EXPERIENCE USING A RETARGETABLE PEEPHOLE OPTIMIZER TO ACHIEVE COMPILER PORTABILITY

by

Janell K. Hancock

CU-CS-287-85 April, 1985

University of Colorado, Department of Computer Science, Boulder, Colorado.

EXPERIENCE USING A RETARGETABLE PEEPHOLE OPTIMIZER TO ACHIEVE COMPILER PORTABILITY

by

Janell Kay Hancock

B.S., Colorado State University, 1979

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

1985

Hancock, Janell Kay (M.S., Computer Science)

Experience Using a Portable Peephole Optimizer to Achieve Complier Portability

Thesis directed by Professor William M. Waite.

With the continual introduction of new microprocessors into the marketplace, compiler portability has become increasingly important. In order to adapt an existing compiler to a new processor architecture, the code generation portion of the compiler must be modified. The purpose of this research is to determine the feasibility of using a retargetable code generator/peephole optimizer to automate this effort.

For this project, the University of Arizona Peephole Optimizer (PO) has been employed as a compiler retargeting tool. It has been interfaced to the PASCAL P4 compiler front-end in order to generate code for the MC68000 microprocessor. This paper describes the implementation of the code generation portion of the compiler using the peephole optimizer. The practicality of this implementation method is assessed based on the required effort, the compiler speed, the applicability of PO, and the quality of the resulting MC68000 assembly code produced by the peephole optimizer.

CONTENTS

FIGURES

Figure

CHAPTER I

INTRODUCTION

Assuming the front-end of a compiler is machine independent, retargeting a compiler entails rewriting its code generation portion. A good code generator must perform complicated case analysis in order to produce efficient code. This takes time to implement and the resulting code may still contain inefficiencies. An alternate approach is to generate code locally without case analysis and then process it with a peephole optimizer to increase its efficiency. In either case, the code generation phase of the compiler must handle the machine-dependent tasks of register and temporary allocation and memory mapping.

Attempts have been made to automate the retargeting process by providing tools to aid in building the code-generation portion of a compiler. One such tool is the University of Arizona Peephole Optimizer (PO) [2,3]. It is a retargetable code generator/peephole optimizer driven by a description of the target machine. Based on the machine description defining each instruction in terms of register transfers, PO can simulate adjacent instructions, combining them into single instructions where possible. By performing a simple flow analysis, it finds logically adjacent instructions that are physically separated. PO is advertised as a fast retargeting method [3]. It allows the implementor to generate straightforward non-optimal assembly code sequences from the source language. Peephole optimization transforms this naive code into production-quality assembly code. The implementor need not be concerned with register allocation but instead may assume an infinite supply of pseudo-registers. In its final phase, PO handles hardware register and temporary allocation. Only the memory mapping task is left to the implementor.

This paper describes a project in which PO was used as a tool to retarget the PASCAL

P4 compiler for the Motorola MC68000 [9] processor. The goal was to build a cross-compiler running on a VAX† but generating assembly code for the MC68000. The purpose of the project was to gain experience with PO in order to evaluate its viability as a retargeting tool.

† VAX is a trademark of Digital Equipment Corporation

CHAPTER II

SYSTEM COMPONENTS

In order to test the adaptability of the retargetable peephole optimizer, other system components were chosen that were not specifically designed to interface with it. Figure 2.1 shows the organization of the components in the complete system design. The following sections briefly describe the major software components used as the starting point for this project. Where applicable, the advertised retargeting method is summarized.

PASCAL P4 Compiler

The portable PASCAL P4 compiler [14] was retargeted in this experiment. It compiles a subset of standard PASCAL and is written in the subset it processes. The features not supported are packed characters, procedures and functions as parameters, GOTO statements leading out of procedure or function bodies, and user-defined files. (Four predefined text files are supported: two input files and two for output.) In addition, the procedures MARK and RELEASE replace the standard procedure DISPOSE.

The output from P4 is object code for a hypothetical stack computer (P-code). It is intended to be adaptable to a wide variety of machine architectures, however, it makes a commitment to certain data representations in memory for activation records. The address assignment performed by the compiler runs from low to high memory for each data segment representing a procedure. In most instances simple data elements and bases of complex data structures are addressed by the pair [relative static level, offset]. Further offsets into data structures are positive integers. In certain cases, global-level data elements are addressed directly. The suggested memory layout for an activation record is shown in Figure 2.2.

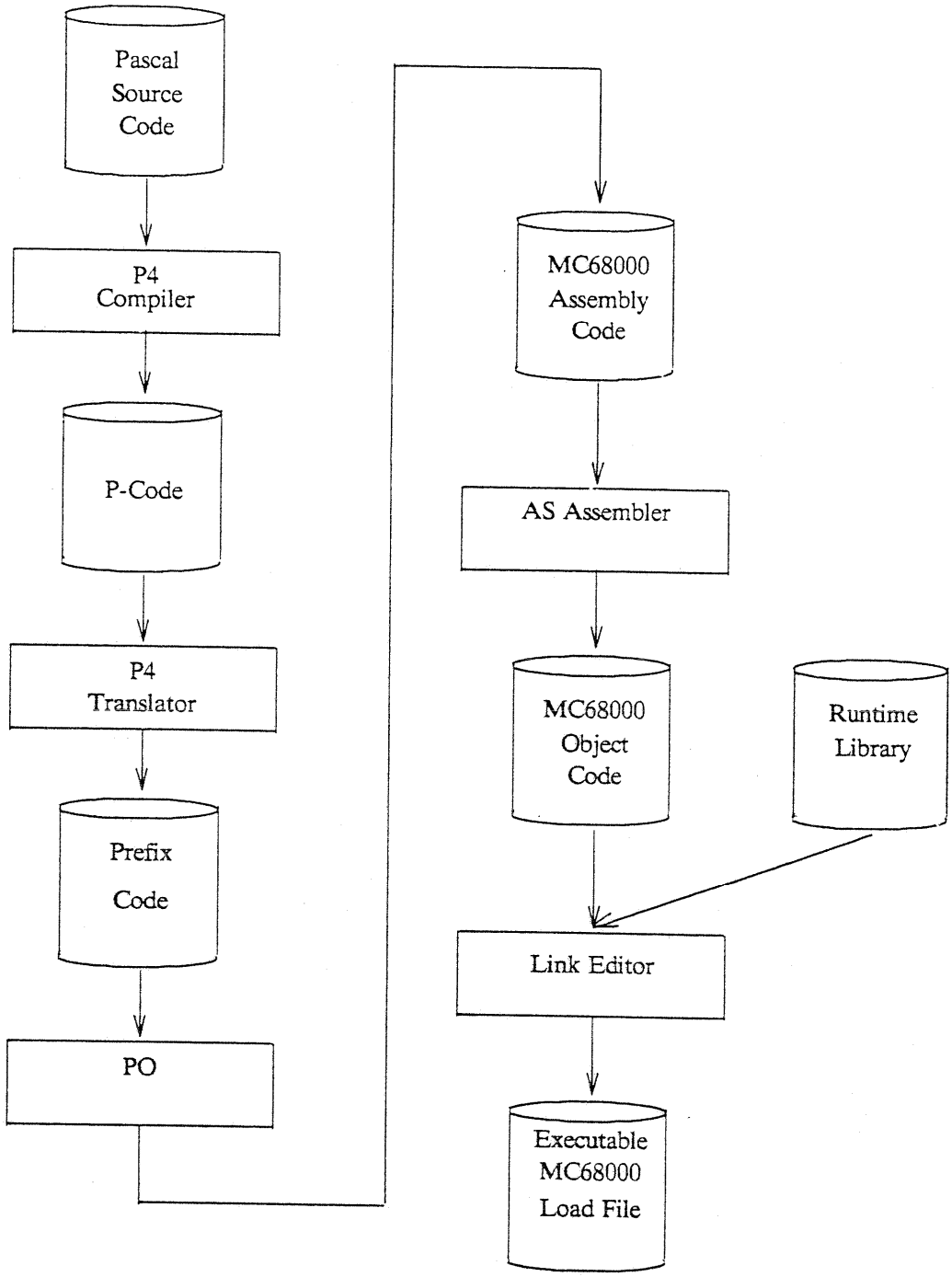

Figure 2.1. System Components

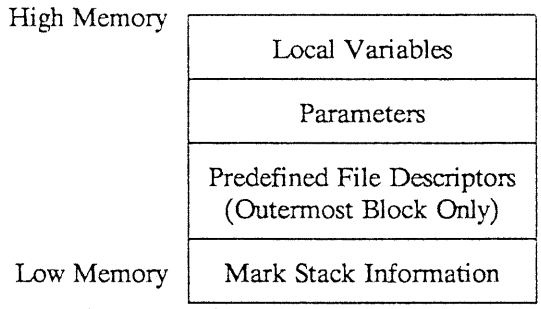

Figure 2.2. Suggested Activation Record Layout

Certain constants must be set in the P4 compiler source in order to retarget it for a new machine. These constants specify the size in storage units of PASCAL data types, as well as stack alignment, stack element size, file descriptor size, and minimum and maximum values of integers and characters. Adjustment of these parameters should be the only compiler source modifications necessary in a retargeting effort.

P4 Translator

The P4 translator accepts the stack machine object code produced by the P4 front end and translates it into a prefix code. Its output is a series of tokens each having a fixed number of operands that are also tokens. Some of the tokens have associated attributes. For this project, the translator output is the intermediate language from which code will be generated.

Not all of the PASCAL features implemented by P4 are supported by the translator at this point. Most notably missing are strings, sets, and floating point numbers. Case statements present a special problem for the translator. The translator assumes that the stack will be empty at the end of a PASCAL statement. However, for a case statement, the selector expression remains on the stack across statement boundaries. This requires special handling and therefore case statements have not yet been implemented.

PO

PO [3] is the retargeting tool used in this project. It is a retargetable code generator/peephole optimizer. Its original application was with a compiler for the Y programming language [5, 8]. PO is driven by a description of the target machine providing the assembler syntax and effects of each machine instruction to be generated. Its software consists of six separate programs that run under UNIX†. Two (the machine description processors) are for self-generation and the remaining four correspond to phases of the code generator/optimizer. Figure 2.3 diagrams the components of PO responsible for code generation and optimization. The following subsections briefly describe the components of PO. Davidson [3] explains them in more detail.

Machine Description Processors

The machine description is a grammar for translation between instructions described in a register transfer notation (used by most stages of PO) and the target machine's assembly language. A target machine is described by first specifying the machine's addressing modes and then its instructions. Machine instructions are described by register transfer lists similar to ISP [1] notation.

Two separate SNOBOL4 [7] programs process the machine description. One creates a recognizer for register transfers and the other produces a transducer used to translate register transfer lists into assembly language instructions. Both SNOBOL4 programs transform the machine description into input grammars for Lex [13], a lexical analyzer generator. Lex in turn generates C [12] routines implementing the recognizer and transducer. The recognizer is used in the peephole optimization phase of PO, and the transducer is part of the final phase that generates the assembly language instructions.

† UNIX is a trademark of AT&T Bell Laboratories

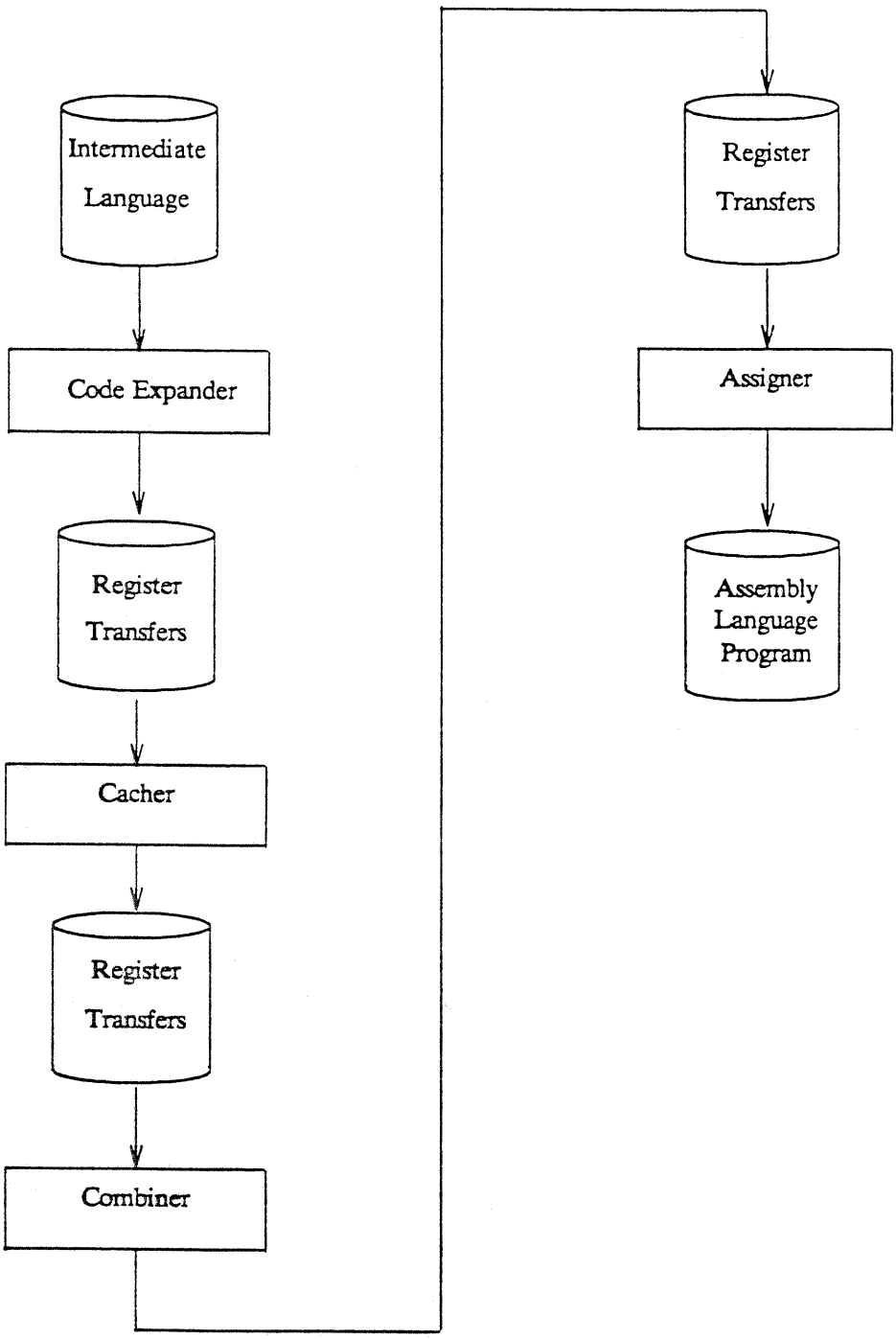

Figure 2.3. PO Organization

Code Expander

The code expander is the first phase of PO. It translates each intermediate language instruction into target machine instructions. The structure of the code expander is dependent on the intermediate representation of the language being processed. It should not vary when retargeting the same compiler for differing processors. A new intermediate language, however, may completely change the structure of this program.

The code expander is also dependent on the target machine, since its output is a sequence of target machine instructions. The machine instructions are expressed as register transfer lists rather than assembly language instructions. Retargeting the code expander involves rewriting the machine-specific register transfer sequence for each intermediate language instruction according to the rules and guidelines outlined in reference [3]. This is intended to be a straightforward task because there is no need to consider register allocation or code optimization, which are managed by a separate phase of PO. The implementor may assume an unlimited supply of registers, leaving the machine register allocation task to a later phase of PO.

If the prescribed guidelines are followed, the code is almost guaranteed to be non-optimal. The guidelines recommend that all values be loaded into registers and that intermediate results of computations be preserved in registers. This allows later phases of the optimizer to determine which values should be retained in registers for future use, and which values can be discarded.

The records output by the expander are input to Cacher, the next phase of PO. They must be produced in the syntactic format expected by Cacher. If necessary, instructions may be expressed in assembly language and tagged so that they are ignored by the remaining phases of PO. These instructions have no chance for optimization.

Cacher

The second phase of PO, Cacher, is responsible for tracking values held in registers. Its job is to eliminate redundant register loads, identify dead variables, and define for each instruction the size of the adjustable peephole for the subsequent peephole optimization phase of PO. It optionally eliminates common subexpressions between label boundaries. Davidson [3] outlines the algorithm used for performing these tasks.

Cacher can be retargeted by rewriting a function that identifies registers and an additional function that chooses the more efficient of two data accesses. (For instance, it may be passed symbolic representations of a register access and a memory access, and would return the register access as the less expensive of the two.) It is important to note that Cacher relies on the compiler front end to pass it information about interference due to aliasing. Without this information, it may make invalid code optimizations.

Combiner

Combiner accepts a sequence of register transfers linked together by the flow analysis performed in Cacher. It is responsible for the actual peephole optimization functions of PO. Combiner attempts to combine logically adjacent register transfer list pairs or triples into single register transfer lists representing valid machine instructions. It removes unreferenced labels and collapses branch chains.

Combiner also has a section of code responsible for simplifying register transfers representing a single instruction. This includes removing additions and subtractions of zero, removing branches to the next location, and simplifying boolean expressions.

The recognizer generated from the target machine description is an integral part of Combiner. Any functions referenced in the machine description must be provided by the implementor in order for Combiner to compile and execute correctly. In addition to providing a machine description, retargeting Combiner requires the recoding of routines to recognize

pseudo-registers and hardware registers represented in the register transfer notation. Also the register transfer simplification routine may require some modifications or additions.

Assigner

The final phase of PO is the register assignment program, Assigner. It is responsible for performing register allocation and generating code to handle register spills. Assigner also translates the optimized code from register transfer notation to assembly code for the target machine. This function is performed using the transducer generated from the machine description.

Retargeting Assigner is accomplished by rewriting part of the register allocation routine. The implementor must supply it with the number and types of target machine registers, and information about which registers may be allocated. This routine also contains machine-dependent templates for storing and loading temporaries.

The output from Assigner is the assembly language representation of the source routine being translated. Its form depends on the syntax required by the assembler for the target machine.

MC68000 Processor

As previously mentioned, the MC68000 microprocessor was the target machine for this project. It is described as having a 16/32 bit architecture [9] because of its 16-bit word size and 32-bit registers. It provides eight data registers used for holding byte (8-bit), word (16-bit) and long word (32-bit) data. Eight address registers, one of which is the user stack pointer, are available for use as base address registers and stack pointers. Address registers may also be used for word and long word operations. Both data and address registers may be used as index registers. Six basic addressing modes are provided. Some of these may be extended with indexing, offsetting, pre-decrementing, or post-incrementing features. Most of

the machine instructions are two-address instructions capable of byte, word and long word operations, and capable of handling most of the available addressing modes. However, some instructions do not allow all operation sizes and many instructions only accommodate a subset of the possible addressing modes. This asymmetry increases the complexity of the architecture.

Motorola 68000 AS Assembler

The Motorola 68000 AS assembler [15] was the interface between the PO output and the processor in this experiment. It is a relatively uncomplicated assembler providing a close correspondence between assembler mnemonics and machine instructions. Most AS mnemonics are suffixed with "b", "w", or "l" to indicate the range of the operand. AS provides extended branch instructions, eliminating the need to consider the length of branches. It also combines certain separate but similar instructions (such as ADD and ADDA) into one mnemonic, determining from the operand types which machine instruction to produce.

AS programs are divided into text, data, and uninitialized data (bss) segments. Directives are provided for switching between segments during assembly. The assembler also handles symbolic expressions.

The output from AS is MC68000 object code. This may be link-edited with other object files to form an executable object file.

CHAPTER III

IMPLEMENTATION EXPERIENCE

The work involved in retargeting the P4 compiler for the MC68000 spanned various tools, programming languages and operating systems. Although most of the work was anticipated in the documentation for each component, some additional work was necessary at certain steps. This chapter describes the actual implementation effort for this project. It provides an account of some of the obstacles encountered and some special considerations that arose from the choice of components. It also provides a subjective indication of the difficulty of each step. This chapter is partitioned into descriptions of the implementation effort for each system component. The order is more or less chronological although the implementation phases were not completely sequential.

P4 Compiler

The P4 compiler work consisted of two tasks. The first was porting the compiler to run under UNIX. This was accomplished by making the necessary modifications for it to compile with the Berkely Pascal Compiler (*pc*) [11]. Approximately 80 lines of code were added or modified. The remainder of this project used the executable object produced by compiling the P4 front end with *pc*.

The second task was to define the compiler constants to reflect the characteristics of the target machine and the desired run-time activation record structure. Some decisions about data representations and the run-time memory layout were made at this point.

The choice was made to implement PASCAL integers as sixteen-bit entities, suggested by the operand sizes of the multiply and divide instructions on the target machine. Thirty-two

bit integers would have been a logical alternative and, in retrospect, may have simplified the target machine description supplied in a subsequent implementation step.

Booleans were implemented with single unaligned bytes (eight bits), using the encoding "0" for **false** and "1" for **true**. These choices were supported by their ease of representation in the machine description and by the P4 representation for booleans. An alternative would have been to use all ones (hexadecimal "ff") to denote **true** as is suggested by the MC68000 "s*cc*" instruction. The latter alternative may have resulted in more optimal code but was more difficult to represent in the machine description.

One incompatibility between the P4 compiler and the MC68000 architecture surfaced when designing the activation record structure. P4 addressing dictates that the run-time stack must grow from low to high memory, whereas the MC68000 provides convenient stacking operations to be used with a hardware stack growing from high to low memory. In fact, this hardware stack must be used in order to utilize the built-in stacking and subroutine linkage instructions ("pea", "jsr", "rts", "link", and "unlnk"). Furthermore, any UNIX system routines called to implement PASCAL standard functions would expect to find their parameters on top of the hardware stack. Since it was not possible to map the P4 stack onto the MC68000 stack, two stacks were used: a software stack growing from low to high memory for frame header information, user data, and temporary storage space; and the built-in hardware stack to save registers, hold subroutine linkage information, and pass parameters to system routines. Figure 3.1 shows the memory layout used in this implementation. The arrows indicate the direction of stack or heap growth.

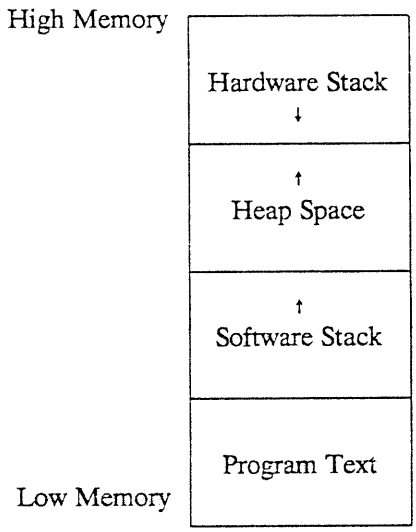

Figure 3.1. Memory Layout

This design allowed appropriate use of the "jsr" and "rts" instructions and a limited use of the "pea" instruction. "Link" and "unlnk", however, could not be used for pointer linkage under this scheme since static and dynamic link pointers are stored in the activation record. Figure 3.2 shows an example of run-time hardware stack contents representing two dynamic nesting levels.

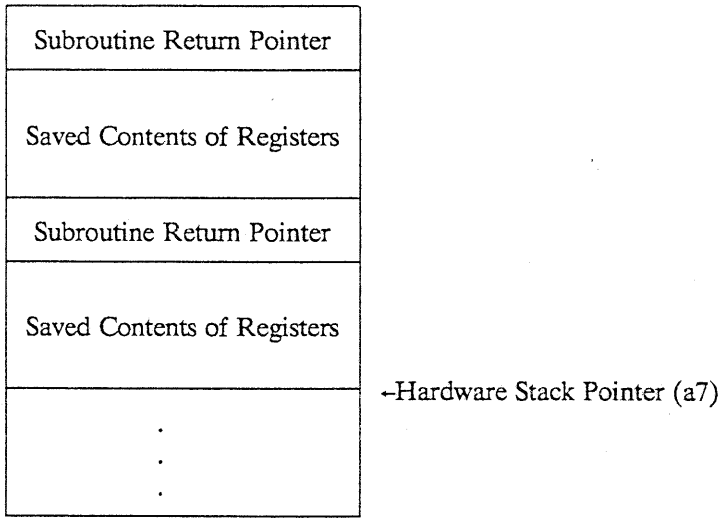

Figure 3.2. Hardware Stack Contents

The hardware stack may also hold parameters for external C subroutines and is used to temporarily save the software stack pointer during the procedure and function call sequence.

The software stack consists of a sequence of activation records, one for the main program and one for each incarnation of a procedure or function. The activation record layout for this implementation contains only a subset of the information suggested in [14]. It is structured as shown if Figure 3.3. The numbers in the figure represent the offset from the start of the record.

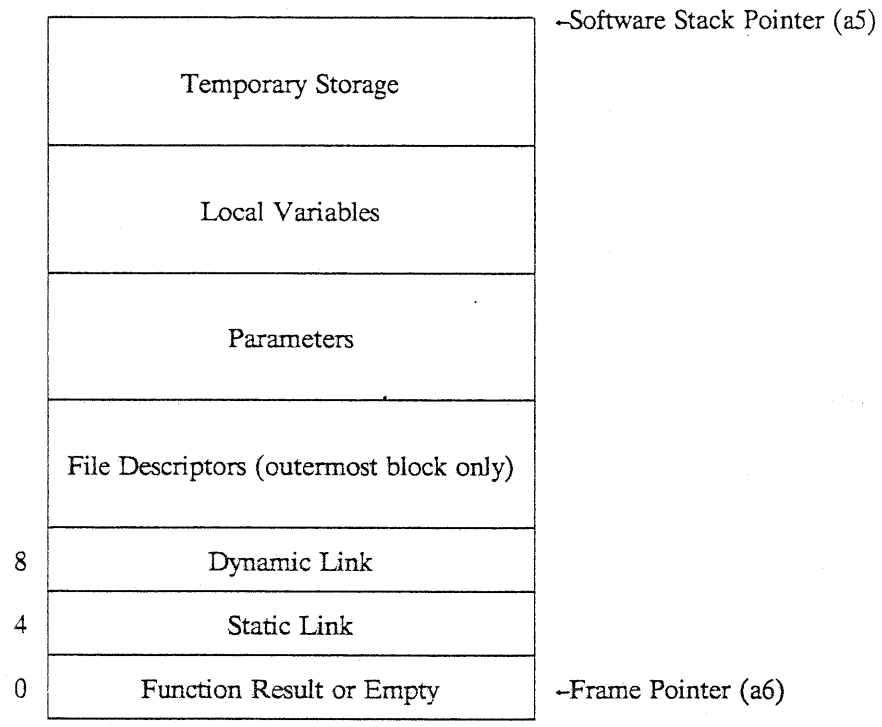

Figure 3.3. Activation Record Layout

For the outermost block, the P4 compiler allocates space for descriptors for the predefined files. The implementor may determine the size and structure of these. For this implementation they were each eight bytes long and structured as shown in Figure 3.4.

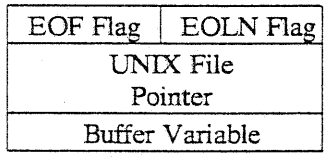

Figure 3.4. File Descriptor Format

The ordering of the activation record information was dictated by the P4 calling sequence. To invoke a procedure or function, P4 emits mark stack information followed by any parameters. Obtaining the parameters may involve calculations or even another function call as in the following example of a call to procedure **p**.

```
p(f(a));
```

In this case the parameter is the value returned by function **f**. Since P4 doesn't produce enough information to determine what size a parameter must be, a long word is reserved on the stack for each parameter.

Following the parameter information the actual branch to the procedure or function is performed. After the return from the procedure or function, the software stack pointer and frame pointer must be restored. An example of MC68000 code for a procedure or function call is:

MC68000 Code		Explanation
pea	a5@	Save software stack pointer on hardware stack. This implicitly decrements the hardware stack pointer (a7).
clrl	a5@+	Reserve space on software stack for function return value. Space is reserved but not used for a procedure call.
movl	a6@(4),a5@+	Push static link on software stack.
movl	a6,a5@+	Push dynamic link on software stack.
	.	
	.	(Push parameters.)
	.	
movl	a7@+,a6	Pop saved software stack pointer from hardware stack; make it the new frame pointer.
jsr	L3	Branch to function saving return address on hardware stack. This implicitly decrements the hardware stack pointer (a7).
movl	a6,a5	(On return from function) restore software stack pointer.
movl	a6@(8),a6	Restore frame pointer.

In this example the called routine is at the same static level as the caller (i.e., it is a sibling).

Corresponding to this calling sequence, the body of the function or procedure must appear as follows.

MC68000 Code		Explanation
tstb	a7@(-40)	Insure hardware stack has enough room.
moveml	#mask1,a7@-	Save registers indicated by "mask1" on hardware stack.
IDstk=4*ID4+ID3		Calculate space needed for local variables (ID3) and temporaries (4*ID4).
addl	IDstk,a5	Add local variable and temporary space to the software stack pointer.
	.	
	.	(Procedure or function body.)
	.	
moveml	a7@+,#mask2	Restore registers indicated by "mask2" from hardware stack.
rts		Return from subroutine through return address on top of hardware stack. Implicitly increments the hardware stack pointer.

The above examples assume that certain address registers are designated for special stack pointers. It was necessary to reserve certain address registers and data registers for special purposes for this implementation while leaving the remaining registers to be allocated by PO. The following shows the use for each address and data register:

Register	Designation
a7	Hardware Stack Pointer
a6	Frame Pointer (Points into Software Stack)
a5	Software Stack Pointer
a4	Allocatable
a3	Allocatable
a2	Allocatable
a1	Reserved Scratch Register
a0	Reserved Scratch Register
d7	Allocatable
d6	Allocatable
d5	Allocatable
d4	Allocatable
d3	Allocatable
d2	Allocatable
d1	Reserved Scratch Register
d0	Reserved Scratch Register

P4 Translator

Like the P4 compiler, the translator also had to be ported to run under UNIX. Only minor changes were made to compile it with *pc*. Since this project was its initial application, some debugging work was also necessary prompting additional code modifications. In all, approximately 40 lines of code were modified.

The original version of the translator made a distinction between two methods of specifying memory locations. One instruction specified the address calculation (static level + offset) while another specified a memory access by the pair [static level, offset]. The distinction is must be made since it is not always necessary to perform an address calculation. Consider the following MC68000 code sequence for storing the integer 4 at offset 46 from the beginning of the current frame. Assume register a6 is the frame pointer.

```
movl a6,a2
addl #46,a2
movw #4,a2@
```

The first two instructions calculate the address while the last instruction stores the integer. This code sequence can be shortened if the code generator "knows" that the address calculation is unnecessary. The following instruction accomplishes the same task:

```
movw #4,a6@(46)
```

By providing separate translator tokens for the two types of memory accesses a conventional code generator could make this optimization. However, to simplify the translator and speed up the debugging process, the capability to specify an access using the level and offset pair was removed. Thus, the code expander for this project always generated instructions that calculated the address in a manner similar to the first code sequence. This ultimately turned out to be completely acceptable since peephole optimization was the final step in code generation. The peephole optimizer was able to remove the inefficiencies and produce code similar to the

second example.

Code Expander

Since PO was to be interfaced to both a new front-end and a different processor, the code expander had to be almost entirely rewritten. This was accomplished in three steps. The first task was to write code to parse the translator output correctly. A recursive descent scheme was used in order to accept the prefix code notation. A few functions left over from previous code expanders were useful for reading input lines.

Secondly, a sequence of MC68000 assembly language instructions was written for each intermediate language token recognized. These were written in the form of comments since the desired output was to be in the form of register transfers. The code sequences were written to be correct, but non-optimal. No attempt was made to be clever in the code expansion process. In fact, the code expansion rules and guidelines [3] prohibited it in most cases. The code was generated from a local viewpoint and no case analysis was performed. Register allocation was not considered since the methodology allows the implementor to assume an unlimited supply of pseudo registers. Only in special cases were specific hardware registers specified by the code expander. Designing code sequences to handle procedure and function calls was the most difficult part of this task.

Although the third step in implementing the code expander was not completed until after the MC68000 machine description was written, it shall nevertheless be mentioned here. It entailed expressing each assembly language instruction as a register transfer list. This process was tedious and error-prone, but was conceptually straightforward.

The code expander is approximately 1000 lines of C code. Comments and code for debugging purposes are not included in this figure.

Run-Time Routines

It was necessary for the code expander to implement the PASCAL standard functions and procedures. Some were handled by in-line assembly code while others were handled most conveniently by separate routines coded in C. These run-time C routines were written in parallel with the code expander in order to coordinate parameter passing. Most of the PASCAL I/O capabilities supported by P4 and the translator were implemented by short C routines calling UNIX I/O functions. It was also convenient to initialize the predefined files and the memory layout with a C program. Although this C program is a part of the run-time library, it calls the assembly program generated by PO as a subroutine.

Machine Description

The target machine description was the most crucial and time-consuming part of this implementation. Although the original version was written in approximately three days, it underwent nearly seventy-five revisions before arriving at the version shown in Appendix A. In the initial version an attempt was made to describe nearly all the possible addressing modes and instructions provided by the MC68000 architecture. However, limitations in the tools used to process the machine description forced a drastic reduction in its size. These limitations will be described later.

Appendix A lists the complete machine description followed by a glossary of symbols used in the description. The machine description is divided into four parts delimited by "%%" symbols. The following sections discuss each of these parts.

Regular Expression Definitions

The first part defines regular expressions to be used in the second part. REGNO and REGN2 state the allowable register numbers. Because an unlimited supply of pseudo-registers is assumed by the code generator, all non-negative integers are represented here. Symbols

QNUM through IDENT define the various sizes of integers and identifiers. Since the MC68000 architecture restricts the size of integers used in some of its addressing modes and operands, it was desirable to recognize certain sizes of integers separately. For this reason, integers were expressed in hexadecimal notation both here and in the code expander. LABEL defines the representation for labels. This does not permit all label representations allowed by the assembler, but only those emitted by the code expander.

Token Definitions

The next set of definitions associates tokens with addressing modes and other patterns to be used in the instruction definition section. These definitions have three fields. The token in the first field is returned if the register transfer pattern in the second field is matched. The assembly language equivalent of the register transfer pattern is given in the third field. If the first field is empty, the token returned is the string matched. An empty third field indicates no equivalent assembly-language syntax.

Symbols AR, DRL, DRW, and DRB define the syntax for registers. It was necessary to keep track of what size data elements were contained in the data registers since they may hold byte, word, or long word operands. Failure to do this would allow PO to make invalid optimizations (e.g., using a register with only 16 bits of valid data in a 32-bit operation). This was accomplished by using differing representations for each data size, hence, the "dl", "dw", and "db" notations. Since all quantities loaded into address registers are sign extended, they always contain 32 bits of valid data following a load. Therefore, only one representation for address registers was necessary.

Groups of patterns may correspond to a single non-terminal in the second section. The MC68000 memory-alterable addressing modes were grouped into three classes representing the three sizes of memory accesses (MLTL, MALTW, MALTB). The pre-increment memory addressing mode using the stack pointer (register a7) was an exception, however. Although it

belongs in the group of long-word memory-alterable addressing modes, it had to be recognized as a special case in some of the instruction definitions in the last section of the machine description. Because of this, it had to be defined separately (PUSH). In general, any patterns to be recognized separately must correspond to a separate token in this section.

It is interesting to note that the addressing mode specifications in the second part go beyond the effective address calculations to partially specify the semantics of the instructions that use them. For example, instructions specifying memory alterable addressing mode operands always fetch from the effective address of the operand. Thus, the fetch is included in the addressing mode specification rather than the instruction description. On the other hand, instructions using control addressing mode operands operate on the effective address itself rather than its contents. Although the assembler syntax of these operands is the same as that for memory addressing mode operands, the patterns defining their meanings differ. (See definition of CTL.) Describing the addressing modes in this manner allows the instruction definitions in the last part of the machine description to be more concise.

The pre-decrement and post-increment addressing modes presented a special problem. Their descriptions for the MC68000 are similar to those for the PDP-11 shown in [3]. For example, the long-word post-increment addressing mode is represented by the following pattern:

ml[a[REGNO]++]

The "ml" indicates a long-word memory access. Memory is addressed by the address register "a[REGNO]" and the address register is incremented after its use. However, the "++" notation is in no way special to PO. Therefore this description disguises the fact that the value of the address register is changed. If not used carefully in the code expander, this can cause incorrect optimizations to be made. For example the code sequence:

```
movl #1,a2
movl a2@+,a3
movl a2@+,a4
movl a4,a2@
```

was optimized into the following:

```
movl #1,a2
movl a2@+,a2@
```

The load of a3 was eliminated because a3 had no subsequent reference. Since a4 only held an intermediate result, PO found a way to eliminate its use also. Both of these optimizations are invalid, however, and occurred because the side effect of the post increment addressing mode (that of incrementing the address register after its use) was not properly defined to PO. In order to use these addressing mode correctly, they should not be defined in this section. Instead, instructions making use of these modes should be defined by two or more register transfers: one or more describing the effect of the instruction and one identifying the side effect of the pre-decrement or post-increment addressing mode. However, this would increase the size of the machine description beyond the size limit. Since these addressing modes were only needed for special cases in this application (e.g. for pushing and popping stack elements), the decision was made to retain their definitions in the second section of the machine description and restrict their usage by the code expander to cases where their chance for invalid optimizations could be controlled.

Most of the program-counter-relative addressing modes were not included in the machine description. The code expander never emits program-counter-relative operands; therefore they need not be defined to PO.

Token Groupings

The third part of the machine description associates groups of previously-defined tokens with new tokens. Tokens that comprise the groups may still be recognized in their own

right in the subsequent instruction definition section. This capability was used heavily in the MC68000 definition since so many instructions use various subsets of the possible addressing modes.

Instruction Definitions

The final part of the machine description defines all the instructions to be produced by the code generation and optimization process. Because of a limit on the size of the machine description, an effort was made to recognize and eliminate instructions that had little possibility of being generated. Minimally, the machine description must include any addressing mode and instruction produced by the code expander. Beyond this, it should have knowledge of more efficient replacements for single instructions and combinations of instructions. For example if, in loading a register with an integer value, the code expander generates the register transfer equivalent for the instruction:

```
movw #0,d3
```

the machine description should have enough information to replace it with the faster instruction:

```
clrw d3
```

Ordering is important in this section; less expensive instructions must be described before their more general counterparts. For example, definitions for the MC68000 "addq" were listed before the more general "addl" and "addw" instruction definitions. The "addb" instruction was not defined at all since bytes are never added in this implementation.

It is interesting to note that, using the machine description shown in Appendix A, the "addql" instruction can never be generated for address registers. Suppose that the following register transfer is emitted by the code expander to add 5 to an address pseudo-register:

a[37] = a[37] + 0x00000005;

Suppose also that the condition code (NZ) is modified by a subsequent instruction before it is used. A possible match for this in the machine description is the pattern for the "addql" instruction which is:

ALTL = ALTL + QD;NZ = ALTL + QD ? 0

Register a[37] is an address pseudo-register (AR in the machine description) and is part of the group of tokens represented by ALTL. The string "0x00000005" matches the token QD representing the range of integers allowed in the immediate data field of "quick" instructions. Since the condition will be modified again before any possible future use, the register transfer:

NZ = ALTL + QD ? 0

is a harmless side effect. (Recall, the Cacher phase of PO tracks dead variables.) Clearly the pattern for "addql" is consistent with the register transfer. However, the source of the register transfer,

a[37] + 0x00000005

matches the following pattern for the control (CTL) addressing mode:

a[REGNO] + WDENT

At this point a[37] corresponds to a[REGNO] and "0x00000005" matches the regular expression for WDENT. Therefore, the pattern to be matched against in the instruction definition section is:

AR = CTL

This matches the "lea" instruction pattern in the instruction definitions and, after the register mapping is performed, the resulting assembly instruction would be

lea a2(0x00000005),a2

(This assumes pseudo-register a[37] is mapped to hardware register a2.) For the MC68000 processor, "lea" has the same execution time as "addql"; therefore, no effort was made to accommodate the generation of "addql" for address registers.

Some instructions were not described in general terms, but rather for specific cases. An example of this is the MC68000 "btst" instruction. In its general form, it sets the condition code register according to the value of a specified bit of the destination operand. Since "btst" is produced by the code expander in only one instance (for implementing the PASCAL ODD function), and that instance tests the least significant bit of its operand, only a specialized definition of "btst" was included the machine description. The instruction was described to test bit zero of the word-length data register and set the condition code accordingly. Furthermore, the actual "btst" instruction operates on a 32-bit data register and the machine description specifies a 16-bit register. Obviously, if only the least significant bit is important, it doesn't matter if the upper bits do not contain valid data. It was convenient to represent the "btst" instruction in this way since the PASCAL ODD function operates on integers which, as previously mentioned are 16 bits in length. An alternative would have been to sign extend the integer in a register and then generate "btst" with a long-word register operand. This choice would be less efficient and PO would not have been able to optimize it into a shorter sequence. Clearly, this was a case of using knowledge about the code expander to influence the machine description. Many other instructions were described (or omitted) based on knowledge of the code expander.

Test conditions were used in describing some of the instructions. Two unexpected problems were encountered when the test conditions were processed by the machine description processor. One problem occurred when attempting to describe the division operation as follows:

```
DRW = DRL / DATW;NZ = DRL / DATW ? 0; := {
        !equivr(DRL,DRW): ABORT
        divs DATW,DRL
        }
```

The first line consists of the register transfers describing the effects of the instruction. It indicates that a data register containing 32 bits of valid data (DRL) is to be divided by a word of data (DATW). DATW represents any of the word-length data addressing modes. The 16-bit result of the division is to be placed in a data register (DRW) and the condition code (NZ) is to be set accordingly. The second line insures that the word register and the long word register are the same physical register, and the third line specifies that the "divs" instruction should be emitted. However, in the C code generated from this description, the two register transfers were split and the test on the second line was applied to them separately. When the test:

```
!equivr(DRL,DRW)
```

was applied to the register transfer:

```
NZ = DRL / DATW ? 0
```

the DRW parameter was undefined. Because of this, the C program would not even compile. An awkward but harmless way around this problem was to describe the second register transfer as:

```
NZ = DRL $ DRW / DATW ? 0
```

where "$" is the symbol for the inclusive OR operation.

A second problem occurred when attempting to define the multiplication operation in a manner similar to the definition in the PDP-11 description in reference [3]. The MC68000 definition was as follows:

```
DRW = DRW * DATW;NZ = DRW * DATW ? 0; := {
        pwr2(DATW): aslw DATW,DRW
        muls DATW,DRW
        }
```

The "aslw" instruction is emitted if "pwr2" is true. The function "pwr2" returns **true** if its argument is determined to be a power of two and, as a side effect, converts its argument to the $\log_2$ of itself. If the argument is not a power of two, **false** is returned and the argument is unchanged. If **false** is returned "muls" is generated. The intent is that if the argument is a power of two, its $\log_2$ will be supplied as the operand to the shift ("aslw") instruction. However, the modified argument (DATW in this case) did not find its way into the "aslw" instruction operand field as advertised. Instead the original value of DATW was inserted. Because of this bug in PO, the possibility of optimizing a multiply instruction into a less expensive shift instruction was removed.

Describing instructions that transformed the data in a register to a different length proved to be difficult. In order to do this, it is necessary to describe one register in two different ways. For instance the sign extent instruction ("extl") extends 16-bit data to 32-bit data. The description for this instruction is

```
DRL = DRW; := {
        !equivr(DRL,DRW): ABORT
        extl DRL
        }
```

The "equivr" function insures that DRL and DRW actually represent the same register. The code expander may generate an instance of the "extl" instruction using register transfers and pseudo-registers as in the following example:

```
dl[57] = dw[57];
```

At this point it is necessary to mark dw[57] as a dead variable since any future changes to the contents of dl[57] would invalidate dw[57] and vice versa. However, in the final stages of PO,

dw[57] is mapped to a hardware register, say d2. If dw[57] is on the dead variable list, then d2 is considered dead after the mapping, and may be reused for something else. If dl[57] is also mapped to d2, its contents could be destroyed before their use (because d2 is considered to be dead and therefore allocatable). Furthermore there is no guarantee that dl[57] will be mapped to d2. The only viable solution to this problem was to use a non-allocatable hardware register for such instructions. Register d0 was reserved for such operations. This added inefficiencies to the code since it required d0 to be loaded with the value to be sign extended.

Machine Description Processors

Since a version of SNOBOL4 was not available on the local UNIX system, the machine description processing programs were transported to another operating system and processed with a SPITBOL [6] interpreter. Some modifications (24 lines of code) were made to the SNOBOL4 programs in order for the SPITBOL interpreter to accept them.

As previously mentioned, the size of the machine description was limited by the machine description processing software. Both the recognizer and the transducer are comprised of three levels of Lex-generated programs. These programs read single-character tokens and return single-character tokens as function results. The SNOBOL4 programs that generate the Lex grammars assign a unique character to each token in the machine description and to each register transfer. Where two or more register transfers are grouped together to describe an instruction, unique character tokens are assigned to each one individually, and then to their combinations.

The first version of the MC68000 machine description rendered an "out of tokens" message from each SNOBOL4 program before even half of the machine description was processed. The programs were modified slightly so they would use nearly all the ASCII characters as tokens; however, the machine description size still had to be reduced drastically. Between the two SNOBOL4 programs, approximately 100 lines of code were modified.

Further problems occurred when Lex complained about "too many right contexts" and "too many positions for one state", again signifying too large a machine description. A private version of Lex was created to handle these problems.

Peephole Optimizer C Code

Aside from rewriting the code expander, only a small amount of recoding was necessary in order to retarget PO. Most of the required modifications were documented in reference [3], however, some additional changes were necessary due to the choices made for register and integer representations.

Machine-dependent PO source code was located in separate directories corresponding to the specific target processors. The machine-dependent code for the PDP-11 processor was used as a starting point for this project. The following sections outline the code modifications made to the Cacher, Combiner, and Assigner components of PO.

Cacher

Modifications to Cacher involved only 12 lines of code. It was changed to recognize the MC68000 register definitions. A function to choose the more efficient of two data accesses was also modified.

Combiner

Approximately 70 lines of code were written or modified in order to retarget Combiner. (This does not include the machine description.) Again, code to recognize registers had to be rewritten as well as code to recognize integers. Also, functions referenced in the machine description had to be supplied if they were not standard UNIX functions.

Assigner

Assigner made use of some of the functions already retargeted for other PO programs.

In addition, 90 lines of code had to be written or modified, most dealing with types and numbers of hardware registers. It was necessary to specify how many data and address registers were available on the MC68000 and which ones could be used for assignment. Patterns for loading and storing temporaries also had to be supplied. Most of the necessary modifications were straightforward and obvious.

CHAPTER IV

RESULTS AND CONCLUSIONS

The previous chapters described the use of a retargetable peephole optimizer to achieve compiler portability. The following sections evaluate this compiler retargeting method based on experience gained from this project. Davidson [3] suggests four measures of the effectiveness of a retargeting technique: the machine applicability, the quality of the generated code, the compiler speed, and the retargeting effort. Each of these areas shall be addressed; however, the topic of applicability shall also be extended to the compiler front-end.

Applicability

PO was adapted to generate code for the MC68000 with relative ease. The machine description format was uncumbersome and fairly flexible although most exotic instructions were not described (e.g. "link", "unlnk", "db*cc*"). Peculiarities of the MC68000 instruction set and addressing modes were, for the most part, possible to represent in a concise manner. One exception was the problem of register overlap (e.g. as in the sign extend instruction) discussed in Chapter III. The solution to this problem was unsatisfactory in that no effective means was found for specifying multiple representations for the same hardware register.

Probably the biggest limitation relating to target machine applicability was the machine description size problem. Additional work on the machine description processing tools could most likely remove this limitation.

Interfacing PO to the compiler front end was also straightforward. Although the code expander had to be rewritten to recognize a new intermediate language, the process for generating code was well-defined and did not rely on the ingenuity of the implementor.

Only in one respect were P4 and PO not well-suited for each other. PO's Cacher program relies on the compiler front end to pass along aliasing information so that it can eliminate common sub-expressions correctly. Without this information, it cannot properly track memory locations and incorrect code may result. Unfortunately, the P4 compiler does not pass along aliasing information; thus, the optional common subexpression removal capability could not be used. This made for less efficient code as is shown by the following sequences generated for the PASCAL statement "a[i] := a[i] + 1":

Without Common Subexpression Removal		With Common Subexpression Removal	
movw	_ustack + 0x00000036,d2	movl	#_ustack,a2
subqw	#0x00000001,d2	movw	a2@(0x00000036),d2
muls	#0x00000002,d2	subqw	#0x00000001,d2
movl	#_ustack + 0x0000002c,a2	muls	#0x00000002,d2
movw	_ustack + 0x00000036,d3	movl	#_ustack + 0x0000002c,a2
subqw	#0x00000001,d3	lea	a2@(0,d2:w),a2
muls	#0x00000002,d3	addqw	#0x00000001,a2@
movl	#_ustack + 0x0000002c,a3		
movw	a3@(0,d3:w),d4		
addqw	#0x00000001,d4		
movw	d4,a2@(0,d2:w)		

In most cases, the difference in code efficiency was not as extreme as in this example. In fact, holding common subexpressions in registers often caused register spills when the final register mapping was performed. Common subexpression removal generally decreased the size of the code by about five percent. However, since it introduced errors, this option was not used.

Code Quality

The resulting code was evaluated by inspection and execution only. Other MC68000 PASCAL implementations utilized different memory layouts and type representations, making direct comparisons impractical. Several programs in Jensen and Wirth's PASCAL manual [10] were compiled and executed on the MC68000. These include the "recursivegcd", "traversal", "minmax3", "complex", and "matrixmul" programs, some of which were slightly modified to

avoid PASCAL features not supported by the front-end. (Appendix B gives an example of the assembly code generated for a shortened version of the "recursivegcd" program found in reference [10].) The output resulting from their execution was correct.

In general the quality of the generated code was good. Examination of the code indicated some places where the code expander and the machine descriptions could be tuned to increase code efficiency. Most of the remaining inefficiencies stemmed from four factors. The first was the P4 procedure and function calling sequence. The P4 compiler generates markstack information prior to the parameter list and the procedure or function call. This causes activation frame initialization code to be generated for each call rather than each procedure or function entry.

The second cause of inefficient code was the address mapping dictated by P4 which prohibits the use of the hardware stack and its associated instructions. Because of this, code associated with parameter passing and procedure and function calls could not use some of the specially-designed stacking and linkage instructions provided by the MC68000 architecture.

A third cause of non-optimal code was the use of hardware registers rather than pseudo-registers in the code emitted by the code expander. Instructions containing hardware registers are not subject to all the possible optimizations. Some optimizations of instructions containing hardware registers were not performed due to bugs in PO. One example of non-optimal code produced by PO is the following sequence:

```
moveq   #0x0000001b,d0
movl    d0,a7@-
moveq   #0x0000000a,d0
        .
        .
        .
```

The first two instructions should be replaced by

```
movl    #0x0000001b,a7@-
```

PO does not recognize, however, that the load of d0 is unnecessary because it sees another occurrence of d0 in the third instruction. It does not realize that the third instruction is a load of d0 rather than a use. Therefore it does not optimize out the load of d0 in the first instruction. This bug seems to occur only with hardware registers since they are reused throughout the program. Unfortunately it was necessary to use hardware registers for such instructions as "ext", and "divs" which deal with different data lengths in the same register. (Refer to Chapter III for a discussion of this problem.)

Finally, non-optimal code also resulted from the decision to encode the boolean value **true** as a "1". The present machine description does not contain enough information to optimize such sequences as

```
cmpw    #0x00000014,d2
slt     d2
negb    d2
jeq     L7
```

into

```
cmpw    #0x00000014,d2
jge     L7
```

The alternate representation of **true** by a hexadecimal "ff" would have eliminated the need for the "negb" instruction and would possibly have allowed the more complete optimization to be performed. However, as mentioned in Chapter III, this representation would have complicated the machine description. Further modifications to the machine description or additions to the routine that simplifies register transfer lists in Combiner may solve this problem.

Compiler Speed

The speed of the compiler resulting from this project was, at best, marginal. It pro-

cessed approximately seven MC68000 instructions per second. For comparison Berkely PASCAL was found to process approximately 60 VAX instructions per second. Both compilers ran on a VAX under UNIX although they did not generate code for the same processor.

PO consumed 87% of the execution time while the P4 compiler and translator only accounted for 13% of the time. The Combiner program consumed 34% of the time, Cacher used 33% of the time and Assigner accounted for 14% of the time. The code expander was responsible for 6% of the time. The Combiner and Assigner programs were both driven by the machine description which was quite large in this implementation. Further research has been done by Davidson and Fraser [4] in an attempt to move some of the work of Combiner and Assigner from compile time to compiler-generation time. Perhaps this scheme would help bring the compiler speed to an acceptable level by eliminating the three levels of Lex programs now included in Combiner and Assigner.

Retargeting Effort

Appendix C shows time estimates for implementing each component of the system based on records kept during the experiment. It is also broken down by learning, porting, coding, and debugging activities. The entire project took approximately 100 full days to complete. The learning curve was roughly 40 days. This included the time taken by false starts and testing the capabilities of the various components. The porting and coding work took 23 days, and debugging activities accounted for 35 days.

The effort to retarget P4 for a new but similar processor would require approximately 65 days assuming no previous experience with the system components. Using the experience gained in this project, it is estimated that a similar retargeting effort using the same front-end would take 30 days including debugging time. With a new front-end, the implementation time estimate is 45 days.

Conclusion

Aside from its speed, PO proved to be an acceptable compiler retargeting tool. Ideally, the user of a tool need only supply it with the necessary information at pre-designated points of interface. With the exception of the machine-description processor problems, PO came close to this ideal. It was flexible in its adaptability both to the compiler front-end and the target machine interface.

The most impressive result of this experiment was the small amount of time required to build a generator of fairly good-quality code. This can be attributed to the removal the register allocation and code optimization responsibilities from the implementor.

The overall implementation experience using PO was positive. This, combined with the quality of the resulting compiler support the conclusion that using PO as a retargeting tool is a viable means of achieving compiler portability.

BIBLIOGRAPHY

1. C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, NY, 1971.

2. J. W. Davidson and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, *ACM Trans. Prog. Lang. and Systems 2*, 2 (Apr. 1980), 191-202.

3. J. W. Davidson, *Simplifying Code Generation Through Peephole Optimization*, Ph.D. Dissertation, Department of Computer Science, University of Arizona, Tucson, Arizona, Dec. 1981.

4. J. W. Davidson and C. W. Fraser, Automatic Generation of Peephole Optimizations, *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction 19*, 6 (June 1984), 111-121.

5. J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *ACM Trans. Prog. Lang. and Systems 6*, 4 (Oct. 1984), 515-526.

6. R. B. K. Dewar and A. P. McCann, MACRO SPITBOL -- A SNOBOL4 Compiler, *Software—Practice & Experience 7*, 1 (Jan. 1977), 95-113.

7. R. E. Griswold, J. J. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice Hall, Englewood Cliffs, NJ, Second Edition 1971.

8. D. R. Hanson, The Y Programming Language, *SIGPLAN Notices 16*, 2 (Feb. 1981), 59-68.

9. Motorola, Inc., *M68000 16/32-Bit Microprocessor Programmer's Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, Fourth Edition 1984.

10. K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, Second Edition -- 1974.

11. W. N. Joy, S. L. Graham and C. B. Haley, *Berkely Pascal User's Manual*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California, Version 1.1 - Apr. 1979.

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

13. M. E. Lesk, Lex -- A Lexical Analyzer Generator, Computer Science Technical Report 39, AT&T Bell Laboratories, Jan. 1979.

14. K. V. Nori, U. Ammann, K. Jensen, H. H. Nägeli and C. Jacobi, *The Pascal (P) Compiler: Implementation Notes*, Institüts fur Informatik, Eidgenössische Technische Hochschule, Zürich, Revised Edition 1976.

15. *AS Assembler Reference Guide*, UNISOFT Corporation, Mar. 27, 1982.

APPENDIX A

MC68000 MACHINE DESCRIPTION

```
REGNO     [0-9]+
REGN2     [0-9]+
QNUM      [1-8]
HXNUM     [0-9a-f]
LONUM     [0-7]
HINUM     [8-9a-f]
QDENT     "0x0000000"{QNUM}
BDENT     (("0xffffff"{HINUM}{HXNUM})|("0x000000"{LONUM}{HXNUM})|0|1)
WDENT     (("0xffff"{HINUM}({HXNUM}{3,3}))|("0x0000"{LONUM}({HXNUM}{3,3}))|0|1)
XDENT     ((("_"|"I")[A-Za-z0-9_]+)|("0x"({HXNUM}{8,8}))|0|1)
IDENT     {XDENT}(" "[-+]" "{XDENT})*
LABEL     (("_P"|"L"|"_m"|"_f")[A-Za-z0-9_]+)
%%
AR   :=   a[REGNO]                          :=   aREGNO

DRL  :=   dl[REGNO]                         :=   dREGNO

DRW  :=   dw[REGNO]                         :=   dREGNO

DRB  :=   db[REGNO]                         :=   dREGNO

     :=   0                                 :=   #0

     :=   1                                 :=   #1

LB   :=   LABEL                             :=   LABEL

NZ   :=   NZ

PC   :=   PC                                :=   pc@(0)

SWAP :=   SWAP

QDT  :=   QDENT                             :=   #QDENT

BDT  :=   BDENT                             :=   #BDENT

WDT  :=   WDENT                             :=   #WDENT

IDT  :=   IDENT                             :=   #IDENT

PUSH :=   ml[a[7]--]                        :=   a7@-
```

```
MLTL :=   ml[a[REGNO]]                          :=  aREGNO@
          ml[a[REGNO] + WDENT]                  :=  aREGNO@(WDENT)
          ml[a[REGNO] + a[REGN2]]               :=  aREGNO@(0,aREGN2:1)
          ml[a[REGNO] + a[REGN2] + BDENT]       :=  aREGNO@(BDENT,aREGN2:1)
          ml[a[REGNO] + dl[REGN2]]              :=  aREGNO@(0,dREGN2:1)
          ml[a[REGNO] + dl[REGN2] + BDENT]      :=  aREGNO@(BDENT,dREGN2:1)
          ml[a[REGNO] + dw[REGN2]]              :=  aREGNO@(0,dREGN2:w)
          ml[a[REGNO] + dw[REGN2] + BDENT]      :=  aREGNO@(BDENT,dREGN2:w)
          ml[a[REGNO]++]                        :=  aREGNO@+
          ml[a[REGNO]--]                        :=  aREGNO@-
          ml[IDENT]                             :=  IDENT

MALTW:=   mw[a[REGNO]]                          :=  aREGNO@
          mw[a[REGNO] + WDENT]                  :=  aREGNO@(WDENT)
          mw[a[REGNO] + a[REGN2]]               :=  aREGNO@(0,aREGN2:1)
          mw[a[REGNO] + a[REGN2] + BDENT]       :=  aREGNO@(BDENT,aREGN2:1)
          mw[a[REGNO] + dl[REGN2]]              :=  aREGNO@(0,dREGN2:1)
          mw[a[REGNO] + dl[REGN2] + BDENT]      :=  aREGNO@(BDENT,dREGN2:1)
          mw[a[REGNO] + dw[REGN2]]              :=  aREGNO@(0,dREGN2:w)
          mw[a[REGNO] + dw[REGN2] + BDENT]      :=  aREGNO@(BDENT,dREGN2:w)
          mw[a[REGNO]++]                        :=  aREGNO@+
          mw[a[REGNO]--]                        :=  aREGNO@-
          mw[IDENT]                             :=  IDENT

MALTB:=   mb[a[REGNO]]                          :=  aREGNO@
          mb[a[REGNO] + WDENT]                  :=  aREGNO@(WDENT)
          mb[a[REGNO] + a[REGN2]]               :=  aREGNO@(0,aREGNO:1)
          mb[a[REGNO] + a[REGN2] + BDENT]       :=  aREGNO@(BDENT,aREGNO:1)
          mb[a[REGNO] + dl[REGN2]]              :=  aREGNO@(0,dREGN2:1)
          mb[a[REGNO] + dl[REGN2] + BDENT]      :=  aREGNO@(BDENT,dREGN2:1)
          mb[a[REGNO] + dw[REGN2]]              :=  aREGNO@(0,dREGN2:w)
          mb[a[REGNO] + dw[REGN2] + BDENT]      :=  aREGNO@(BDENT,dREGN2:w)
          mb[a[REGNO]++]                        :=  aREGNO@+
          mb[a[REGNO]--]                        :=  aREGNO@-
          mb[IDENT]                             :=  IDENT

CTL  :=   a[REGNO] + WDENT                      :=  aREGNO@(WDENT)
          a[REGNO] + a[REGN2]                   :=  aREGNO@(0,aREGN2:1)
          a[REGNO] + a[REGN2] + BDENT           :=  aREGNO@(BDENT,aREGN2:1)
          a[REGNO] + dl[REGN2]                  :=  aREGNO@(0,dREGN2:1)
          a[REGNO] + dl[REGN2] + BDENT          :=  aREGNO@(BDENT,dREGN2:1)
          a[REGNO] + dw[REGN2]                  :=  aREGNO@(0,dREGN2:w)
          a[REGNO] + dw[REGN2] + BDENT          :=  aREGNO@(BDENT,dREGN2:w)

REL  :=   ==                                    :=  eq
          !=                                    :=  ne
          >=                                    :=  ge
          <=                                    :=  le
          <                                     :=  lt
          >                                     :=  gt

IMP  :=   ->
%%
```

```
QD          :=   0|1|QDT
BD          :=   0|1|QDT|BDT
WD          :=   0|1|QDT|BDT|WDT
ID          :=   0|1|QDT|BDT|WDT|IDT
MALTL       :=   MLTL|PUSH
ALLL        :=   0|1|QDT|BDT|WDT|IDT|AR|DRL|MLTL|PUSH|PC
ALLW        :=   0|1|QDT|BDT|WDT|AR|DRW|MALTW
DATW        :=   0|1|QDT|BDT|WDT|DRW|MALTW
DATB        :=   0|1|QDT|BDT|DRB|MALTB
ALTL        :=   AR|DRL|MLTL|PUSH
ALTW        :=   AR|DRW|MALTW
ALTB        :=   AR|DRB|MALTB
DALTL       :=   DRL|MLTL|PUSH
DALTW       :=   DRW|MALTW
DALTB       :=   DRB|MALTB
%%
NZ = DRW & 1 ? 0;                                   :=   btst #0,DRW
DALTL = 0;NZ = 0 ? 0;                               :=   clrl DALTL
DALTW = 0;NZ = 0 ? 0;                               :=   clrw DALTW
DALTB = 0;NZ = 0 ? 0;                               :=   clrb DALTB
NZ = DALTL ? 0;                                     :=   tstl DALTL
NZ = DALTW ? 0;                                     :=   tstw DALTW
NZ = DALTB ? 0;                                     :=   tstb DALTB
NZ = DRL ? ALLL;                                    :=   cmpl ALLL,DRL
NZ = DRW ? ALLW;                                    :=   cmpw ALLW,DRW
NZ = DRB ? DATB;                                    :=   cmpb DATB,DRB
NZ = AR ? ALLL;                                     :=   cmpl ALLL,AR
NZ = AR ? ALLW;                                     :=   cmpw ALLW,AR
ALTL = ALTL + QD;NZ = ALTL + QD ? 0;                :=   addql QD,ALTL
ALTW = ALTW + QD;NZ = ALTW + QD ? 0;                :=   addqw QD,ALTW
AR = CTL;                                           :=   lea CTL,AR
DRL = DRL + ALLL;NZ = DRL + ALLL ? 0;               :=   addl ALLL,DRL
MALTL = MALTL + DRL;NZ = MALTL + DRL ? 0;           :=   addl DRL,MALTL
DRW = DRW + ALLW;NZ = DRW + ALLW ? 0;               :=   addw ALLW,DRW
MALTW = MALTW + DRW;NZ = MALTW + DRW ? 0;           :=   addw DRW,MALTW
AR = AR + ALLL;                                     :=   addl ALLL,AR
AR = AR + ALLW;                                     :=   addw ALLW,AR
DRB = DRB & DATB;NZ = DRB & DATB ? 0;               :=   andb DATB,DRB
DALTW = DALTW & ID;NZ = DALTW & ID ? 0;             :=   andw ID,DALTW
DALTB = DALTB & BD;NZ = DALTB & BD ? 0;             :=   andb BD,DALTB
MALTB = MALTB & DRB;NZ = MALTB & DRB ? 0;           :=   andb DRB,MALTB
MALTW = MALTW / QD;NZ = MALTW / QD;                 :=   {
     strcmp(QD, "0x00000002"): ABORT
     asrw MALTW
     }
DRW = DRL / DATW;NZ = DRL $ DRW / DATW ? 0;         :=   {
     !equivr(DRL,DRW): ABORT
     divs DATW,DRL
     }
DRW = DRL % DATW;NZ = DRL $ DRW % DATW ? 0;         :=   {
     !equivr(DRL,DRW): ABORT
     divs DATW,DRL
     }
```

```
DRL = DRW;                                          :=   {
      !equivr(DRL,DRW): ABORT
      extl DRL
      }
DRW = DRB;                                          :=   {
      !equivr(DRW,DRB): ABORT
      extw DRW
      }
PC = NZ REL O IMP LB | PC;                          :=   jREL LB
PC = LB;                                            :=   jra LB
PUSH = PC;PC = LB;                                  :=   jsr LB
PC = CTL;                                           :=   jra CTL
DRL = BD;NZ = BD ? O;                               :=   moveq BD,DRL
DRW = BD;NZ = BD ? O;                               :=   moveq BD,DRW
DRB = BD;NZ = BD ? O;                               :=   moveq BD,DRB
PUSH = AR;                                          :=   pea AR@
DALTL = ALLL;NZ = ALLL ? O;                         :=   movl ALLL,DALTL
DALTW = ALLW;NZ = ALLW ? O;                         :=   movw ALLW,DALTW
DALTB = WD;                                         :=   {
      strcmp(WD, "0x000000ff"): ABORT
      st DALTB
      }
DALTB = DATB;NZ = DATB ? O;                         :=   movb DATB,DALTB
AR = ALLL;                                          :=   movl ALLL,AR
AR = ALLW;                                          :=   movw ALLW,AR
MALTW = MALTW * QD;NZ = MALTW * QD;                 :=   {
      strcmp(QD, "0x00000002"): ABORT
      aslw MALTW
      }
DRW = DRW * DATW;NZ = DRW * DATW ? O;               :=   muls DATW,DRW
DALTL = -DALTL;NZ = -DALTL ? O;                     :=   negl DALTL
DALTW = -DALTW;NZ = -DALTW ? O;                     :=   negw DALTW
DALTB = -DALTB;NZ = -DALTB ? O;                     :=   negb DALTB
DALTB = ~DALTB;NZ = ~DALTB ? O;                     :=   notb DALTB
DRB = DRB $ DATB;NZ = DRB $ DATB ? O;               :=   orb DATB,DRB
ALTB = ALTB $ DRB;NZ = ALTB $ DRB ? O;              :=   orb DRB,ALTB
PUSH = LB;                                          :=   pea LB
PUSH = CTL;                                         :=   pea CTL
PC = MALTL;                                         :=   {
      strcmp(MALTL, "ml[a[7]++]"): ABORT
      rts
      }
DALTB = NZ REL O IMP WD | O;                        :=   {
      strcmp(WD, "0x000000ff"): ABORT
      sREL DALTB
      }
ALTL = ALTL - QD;NZ = ALTL - QD ? O;                :=   subql QD,ALTL
DRL = DRL - ALLL;NZ = DRL - ALLL ? O;               :=   subl ALLL,DRL
MALTL = MALTL - DRL;NZ = MALTL - DRL ? O;           :=   subl DRL,MALTL
ALTW = ALTW - QD;NZ = ALTW - QD ? O;                :=   subqw QD,ALTW
DRW = DRW - ALLW;NZ = DRW - ALLW ? O;               :=   subw ALLW,DRW
MALTW = MALTW - DRW;NZ = MALTW - DRW ? O;           :=   subw DRW,MALTW
AR = AR - ALTL;                                     :=   subl ALTL,AR
AR = AR - ALTW;                                     :=   subw ALTW,AR
DRW = SWAP(DRW);NZ = SWAP(DRW) ? O;                 :=   swap DRW
```

GLOSSARY OF SYMBOLS

Regular Expression Section

REGNO	Integer used in a hardware or pseudo-register name.
REGN2	Second instance of REGNO.
QNUM	Valid integer for most "quick" instructions.
HXNUM	Single hexadecimal digit.
LONUM	Hexadecimal digits 0 through 7.
HINUM	Hexadecimal digits 8 through f.
QDENT	Hexadecimal identifier representing the numbers 1 through 8 suitable for "quick" instructions.
BDENT	Byte-length identifier. Represents all signed integers. capable of fitting into one byte.
WDENT	Word-length identifier. Represents all signed integers. capable of fitting into a single word.
XDENT	Represents a term of the general case of identifiers. Can either be a long-word length hexadecimal identifier or a symbol beginning with an underscore or the letter "T" followed by any number of letters, digits, or underscores.
IDENT	The most general case of identifier. Represented by a term or by the addition and/or subtraction of terms.
LABEL	A valid label.

Token Definition Section

AR	Address register
DRL	Long-word data register. Holds 32 bits of data.
DRW	Word-length data register. Holds 16 bits of data.
DRB	Byte-length data register. Holds 8 bits of data.
LB	Label.
NZ	Condition Code.
PC	Program Counter.
SWAP	Function to swap the upper half of a long-word register with the lower half.
QDT	Integers 1 through 8 capable of being used in "quick" instructions.
BDT	Byte-length signed integers
WDT	Word-length signed integers.
IDT	Identifier.
PUSH	Long-word memory access using the hardware stack pointer as an index and incrementing it before its used. Used to push a data element onto the hardware stack.
MLTL	Long-word memory-alterable addressing modes excluding the specific addressing mode defined by PUSH.
MALTW	Word-length memory-alterable addressing modes.
MALTB	Byte-length memory-alterable addressing modes.
CTL	Control addressing modes.
REL	Relational operators.
IMP	"Implies".

Token Grouping Section

QD	Integer identifiers 1 through 8 suitable for "quick" instructions.
BD	Byte-length signed integer identifiers.
WD	Word-length signed integer identifiers.
ID	All possible identifiers
MALTL	All long-word memory-alterable addressing modes.
ALLL	All long-word addressing modes.
ALLW	All word-length addressing modes.
ALLB	All byte-length addressing modes.
DATW	All word-length data addressing modes.
DATB	All byte-length data addressing modes.
ALTL	All long-word alterable addressing modes.
ALTW	All word-length alterable addressing modes.
ALTB	All byte-length alterable addressing modes.
DALTL	All long-word data-alterable addressing modes.
DALTW	All word-length data-alterable addressing modes.
DALTB	All byte-length data-alterable addressing modes.

APPENDIX B

ASSEMBLY CODE EXAMPLE

Pascal Source Program

```
program recursivegcd(output);
function gcd(m, n: integer): integer;
begin
        if n = 0 then gcd := m
        else gcd := gcd(n, m mod n)
end;
begin
        writeln(18, 27, (gcd(18, 27)));
end.
```

Corresponding Assembly Code

Unoptimized Code

```
        .text
        .globl _P_PGM
_P_PGM:
        mask1=0x3f38
        mask2=0x1cfc
        jra L1
L3:
        moveml #mask1,a7@-
        IDstk=4*ID5+ID4-12
        addl IDstk,a5
        movl a6,a8
        movw a8,a9
        addl #16,a9
        movw a9@,d10
        movw #0,d11
        cmpw d11,d10
        seq d12
        movb d12,d13
        negb d13
        movb d13,d14
        cmpb 0,d14
        jeq L6
```

Optimized Code

```
        .text
        .globl _P_PGM
_P_PGM:
        mask1=0x3f38
        mask2=0x1cfc
        jra L1
L3:
        moveml #mask1,a7@-
        IDstk=4*ID5+ID4-12
        addl #IDstk,a5




        tstw a6@(0x00000010)

        seq d2


        negb d2
        jeq L6
```

```
        movl a6,a15
        movl a15,a16
        addl #0,a16
        movl a6,a17
        movw a17,a18
        addl #12,a18
        movw a18@,d19
        movw d19,a16@
        jra L7
L6:
        movl a6,a20
        movl a20,a21
        addl #0,a21
        movl a6,a22
        movl a22@(4),a23
        pea a5@
        clrl a5@+
        movl a23,a5@+
        movl a6,a5@+
        movl a6,a24
        movw a24,a25
        addl #16,a25
        movw a25@,d26
        movw d26,a5@+
        clrw a5@+
        movl a6,a27
        movw a27,a28
        addl #12,a28
        movw a28@,d29
        movl a6,a30
        movw a30,a31
        addl #16,a31
        movw a31@,d32
        movw d29,d0
        extl d0
        divs d32,d0
        swap d0
        movw d0,d33
        movw d33,a5@+
        clrw a5@+
        movl a7@+,a6
        jsr L3
        movl a6,a5
        movl a6@(8),a6
        movl a5@,a34
        movw d0,d35
        movw d35,a21@
L7:
        movw a6@,d36
        movw d36,d0
        extl d0
        moveml a7@+,#mask2
        rts
        ID4=20
        ID5=16
L8:
```

```
        movw a6@(0x0000000c),a6@
        jra L7
L6:

        movl a6,a2

        pea a5@
        clrl a5@+
        movl a6@(0x00000004),a5@+
        movl a6,a5@+

        movw a6@(0x00000010),a5@+
        clrw a5@+

        movw a6@(0x0000000c),d0
        extl d0
        divs a6@(0x00000010),d0
        swap d0

        movw d0,a5@+
        clrw a5@+
        movl a7@+,a6
        jsr L3
        movl a6,a5
        movl a6@(0x00000008),a6
        movl a5@,a3
        movw d0,d2
        movw d2,a2@
L7:

        movw a6@,d0
        extl d0
        moveml a7@+,#mask2
        rts
        ID4=20
        ID5=16
L8:
```

```
moveml #mask1,a7@-
IDstk=4*ID10+ID9-12
addl IDstk,a5
movw #18,d37
movw d37,d0
extl d0
movl d0,a7@-
movw #10,d38
movw d38,d0
extl d0
movl d0,a7@-
movl a6,a39
movl a39,a40
addl #20,a40
movl a40,a7@-
jsr _P_WRI
addl #12,a7
movw #27,d41
movw d41,d0
extl d0
movl d0,a7@-
movw #10,d42
movw d42,d0
extl d0
movl d0,a7@-
movl a6,a43
movl a43,a44
addl #20,a44
movl a44,a7@-
jsr _P_WRI
addl #12,a7
movl a6,a45
pea a5@
clrl a5@+
movl a45,a5@+
movl a6,a5@+
movw #18,d46
movw d46,a5@+
clrw a5@+
movw #27,d47
movw d47,a5@+
clrw a5@+
movl a7@+,a6
jsr L3
movl a6,a5
movl a6@(8),a6
movl a5@,a48
movl d0,a7@-
movw #10,d49
movw d49,d0
extl d0
movl d0,a7@-
movl a6,a50
movl a50,a51
addl #20,a51
movl a51,a7@-
```

```
moveml #mask1,a7@-
IDstk=4*ID10+ID9-12
addl #IDstk,a5

moveq #0x00000012,d0
movl d0,a7@-

moveq #0x0000000a,d0
movl d0,a7@-

pea a6@(0x00000014)
jsr _P_WRI
lea a7@(0x0000000c),a7

moveq #0x0000001b,d0
movl d0,a7@-

moveq #0x0000000a,d0
movl d0,a7@-

pea a6@(0x00000014)
jsr _P_WRI
lea a7@(0x0000000c),a7

pea a5@
clrl a5@+
movl a6,a5@+
movl a6,a5@+

movw #0x00000012,a5@+
clrw a5@+

movw #0x0000001b,a5@+
clrw a5@+
movl a7@+,a6
jsr L3
movl a6,a5
movl a6@(0x00000008),a6
movl a5@,a2
movl d0,a7@-

moveq #0x0000000a,d0
movl d0,a7@-

pea a6@(0x00000014)
```

```
        jsr _P_WRI
        addl #12,a7
        movl a6,a52
        movl a52,a53
        addl #20,a53
        movl a53,a7@-
        jsr _P_WLN
        addql #4,a7
        moveml a7@+,#mask2
        rts
        ID9=44
        ID10=16
L1:
        tstb a7@(-40)
        moveml #0x3f3e,a7@-
        lea _ustack,a5
        lea _ustack,a6
        movl a6,a54
        pea a5@
        clrl a5@+
        movl a54,a5@+
        movl a6,a5@+
        addl 32,a5
        movl a7@+,a6
        jsr L8
        movl a6,a5
        movl a6@(8),a6
        moveml a7@+,#0x7cfc
        rts
```

```
        jsr _P_WRI
        lea a7@(0x0000000c),a7

        pea a6@(0x00000014)
        jsr _P_WLN
        lea a7@(0x00000004),a7
        moveml a7@+,#mask2
        rts
        ID9=44
        ID10=16
L1:
        tstb a7@(-40)
        moveml #0x3f3e,a7@-
        movl #_ustack,a5
        movl #_ustack,a6

        pea a5@
        clrl a5@+
        movl a6,a5@+
        movl a6,a5@+
        lea a5@(0x00000020),a5
        movl a7@+,a6
        jsr L8
        movl a6,a5
        movl a6@(0x00000008),a6
        moveml a7@+,#0x7cfc
        rts
```

APPENDIX C

IMPLEMENTATION EFFORT IN DAYS

System Component	Learning Curve	Porting	Coding	Debugging	Total
P4 Compiler	7	2	¼	N/A	9 ¼
P4 Translator	3	1	½	10	14 ½
Code Expander	5	½	10	5	20 ½
Machine Description	10	N/A	3	10	23
Machine Descr. Processor	3	¼	1	5	9 ¼
Cacher	2	½	¼	½	3 ¼
Combiner	2	½	1	½	4
Assigner	2	¼	1	2	5 ¼
Runtime Routines	3	N/A	1	2	6
Target Machine Instr. Set	3	N/A	N/A	N/A	3
Total	**40**	**5**	**18**	**35**	**98**