# Using Peephole Optimization on Intermediate Code

ANDREW S. TANENBAUM, HANS van STAVEREN, and JOHAN W. STEVENSON
Vrije Universiteit, Amsterdam, The Netherlands

Many portable compilers generate an intermediate code that is subsequently translated into the target machine's assembly language. In this paper a stack-machine-based intermediate code suitable for algebraic languages (e.g., PASCAL, C, FORTRAN) and most byte-addressed mini- and microcomputers is described. A table-driven peephole optimizer that improves this intermediate code is then discussed in detail and compared with other local optimization methods. Measurements show an improvement of about 15 percent, depending on the precise metric used.

Categories and Subject Descriptors: D.3.4 **[Programming Languages]**: Processors—*compilers; interpreters; optimization*

General Terms: Experimentation, Languages

Additional Key Words and Phrases: abstract machine, intermediate code, peephole optimizer

## 1. INTRODUCTION

As the computer science field matures, software designers are coming to realize that writing a distinct compiler for each (programming language, machine) pair is an expensive way to do business. Accordingly, there has been increasing interest recently in compilers that can be easily adapted to produce code for a variety of target machines, instead of just one machine [2, 3, 5, 9].

A common method for implementing a group of languages on a collection of machines is to have one front end per language and one back end per machine. Each front end translates from its source language to a common intermediate code, called an UNCOL [11], and each back end translates from the common intermediate code to a specific target machine's assembly language (or object format). Thus a "compiler" consists of a front end, a back end, and possibly programs that optimize the intermediate code. When this model has been adopted, adding a new language or new machine only requires writing one new program (front end or back end) to maintain the property that all languages are available on all machines.

Although squeezing the last drop of performance out of the CPU is no longer the dominant requirement it once was, there are still situations in which fast

Authors' present addresses: A. S. Tanenbaum and H. van Staveren, Department of Mathematics, Vrije Universiteit, Postbus 7161, 1007 MC Amsterdam, The Netherlands; J. W. Stevenson, Arsycom BV, Kabelweg 43, Amsterdam, The Netherlands.

execution or compact code is important. Consequently, the question of where to perform the optimization arises. There are three conceptual possibilities:

1. in the front ends;
2. on the intermediate code;
3. in the back ends.

If the first possibility is chosen, many common optimizations will have to be programmed into each front end, increasing development effort. Similarly, putting the optimizations in the back ends will also require a duplication of effort. However, any optimizations that can be performed on the intermediate code itself only need be done once, with the results being applicable to all front ends and all machines being used. Although it is not possible to perform all optimizations on the intermediate code (e.g., making efficient use of target machine idiosyncracies is hard), it is clear that every optimization that can be performed on the intermediate code should be, unless it is very difficult or expensive to do so. Such optimizations are the subject of this paper.

In particular, we discuss performing peephole [8] optimizations on the intermediate code. A peephole optimization is one that replaces a sequence of consecutive instructions by a semantically equivalent but more efficient sequence. The sequences to be matched and their replacements are described in a driving table.

Other researchers [4, 7, 14] have also examined peephole optimization, but generally on the object code rather than on the intermediate code. We compare our work to theirs after describing our method and results in detail.

## 2. THE ARCHITECTURE OF THE INTERMEDIATE CODE MACHINE

The intermediate code produced by our front ends and accepted by our back ends [12, 13] is the assembly language of a simple stack machine called EM. It has been designed to be suitable for algebraic languages and byte-addressable target machines. Front ends exist, are currently being developed, or are at least being contemplated for ADA, ALGOL 68, BASIC, BCPL, C, FORTRAN, PASCAL, and PLAIN. Back ends will eventually include the Intel 8080 and 8086, Zilog Z80 and Z8000, Motorola 6800, 6809, and 68000, TI 9900, DEC PDP-11 and VAX, and IBM 370. These lists are not exhaustive but should give a reasonable idea of the scope of EM. On the other hand, EM is not especially well suited for radically different languages (e.g., COBOL, LISP, SNOBOL) or machines that are not byte addressable (e.g., CDC Cyber, DEC PDP-10, Univac 1108) or are completely different (e.g., data flow, Turing machine).

Briefly, each procedure invocation creates a frame on the stack. The Local Base register (LB) points to the base of the current frame, and the Stack Pointer (SP) points to the top of the frame. The External Base register (EB) points to the bottom of the outermost stack frame. Variables at intermediate levels of lexical nesting are accessed by following the static chain backward.

All arithmetic instructions fetch their operands from the top of the stack and put their results back on the stack. Expressions are evaluated merely by converting them to reverse Polish. There are no general registers. Instructions are provided for manipulating integers of various lengths, floating-point numbers, pointers, and multiword unsigned quantities (e.g., for representing sets).

A variety of instructions has been provided for loading operands onto the stack and popping them off the stack for storage. These instructions have been chosen to match closely the semantic primitives of common algebraic languages. There are instructions for use with constants, local variables, parameters, array elements, record fields, etc.

A list of the most important machine instructions is given in Table I. The meaning of most of the instructions is provided by the PASCAL fragment following it. Throughout the definitions, the variable $B$ is used to avoid making a distinction between LB and EB, since that aspect of the machine has little significance for the optimizations discussed below. Correspondingly, the distinction between instructions that access local variables and those that access external variables has been eliminated.

## 3. CODE GENERATION STRATEGY

In order to understand the significance of the optimizations, it is necessary to understand something about typical code sequences produced by front ends for EM. Various studies [1, 6, 12] have shown that most programs tend to contain mostly simple expressions and statements. Wulf et al. [14] have pointed out that in the final analysis the quality of the local code has a greater impact on both the size and speed of the final program than does any optimization. The conclusion we draw from these two observations is that the efficient compilation of simple, commonly occurring source statements is the key to producing good code.

The traditional way to generate good code for commonly occurring statements is to build a myriad of special cases into all the front ends. For example, the statement $N := N + 1$ occurs often in many programs; so EM has a special INCREMENT VARIABLE instruction. The normal approach would be to have the front end check all assignments to see if they can use this instruction. It is our belief that this approach is a mistake and that recognition and replacement of important instruction sequences should be done by the optimizer. In fact, our basic intermediate file optimizer performs only this kind of peephole optimization; data flow and other optimizations can and should be done by other programs, cascading them in the manner of UNIX™[1] [10] filters.

Coming back to the case of assignment statements, in general, assignment statements can be complicated, such as

$$A[I + 1].FIELD1 := B[J * K].FIELD2[M]$$

Although such statements are rare, front ends must be prepared to deal with them. Consequently, the general strategy used by our front ends is to evaluate the address of the right-hand side, push this address onto the stack, and then do a LOAD INDIRECT $n$ instruction, which pops the address and pushes the $n$-byte-long object starting at the address. After that, the address of the left-hand side is also evaluated and pushed onto the stack, and then a STORE INDIRECT $n$ instruction, which fetches the destination address and $n$-byte object from the stack and performs the store, is executed. There are many special cases of the assignment statement that can be optimized, but the front ends ignore most of them, leaving the work to the optimizer.

[1] UNIX is a trademark of Bell Laboratories.

Table I. A Simplified Summary of the EM Instruction Set

| Mnemonic | Instruction | Meaning |
|---|---|---|
| ADD | Add | $t := pop; s := pop; push(s + t)$ |
| ADI $k$ | Add immediate | $t := pop; push(t + k)$ |
| AND 2 | And | $t := pop; s := pop; push(booland(s, t))$ |
| BEG $k$ | Begin procedure | $sp := sp + k$ |
| BEQ $k$ | Branch equal to | $t := pop; s := pop;$ **if** $s = t$ **then** $pc := pc + k$ |
| BGE $k$ | Branch greater/equal | $t := pop; s := pop;$ **if** $s \geq t$ **then** $pc := pc + k$ |
| BGT $k$ | Branch greater | $t := pop; s := pop;$ **if** $s > t$ **then** $pc := pc + k$ |
| BLE $k$ | Branch less/equal | $t := pop; s := pop;$ **if** $s \leq t$ **then** $pc := pc + k$ |
| BLT $k$ | Branch less than | $t := pop; s := pop;$ **if** $s < t$ **then** $pc := pc + k$ |
| BNE $k$ | Branch not equal | $t := pop; s := pop;$ **if** $s \neq t$ **then** $pc := pc + k$ |
| BRA $k$ | Branch unconditionally | $pc := pc + k$ |
| BLM $k$ | Block move | $t := pop; s := pop;$ **for** $j := 1$ **to** $k$ **div** 2 **do** $m[t - 2 + 2 * j] := m[s - 2 + 2 * j]$ |
| CMI | Compare | $t := pop; s := pop;$ **if** $s < t$ **then** $push(-1)$ **else if** $s = t$ **then** $push(0)$ **else** $push(1)$ |
| COM 2 | Complement | $t := pop; push(boolxor(-1, t))$ |
| DEC | Decrement stack | $t := pop; push(t - 1)$ |
| DEV $k$ | Decrement variable | $m[B + k] := m[B + k] - 1$ |
| DIV | Divide | $t := pop; s := pop; push(s/t)$ |
| DUP 2 | Duplicate | $t := pop; push(t); push(t)$ |
| INC | Increment stack | $t := pop; push(t + 1)$ |
| INV $k$ | Increment variable | $m[B + k] := m[B + k] + 1$ |
| IOR 2 | Inclusive or | $t := pop; s := pop; push(boolior(s, t))$ |
| LAV $k$ | Load address of variable | $push(B + k)$ |
| LDF $k$ | Load double offsetted | $t := pop; push(m[t + k]); push(m[t + k + 2])$ |
| LDV $k$ | Load double variable | $push(m[B + k]); push(m[B + k + 2])$ |
| LOC $k$ | Load constant | $push(k)$ |
| LOF $k$ | Load offsetted | $t := pop; push(m[t + k])$ |
| LOI $k$ | Load indirect | $t := pop;$ **for** $j := 1$ **to** $k$ **div** 2 **do** $push(m[t - 2 + 2 * j])$ |
| LOP $k$ | Load parameter | $push(m[m[B + k]])$ |
| LOV $k$ | Load variable | $push(m[B + k])$ |
| MOD | Modulo | $t := pop; s := pop; push(s \bmod t)$ |
| MUL | Multiply | $t := pop; s := pop; push(s * t)$ |
| NEG | Negate | $t := pop; push(-t)$ |
| NOT | Boolean complement | $t := pop;$ **if** $t = 0$ **then** $push(1)$ **else** $push(0)$ |
| RET $k$ | Return | {Pop $k$ bytes, remove a stack frame, then push the $k$ bytes} |
| ROL | Rotate left | {All bits are left-circularly shifted} |

Next to assignment statements, **if** statements are most common. Statements of the form **if** $A = B$ **then** . . . occur far more frequently than other types; so the obvious EM code sequence consists of instructions to push $A$ and $B$ onto the stack followed by a BNE instruction, which pops two operands and branches to the **else** part if they are unequal.

At first glance it would seem that six B$xx$ instructions would be needed in the EM architecture, for $xx$ = EQ, NE, LT, LE, GT, and GE, but in fact many more are needed, since a complete set is needed for single-precision integers, double-precision integers, unsigned integers, pointers, floating-point numbers, sets, etc. To avoid this proliferation, EM has one compare instruction for each data type (CM$x$) that pops two operands and replaces them with a −1, 0, or +1, depending on whether the first is less than, equal to, or greater than the second. Then there are six T$xx$ instructions for replacing this number with **true** (represented by 1)

Table I continued

| Mnemonic | Instruction | Meaning |
|---|---|---|
| ROR | Rotate right | {All bits are right-circularly shifted} |
| SDF $k$ | Store double offsetted | $t := pop$; $m[t + 2 + k] := pop$; $m[t + k] := pop$ |
| SDV $k$ | Store double variable | $m[B + k + 2] := pop$; $m[B + k] := pop$ |
| SHL | Shift left | $t := pop$; $s := pop$; **for** $j := 1$ **to** $t$ **do** $s := s + s$; $push(s)$ |
| SHR | Shift right | $t := pop$; $s := pop$; **for** $j := 1$ **to** $t$ **do** $s := s$ **div** 2; $push(s)$ |
| STF $k$ | Store offsetted | $t := pop$; $m[t + k] := pop$ |
| STI $k$ | Store indirect | $t := pop$; **for** $j := 1$ **to** $k$ **div** 2 **do** $m[t + k - 2 * j] := pop$ |
| STP $k$ | Store parameter | $m[m[B + k]] := pop$ |
| STV $k$ | Store variable | $m[B + k] := pop$ |
| SUB | Subtract | $t := pop$; $s := pop$; $push(s - t)$ |
| TEQ | True if equal to | $t := pop$; **if** $t = 0$ **then** $push(1)$ **else** $push(0)$ |
| TGE | True if greater/equal | $t := pop$; **if** $t \geq 0$ **then** $push(1)$ **else** $push(0)$ |
| TGT | True if greater | $t := pop$; **if** $t > 0$ **then** $push(1)$ **else** $push(0)$ |
| TLE | True if less/equal | $t := pop$; **if** $t \leq 0$ **then** $push(1)$ **else** $push(0)$ |
| TLT | True if less than | $t := pop$; **if** $t < 0$ **then** $push(1)$ **else** $push(0)$ |
| TNE | True if not equal | $t := pop$; **if** $t \neq 0$ **then** $push(1)$ **else** $push(0)$ |
| XOR 2 | Exclusive or | $t := pop$; $s := pop$; $push(boolxor(s, t))$ |
| ZEQ $k$ | Branch equal to zero | $t := pop$; **if** $t = 0$ **then** $pc := pc + k$ |
| ZGE $k$ | Branch greater/equal zero | $t := pop$; **if** $t \geq 0$ **then** $pc := pc + k$ |
| ZGT $k$ | Branch greater zero | $t := pop$; **if** $t > 0$ **then** $pc := pc + k$ |
| ZLE $k$ | Branch less/equal zero | $t := pop$; **if** $t \leq 0$ **then** $pc := pc + k$ |
| ZLT $k$ | Branch less than zero | $t := pop$; **if** $t < 0$ **then** $pc := pc + k$ |
| ZNE $k$ | Branch not equal zero | $t := pop$; **if** $t \neq 0$ **then** $pc := pc + k$ |
| ZRV $k$ | Zero variable | $m[B + k] := 0$ |

Notes

1. The offset $k$ is in bytes.
2. $m[k]$ addresses the word at address $k$; $k$ must be even.
3. *pc* is the program counter.
4. *sp* is the stack pointer.
5. EM uses two's complement arithmetic.
6. $push(k)$ pushes one word onto the stack.
7. *pop* is a function that removes and returns the top word on the stack.
8. *booland*, *boolior*, and *boolxor* each return the indicated Boolean function.
9. No distinction is made here between local and external variables.
10. No distinction is made here between integers and addresses.
11. No checking and trapping is indicated here.

or **false** (represented by 0), depending on the relational operator. For example, the PASCAL statement

**if** $(I = J)$ **and** $(X < 3.14)$ **and** $((FLAG =$ **false**$)$ **or** $(K > 0))$ **then** ...

is trivially translated to the reverse Polish string

$I$, $J$, =, $X$, 3.14, *FLOATING* <, **and**, *FLAG*, **false**, =, $K$, 0, >, **or**, **and**

and then to the EM code sequence

```
LOV I       ; stack I
LOV J       ; stack J
CMI         ; compare integer, pushing −1, 0, or +1
TEQ         ; relational operator was =
LDV X       ; stack X (floating-point is double length)
```

```
LDV c1      ; stack floating-point constant 3.14 from memory
CMF         ; compare floating, pushing -1, 0, or +1
TLT         ; relational operator was <
AND 2       ; "Boolean and" on 2-byte operands
LOV FLAG    ; stack FLAG
LOC 0       ; stack false
CMI         ; compare integer FLAG and false
TEQ         ; test for equality
LOV K       ; stack K
LOC 0       ; stack 0
CMI         ; compare integer K and 0
TGT         ; relational operator was >
IOR 2       ; "inclusive or" on 2-byte operands
AND 2       ; "Boolean and" on 2-byte operands
```

Here, as in the case of assignment statements, optimizations are possible.

In summary, our approach is to have front ends deal with only a single case, the most general one, and to let the optimizer convert the often clumsy code sequence that results into the optimal one.

## 4. OPTIMIZATION PATTERNS

The optimizer is driven by a pattern/replacement table consisting of a collection of lines. Each line contains a pattern part and a replacement part. A pattern or replacement part is composed of a consecutive sequence of EM instructions, all of which designate an opcode and some of which designate an operand. (By design, no EM instruction has more than one operand.)

The operands can be constants, references to other operands within the line, or expressions involving both. For example, consider the three lines (1)-(3):

| Pattern | | | | Replacement |
|---|---|---|---|---|
| (1) LOV *A* | INC | STV *A* | ⇒ | INV *A* |
| (2) LOV *A* | LOV *A* + 2 | | ⇒ | LDV *A* |
| (3) LOC *A* | NEG | | ⇒ | LOC −*A* |

In each line the ⇒ symbol separates the pattern part (left) from the replacement part (right). Throughout the optimizations we use the symbols *A*, *B*, *C*, etc., as formal parameters to refer to operands. In line (1), a variable is loaded (pushed) onto the stack (LOV *A*), it is incremented by one (INC), and the result is stored in the variable (STV *A*). The whole sequence can be replaced by a single increment variable instruction (INV). The pattern of (1) only matches a sequence of three EM instructions if the three opcodes are LOV, INC, and STV, in that order; furthermore, the first and third operands are the same, whatever their value may be. If the pattern matches, the operand of the resulting INV instruction is copied from that of the LOV instruction.

In (2) an example of operand arithmetic is given. The pattern only matches if the address of the second LOV instruction is two higher than that of the first. The effect of this line is to replace two consecutive one-word pushes by a single two-word push.

The third example shows that loading a constant onto the stack and then negating it can be replaced by loading the negative of the constant onto the stack in the first place. The noteworthy feature here is an expression, albeit a simple one in this case $(-A)$, in the replacement part.

We can now describe the operation of the optimizer in detail. Initially, an entire procedure is read into an internal array, one instruction per array element. The optimizer maintains a pointer to the "current" instruction. Starting at this position, it tries to find optimizations that begin with the current instruction. If a pattern match can be found, the replacement part is substituted for the matched instructions.

It often occurs that the output of one optimization produces a pattern that itself can be optimized. In fact, this principle has been extensively used in the design of the optimization table to reduce its size. By repeating the matching process until no more matches can be found, patterns much longer than the longest optimization table entry can ultimately be replaced. After a replacement, the code pointer is moved back a distance equal to the longest pattern to make sure that no newly created matches are missed.

## 5. THE OPTIMIZATION TABLE

In this section we present and discuss a major portion of the EM optimization table. As an aid to the reader, we have divided the optimizations into ten major groups, as shown in Table II. Each optimization is numbered for reference in the text. In addition, the occurrence rate of each optimization per 100,000 optimizations is given in parentheses. The data are based on a collection of about 500 PASCAL programs. Obviously, the front end's code generation strategy has an effect on these frequencies. The PASCAL front end used for these measurements used the strategy described above.

The first group, lines 1–21, represent computations that can be performed at compile time (i.e., optimize time) instead of at run time. Lines 1–13 are all of the form LOC *a*; LOC *b*; operator; so it is straightforward to carry out the operation. More complicated expressions can also be folded because the pattern scanning continues until no more matches are found. For example, the source statement $K := 3 * 5 + 6$ would initially compile to the EM code sequence

```
LOC 3
LOC 5
MUL
LOC 6
ADD
```

Line 3 would replace the first 3 instructions by LOC 15; then optimization 1 would replace the remaining sequence by LOC 21. Arbitrary constant expressions involving all the operators can be folded to a single LOC.

Lines 14 and 15 represent the unary operators for two's complement and one's complement, respectively. Lines 16 and 17 reflect the fact that negation followed by addition is subtraction, and vice versa. Lines 18 and 19 might have been useful to convert **not true** (1) to **false** (0) and **not false** to **true**, but these never occurred. The BEG instruction is typically used to advance the stack upon

Table II. The Optimizations and Their Replacements

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *Constant folding* | | | | | | | |
| 1. | (225) | LOC *A* | LOC *B* | ADD | ⇒ LOC *A* + *B* | | |
| 2. | (181) | LOC *A* | LOC *B* | SUB | ⇒ LOC *A* − *B* | | |
| 3. | (125) | LOC *A* | LOC *B* | MUL | ⇒ LOC *A* * *B* | | |
| 4. | (65) | LOC *A* | LOC *B* | DIV | ⇒ LOC *A*/*B* | | |
| 5. | (12) | LOC *A* | LOC *B* | MOD | ⇒ LOC *A* MOD *B* | | |
| 6. | (52) | LOC *A* | LOC *B* | AND 2 | ⇒ LOC *A* AND *B* | | |
| 7. | (67) | LOC *A* | LOC *B* | IOR 2 | ⇒ LOC *A* IOR *B* | | |
| 8. | (0) | LOC *A* | LOC *B* | XOR 2 | ⇒ LOC *A* XOR *B* | | |
| 9. | (0) | LOC *A* | LOC *B* | SHL | ⇒ LOC *A* SHL *B* | | |
| 10. | (0) | LOC *A* | LOC *B* | SHR | ⇒ LOC *A* SHR *B* | | |
| 11. | (0) | LOC *A* | LOC *B* | ROL | ⇒ LOC *A* ROL *B* | | |
| 12. | (0) | LOC *A* | LOC *B* | ROR | ⇒ LOC *A* ROR *B* | | |
| 13. | (191) | LOC *A* | LOC *B* | CMI | ⇒ LOC *A* CMI *B* | | |
| 14. | (1330) | LOC *A* | NEG | | ⇒ LOC −*A* | | |
| 15. | (39) | LOC *A* | COM 2 | | ⇒ LOC ~*A* | | |
| 16. | (0) | NEG | SUB | | ⇒ ADD | | |
| 17. | (0) | NEG | ADD | | ⇒ SUB | | |
| 18. | (0) | LOC 1 | NOT | | ⇒ LOC 0 | | |
| 19. | (0) | LOC 0 | NOT | | ⇒ LOC 1 | | |
| 20. | (0) | BEG *A* | BEG *B* | | ⇒ BEG *A* + *B* | | |
| 21. | (496) | ADI *A* | ADI *B* | | ⇒ ADI *A* + *B* | | |
| *Operator strength reduction* | | | | | | | |
| 22. | (85) | LOC 2 | LOV *A* | MUL | ⇒ LOV *A* | LOC 1 | SHL |
| 23. | (29) | LOC 2 | MUL | | ⇒ LOC 1 | SHL | |
| 24. | (17) | LOC 4 | MUL | | ⇒ LOC 2 | SHL | |
| 25. | (18) | LOC 8 | MUL | | ⇒ LOC 3 | SHL | |
| 26. | (12) | LOC 16 | MUL | | ⇒ LOC 4 | SHL | |
| 27. | (0) | LOC 32 | MUL | | ⇒ LOC 5 | SHL | |
| 28. | (0) | LOC −1 | LOV *A* | MUL | ⇒ LOV *A* | NEG | |
| *Null sequences* | | | | | | | |
| 29. | (4248) | ADI 0 | | | ⇒ — | | |
| 30. | (1638) | BEG 0 | | | ⇒ — | | |
| 31. | (12) | NEG | NEG | | ⇒ — | | |
| 32. | (1) | LOC 0 | ADD | | ⇒ — | | |
| 33. | (0) | LOC 0 | SUB | | ⇒ — | | |
| 34. | (4) | LOC 1 | MUL | | ⇒ — | | |
| 35. | (1) | LOC 1 | DIV | | ⇒ — | | |
| 36. | (14) | LOC 0 | IOR 2 | | ⇒ — | | |
| 37. | (0) | LOC 0 | XOR 2 | | ⇒ — | | |
| 38. | (5) | LOV *A* | STV *A* | | ⇒ — | | |
| 39. | (9) | LDV *A* | SDV *A* | | ⇒ — | | |
| 40. | (5) | LOC 0 | LOV *A* | ADD | ⇒ LOV *A* | | |
| 41. | (0) | LOC 1 | LOV *A* | MUL | ⇒ LOV *A* | | |
| 42. | (0) | LOC 0 | LOV *A* | MUL | ⇒ LOC 0 | | |
| 43. | (0) | LOV *A* | LOC 0 | MUL | ⇒ LOC 0 | | |
| 44. | (454) | STV *A* | LOV *A* | RET 2 | ⇒ RET 2 | | |
| 45. | (256) | SDV *A* | LDV *A* | RET 4 | ⇒ RET 4 | | |
| 46. | (3121) | LOC 0 | CMI | T*xx* | ⇒ T*xx* | | |
| 47. | (323) | BRA *A* | LAB *A* | | ⇒ LAB *A* | | |

procedure entry. In line 20 two consecutive BEGs are reduced to one. Typically, ADI (add immediate) is used to offset from a pointer by a known distance, for example, to access a field of a record. If the field is itself a record, the front end may generate two consecutive ADIs, in which case line 21 can be used to make one ADI from them.

The operator strength reduction group replaces multiplications by powers of two with shifts, and multiplications by −1 with negates.

Table II continued

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *Combined moves* | | | | | | | | |
| 48. | (2555) | LOV *A* | LOV *A* + 2 | | | ⇒ LDV *A* | | |
| 49. | (15) | LDV *A* | LOV *A* + 4 | | | ⇒ LAV *A* | LOI 6 | |
| 50. | (217) | LDV *A* | LDV *A* + 4 | | | ⇒ LAV *A* | LOI 8 | |
| 51. | (98) | LAV *A* | LOI *B* | LOV *A* + *B* | | ⇒ LAV *A* | LOI *B* + 2 | |
| 52. | (0) | LAV *A* | LOI *B* | LDV *A* + *B* | | ⇒ LAV *A* | LOI *B* + 4 | |
| 53. | (89) | LAV *A* | LOI *B* | LAV *A* + *B* | LOI *C* | ⇒ LAV *A* | LOI *B* + *C* | |
| 54. | (260) | LAV *A* | STI *B* | LOC *C* | STV *A* + *B* | ⇒ LOC *C* | LAV *A* | STI *B* + 2 |
| 55. | (9) | LAV *A* | STI *B* | LOV *C* | STV *A* + *B* | ⇒ LOV *C* | LAV *A* | STI *B* + 2 |
| 56. | (3) | STV *A* | STV *A* − 2 | | | ⇒ SDV *A* − 2 | | |
| 57. | (0) | SDV *A* | STV *A* − 2 | | | ⇒ LAV *A* − 2 | STI 6 | |
| 58. | (8) | SDV *A* | SDV *A* − 4 | | | ⇒ LAV *A* − 4 | STI 8 | |
| 59. | (0) | LAV *A* | STI *B* | STV *A* − 2 | | ⇒ LAV *A* − 2 | STI *B* + 2 | |
| 60. | (0) | LAV *A* | STI *B* | SDV *A* − 4 | | ⇒ LAV *A* − 2 | STI *B* + 4 | |
| 61. | (385) | STV *A* | LOC *B* | STV *A* + 2 | | ⇒ LOC *B* | SDV *A* | |
| 62. | (20) | STV *A* | LOV *B* | STV *A* + 2 | | ⇒ LOV *B* | SDV *A* | |
| 63. | (203) | SDV *A* | LOC *B* | STV *A* + 4 | | ⇒ LOC *B* | LAV *A* | STI 6 |
| 64. | (4) | LAV *A* | BLM 2 | | | ⇒ LOI 2 | STV *A* | |
| 65. | (86) | LAV *A* | BLM 4 | | | ⇒ LOI 4 | SDV *A* | |
| 66. | (4) | LOV *A* | BLM 2 | | | ⇒ LOI 2 | STP *A* | |
| *Commutative law* | | | | | | | | |
| 67. | (68) | LOV *A* | LOV *A* − 2 | ADD | | ⇒ LDV *A* − 2 | ADD | |
| 68. | (8) | LOV *A* | LOV *A* − 2 | MUL | | ⇒ LDV *A* − 2 | MUL | |
| 69. | (181) | LOV *A* | LOV *A* − 2 | CMI | T*xx* | ⇒ LDV *A* − 2 | CMI | T*zz* |
| 70. | (0) | LOV *A* | LOV *A* − 2 | B*xx B* | | ⇒ LDV *A* − 2 | B*zz B* | |
| *Indirect moves* | | | | | | | | |
| 71. | (994) | LAV *A* | LOI 2 | | | ⇒ LOV *A* | | |
| 72. | (874) | LAV *A* | STI 2 | | | ⇒ STV *A* | | |
| 73. | (0) | LAV *A* | LOF *B* | | | ⇒ LOV *A* + *B* | | |
| 74. | (0) | LAV *A* | STF *B* | | | ⇒ STF *A* + *B* | | |
| 75. | (2785) | LOV *A* | LOI 2 | | | ⇒ LOP *A* | | |
| 76. | (1353) | LOV *A* | STI 2 | | | ⇒ STP *A* | | |
| 77. | (3991) | LAV *A* | ADI *B* | | | ⇒ LAV *A* + *B* | | |
| 78. | (0) | LAV *A* | STI *B* | STV *A* − 2 | | ⇒ LAV *A* | STI *B* + 2 | |
| 79. | (0) | LAV *A* | STI *B* | SDV *A* − 4 | | ⇒ LAV *A* | STI *B* + 4 | |
| 80. | (9834) | LAV *A* | LOI 4 | | | ⇒ LDV *A* | | |
| 81. | (4211) | LAV *A* | STI 4 | | | ⇒ SDF *A* | | |
| 82. | (0) | LAV *A* | LDF *B* | | | ⇒ LDV *A* + *B* | | |
| 83. | (0) | LAV *A* | SDF *B* | | | ⇒ SDV *A* + *B* | | |
| 84. | (5988) | ADI *A* | LOI 2 | | | ⇒ LOF *A* | | |
| 85. | (0) | ADI *A* | LOF *B* | | | ⇒ LOF *A* + *B* | | |
| 86. | (2562) | ADI *A* | STI 2 | | | ⇒ STF *A* | | |
| 87. | (0) | ADI *A* | STF *B* | | | ⇒ STF *A* + *B* | | |
| 88. | (891) | ADI *A* | LOI 4 | | | ⇒ LDF *A* | | |
| 89. | (526) | ADI *A* | STI 4 | | | ⇒ SDF *A* | | |
| 90. | (0) | ADI *A* | LDF *B* | | | ⇒ LDF *A* + *B* | | |
| 91. | (0) | ADI *A* | SDF *B* | | | ⇒ SDF *A* + *B* | | |

Note: *xx* = LT, LE, EQ, NE, GE, GT
*yy* = GE, GT, NE, EQ, LT, LE
*zz* = GT, GE, EQ, NE, LE, LT

The third group (lines 29–47) eliminates sequences or partial sequences of code that are redundant. ADI 0 (line 29) is typically generated when accessing the first field of a record. The front end arranges for the address of the record to appear on the stack and then increases this address by the relative position of the desired field in the record, which for the first field is 0.

Upon procedure entry, the stack pointer is advanced to reserve storage for local variables by emitting BEG *SIZE*, where *SIZE* is subsequently defined as the

Table II continued

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *Comparison* | | | | | | | | |
| 92. | (9625) | T*xx* | ZEQ *A* | | | ⇒ Z*yy A* | | |
| 93. | (4824) | CMI | Z*xx A* | | | ⇒ B*xx A* | | |
| 94. | (726) | T*xx* | TEQ | | | ⇒ T*yy* | | |
| 95. | (0) | NOT | ZEQ *A* | | | ⇒ ZNE *A* | | |
| 96. | (0) | NOT | ZNE *A* | | | ⇒ ZEQ *A* | | |
| *Special instructions* | | | | | | | | |
| 97. | (8546) | LOC 1 | ADD | | | ⇒ INC | | |
| 98. | (1504) | LOC 1 | SUB | | | ⇒ DEC | | |
| 99. | (5727) | LOV *A* | INC | STV *A* | | ⇒ INV *A* | | |
| 100. | (347) | LOV *A* | DEC | STV *A* | | ⇒ DEV *A* | | |
| 101. | (0) | LOC −1 | SUB | | | ⇒ INC | | |
| 102. | (0) | LOC −1 | ADD | | | ⇒ DEC | | |
| 103. | (8) | LOC 1 | LOV *A* | ADD | | ⇒ LOV *A* | INC | |
| 104. | (0) | LOC −1 | LOV *A* | ADD | | ⇒ LOV *A* | DEC | |
| 105. | (11) | LOV *A* | LOC 2 | ADD | STV *A* | ⇒ INV *A* | INV *A* | |
| 106. | (0) | LOC 2 | LOV *A* | ADD | STV *A* | ⇒ INV *A* | INV *A* | |
| 107. | (5) | LOV *A* | LOC 2 | SUB | STV *A* | ⇒ DEV *A* | DEV *A* | |
| 108. | (0) | LOC −1 | MUL | | | ⇒ NEG | | |
| 109. | (0) | LOC −1 | DIV | | | ⇒ NEG | | |
| 110. | (4082) | LOC 0 | STV *A* | | | ⇒ ZRV *A* | | |
| 111. | (22) | LOC 0 | B*xx A* | | | ⇒ Z*xx A* | | |
| 112. | (46) | LOC 1 | BLT *A* | | | ⇒ ZLE *A* | | |
| 113. | (17) | LOC 1 | BGE *A* | | | ⇒ ZGT *A* | | |
| 114. | (0) | LOC −1 | BGT *A* | | | ⇒ ZGE *A* | | |
| 115. | (1) | LOC −1 | BLE *A* | | | ⇒ ZLT *A* | | |
| *DUP instruction* | | | | | | | | |
| 116. | (263) | STV *A* | LOV *A* | | | ⇒ DUP 2 | STV *A* | |
| 117. | (199) | LOV *A* | LOV *A* | | | ⇒ LOV *A* | DUP 2 | |
| *Reordering* | | | | | | | | |
| 118. | (53) | ADD | LOC *A* | ADD | | ⇒ LOC *A* | ADD | ADD |
| 119. | (107) | ADD | LOC *A* | SUB | | ⇒ LOC *A* | SUB | ADD |
| 120. | (56) | SUB | LOC *A* | ADD | | ⇒ LOC *A* | SUB | SUB |
| 121. | (27) | SUB | LOC *A* | SUB | | ⇒ LOC *A* | ADD | SUB |
| 122. | (24) | MUL | NEG | | | ⇒ NEG | MUL | |
| 123. | (0) | DIV | NEG | | | ⇒ NEG | DIV | |

Note: *xx* = LT, LE, EQ, NE, GE, GT
*yy* = GE, GT, NE, EQ, LT, LE
*zz* = GT, GE, EQ, NE, LE, LT

amount of storage needed. If there are no locals, *SIZE* will turn out to be 0; so line 30 removes the BEG.

The next five lines deal with expressions of the form $-(-n)$, $n + 0$, $n - 0$, $n * 1$, and $n/1$, respectively. These expressions might occur when the constant is actually a manifest constant. Inclusive and exclusive **or**-ing a variable with 0 does not change it, as indicated in lines 36 and 37.

Lines 38 and 39 eliminate redundant load/store sequences.

The next four optimizations deal with pushing two operands onto the stack and then performing an operation on them. In these cases the operation does nothing; so one push and the operation can be deleted.

Lines 44 and 45 concern the way functions return values in EM: they push the values onto the stack and then execute a RET instruction. A store into a local variable prior to the return is obviously wasted; so it is eliminated.

The instruction CMI in line 46 puts a negative number, 0, or a positive number onto the stack for use by a subsequent T*xx*, where *xx* stands for one of the relational operators LT, LE, EQ, NE, GE, or GT. In this case the comparison with 0 is redundant, since the operand being compared to 0 can itself be used for the succeeding test.

Line 47 uses the fact that labels are represented as pseudoinstructions with the opcode LAB. Thus, if a branch forward to *A* is followed by *A* itself, the branch can be eliminated.

The combined move group tries to combine consecutive push or pop operations into a single one. When the EM code is to be interpreted, replacing two instructions by one is always worth doing. Similarly, when EM is translated to some target machine, combining two moves may be useful, especially if the target machine has double- or multiple-word moves. If enough moves can be combined, the optimization is worth doing on all machines, since it gives the back end the option of using a loop instead of in-line code. In our PASCAL compiler, three pages of source code initializing various tables are ultimately reduced to a single move (BLM) instruction.

The basic strategy followed by lines 48–63 is to combine single-word, double-word, and multiword moves (LOV, LDV, and LOI, respectively) into longer units. Lines 64–66 convert short BLM instructions into LOIs to permit them to combine more easily with other instructions (there are many patterns with LOI, but few with BLM).

Lines 67–70 take advantage of the commutative law. Since the operands of ADD and MUL can be reversed without damage, they are reversed when doing so is worthwhile. The operands of CMI can also be reversed (line 69), but only if the sense of the following T*xx* is also reversed, where *xx* again stands for LT, LE, EQ, NE, GE, or GT, and *zz* stands for the complementary operators, GT, GE, EQ, NE, LE, or LT, respectively. For example,

```
LOV 8                LDV 6
LOV 6     becomes    CMI
CMI                  TGT
TLT
```

Line 70 uses the same principle as does line 69.

The indirect move group is largely concerned with replacing indirect moves, generated by the general case of the assignment statement, with more efficient direct ones. Lines 71–83 fall in this category. The remaining optimizations in the group combine ADI with the following instruction and improve the code. The ADIs are often generated by accessing record fields, as mentioned earlier.

Line 92 is one of the most frequent optimizations. T*xx* normally converts the output of a compare instruction (−1, 0, or +1) to **true** (1) or **false** (0). The ZEQ following it can be thought of as "branch if false." Instead of first converting the negative, 0, or positive value on the stack to **true** or **false**, the branch uses the value itself. The code *yy* stands for GE, GT, NE, EQ, LT, or LE, respectively, depending on the corresponding *xx* code: LT, LE, EQ, NE, GE, or GT. This optimization is used for **if** statements with only a single relational operator in the condition.

Line 93 is useful when comparing two integer operands. Instead of comparing them and then testing the result of the comparison, one can use the B*xx* instructions, which compare and branch in one instruction. The optimization on line 94 replaces two cascaded T*xx* operations by a single equivalent one. The remaining two in this group are able to simulate the effect of the PASCAL **not** operator by changing the sense of the branch.

EM has several one-operand instructions that are actually special cases of two-operand instructions. Lines 97–104 handle addition and subtraction of +1 or −1. Note that the source statement $N := N + 1$ is reduced to a single INV instruction in two stages: first, optimization 97 is applied, and then optimization 99. This replacement may even be useful on target machines lacking an increment instruction (e.g., on machines that can add a register to memory). In general we cannot guarantee that every optimization is useful on every machine, but in practice the ones listed in Table II are useful on many machines of the class being considered.

Lines 105–107 replace addition or subtraction by two with a pair of increments or decrements. In retrospect, their inclusion in the table was probably not such a good idea, since on some target machines they are likely to lead to worse rather than better code. (A machine lacking an INCREMENT but able to add a register to memory would probably be better off adding 2 to the variable once than adding 1 to the variable twice.) On the other hand, these patterns are rarely used in practice. Lines 108 and 109 substitute the one-operand NEG instruction for the two-operand MUL and DIV instructions where possible.

The optimization on line 110 is intended for source statements of the form $N := 0$. The ZRV instruction does the work of two other instructions. The next line represents six optimizations for comparing an integer operand against zero. The last four lines are similar but slightly tricky. Line 112 implicitly replaces the test **if** $N < 1$ **then** with the equivalent test **if** $N \leq 0$ **then** so that the one-operand ZLE instruction instead of the two-operand BLT instruction can be used. Lines 113–115 are analogous to line 112.

The DUP group is useful to avoid refetching an operand that is already on the stack. On many target machines, duplicating the top word or two words on the stack is a cheaper operation than addressing memory, since no memory address is required in the former, but it is in the latter.

The reordering group merely reorders the instructions in each pattern without changing them. The first four move a LOC from the middle of an operation sequence to the start of it. By moving LOCs forward, the chances of another optimization becoming possible are increased. For example, the source expression $A := A + SIZE - 5$, where *SIZE* is a manifest constant with value 6, normally is compiled and optimized as follows:

| Original | 1st opt. (119) | 2nd opt. (2) | 3rd opt. (97) | Final code (99) |
|---|---|---|---|---|
| LOV *A* | LOV *A* | LOV *A* | LOV *A* | INV *A* |
| LOC 6 | LOC 6 | LOC 1 | INC | |
| ADD | LOC 5 | ADD | STV *A* | |
| LOC 5 | SUB | STV *A* | | |
| SUB | ADD | | | |
| STV *A* | STV *A* | | | |

Lines 122 and 123 operate in a spirit similar to that of lines 118–121. The NEG is pushed forward to allow it to combine with a possible LOC preceding it. For example, the source statement $N := 5/-K$ in essence is transformed into $N := -5/K$, which is more efficient since the negation can be carried out at compile time instead of run time.

We would like to point out that the optimizer is slightly more general than described so far. Each optimization line may contain a number indicating that it requires some special processing. This feature is used to specify replacement parts that are difficult or impossible to express as a pattern. Among these optimizations are bounds checking of subscripts or subrange variables that can be evaluated at compile time, comparisons that can be evaluated at compile time, and branches to other branches or to return instructions. In addition, EM has a full complement of instructions one of whose operands is taken from the stack instead of as an immediate operand; for example, AND 2 has the 2 as an immediate operand, but there is also a form where the size is fetched from the stack. A collection of optimizations exists to replace the more general form by the immediate operand form where possible. Together, the optimizations not listed account for 12.5 percent of all optimizations.

## 6. MEASURED RESULTS

To measure the effect of the peephole optimizer, we have run two tests. In the first we compared the number of machine instructions in each optimized EM program with the number in the unoptimized EM program. Thus, for each program we have a number between 0.00 and 1.00 giving the number of instructions in the optimized program as a fraction of the original. This metric was chosen since it is independent of both the source language and the target machine and directly measures what the optimizer primarily attempts to do, namely, eliminate EM instructions. This metric can also be defended on theoretical grounds. EM code is really just glorified reverse Polish, in other words, the parse tree linearized into postfix order. Removal of an EM instruction typically corresponds to a removal of a node in the parse tree. Since object code size is typically proportional to parse tree size, such removal normally has a direct impact on the final object code size. The measurements presented below bear this out.

The occurrence frequencies per 1000 optimizations are shown in Table III in the column labeled EM. The median saving is 16 percent: one in six EM instructions is eliminated.

The second test consisted of translating the optimized and unoptimized EM code into PDP-11 object code and comparing the number of bytes produced in each case. These results are given in Table III in the column labeled PDP-11. The median reduction in the object code is 14 percent, close to the EM result. This closeness suggests that nearly all the EM optimizations are indeed reflected in the final object code. In two test programs, the optimized PDP-11 code was increased by 2 percent over the unoptimized code due to optimization 50; this was traced to a design error in the (original) EM to PDP-11 back end. (With the optimization the operands were actually stacked, whereas without it they were not.) This defect can easily be fixed, however.

On the basis of these results, we believe peepholing the intermediate code to be worthwhile, since the optimizer need only be written once, for all languages and

Table III. Distribution of Amount of Reduction in Size

| Ratio | EM | PDP-11 |
|---|---|---|
| <0.60 | 0 | 12 |
| 0.60–0.64 | 3 | 6 |
| 0.65–0.69 | 22 | 25 |
| 0.70–0.74 | 24 | 35 |
| 0.75–0.79 | 205 | 71 |
| 0.80–0.84 | 283 | 191 |
| 0.85–0.89 | 298 | 333 |
| 0.90–0.94 | 126 | 181 |
| 0.95–1.00 | 39 | 141 |
| >1.00 | 0 | 4 |
| Total | 1,000 | 999[a] |

[a] Not 1000 due to roundoff.

all machines, and it in no way inhibits additional, more sophisticated optimizers, either on the source code, on the EM code, or on the target code. Moreover, the peephole optimizer is fast: 1140 EM instructions per CPU second on a PDP-11/45 excluding certain overhead not related to peephole optimization and 650 instructions per CPU second including all overhead. This speed was achieved without any special effort to tune the program. It could easily be made faster still by hashing the pattern table instead of examining all patterns starting with the current opcode.

## 7. DISCUSSION

Davidson and Fraser [4] have recently described a peephole optimizer that looks for pairs of consecutive instructions that can be replaced by a single instruction having the same effect. As an example, they note that the PDP-11 sequence SUB #2,R3; CLR (R3) can be replaced by CLR −(R3). In essence, during each compilation the optimizer dynamically "discovers" various pattern/replacement pairs. Their algorithm is so designed that it is complete in the sense that it catches all possible object code pairs that can be reduced to one instruction. In our method, in contrast, the patterns and replacements are determined in advance and looked up in a table during compilation. The table could be constructed using their method of examining all instruction pairs, of course. However, in this case completeness means that all reducible EM pairs, rather than target code pairs, have been eliminated.

The advantage of computing the pattern/replacement pairs in advance instead of on-the-fly is faster performance. They quote an optimization speed of 1 to 10 instructions per second on a PDP-11/70, whereas we have measured a speed of over 1000 instructions per second on the (considerably slower) PDP-11/45. Their use of SNOBOL (versus our use of C) no doubt accounts for part of this two or three order-of-magnitude effect, but even with the same language, computing the table once and for all in advance is surely much faster than rediscovering it piecemeal during compilation. Both our method and theirs require driving tables, as well as code to process them, to be available at run time. The relative sizes of these are too implementation-dependent to make any general remarks about.

Another difference is their decision to optimize the object code versus our decision to optimize the intermediate code. This difference is strongly felt when one is trying to decide whether two sequences are equivalent or not. For example, the PDP-11 instruction ADD #1,R0 is not quite equivalent to (the cheaper) INC R0 because the former sets the "carry" condition code bit and the latter does not. The optimizer must therefore perform some live/dead analysis for each occurrence of ADD #1,R0 to see if the carry bit is used later. If it is, INC cannot be used. Since the intermediate code can be designed to be free of such idiosyncracies, optimizations on it can be done without requiring the live/dead context information that Davidson and Fraser need (cf. line 97 in Table II).

Another research project that has reported work with peephole optimization is Bliss/PQCC at Carnegie-Mellon University [7, 14]. In many cases we have done optimizations similar to theirs, only we have done them entirely on the source- and machine-independent intermediate code, whereas they have largely done them on the (machine-dependent) object code.

In the compiler pass called DELAY, they do constant folding, unary-minus propagation, and reordering using the commutative law. We do similar optimizations in lines 1–21, 122–123, and 67–70, respectively. In the pass called FINAL, they do crossjumping (which we do not presently do, but easily could), branch chain collapsing, null branch removal, and conditional branch reversal, all of which we do do. They also do various special case analyses on the object code that we do on the intermediate code. In addition, FINAL performs certain inherently machine-dependent optimizations, such as manipulating the addressing modes and determining whether to use short or long branches, neither of which is possible on the intermediate code.

Consequently, some machine-dependent optimization may be required in our system too. However, we have never claimed that optimizing the intermediate code eliminates the need for all target code optimization, just that it is desirable to do as much optimization as possible on the intermediate code, because that optimizer can be written once and for all and used without change as a filter for subsequent front ends and back ends.

## REFERENCES


1. ALEXANDER, W.G., AND WORTMAN, D.B. Static and dynamic characteristics of XPL programs. *Computer 8,* 11 (Nov. 1975), 41–46.
2. AMMANN, U. On code generation in a Pascal compiler. *Softw. Pract. Exper. 7,* 4 (June–July 1977), 391–423.
3. COLEMAN, S.S., POOLE, P.C., AND WAITE, W.M. The mobile programming system: Janus. *Softw. Pract. Exper. 41* (Jan.–March 1974), 5–23.
4. DAVIDSON, J.W., AND FRASER, C.W. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst. 2,* 2 (April 1980), 191–202.
5. JOHNSON, S.C. A portable compiler: Theory and practice. In Conf. Rec., 5th Ann. ACM Symp. Principles of Programming Languages, Tucson, Ariz., Jan. 23–25, 1978, pp. 97–104.
6. KNUTH, D.E. An empirical study of FORTRAN programs. *Softw. Pract. Exper. 1,* 1 (Jan.–March 1971), 105–133.
7. LEVERETT, B.W., CATTELL, R.G.G., HOBBS, S.O., NEWCOMER, J.M., REINER, A.H., SCHATZ, B.R., AND WULF, W.A. An overview of the production-quality compiler-compiler project. *Computer 13,* 8 (Aug. 1980), 38–49.
8. MCKEEMAN, W.M. Peephole optimization. *Commun. ACM 8,* 7 (July 1965), 443–444.
9. RICHARDS, M. The portability of the BCPL compiler. *Softw. Pract. Exper. 1,* 2 (April–June 1971), 135–146.

10. RITCHIE, D.M., AND THOMPSON, K. The UNIX time-sharing system. *Commun. ACM 17,* 7 (July 1974), 365–375.
11. STEEL, T.B., JR. UNCOL: The myth and the fact. *Annu. Rev. Autom. Program. 2* (1960), 325–344.
12. TANENBAUM, A.S. Implications of structured programming for machine architecture. *Commun. ACM 21,* 3 (March 1978), 237–246.
13. TANENBAUM, A.S., STEVENSON, J.W., AND VAN STAVEREN, H. Description of an experimental machine architecture for use with block structured languages. Inf. Rapp. 54, Vrije Univ., Amsterdam, 1980.
14. WULF, W., JOHNSSON, R.K., WEINSTOCK, C.B., HOBBS, C.B., AND GESCHKE, C.M. *Design of an Optimizing Compiler.* Elsevier North-Holland, New York, 1975.

Received April 1980; revised January and July 1981; accepted July 1981