A COMPACT, MACHINE-INDEPENDENT PEEPHOLE OPTIMIZER

Christopher W. Fraser

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Abstract

Object code optimizers pay dividends but are usually ad hoc and machine-dependent. They would be easier to understand if, instead of performing many ad hoc optimizations, they performed a few general optimizations that give the same effect. They would be easier to implement if they were machine-independent and parametrized by symbolic machine descriptions. This paper describes such a compact, machine-independent peephole optimizer.

Introduction

Of all optimizations, those applied to object code are among the least-understood. Ad hoc instruction sets complicate elegant treatment and portability, diverting research to higher-level, machine-independent global optimization. However, experience shows the value of object code optimization; even the BLISS11 compiler [Wulf], with thorough global optimization, reduces code size by 15-40% with object code optimization.

Examining compiler design shows why. To be machine-independent, global optimization usually precedes code generation; to be simple and fast, code generators usually operate locally; so the code generator produces, perhaps, locally optimal fragments, but these may look silly when juxtaposed. For example, local code for a conditional ends with a branch; so does local code for the end of a loop. So a conditional at the end of a loop becomes a branch to a branch. Correcting this in the code generator complicates its case analysis combinatorially, since each combination of language features may admit some optimization [Harrison]. It is better to simplify the code generator and optimize object code. Consequently, object code optimization and its problems -- a machine-dependent, informal nature -- deserve attention.

This work was supported in part by the National Science Foundation under contract MCS78-02545.

Background

Little has been published on object code optimization, and many early object code optimizations [Bagwell, Lowry, McKeeman] (eg, constant folding, exponentiation via multiplication) are now performed at a higher level [Allen, Standish]. A notable exception is redundant load elimination, the poor man's global register allocation; many code generators simulate register contents to do this and to replace, where possible, memory references with equivalent register references.

FINAL optimizes the object code generated by the optimizing BLISS11 compiler [Wulf]. FINAL collects several effective but ad hoc optimizations: deleting comparisons that are unnecessary because a previous instruction incidentally set the condition code; exploiting special case instructions and exotic address calculations; coalescing chains of branches; and deleting unreachable code. The optimizer described below complements the more ambitious FINAL by concentrating on one general optimization; it sacrifices a little code quality for simplicity and machine-independence. Comparisons with FINAL quantify this trade-off.

Overview

PO is a compact, machine-independent peephole optimizer. Given an assembly language program and a symbolic machine description, PO simulates pairs of adjacent instructions and, where possible, replaces them with an equivalent single instruction. PO makes one pass to determine the effect of each instruction, a second to reduce pairs, and a third to replace each instruction with its cheapest equivalent.

Compared with conventional object code optimizers, PO is organized in a simple manner and is easily retargetted by changing machine descriptions. Moreover, it is not cluttered by ad hoc case analysis because it combines all possible adjacent pairs, not just branch chains or constant computations or any other special cases. Compared with conventional object code optimizers, PO's effect can be described especially concisely:

> PO replaces each pair of adjacent instructions with an equivalent single instruction, if possible.
>
> PO replaces each instruction with its cheapest equivalent.

Later sections explain "adjacent", "equivalent", and "cheapest".

The two-instruction window catches inefficiencies at the boundaries between fragments of locally-generated code. It misses many others, so PO is best used with a high-level, machine-independent global optimizer.

Machine descriptions

To simulate an instruction, PO must know its syntax and its effect. Examples illustrate the notation, which is based on Bell and Newell's ISP [Bell]. The example below defines one of the PDP11 CLR instructions, which clears ("← 0") the memory cell ("M[...]") addressed by a register ("R[d]", where d is a register index from an instruction field). It also sets the condition code (N and Z bits).

```
CLR @Rd      M[R[d]] ← 0; N ← 0; Z ← 1
```

The first column gives an assembler language syntax pattern with lower case pattern variables ("d") for variable fields. The corresponding pattern in the second column describes the effect of the instruction using these variables. PO assumes that the program counter is automatically incremented, so this needn't be made explicit. Other details irrelevant to the object code (eg, setting the carry bit) are omitted for conciseness.

This second example defines the PDP11 BEQ instruction, which branches ("PC ← 1") if the Z bit is set ("Z → "). Read "→ " as "implies".

```
BEQ 1        Z → PC ← 1
```

PO assumes that "PC" names the machine's program counter.

To simplify the simulator, all register transfers occur simultaneously, so each reference to R[d] refers to its initial value, even if the instruction changes it. Further, all instructions are fully decoded: m×n pairs of patterns are used to define m instructions with n addressing variants each. PO scans the instruction list in order, so the person who describes the machine should decide which instructions are cheapest and put them first.

Since PO knows target machines only through these patterns, it is retargetted by supplying a different instruction set. Its few machine-dependencies are assumptions built into its algorithms and machine description language. For example, the simulator assumes that the machine uses a program counter and that variables, once set, stay set; PO cannot optimize code that uses changing device registers. PO assumes that one instruction is better than two; adding instruction timings to machine descriptions would correct this. Finally, instructions with internal loops (eg, block moves) are hard to describe in the language above. In general, such assumptions are removed by extending PO. As it stands, PO optimizes a useful class of assembly language programs.

Determining the effects of instructions

Initially, PO determines the effect of each assembler statement in isolation (so PO assumes that programs do not modify themselves). Given an assembler statement, the simulator seeks a matching assembler syntax pattern and returns the corresponding register transfer pattern, with pattern variables evaluated. For example, the instruction

```
ADD #2,R3
```

matches the syntax pattern

```
ADD #s,Rd
```

so PO substitutes 2 for s and 3 for d in the register transfer pattern

```
R[d] ← R[d] + s; N ← R[d]+s < 0; Z ← R[d]+s = 0
```

and obtains

```
R[3] ← R[3] + 2; N ← R[3]+2 < 0; Z ← R[3]+2 = 0
```

Programs typically ignore some effects of some instructions. For example, a chain of arithmetic instructions may set and reset condition codes without ever testing them. PO can do a better job if such useless register transfers are removed from an instruction's register transfer list. For example, the full effect of the instruction above includes assignments to the N and Z bits. If the next instruction changes N and Z without testing them, its useful effect is just

```
R[3] ← R[3] + 2
```

If the previous instruction references R[3] indirectly, the useful effect may be had by auto-incrementing instead and removing the ADD instruction (auto-incrementing is a PDP11 address calculation that references indirectly through a register and then increments the register). The full effect requires the ADD instruction, since auto-incrementing does not set the condition code. Consequently, when initially determining each instruction's effect, PO ignores effects on such "dead" variables. To do this, the initial pass scans the program backwards and associates with each instruction its useful effect and a list of variables that are dead from that instruction forward. Each instruction's list is that of its lexical successor, plus the variables it sets, minus the variables it examines. If the instruction branches, its list is just the variables it sets without examining, since those that are dead from the branch on depend on where it jumps. Full dead variable elimination (considering control flow and subscripted variables) [Hecht] is an unnecessary expense; this simpler analysis permits the first pass over the code to eliminate most "extra" effects such as condition code setting. As a bonus, code not subjected to dead variable elimination at a higher level enjoys a measure of it now: instructions with no effect are removed.

Choosing instruction pairs

Once the initial pass determines the isolated effect of each instruction, PO passes forward over the program and considers the combined effect of lexically adjacent instructions; where possible, it replaces such pairs with a single instruction having the same effect. PO learns a pair's effect by combining their independent effects and substituting the values stored in variables in the first for instances of those variables in the second. The effect of

```
ADD #177776,R3
CLR @R3
```

is (ignoring dead variable elimination)

```
R[3] ← R[3] + 177776;
  N ← R[3]+177776 < 0; Z ← R[3]+177776 = 0
M[R[3]] ← 0; N ← 0; Z ← 1
```

which simplifies to

```
R[3] ← R[3] + 177776; M[R[3] + 177776] ← 0;
  N ← 0; Z ← 1
```

PO now seeks a single instruction with a register transfer pattern matching this effect. It finds the auto-decrement version of CLR

```
CLR -(R3).
```

A register transfer pattern matches if it performs all register transfers requested and if the rest of its register transfers set harmless dead variables (eg, the condition code). After each replacement, PO backs up one instruction -- to consider the new adjacency between the new instruction and its predecessor -- and continues.

It is harder to combine pairs that start with a branch. The combined effect of

```
    Z → PC ← L1
    PC ← L2
L1:
```

is

```
    Z → PC ← L1; ~Z → PC ← L2
L1:
```

or just

```
    ~Z → PC ← L2
L1:
```

When PO combines instructions, it treats assignments to the PC as a special case, adding inverted relationals and removing useless assignments to the PC.

Labels prevent the consideration of some pairs. Combining pairs whose second instruction is labelled changes, erroneously, the effect of programs that jump to the label to include the effect of the first instruction. PO must ignore such pairs and assume that all branches are to explicit labels. To improve its chances PO removes any labels it can. When it encounters a label, it looks for a reference to it; if it finds none -- possibly because optimizations like the one above have removed them all -- PO removes the label and tries combining the two instructions that it separated. This enabled PO to remove the last three branches in the large example in the appendix.

When PO removes the last reference to a label that it has passed, it could back up to reconsider the instructions the label separated: new optimizations are possible with the label gone. This happens only with labels referenced after their definition. However, when optimizing code generated locally from a program with "structured" control flow, loop and subroutine heads are the only such labels, and PO seldom removes these. So backup was discarded as an excessive generality.

Branches make extra pairs. If an instruction branches to L, PO simulates it with instruction L and replaces it (leaving L alone) if possible, For example,

```
    BEQ L1
    ...
L1: BR L2
```

has the effect

```
      Z → PC ← L1
      ...
   L1: PC ← L2
```

This combines to

```
      Z → PC ← L2
      ...
   L1: PC ← L2
```

and PO replaces L1 with L2 in the first instruction. Note that the second instruction may now be unreachable. Had the second instruction done

```
   L1: R[3] ← 0
```

the combined effect would have been

```
      Z → R[3] ← 0; Z → PC ← next(L1)
      ...
   L1: R[3] ← 0
```

That is, PO behaves as though the first branch jumped over one extra instruction and the target were conditional on that branch, and then it simulates as before. However, even if there were an instruction with the first effect, PO would not replace the first instruction, because introducing the new label it requires ("next(L1)") complicates other optimizations. PO combines only physically adjacent instructions and branch chains.

Example

The appendices show PO optimizing a program that has been used to illustrate FINAL. Appendix 1 gives the initial code, produced by earlier phases of the BLISS11 compiler for a program that prints trees. Comments guide the user new to the PDP11. In addition to the effects shown, each non-branch sets the condition code according the value it assigns; TSTs set the condition code but do nothing else.

PO optimizes the pairs shown in Appendix 2, in the order given. Each line gives the pair reduced, the resulting instruction, and some explanation. In most cases, it replaces the pair with one equivalent instruction. Three pairs are non-adjacent branch chain members and so only the first instruction is changed; comments note these. Note that, by simply combining adjacent instructions, PO collects branch chains, uses special-purpose addressing modes, combines jumps-over-jumps, and deletes useless TSTs and unreachable code. Appendix 3 gives the result.

BLISS11's FINAL goes one step further with an optimization called "cross-jumping". It changes the last branch to go to L8 instead of L9 and eliminates the second MOV/JSR sequence. This, in turn, admits one last optimization -- the now-adjacent BEQ and BR can be combined into a BNE -- but, by itself, it does not make the program faster, only smaller. Hence, it differs fundamentally from PO's optimizations; even a wider window would not help. Cross-jumping could be added to PO, but the larger need is for a space-optimizer that reduces code size through more general reorderings.

Implementation

PO is a 180-line LISP program that runs in 128K bytes on a PDP11/70. It was developed in three man-weeks. A description of the PDP11 sufficient to optimize all examples in this paper is 40 lines and was written in an hour. PO has also optimized several short PDP10 and IBM360 programs.

PO is experimental, so it still needs thorough documentation, diagnostics, testing and, most of all, optimization. PO treats only 2.5 instructions each second. LISP, which simplified programming, encourages sub-optimal algorithms: ideally, programs would be stored as doubly-linked lists and, for a program of length n, PO would take O(n) steps; using LISP's singly-linked lists requires so many extra implicit passes that PO takes $O(n^2)$ steps. Quick-and-dirty algorithms slow PO even more (eg, PO uses linear search to find instructions in the machine description) and hide some machine-dependencies (eg, PO doesn't know that some multiplications can be done by shifting). These problems seem simple, and dramatic speedups seem possible, but further development is needed.

Conclusions

The success with PO suggest re-examining the division of labor between the global optimizer, code generator and object code optimizer. For example, the easy availability of a peephole optimizer may simplify code generators: they might produce only load/add-register sequences for additions and rely on a peephole optimizer to, where possible, discard them in favor of add-memory, add-immediate or increment instructions. Experiments underway indicate that a very naive code generator can give good code if used with PO.

Global optimizers might also be simplified: since PO eliminates (much) unreachable code anyway, should global optimizers bother? Perhaps other global optimizations should be pushed down to the object code level: register transfers resemble quadruples; perhaps a machine-independent, global optimizer [Hecht] could be adapted to take a more global view of object code and so catch inefficiencies missed by PO's narrow window.

Appendix 1. Tree printer (with thanks to Elsevier Publishing)

```
1)                  JSR     R1,SAV3          # call SAV3
2)                  MOV     S+310,R3         # R[3] ← M[S+310]
3)                  MOV     12(R5),R2        # R[2] ← M[R[5]+12]
4)                  ADD     #177776,R3       # R[3] ← R[3] - 2
5)                  CLR     @R3              # M[R[3]] ← 0
6)        L5:L6:    TST     LEFT(R2)         # test M[R[2]+LEFT]
7)                  BNE     L7               # ~Z → PC ← L7
8)                  BR      L8               # PC ← L8
9)        L7:       ADD     #177776,R3       # R[3] ← R[3] - 2
10)                 MOV     R2,@R3           # M[R[3]] ← R[2]
11)                 MOV     LEFT(R2),R2      # R[2] ← M[R[2]+LEFT]
12)                 BR      L6               # PC ← L6
13)       L8:       MOV     INFO(R2),R1      # R[1] ← M[R[2]+INFO]
14)                 JSR     R7,PRINT         # call PRINT
15)       L9:       MOV     RIGHT(R2),R2     # R[2] ← M[R[2]+RIGHT]
16)                 TST     R2               # test R2
17)                 BEQ     L10              # Z → PC ← L10
18)                 BR      L11              # PC ← L11
19)       L10:      MOV     @R3,R2           # R[2] ←  M[R[3]]
20)                 ADD     #2,R3            # R[3] ←  R[3] + 2
21)                 TST     R2               # test R2
22)                 BNE     L12              # ~Z → PC ← L12
23)                 BR      L13              # PC ← L13
24)       L12:      MOV     INFO(R2),R1      # R[1] ← M[R[2]+INFO]
25)                 JSR     R7,PRINT         # call PRINT
26)                 BR      L14              # PC ← L14
27)       L13:      BR      L4               # PC ← L4
28)       L14:      BR      L9               # PC ← L9
29)       L11:      BR      L5               # PC ← L5
30)       L4:       RTS     R7               # return
```

Appendix 2. Pair-wise optimizations on tree printer

```
a) 4,5              CLR     -(R3)            # use auto-decrement
b) 7,8              BEQ     L8               # remove label L7
c) 9,10             MOV     R2,-(R3)         # use auto-decrement
d) 15,16  L9:       MOV     RIGHT(R2),R2     # remove TST
e) 17,18            BNE     L11              # remove label L10
f) e,29             BNE     L5               # remove label L11, retain 29
g) 19,20            MOV     (R3)+,R2         # use auto-increment
h) g,21             MOV     (R3)+,R2         # remove TST
i) 22,23            BEQ     L13              # remove label L12
j) i,27             BEQ     L4               # remove label L13, retain 27
k) 26,27            BR      L14              # 27 unreachable without L13
l) k,28             BR      L9               # remove label L14, retain 28
m) 1,28             BR      L9               # 28 unreachable without L14
n) m,29             BR      L9               # 29 unreachable without L11
```

Appendix 3. Optimized tree printer

```
1)                  JSR     R1,SAV3          # call SAV3
2)                  MOV     S+310,R3         # R[3] ← M[S+310]
3)                  MOV     12(R5),R2        # R[2] ← M[R[5]+12]
4)                  CLR     -(R3)            # M[R[3]-2] ← 0; decrement R[3]
5)        L5:L6:    TST     LEFT(R2)         # test M[R[2]+LEFT]
6)                  BEQ     L8               # Z → PC ← L8
7)                  MOV     R2,-(R3)         # M[R[3]-2] ← R[2]; decrement R[3]
8)                  MOV     LEFT(R2),R2      # R[2] ← M[R[2]+LEFT]
9)                  BR      L6               # PC ← L6
10)       L8:       MOV     INFO(R2),R1      # R[1] ← M[R[2]+INFO]
11)                 JSR     R7,PRINT         # call PRINT
12)       L9:       MOV     RIGHT(R2),R2     # R[2] ← M[R[2]+RIGHT]
13)                 BNE     L5               # ~Z → PC ← L5
14)                 MOV     (R3)+,R2         # R[2] ← M[R[3]]; increment R[3]
15)                 BEQ     L4               # Z → PC ← L4
16)                 MOV     INFO(R2),R1      # R[1] ← M[R[2]+INFO]
17)                 JSR     R7,PRINT         # call PRINT
18)                 BR      L9               # PC ← L9
19)       L4:       RTS     R7               # return
```

References

[Allen] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, Design and Optimization of Compilers, 1-30. Prentice-Hall, 1972.

[Bagwell] J. T. Bagwell. Local optimizations. SIGPLAN Notices 5(7):52-66, July 1970.

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

[Harrison] W. Harrison. A new strategy for code generation -- the general purpose optimizing compiler. POPL 4:29-37, 1977.

[Hecht] M. S. Hecht. Flow Analysis of Computer Programs. North-Holland, 1977.

[Lowry] E. S. Lowry and C. W. Medlock. Object code optimization. CACM 12(1):13-22, January 1969.

[McKeeman] W. M. McKeeman. Peephole optimization. CACM 8(7):443-444.

[Standish] T. A. Standish, D. C. Harriman, D. F. Kibler and J. M. Neighbors. The Irvine program transformation catalogue. Dept. of Information and Computer Science, UC Irvine, 1976.

[Wulf] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke. The Design of an Optimizing Compiler. American Elsevier, 1975.