Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction
SIGPLAN Notices Vol. 19, No. 6, June 1984

Automatic Generation of Peephole Optimizations†

Jack W. Davidson
Dept. of Applied Mathematics and Computer Science
University of Virginia
Charlottesville, VA 22901

Christopher W. Fraser
Dept. of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper describes a system that automatically generates peephole optimizations. A general peephole optimizer driven by a machine description produces optimizations at compile-compile time for a fast, pattern-directed, compile-time optimizer. They form part of a compiler that simplifies retargeting by substituting peephole optimization for case analysis.

1. Introduction

Code generators often create inefficient juxtapositions. For example, incrementing and testing a variable can create a redundant comparison if the code for the increment automatically sets a condition code register. Correcting this in the code generator complicates case analysis combinatorially, since each combination of language features may generate a unique juxtaposition [9]. It is often cheaper to generate code locally and then use a peephole optimizer to improve inefficient juxtapositions. Peephole optimization typically reduces code size by 10-50% [14, 17]. Even the new code generators driven by machine descriptions [6] benefit from peephole optimization [2].

Classical peephole optimizers [1, 14, 15, 17] rapidly correct a few hand-written, machine-specific patterns. For example, the ambitious "FINAL" optimizer in the BLISS-11 compiler [17] deletes unnecessary comparisons, exploits special-case instructions and exotic addressing modes, coalesces chains of branches, and deletes unreachable code. Unfortunately, good patterns can be hard to identify and are language-, compiler-, and machine-specific.

A recent alternative to classical peephole optimizers [3] uses a machine description to simulate adjacent instructions, replacing them, wherever possible, with an equivalent singleton. Such machine-directed optimizers use no patterns, so they are more thorough and portable than their classical counterparts, but they are slower. Their thoroughness allows the use of naive, easily retargeted code generators, but verbose code makes optimization speed even more crucial.

This paper describes a system that automatically generates patterns for a fast classical peephole optimizer. A modern machine-directed optimizer is run at *compile-compile* time, and patterns for a fast, classical *compile-time* peephole optimizer are automatically inferred from its output. This combines the thoroughness and retargetability of a machine-directed peephole optimizer with the speed of a classical peephole optimizer. This has sped up the peephole optimization phase of a retargetable compiler by a factor of five.

2. A Machine-Directed Optimizer

The system uses a retargetable peephole optimizer called PO. Other documents elaborate on PO itself [3, 4]; this paper summarizes it only enough to introduce a new application: generating patterns for a fast, classical peephole optimizer.

†This work was supported in part by the National Science Foundation under Grant MCS-7802545.

Given an assembly language program and a symbolic machine description, PO simulates adjacent instructions and, where possible, replaces them with an equivalent single instruction. Each machine description is a grammar for syntax-directed translation between assembly language and register transfers. For example, the production

```
movl src,dst  :=:  dst = src;  NZ = src ? 0;
```

describes the VAX movl instruction, which copies its first operand onto its second and sets the condition code to reflect the sign of the result. Similar productions describe addressing modes.

To improve an instruction, PO must know its effect, that is, the register transfers that it performs. Early versions of PO computed effects by matching assembler instructions against the assembler syntax patterns above and instantiating the corresponding register transfer patterns. The most recent version skips this with a compiler that emits register transfers directly. Register transfers are no harder to emit than assembly code.

Once PO has the effect of each instruction, it symbolically simulates two- and three-instruction sequences to form their combined effect. PO then searches the machine description for an instruction with this combined effect. If it finds one, it replaces the original instructions with the new one. For example, the effects of the VAX instructions

```
movl X,r1
subl2 Y,r1
```

are

```
r[1] = m[X];  NZ = m[X] ? 0;
r[1] = r[1] - m[Y];  NZ = r[1] - m[Y] ? 0;
```

Symbolic simulation combines these to yield

```
r[1] = m[X] - m[Y];  NZ = m[X] - m[Y] ? 0;
```

which is realized by the instruction

```
subl3 Y,X,r1
```

so this instruction replaces the two above.

Unlike classical peephole optimizers, PO *has no patterns*: it combines *all* possible pairs and triples. As a result, its effect can be described formally and concisely: when it is finished, no one-, two-, or three-instruction sequence can be replaced with a cheaper single instruction having the same effect. This thoroughness allows code generators to forgo case analysis and emit only a small subset of the machine's instructions and addressing modes (e.g., one form of add, one form of subtract). PO replaces them with better instructions as it combines adjacencies. A compiler for the programming language Y [8] based on this technique [4, 5] has been retargeted to seven different architectures, some in as few as three man-days. It emits code comparable to host-specific compilers.

This reliance on peephole optimization makes optimization speed especially crucial, and PO is slower than classical target-specific peephole optimizers. The Y compiler runs at a fourth the speed of the UNIX portable C compiler [10], and PO uses almost half of its time. Proposals to speed up optimizers like PO are already emerging† [7, 11, 12]. They propose to perform at compile-compile time some of the symbolic simulation that PO performs at compile time. This entails considering at compile-compile time all possible pairs of instructions [12] or all that use certain rules (like "eliminate redundant instructions" [7, 11]). Naturally, trade-offs appear likely — the first approach may be costly on some machines, the second may miss optimizations, and both may generate unused optimizations — though the proposals certainly merit further investigation. The software described below complements these approaches by automatically inferring patterns from PO's behavior on sample data.

3. Automatic Generation of Patterns

To improve speed, PO is now used at compile-compile time to generate patterns for a fast compile-time optimizer, called HOP, which may then be used in PO's place. HOP patterns are encoded as text with embedded pattern variables of the form $\$i$ to denote context-sensitive operands. Thus the pattern

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$3]
=
r[$1] = m[$2] - m[$3]
```

specifies that register transfers like

```
r[2] = m[X]
r[2] = r[2] - m[Y]
```

should be replaced with

```
r[2] = m[X] - m[Y]
```

Other classical peephole optimizers use similar encodings [14, 16]. An appendix gives further examples of such optimizations and their application.

†Only one of these proposals reports a prototype [12]. It is more powerful than an early version of PO, though not the current version. It considers $O(N^2)$ pairs to PO's $O(N)$, and, though it appears likely that adaptations could run in linear time, it is too early to compare their speed with PO's.

HOP patterns are inferred from PO's behavior on a "training" set. As an option, PO can record each replacement it makes. For example, when PO makes a replacement like the one above, it writes

```
r[2] = m[X]
r[2] = r[2] - m[Y]
=
r[2] = m[X] - m[Y]
```

to a diagnostic file.

This output is automatically reduced to patterns by replacing each distinct assembly-time constant with $\$i$. For example, the diagnostic output above would become

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$3]
=
r[$1] = m[$2] - m[$3]
```

which is the pattern at the head of this section. The syntax of assembly-time constants is potentially target-specific. HOP is retargeted by specifying this syntax.

PO records the last use of each register in each block, because this allows it to make replacements that would otherwise change the effect of the program. When this information is used, it is also recorded in the diagnostic output:

```
r[2] = i
r[3] = m[r[2]]  (r[2] dead)
=
r[3] = m[i]
```

These "obituaries" are automatically reduced to patterns with the rest of the diagnostic output. Thus the example above yields the pattern

```
r[$1] = $2
r[$3] = m[r[$1]]  (r[$1] dead)
=
r[$3] = m[$2]
```

The appendix displays several such optimizations.

A few proposed patterns are too general. For example, the DECSystem-10 diagnostic output

```
r[2] = m[X]
r[2] = r[2] + 1
m[X] = r[2]  (r[2] dead)
=
m[X] = m[X] + 1
```

should not yield the pattern

```
r[$1] = m[$2]
r[$1] = r[$1] + $3
m[$2] = r[$1]  (r[$1] dead)
=
m[$2] = m[$2] + $3
```

because the replacement is only valid if the increment $3 is 1. The validity of proposed patterns like the one above could be checked with the machine description much as PO checks proposed combinations of instructions. When the instruction checker determined that $3 could only match 1, it could rewrite the pattern accordingly. At present, a simpler expedient is used: constants like zero and one that are special to some instructions (i.e., that appear explicitly in the machine description) are added to an exception list and never replaced with $\$i$. This generates a few extra patterns when these constants appear in contexts where they are not special (e.g., as register indices), but the number of these is small.

Given the established simplicity of typical programs [13], compiling a large, varied "training" testbed with PO should yield enough diagnostic output to generate most needed patterns. At present, the testbed is the Y compiler's front end, which compiles Y into a simple abstract machine code, plus a few extra test cases, which exercise the few operators seldom used in the compiler. Figure 1 plots for this testbed the number of VAX patterns generated versus the number of actual replacements from which the patterns are generated. The pattern file grows rapidly at first and then levels off. The 17,138 replacements generate only 627 distinct patterns. Using this pattern file, HOP yields the same result as PO when compiling routines from the testbed. When compiling other typical routines, HOP's results are only about 2% larger than PO's, which suggests that even this small testbed is adequate.

Ultimately, it should be possible to do without a testbed, by using an incremental training phase. This could be implemented by the following changes to PO. After replacing a pair or triple, PO would internally record the pattern represented by the replacement; if the pair or triple could not be replaced, PO would note this as well. Also, PO would be changed to consult this record and use the fast algorithm described below to replace or reject juxtapositions that have appeared before; it would fall back on its original, slower algorithm only for juxtapositions that had never appeared before. Thus PO would reach HOP's speed after a few compilations, and it would never miss an optimization due to insufficient training because PO's general mechanism would be available for new juxtapositions.

4. A Pattern-Directed Optimizer

HOP matches patterns without actual string manipulation, by separating each instruction's pattern or "skeleton" from its operands as it reads them. This is accomplished at compile time by the same procedure used to form patterns at compile-compile time. For example, the instruction

```
r[2] = r[2] - m[Y]
```

is reduced to the skeleton

```
r[$1] = r[$1] - m[$2]
```

plus the operands 2 and Y, respectively. That is, the instruction is represented by the triple

```
r[$1] = r[$1] - m[$2], 2, Y
```

This representation is a little like conventional assembly code. The skeleton in the first field is determined roughly by the instruction's opcode and mode bits. The operands in the remaining fields are determined roughly by the instruction's address and register fields.

Hashing helps HOP match patterns and form replacements fast. HOP stores skeletons and operands uniquely in a hash table, so an input skeleton is compared with a line from a pattern by merely comparing two addresses. This operation is logically similar to, and costs about the same as, comparing two binary opcodes in a classical peephole optimizer. If a run of input skeletons matches some complete pattern, then inter-instruction operand consistency is checked, again by comparing addresses. Finally, HOP forms replacements without actual string manipulation. The skeleton for the replacement instruction is the last line of the successful pattern, and the operands for the replacement instruction are formed by reordering the input operands. Thus the typical pattern is matched and, if successful, replaced, by comparing and moving about a dozen pointers.

One detail complicates this procedure. The $\$i$ in input skeletons are numbered from one, so pattern-matching without string operations requires renumbering the $\$i$ from each line of each pattern when the pattern file is read. For example, the input

```
r[4] = m[A]
r[4] = r[4] - m[B]
```

is translated into the triples

```
r[$1] = m[$2], 4, A
r[$1] = r[$1] - m[$2], 4, B
```

as it is read. To compare such triples with the pattern

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$3]
=
r[$1] = m[$2] - m[$3]
```

without string operations, the $\$i$ of the second line of the pattern are renumbered to yield

```
r[$1] = r[$1] - m[$2]
```

as the pattern file is read. The two strings are now identically equal and can be compared by comparing addresses in the hash table. A record of the renumbering is retained for checking inter-instruction operand consistency.

The input triples above are compared with the pattern above as follows. First, the two input skeletons

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$2]
```

are compared with the first two (renumbered) lines of the pattern

```
r[$1] = m[$2]
r[$1] = r[$1] - m[$2]
```

by comparing two pairs of pointers. Next, HOP checks that $\$i$ denotes the same operand in both input instructions. Since $1 is the only $\$i$ that appears more than once in the original (unrenumbered) pattern†, this merely compares the first operand from the first instruction (the first 4) with the first operand from the second instruction (the second 4), again by comparing two string table addresses. Since all comparisons have succeeded, a replacement instruction is formed. Its skeleton is the last line of the pattern

```
r[$1] = m[$2] - m[$3]
```

and its three operands are the 4 and A from the first instruction and the B from the second instruction. This represents the instruction

```
r[4] = m[A] - m[B]
```

which is the desired replacement for the two instructions above.

Hashing also helps locate applicable patterns rapidly. HOP stores its patterns in a hash table keyed by the hashed addresses of the (uniquely stored) skeletons that each matches. Thus HOP identifies the patterns that apply to a given input sequence by hashing the addresses of the skeletons from the input

†$2 appears more than once in the *renumbered* pattern, but this is an artifact of renumbering and so does not require consistency checking.

sequence. If this hash table is made large enough to make collisions rare, HOP identifies any applicable patterns in nearly constant time.

These measures make HOP fast, about 5 times faster than PO. In a typical application, it read 269 lines, performed 136 replacements, and wrote out the results in 1.3 CPU seconds on a VAX-11/780. It spends most of its time reading its input and building the structures above. The actual matching and replacements take less than 5% of its time. To save time, the pattern file is incorporated into HOP at compile-compile time. For the VAX, HOP plus these incorporated patterns take 150K bytes where PO takes 120K bytes.

HOP can also be used for code generation. Abstract machines are often mapped onto real machines by macros, and single-input replacement patterns are essentially macros. A compiler can thus be retargeted by writing a machine description and some patterns for naive code generation. These will be augmented by automatically generated optimization patterns. The use of a single program for code generation and optimization should make compilers faster, simpler, and easier to retarget.

HOP can also be used on assembly code. The hand-written patterns for code generation could emit assembly code, for this can be mapped to and from register transfers for PO by translators automatically generated from the machine description [3]. Translating assembly code to register transfers would slow PO, but this is unimportant now that HOP has replaced PO at compile time.

Acknowledgments

The authors thank Dave Hanson for his many helpful comments, and Torben Nielsen for his technical assistance.

Appendix

This appendix traces the optimization of the VAX code for

```
j = i + 4
```

The figure below gives postfix intermediate code and corresponding naive object code for this statement.

```
    postfix      object code

1.  push i       r[2] = m[i]
2.  pushc 4      r[3] = 4
3.  add          r[2] = r[2] + r[3]  (r[3] dead)
4.  pop j        m[j] = r[2]  (r[2] dead)
```

Initially, the pattern

```
r[$1] = $2
r[$3] = r[$3] + r[$1]  (r[$1] dead)
=
r[$3] = r[$3] + $2
```

replaces instructions 2 and 3 with

```
r[2] = r[2] + 4
```

Next, the pattern

```
r[$1] = m[$2]
r[$1] = r[$1] + $3
=
r[$1] = m[$2] + $3
```

combines instruction 1 with this new instruction, yielding

```
r[2] = m[i] + 4
```

Finally, the pattern

```
r[$1] = m[$2] + $3
m[$4] = r[$1]  (r[$1] dead)
=
m[$4] = m[$2] + $3
```

replaces this last instruction and instruction 4 with

```
m[j] = m[i] + 4
```

which represents the VAX instruction

```
addl3 $4,i,j
```

Thus the four original instructions have been replaced with one.

References

1. J. T. Bagwell, Jr., Local Optimizations, *SIGPLAN Notices 5*, 7 (July 1970), 52-66.
2. T. Crowley, Combining Table-driven Effect Selection and Description-Driven Peephole Optimization for Automatic Code Generation, MS thesis, MIT, September 1982.
3. J. W. Davidson and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, *ACM Trans. Prog. Lang. and Systems 2*, 2 (April 1980), 191-202.
4. J. W. Davidson, *Simplifying Code Generation Through Peephole Optimization*, PhD dissertation, University of Arizona, December 1981.
5. J. W. Davidson and C. W. Fraser, Code Selection Through Object Code Optimization, *ACM Trans. Prog. Lang. and Systems*, to appear.

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

7. R. Giegerich, A Formal Framework for the Derivation of Machine-Specific Optimizers, *ACM Trans. Prog. Lang. and Systems 5*, 3 (July 1983), 478-498.

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

9. W. Harrison, A New Strategy for Code Generation - The General Purpose Optimizing Compiler, *Conf. Rec. 4th ACM Symp. on Prin. of Programming Languages*, January 1977, 29-37.

10. S. C. Johnson, A Portable Compiler: Theory and Practice, *Conf. Rec. 5th ACM Symp. on Prin. of Programming Languages*, Jan. 1978, 97-104.

11. P. B. Kessler, Machine Dependencies in Retargetable Compiler Construction, Dissertation proposal, Department of Electrical Engineering and Computer Science, University of California, Berkeley, May 1982.

12. R. R. Kessler, Peephole Optimization in COG, Operating Note 76, Utah Symbolic Computation Group, Computer Science Department, University of Utah, June 1983.

13. D. E. Knuth, An Empirical Study of Fortran Programs, *Software—Practice & Experience 1*, 2 (April-June 1971), 105-133.

14. D. A. Lamb, Construction of a Peephole Optimizer, *Software—Practice & Experience 11*(1981), 638-647.

15. W. M. McKeeman, Peephole Optimization, *Comm. ACM 8*, 7 (July 1965), 443-444.

16. A. S. Tanenbaum, H. van Staveren and J. W. Stevenson, Using Peephole Optimization on Intermediate Code, *ACM Trans. Prog. Lang. and Systems 4*, 1 (January 1982), 21-36.

17. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, North Holland, 1975.

Figure 1: VAX Pattern File Growth