff	19.43	18.92	0.97	8.54	8.44	0.99	24.50	24.86	1.01

Table II. Links Across Basic Blocks Comparison of Execution Times (in seconds).

caller-saves scheme is used, but these are generated only because the spill mechanism built into the register allocator conveniently handles saving live registers across procedure calls and not because the register allocator runs out of allocable registers.

4.3.3. Global Register Allocation

The global register allocator attempts to do what *register* type declarations are intended to do, but, since the global register allocator knows how many registers are available on the target machine, it should be able to do a better job of assigning local variables to registers than the user. One could argue that declaring which variables should be assigned to registers at the *C* source code level is flawed since it introduces machine dependencies in a language that is intended to be portable. Additionally, the *register* declaration scheme is not fully transparent because it prevents the user from declaring both an argument and a local variable as candidates for registers with the implication that the local variable should be given preference over the argument.

Table III presents the results of comparing the execution times of our test suite compiled with and without global register allocation. The respectable improvement in execution times suggests that global register allocation is a worthwhile optimization. The data also suggests that global register allocation is more effective on some machines than others. On the VAX-11/780, global register allocation is not as effective as on the other machines. In some cases, global register allocation results in a slower program because it fails to take into account the exact costs of saving and restoring a register against referencing a local in memory.

4.3.4. Phase Iteration

In order to be able to apply phase iteration to solve phase ordering problems, *vpo* maintains the *RTL* notation through all of the optimization phases. This strategy assumes that phase iteration yields enough improvements in the quality of the code to make the effort of implementing it worthwhile. Table IV presents the results of executing our test suite com-

Program	VAX-11/780			SUN-3/75			Concurrent 3230		
	none	regs	reg/no	none	regs	reg/no	none	regs	reg/no
ackerman	4.15	4.30	1.04	1.07	1.06	0.99	3.54	3.46	0.98
sieve	2.86	1.53	0.54	1.59	0.61	0.38	3.38	2.44	0.72
queens	0.62	0.56	0.90	0.40	0.26	0.65	0.66	0.58	0.88
puzzle	10.13	6.98	0.69	5.08	3.07	0.60	12.92	9.71	0.75
mincost	12.76	11.73	0.92	5.33	4.56	0.86	19.29	18.47	0.96
tsp	216.66	218.09	1.01	127.28	112.80	0.87	228.90	228.70	1.00
cache	14.87	14.53	0.98	40.36	39.41	0.98	22.82	21.97	0.96
wc	1.30	1.21	0.93	0.77	0.51	0.66	1.17	1.10	0.94
grep	5.72	4.07	0.71	3.24	2.01	0.62	8.14	6.90	0.85
sort	87.89	73.60	0.84	47.43	35.81	0.76	129.92	109.49	0.84
nroff	19.88	18.92	0.95	9.92	8.44	0.85	26.95	24.86	0.92

Table III. Global Register Allocation Comparison of Execution Times (in seconds).

piled with and without phase iteration. The results suggest that phase iteration is effective on machines with a small number of allocable registers. Even though the MC68020 has more registers than the VAX-11, the fact that the VAX-11 has twice as many general purpose registers than the MC68020 has of any one given type of register appears to make a difference.

The majority (80%) of the procedures in our test suite required no iteration. When iteration was required, an additional local variable was allocated to a register in about half of the cases. Except in rare circumstances, the code improvements yielded a 1% to 3% improvement in the total execution time of the program.

4.4. Retargetability

The primary intent of the *vpcc/vpo* compiler system is to provide a compiler capable of generating high-quality code in a short period of time. Obviously, familiarity with the system decreases the amount of time required to retarget the compiler. A general overview of the

Program	VAX-11/780			SUN-3/75			Concurrent 3230		
	none	iter	iter/no	none	iter	iter/no	none	iter	iter/no
ackerman	4.30	4.30	1.00	1.06	1.06	1.00	3.46	3.46	1.00
sieve	1.53	1.53	1.00	0.61	0.61	1.00	2.44	2.44	1.00
queens	0.56	0.56	1.00	0.26	0.26	1.00	0.58	0.58	1.00
puzzle	6.98	6.98	1.00	3.07	3.07	1.00	9.74	9.71	1.00
mincost	11.73	11.73	1.00	4.67	4.56	0.98	18.45	18.47	1.00
tsp	218.09	218.09	1.00	115.30	112.80	0.98	228.90	228.70	1.00
cache	14.53	14.53	1.00	39.46	39.41	1.00	21.92	21.97	1.00
wc	1.21	1.21	1.00	0.51	0.51	1.00	1.10	1.10	1.00
grep	4.07	4.07	1.00	2.34	2.01	0.86	7.15	6.90	0.97
sort	73.60	73.60	1.00	35.39	35.81	1.01	110.19	109.49	0.99
nroff	19.44	18.92	0.97	8.74	8.44	0.97	25.09	24.86	0.99

Table IV. Iteration Comparison of Execution Times (in seconds).

procedure required to retarget the compiler system to a new machine is presented here. A more detailed, step-by-step description will appear in a future technical report.

4.4.1. The Front End

The front end needs to know only two things about the target machine: the sizes of the basic data types (integers, floating and double) and the order of evaluating arguments to functions (left-to-right or right-to-left) that best suits the underlying machine architecture.

4.4.2. The Code Expander

The code expander is organized as a *switch* statement where each of the 46 intermediate language operations is a *case* limb. Many of the operations have a one-to-one correspondence with an instruction on the target machine. Code expanders vary in size depending on the capabilities of the target machine. The code expanders for the VAX-11/780, SUN-3/75 and Concurrent 3230 are 1068, 938 and 950 lines long, respectively. Only 40 to 60 percent of the code expander is machine dependent, the rest consists of support code that is identical on all machines. Similarities in machines can be exploited to simplify the task of retargeting the code expander. For example, the Concurrent 3230 is very similar to some IBM processors and would be a logical version to modify into a code expander for these machines.

Machine dependencies affecting the code generator are: types and number of registers available on the target machine, calling conventions (caller-saves or callee-saves), the calling stack (some machines provide no instructions that efficiently implement a stack) and, most importantly, the instruction set available to the user. Our experience has been that given some familiarity with existing code expanders, a code expander for a new target machine can be developed by one person in less than a week. Additional tools and documentation could reduce this time to a single day.

4.4.3. The Optimizer

Most of the effort required to retarget *vpo* goes into writing the machine description and the associated semantic actions that comprise the core of the instruction selection mechanism. Once the machine description is written, *yacc* builds the parser that, along with the semantic actions and a simple lexical scanner, determine if a given *RTL* denotes a valid instruction on the target machine. The semantic actions must be able to output the machine instruction in assembler that the *RTL* represents and calculate a cost for the instruction. The assembly output is required in order to create a file that the assembler for the target machine can assemble into an object file. The cost calculation is used by the common subexpression elimination phase to choose the "cheapest" operands out of a set (for example, if an expression is available in both a register and a memory location, the cost would determine which of the two can be accessed faster). Also, since it is not always the case that the result of combining two instructions yields an instruction that executes faster than the sequential execution of the instructions that combined to form it, instruction costs prevent such combinations.

Like the code expander, *vpo* also needs to know the type and number of registers available on the target machine. Functions capable of mapping register types to actual machine registers must be provided, since on some machines, like the VAX-11, floating point register types are mapped onto the general purpose register set. The number of allocable registers and scratch registers is also needed. Register pairs and triples are defined by writing functions that, given a register in a pair or triple, returns the next register in the series. The function calling scheme (caller-saves or callee-saves) affects the payoff functions associated with allocating local variables to registers. A function that is called when *vpo* has finished optimizing a procedure must be written to insert code to set up and tear down an activation record. We delay inserting activation record code until a procedure is optimized because allocating locals to variables can affect the selection of the optimal code sequence required to set up the

activation record. For example, when using the callee-saves scheme, if many registers have to be saved when entering the procedure, then a store multiple instruction might be utilized, whereas if only one register needs saving, a normal store instruction is more appropriate.

Roughly 1000 lines of *vpo*'s 10000 lines of *C* source code and *yacc* machine description need to be written or modified to retarget the optimizer. Although the actual time required to retarget *vpo* depends on the complexity of the target machine, a group of two or three inexperienced persons can retarget the complete compiler in less than a month. A single experienced person can retarget the system in two weeks.

CHAPTER 5

CONCLUSIONS

This thesis has described a successful portable optimizing compiler system. In conclusion, we wish to point out the strategies that enabled us to develop this system and the areas that we feel require further investigation.

5.1. Strategies

Building portable optimizing compilers has become an easier problem. The strategies and tools available have changed to ensure that many of the common pitfalls can be avoided. The following strategies have been applied during the development of *vpcc/vpo*:

(1) Use a simple front end to perform a naive but correct translation of the source language to a simple intermediate language. This strategy simplifies the task of writing a front end.

(2) Use a simple code expander to translate the intermediate language into a sequence of target machine instructions. No optimizations are applied to the code by the code expander. This strategy allows a code expander for a new target machine to be written in a few days.

(3) Use the *RTL* notation to represent machine-dependent instructions. The machine-independent nature of *RTLs* allows them to be manipulated by machine-independent algorithms.

(4) Perform all optimizations at the machine level where all of the factors affecting code selection are exposed to the optimizer. Use a single mechanism, such as a machine description, to make all machine-dependent decisions.

(5) Maintain the *RTL* notation throughout every phase of the optimizer, so that phase iteration can be applied to eliminate phase ordering problems.

By adhering to these strategies, we have been able to develop a portable optimizing compiler. Each strategy emphasizes the need to delay code generation decisions until all aspects of the target machine can be taken into consideration and to concentrate knowledge of the target machine into a simple but powerful mechanism that can be quickly retargeted. Note that none of these strategies simplify the problems of code generation and optimization, they merely factor out the machine-dependent aspects of the problem and allow the user to provide his knowledge of these aspects through a well-defined mechanism. To enhance portability, we need only to provide tools that more quickly capture the knowledge of the machine dependencies from the user. Currently, *yacc* has been the primary tool, but as our appreciation of the capture process increases, the need for a more powerful tool becomes apparent.

5.2. Further Research

We believe *vpcc/vpo* to be a step towards a highly-optimizing, highly-portable compiler system. We are encouraged by its capabilities, yet quite aware that further development is needed to reach the elusive goal of a compiler whose code quality is at least equal to the best optimizing compilers, whose compilation speed rivals that of the quickest compilers and whose portability permits it to be retargeted in a few days, perhaps even in a few hours, by the average user. Currently, our system falls short on all three aspects, but *vpcc/vpo* is not a dead end and every improvement we make seems to suggest even more possibilities for improvement.

5.2.1. Code Quality

We have only implemented a small number of the machine independent optimizations that have been employed in optimizers. Code improvement techniques including induction variable elimination, code motion, loop unrolling and evaluation order determination will make even further improvements to the quality of the code. Improvements can also be made to some of the existing phases. The global register allocator, for example, should be modified to use a map coloring approach to register allocation. Further optimizations can be realized by expanding the use of *RTL*'s to the linking phase of the program. Unlike *vpo*, an *RTL* linker can operate on the whole program, applying optimizations to utilize registers more effectively across procedure calls, pass procedure arguments in registers, allocate global variables to register, remove unreferenced procedures and eliminate tail recursion.

5.2.2. Compilation Speed

Compilation speed has steadily improved even as code quality has improved. Early versions of the system would require as much as twenty times the amount of time required by existing compilers to compile a program. The lastest versions of *vpcc/vpo* take a little less than twice as long to compile a program as the existing compilers that we compared our test suite against. Further improvements can be made, especially in the *RTL* parser. In the near future, we will use a parser generated not by *yacc* which creates a table-driven parser, but by a compiler-compiler that emits code that directly interprets each parsing action. We expect no less than a twenty to thirty percent speed increase with this improvement. Additional improvements can be realized by making changes to *RTLs* to reduce their size and reduce the work required to locate important elements embedded in them, such as registers and local variable references.

5.2.3. Retargetability

Significant improvements in retargetability hinge on the development of new tools that will simplify writing code expanders and machine descriptions. Each new machine that the system is retargeted to yields further insight on which aspects of the system need improvements. Ad-hoc methods of dealing with machine idiosyncrasies have to be isolated and replaced with general mechanisms.

BIBLIOGRAPHY

[AHO86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[AIGR84] P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick and E. Petegri-Llopart, Experience with a Graham-Glanville Style Code Generator, , June 1984.

[AUSL82] M. Auslander and M. Hopkins, An Overview of the PL.8 Compiler, *Proceedings of the SIGPLAN Notices '82 Symposium on Compiler Construction*, Boston, MA, June 1982, 22-31.

[BACK81] J. W. Backus, History of Fortran I, II, and III, in *History of Programming Languages*, R. L. Wexelblat (ed.), New York Academic Press, 1981, 45-66.

[BELL71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[BENI88] M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 329-338.

[CHOW83] F. C. Chow, *A Portable Machine-Independent Global Optimizer—Design and Measurements*, PhD Dissertation, Stanford University, Stanford, CA, 1983.

[DAVI81] J. W. Davidson, *Simplifying Code Generation Through Peephole Optimization*, PhD Dissertation, University of Arizona, December 1981.

[DAVI88] J. W. Davidson and D. B. Whalley, Quick Compilers Using Peephole Optimization, *to appear in Software—Practice & Experience*, , December 1988.

[DIGI76] *DecSystem10 Hardware Reference Manual*, Digital Equipment Corporation, Maynard, MA, 1976.

[DIGI81] *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, MA, 1981.

[GANA82] M. Ganapathi, C. N. Fischer and J. L. Hennessy, Retargetable Compiler Code Generation, *Computing Surveys 14*, 4 (December 1982), 573-592.

[GRAH82] S. L. Graham, R. R. Henry and R. A. Schulman, An Experiment in Table Driven Code Generation, *Proceedings SIGPLAN Notices '82 Symposium on Compiler Construction*, Boston, MA, June 1982, 32-42.

[JENS75] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, NY, 1975.

[JOHN78] S. C. Johnson, Yacc: Yet Another Compiler-Compiler, *Unix Programmer's Manual 2B*, Section 19 (July 1978), 1-34.

[JOHN86] M. S. Johnson and T. C. Miller, Effectiveness of a Machine-Level, Global Optimizer, *Proceedings of the SIGPLAN Notices '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986, 99-108.

[KERN78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KNUT68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2*, 2 (1968), 127-145.

[LESK79] M. E. Lesk and E. Schmidt, Lex - A Lexical Analyzer Generator, *Unix Programmer's Manual 2B*, Section 20 (January 1979), 1-13.

[PERK82] *Model 3230 Processor User's Manual*, Perkin-Elmer, Inc., Oceanport, NJ, 1982.

[PERK79] D. R. Perkins and R. L. Sites, Machine-Independent Pascal Code Optimization, *Proceedings of the SIGPLAN Notices '79 Symposium on Compiler Construction*, Denver, CO, August 1979, 201-207.

[PREN85] *MC68020 32-Bit Microprocessor User's Manual*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[STEE61] T. B. Steel, A First Version of Uncol, *Proceedings of the Western Joint Computer Conference*, , May 1961, 371-378.

[WATT86] J. S. Watts, *Construction of a Retargetable C Language Front-end*, Masters Thesis, University of Virginia, Charlottesville, VA, 1986.