Integrating Code Generation and Optimization†

Christopher W. Fraser
Alan L. Wendt

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

Abstract

This paper describes a compiler with a code generator and machine-directed peephole optimizer that are tightly integrated. Both functions are performed by a single rule-based rewriting system that matches and replaces patterns. This organization helps make the compiler simple, fast, and retargetable. It also corrects certain phase-ordering problems.

Introduction

Many code generation and optimization phases are simply pattern matchers: code generators match patterns in intermediate code and replace them with object code; common subexpression eliminators seek repeated expressions and replace them with register references; peephole optimizers match patterns in object code and replace them with more efficient object code.

A compiler exploiting these observations is being built. The code generation and peephole optimization phases have been subsumed by a single, more general rewriting system that is driven by rules that match and replace patterns. One set of rules generates naive code, and another — which is generally generated automatically at compile-compile time — optimizes this code as soon as it is generated.

Most compilers implement these phases separately, but integrating them has made this compiler simpler, faster, and better able to yield good code. It is simpler because one general-purpose phase replaces two special-purpose phases. It can yield better code because certain phase-ordering problems disappear. Phase-ordering problems occur when one phase generates an inefficiency of a kind that is only corrected by an earlier phase. They simply cannot occur when the two phases are tightly integrated.

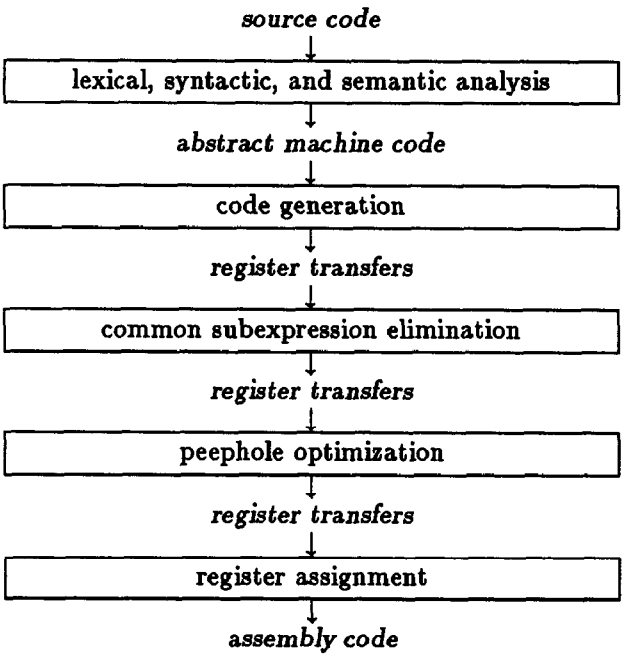
The compiler is faster because it no longer has one phase dismantle its structures and output them only to have another phase read and create an often similar structure. These particular phases — code generation and peephole optimization — can spend a large fraction of their time doing i/o and building and dismantling their structures, so shareable structures turn out to be particularly important.

This project began with a rule-directed peephole optimizer and generalized it to assume responsibility for code generation as well. In many cases, generalizing a program slows it down, but this particular generalization improved performance. For example, the version of the new program that was most comparable with the original rule-directed optimizer took no longer than that optimizer alone, so code generation was "free". This apparent anomaly is explained by the fact that the original optimizer spent much more time reading its input and building its structures than it did matching and replacing patterns. The generalized optimizer reads a shorter, more regular input, so it can apply more rules in the same amount of time.

The compiler is retargetable. Nearly all of its machine-dependencies are isolated in a largely nonprocedural machine description and rule database, not code. The code generation rules are written by hand, but this task is simplified by the absence of case analysis. The necessity of writing these rules is offset by the fact that the required machine descriptions are short enough — generally 2-4 pages — to make the method described herein competitive with other current methods for retargeting compilers [1, 3, 4, 8], while also implementing both code generation and peephole optimization as special cases of one more general process.

†This work was supported in part by the National Science Foundation under Grant DCR-8320257.

Figure 1



Background

Modern machine-directed peephole optimizers [5, 9, 11, 12] are based on a process similar to symbolic simulation. They represent instructions using "register transfers", which are simple expressions and assignments involving the machine's operators and cells. For example, one of these optimizers, PO, represents an instruction that loads the address of x into register 1 as

```
r[1] = x
```

and an instruction that loads register 1 with the word at which it had pointed as

```
r[1] = m[r[1]]
```

On most machines, this pair of instructions can be usefully replaced with one. Symbolic simulation is used to compute their combined effect

```
r[1] = m[x]
```

and then a machine description is searched to determine if the combined register transfers represent a valid instruction on the given machine. Different machine-directed optimizers differ on whether to perform simulation at compile time or compile-compile time; on what juxtapositions are considered for combination; and on whether to seek the output instructions equivalent to some given input instructions or vice versa. All, however, use register transfers and a process similar to symbolic simulation.

This organization makes machine-directed optimizers thorough. When PO, for example, is finished, no one-, two-, or three-instruction sequence can be replaced with a cheaper singleton. This thoroughness allows code generators to emit naive code, because the peephole optimizer can automatically perform much case analysis. For example, one compiler based on PO [7] is organized as shown in Figure 1. The front end emits naive abstract machine code, which the code generator translates into naive target machine code, represented as register transfers. The code generator assumes an infinite register set, which simplifies code generation and postpones register assignment until optimization has reduced the requirements for registers. The third phase eliminates common subexpressions and performs dataflow analysis for the fourth phase, PO. The last phase maps the registers still used in the optimized code down onto the actual registers, introducing spills and reloads as necessary. It also uses the machine description to map the register transfers to the equivalent assembly code.

This compiler has been retargeted to at least 10 different architectures, some in as few as three man-days. It emits code comparable to target-specific compilers, but it is slower: it takes about 4 times as long as PCC [10], partly because PO performs symbolic simulation at compile time.

To speed up the compiler, PO has been augmented to generate rules that describe the optimizations that it has made [6]. A fixed set of rules is generated at compile-compile time and loaded into a fast, rule-directed, compile-time optimizer called HOP. This procedure reduces the time for peephole optimization by about 80%, and it reduces overall compile time by about one third.

HOP runs fast by avoiding much of the string scanning traditionally associated with pattern matching. Its rules are encoded as text, with embedded pattern variables of the form %*n* to denote context-sensitive operands. For example, the rule

```
r[%1] = %2
r[%1] = m[r[%1]]
=
r[%1] = m[%2]
```

describes the optimization above. Rules are generated automatically by replacing each distinct constant in an optimization trace with a unique pattern variable. For example, the rule above was automatically generated by generalizing the optimization trace that began this section.

HOP also represents instructions using patterns. For example, it represents the instruction

```
r[1] = x
```

with the pattern

```
r[%1] = %2
```

plus a record that %1 denotes 1 and %2 denotes x. That is, when it reads the instruction above, it immediately "patternizes" it and henceforth represents it with the tuple

```
r[%1] = %2, 1, x
```

This representation gives HOP two important ways to achieve speed. First, HOP uses a hash table to store exactly one copy of any given string. Thus the representation for instructions above allows HOP to compare instructions with patterns by simply comparing two hash addresses. Second, HOP stores rules in another hash table keyed by the patterns to which the rule applies. Thus when HOP is presented with some instructions to combine, it merely adds up the (unique) addresses of the instruction patterns, discards the high-order bits, and uses the result to index the rule table. If the rule table is large enough to make collisions rare, this mechanism identifies any applicable rule quickly. HOP thus locates and applies optimizations in nearly constant time, without character scanning or recursive tree traversal.

Combining Phases

HOP's speed and generality make it natural to try to use the same techniques to improve other phases as well. This observation is being exploited in a new compiler.

The front end of the new compiler is from PCC, the front end of which was adapted to yield code for a RISC-like stack machine that is tailored to C but is machine-independent. The code is little more than a postfix encoding of the front end's syntax tree, so it is easy to generate. The postfix code requires neither allocation of registers nor packing of operands into instructions, and the RISC design makes case analysis moot.

The combined code generator and optimizer is a general rule-based rewriting system that matches patterns and substitutes new text for them. Some rules implement code generation by replacing intermediate code with machine instructions represented as register transfers; other rules implement peephole optimization by replacing juxtaposed instructions with one; still other rules translate the optimized register transfers to assembly code.

Logically, the back end creates a tree from its postfix input and applies rules that rewrite pieces of the tree. The hand-written code generation rules generate naive code, so few need much context and most need only match a single input node. The automatically-generated optimization rules, however, match pairs of instructions, so they match nodes on linear paths down through the tree. For example, consider the rule

```
a
b
=
c
```

Since code appears in post-order, the rewriting system applies this rule at a node *b* if *b* has a child equal to a. If it finds a match, it deletes a, replaces *b* with c, reattaches all the other children of the input nodes as children of the new node, and then iterates to test for further optimizations. The rules can only descend along a single path, so a simple, non-recursive tree-matching algorithm suffices.

The rewriting system extends HOP. Formerly, HOP could miss optimizations when a new program used a juxtaposition of instructions that had not been seen when the rules were generated at compile-compile time. The new compiler corrects this difficulty by integrating HOP and PO. Now HOP's rule cache is extended incrementally by calling on PO to generate rules that replace or reject previously unseen juxtapositions. The rules are now generated at compile time, but no rule generated earlier need ever be regenerated, so the effect is nearly that of generation at compile-compile time. Thus the integrated system optimizes code as thoroughly as PO, but it is not appreciably slower than HOP.

The rewriting system's rules are like HOP's, but now they may call "built-in" routines to perform operations that cannot be conveniently implemented as pattern matching and substitution [13]. For example, the rule

```
r[%1] = r[%1] * %2
LOG %2 %3
=
r[%1] = r[%1] << %3
```

calls the routine LOG to bind %3 to the log_2 of the number that is bound to %2. If %2 does not denote a number, or if that number is not a power of two, then LOG "fails", which causes the rule to fail just as if a pattern had failed to match. Rules can call any routine that has been link-edited into the rewriting system.

Built-in routines are used extensively by the rules that control code generation. For example, the rule that generates the naive code to add two integers is logically equivalent to

```
IntegerAdd
=
BINOP r[%0] = r[%1] + r[%2]
```

BINOP is a built-in routine that replaces the opcode in the current node (IntegerAdd) with the given register transfer pattern (r[%0] = r[%1] + r[%2]), and binds %0 and %1 to the result register from the node's first child and %2 to the result register from the node's second child.

This is naive code: the code generation rules use only register-to-register addition, because the automatically generated optimization rules are responsible for combining these instructions with their neighbors to better exploit the instruction set. The compiler generates naive code temporarily but optimizes it as much as possible before moving on to another point in the tree. Once a code generation rule has been applied, the node holds register transfers and not intermediate code, and thus optimization rules and not the code generation rules will be the ones that will happen to apply.

Consider the compilation of the source language statement

```
i = i + 1;
```

for the VAX. The front end produces the syntax tree below, where indentation displays the tree structure.

```
IntegerStore
  Address i
  IntegerAdd
    IntegerFetch
      Address i
    IntegerConstant 1
```

The rewriting system does a post-order traversal. The first three rules that apply are merely code generation rules that rewrite one node each. They yield the tree below, in which register transfer patterns have been instantiated to simplify reading.

```
IntegerStore
  r[2] = i
  IntegerAdd
    r[3] = m[r[3]]
      r[3] = i
    IntegerConstant 1
```

Next, the first optimization rule fires, combining the load-indirect with its child and yielding

```
IntegerStore
  r[2] = i
  IntegerAdd
    r[3] = m[i]
    IntegerConstant 1
```

The next two rules generate code for the opcodes IntegerConstant and IntegerAdd, yielding

```
IntegerStore
  r[2] = i
  r[3] = r[3] + r[4]
    r[3] = m[i]
    r[4] = 1
```

Now one optimization rule combines the two instructions that set r[3], using one of the VAX's three-operand instructions, and yielding

```
IntegerStore
  r[2] = i
  r[3] = m[i] + r[4]
    r[4] = 1
```

and another combines the result with the instruction that loaded r[4], yielding

```
IntegerStore
  r[2] = i
  r[3] = m[i] + 1
```

Next, the code generation rule for IntegerStore fires, yielding

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

Then optimization rules yield

```
m[i] = r[3]
  r[3] = m[i] + 1
```

and finally

```
m[i] = m[i] + 1
```

The examples above accurately trace the compiler's sequence of tree replacements, but for clarity they omit one implementation detail: the code generation rules, in addition to their functions displayed above, also *construct* the tree from the postfix input as they go. For example, the actual rule that generates VAX code for integer adds is:

```
IntegerAdd
=
POP 2
PUSH r[%0] = r[%1] + r[%2]
```

The built-in routines PUSH and POP maintain a stack to recreate a tree similar to the one created by the front end. The front and back ends are normally tightly coupled as one program, but using a postfix encoding of the trees — rather than the actual tree structures themselves — for inter-module communication allows separating the front and back ends into two separate programs, which is occasionally helpful during development. Ultimately, this step will no longer be needed, and the back end will use tree structures created by the front end.

Because code generation and optimization have been integrated, it is no longer necessary to assume an infinite set of pseudo-registers. At present, each new node tries to take the result register of one of its children as its own. (This policy is likely to change when the rewriting system is extended to subsume common subexpression elimination as well.) If this is not possible — and it never is for leaves because they have no children — the built-in procedure gets its result register from a list of free registers. In the presence of separate register classes, the built-in would accept an extra argument to define the type of register that is required by the instruction that is now the only argument to the built-in.

A mechanism for handling spills has been designed. Consistent with the policy of emitting naive code and then improving it automatically, the register allocator generates more spills than may be needed, but automatically removes those that subsequent optimization exposes as unnecessary. When the built-in needs a register and none is available (perhaps because it is a leaf), it effectively spills the most distantly used register. The busy registers are the result registers of the trees on the stack. The deeper a result register is in the stack, the longer it will be before it is needed again, so the built-in spills the result register from the deepest stack entry that has not already been spilled.

Because subsequent optimization may eliminate the need for this register, the spill is not emitted immediately. Instead, the built-in merely increments a "spill count" in the node whose result register is to be spilled. Optimization rules that remove a register definition (such as an optimization that replaces a load-store sequence with a memory-to-memory move) decrement the spill count for the last node that defined that register. This procedure effectively deletes spills that were needed in naive code but not in optimized code. But if POP ever pops a node whose spill count still exceeds zero, it attaches the node beneath a store instruction, optimizes the pair together, and emits the resulting tree. POP then substitutes a reload for the node that it had originally popped. Both of the new instructions — the store and the reload — are optimized with their neighbors, so they may combine with other instructions on machines with memory-to-memory arithmetic. A phase-ordering problem prevented the previous compiler from making such optimizations: spills and reloads were introduced during register assignment and thus could not be optimized because register assignment followed peephole optimization. Combining these phases has eliminated this phase-ordering problem.

The integration of code generation and peephole optimization reduces much of the character processing in the back end, but additional steps were needed to minimize it. First, the front and back ends, though separable, are generally connected directly, and the front end does not actually *emit* intermediate code. If it did, its last step for each instruction would use a routine for formatted printing to assemble the opcode and any operands into a postfix instruction for output. For example, it would combine the opcode `IntegerConstant` and the operand 1 to yield the postfix instruction

```
IntegerConstant 1
```

The first step of the back end, however, would read and patternize this instruction, producing the tuple

```
IntegerConstant %1, 1
```

and effectively reversing the effect of the formatting and printing. To avoid these expensive, self-cancelling operations, the front end passes tuples like the one above directly to the back end. The strings are even already hashed for the HOP-style address comparisons — the opcodes being hashed at compile-compile time because they are constant — so the back end need not touch individual input characters.

At the other end, the translation from optimized register transfers to assembly language for output is also designed to minimize character handling, by automatically generating rules that translate register transfers to assembly code. Whenever the machine description is used to translate a register transfer to assembly language, the input and output strings are patternized to form a rule that henceforth implements the same translation without using the machine description. Ultimately, even this step could be avoided by using assembly language

throughout: code generation rules and optimization rules could be specified in assembly code, and PO could use the machine description to translate the assembly language to and from register transfers [5] on those rare occasions when a rule needs to be generated. But even now, the back end rewriting system avoids character handling entirely until it needs to form strings for output or it needs to fall back on the machine description to generate a rule for a previously unseen juxtaposition or output instruction.

Discussion

The rewriting system is about 1700 lines of code, and it calls upon another 1700 lines borrowed (with a few adaptations) from the old compiler to infer new optimization and assembly rules. It uses about 100 code generation rules, which cover most of C except for floating point. It is generally primed with about 500 optimization rules from previous compilations, because experiments on the VAX — which requires about as many optimization rules as any machine — found hit rates over 95% for typical systems programs once about 500 optimization rules were generated.

The new compiler runs as fast as PCC in typical compilations, which start with the needed optimization rules. Pathological cases — starting without any optimization rules whatsoever, and compiling input files too short to build up many optimizations on their own — can take over three times as long as PCC, but the compiler is always primed, and hit rates are so high that this figure is mainly of theoretical interest. This compiler thus may be the first to rely on a machine-directed optimizer for its machine-specific case analysis and still run as fast as existing production compilers. The implementation can probably be made faster still.

At this writing, the new compiler generates code for only the VAX. Its code generation rules are, however, simply rule-based implementations of PO's code generators, and its optimization rules are generated using PO. Therefore, the new system is just as retargetable as the older compiler based on PO.

The original compiler's code was about as good as PCC's, but the new compiler is not yet to quite the same level. The main omission is common-subexpression elimination, which is simply not implemented yet. The planned implementation will create temporary rules that substitute register references for computations that develop values already in registers.

The only other significant omission is the generation of VAX addressing modes that shift an index register before adding it to a base address. This problem can be traced to a problem with rule ordering. The postfix opcode that should, after optimization, result in these addressing modes is one of the few that expand into more than one instruction: one to multiply an index by a constant, and another to add the result to the base address. The multiplication is generated first; when optimized with its child, it generally becomes a memory-to-register multiply that is too complex to combine with the subsequent addition. This code is optimal for most multiplicative factors, but for small powers of two, it would be better to postpone the optimization of the multiplication to allow it combine with the addition. These cases can be caught with a few special-case rules that pre-empt the generation of the multiplication. This solution is effective but inconsistent with naive case analysis. Ultimately, it may be necessary to consider a less greedy optimization strategy, but since there has only been this one rule-ordering problem, this general solution is not yet justified.

Many other optimizations — constant folding, elimination of unary complement operators, execution ordering — can be cast as pattern substitution on trees, so it may be appropriate to include them as code or, better yet, rules in the current system. The current pattern-matching model would need generalization to allow arbitrary tree patterns, not just those that reach down along one path at a time. New kinds of rules may increase rule-ordering problems and require some mechanism for choosing between competing rules.

Related Work

Code generators that rely on modern peephole optimizers for their case analysis had been developing separately from code generators based on tree-matching [1, 2, 14] or parsing [3, 8]. The rewriting system above suggests a way in which these developments may converge.

Code generators based on tree-matching generally match a subtree corresponding to an instruction, emit that instruction, and replace the subtree with a single node identifying where that instruction left its result. The instruction does not normally participate in subsequent matching. Code generators based on parsing might also be viewed as performing a similar matching operation on a linearized tree. Both of these kinds of code generators generally benefit from some peephole optimization.

The rewriting system presented here is also based on tree-matching, but it recycles its output. That is, it normally applies one rule to generate an instruction and then other rules to improve it. Recycling helps make one algorithm perform peephole optimization as well as code generation.

This difference results in different styles of use. The code generators based on tree-matching or parsing are normally used to match larger subtrees than are typical in the current system. For example, there are VAX instructions that take three operands involving two registers and one constant each. Given an instruction like this, the code generators based in tree-matching or parsing will match a much larger subtree than any of the code generation rules in the current system, in which code generation rules generally match one node each but optimization rules simulate the effect of a larger match.

However, the code generators based on tree-rewriting or parsing could be confined to smaller trees by leaving more to the peephole optimizer, and the current rewriting system could be used on larger trees by extending it to match arbitrary subtrees instead of just linear paths. Thus the techniques may be seen to be similar, but differing in the implementation of the tree-matching, in the size of the typical tree pattern, and the support of recycling.

Acknowledgments

Gregg Townsend and Mike Brown adapted PCC to serve as a front end to the rewriting system. Dave Hanson and Bill Waite helped clarify several issues.

References

1. A. V. Aho and M. Ganapathi, Efficient Tree Pattern Matching: An Aid to Code Generation, *Conf. Rec. 12th ACM Symp. on Prin. of Programming Languages*, Jan. 1985, 334–340.
2. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, Code Generation Using Tree Matching and Dynamic Programming, Technical report, Bell Laboratories, 1986.
3. P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick, and E. Pelegri-Llopart, Experience with a Graham-Glanville Code Generator, *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984, 13–24.
4. R. G. G. Cattell, Automatic Derivation of Code Generators from Machine Descriptions, *ACM Trans. Prog. Lang. and Systems* **2**, 2 (Apr. 1980) 173–190.
5. 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.
6. J. W. Davidson and C. W. Fraser, Automatic Generation of Peephole Optimizations, *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* **19**, 6 (June 1984) 111–116.
7. J. W. Davidson and C. W. Fraser, Code Selection Through Object Code Optimization, *ACM Trans. Prog. Lang. and Systems* **6**, 4 (Oct. 1984) 505–526.
8. M. Ganapathi and C. N. Fischer, Affix Grammar Driven Code Generation, *ACM Trans. Prog. Lang. and Systems* **7**, 4 (Oct. 1985) 560–599.
9. R. Giegerich, A Formal Framework for the Derivation of Machine-Specific Optimizers, *ACM Trans. Prog. Lang. and Systems* **5**, 3 (July 1983) 478–498.
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. R. R. Kessler, Peep — An Architectural Description Driven Peephole Optimizer, *SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, June 1984, 106–110.
12. P. B. Kessler, *Automated Discovery of Machine-Specific Code Improvements*, PhD dissertation, University of California, Berkeley (UCB/Computer Science Dept. 84/214, Computer Science Division — EECS), Dec. 1984.
13. D. A. Lamb, Construction of a Peephole Optimizer, *Software—Practice & Experience* **11**, (1981) 638–647.
14. S. W. Weingart, *An Efficient and Systematic Method of Compiler Code Generation*, PhD dissertation, Yale University, June 1973.