Automatic Generation of Fast Optimizing Code Generators

Christopher W. Fraser
AT&T Bell Laboratories
Murray Hill, NJ 07974

Alan L. Wendt
Department of Computer Science
University of Arizona
Tucson, AZ 85721

Introduction

This paper describes a system that accepts compact specifications of an intermediate code and target machine and produces program code for an integrated code generator and peephole optimizer. A compiler for most of C uses this package. It emits code comparable to `pcc1`'s, but it runs over five times faster on preliminary benchmarks. This compiler also runs over twice as fast as a version of `pcc2` with a hand-coded, VAX-specific code generator.

The code generators are produced as follows. A programmer describes a naive code generator by means of a non-procedural specification. The programmer also prepares a machine description for a retargetable peephole optimizer [2]. These two systems are used together to compile a testbed, and the compiler records each peephole optimization as it is made. This record and the specification of the naive code generator are compiled into a fast, integrated code generator and optimizer. This production code generator then takes the place of the slower "training" version. The production code generator and optimizer are integrated to the point that the code to be generated is communicated from one to the other by encoding it in the program counter, which obviates most inter-phase communication costs.

Interpretive peephole optimizers have been driven by traces from retargetable peephole optimizers [3] and integrated with interpretive code generators [4], but the current work is distinguished by the production of a hard-coded, optimizing code generator. Historically, retargetable code generators (i.e., those not largely rewritten for each new machine) have applied a fixed, compile-time interpreter to tables that have been automatically generated from formal specifications [5]. The code generators described below interpret no tables, which helps them run fast.

Representation

Both the training and production code generators accept the same input — an "abstract syntax dag" built by the front end. They use dags rather than trees to accommodate source language features that implicitly reuse values (like C's auto-increment and augmented and multiple assignment) as well as front ends that eliminate common subexpressions as they create nodes. Front ends may confine themselves to trees if the source language permits and if common subexpression elimination is not desired.

The front end compiles, for example, the C statement `up[r-c+7]=0` into a tree annotated with intermediate code:

$$\begin{tikzcd}
\texttt{ISET} \arrow[r] \arrow[d] & \texttt{ICONST 0} & \\
\texttt{IADD} \arrow[r] \arrow[d] & \texttt{GLOBAL up} & \\
\texttt{IMUL} \arrow[r] \arrow[d] & \texttt{ICONST 4} & \\
\texttt{IADD} \arrow[r] \arrow[d] & \texttt{ICONST 7} & \\
\texttt{ISUB} \arrow[r] \arrow[d] & \texttt{IDEREF} \arrow[r] & \texttt{GLOBAL c} \\
\texttt{IDEREF} \arrow[d] & & \\
\texttt{GLOBAL r} & &
\end{tikzcd}$$

The front end has propagated types and folded them into the opcodes (e.g. the `I` prefix flags integer opcodes) so that the back end need not understand the front end's type system, which is typically more complex than the back end's. The front end has

Proceedings of the SIGPLAN '88
Conference on Programming
Language Design and Implementation
Atlanta, Georgia, June 22–24, 1988

also exposed the multiplication implicit in array indexing, so it needs the sizes and alignments of the basic datatypes, but these are easily isolated in a small table.

The code generators rewrite dag nodes in place, replacing the intermediate code with naive and then optimized assembly code. In the example above, each node is first rewritten with a single instruction and then combined with one or more of its descendants via peephole optimization. On the VAX, for example, the subtree rooted at the ISUB above is ultimately replaced with the instruction subl3 _c,_r,r4, and the rest of the tree is replaced with clrl _up+4*7[r4]. That is, the final tree is:

```
clrl _up+4*7[r4]
       ↓
subl3 _c,_r,r4
```

The clrl occupies the node originally occupied by the ISET, and the subl3 occupies the node originally occupied by the ISUB. The actual register assignment for temporaries (like r4 above) is not needed during code generation and optimization, so this task is postponed until these phases complete.

Since the same nodes represent intermediate and assembly code, the code generator needs one representation for both. Assembly code is text, so intermediate opcodes are also represented as text. To avoid the necessity of creating new strings at compile time, the system abstracts constants, identifiers, and register numbers out of the text. For example, the instruction subl3 r2,r3,r4 is represented with the "skeleton" subl3 r%1,r%0,r%2 plus bindings for the "pattern variables" %*i*. The system enumerates all useful skeletons during training and stores them in a table. Opcodes are thus represented as indices into this string table. The compiler has not yet accommodated full C, but the size of the table may be estimated. A production C compiler generated over 26,000 instructions for an 11,000-line testbed, but used fewer than 900 distinct instruction variants. Intermediate codes and target instructions that are always optimized out might increase this figure somewhat, but even so the table should not exceed 40kb even on the VAX, because the average skeleton takes less than 25 bytes, including four bytes for the pointer to each.

For nodes with *n* children, the first *n* pattern variables denote the result registers of the children, and bindings for the rest are stored locally. For example, the instruction subl3 r2,r3,r4 is represented as a node with the following fields:

```
op = 39 where opcode[39] =
                "subl3 r%1,r%0,r%2"
kids[0] = pointer to first child
kids[1] = pointer to second child
vars[0] = "4"
```

The bindings for the pattern variables %0 and %1 are never stored in this node because they are available (after register assignment) in the children's vars fields. Pattern variable %2 is stored in vars[0] because it is the first (and only) pattern variable that needs local storage; this cell is empty until registers are assigned.

Specifying the Code Generator

Here are a few lines from the specification that defines the intermediate code and the naive VAX code generator:

```
%shape 0 1
GLOBAL  moval _%0,r%1
...
%shape 2 2
IADD    addl3 r%1,r%0,r%2
ISUB    subl3 r%1,r%0,r%2
...
%shape 2
ILT     cmpl r%0,r%1; jlss L%2
ISET    movl r%1,(r%0)
...
```

Except for the %shape directives, this specification forms two columns. The first lists the intermediate code's opcodes, and the second gives equivalent but naive assembly code. Thus the intermediate code IADD is to be replaced with the VAX skeleton addl3 r%1,r%0,r%2, and the intermediate code ILT (for "integer less-than") is to be replaced with the instructions cmpl r%0,r%1 and jlss L%2.

The %shape directives describe features shared by the opcodes that follow. Each lists one or two numbers. The first number specifies the number of children of subsequent opcodes. For example, opcodes GLOBAL and moval _%0,r%1 are leaves, and the remaining opcodes above are binary.

The presence of a second number indicates that a register must be allocated to hold the target instruction's result. The number specifies the pattern variable to which the index of the register must be bound. For example, moval _%0,r%1 needs a register allocated and bound to %1, opcodes addl3 r%1,r%0,r%2 and subl3 r%1,r%0,r%2 need a register allocated and bound to %2, and the remaining instructions above need no result register at all.

When building an abstract syntax dag, the front end sets the opcode fields using values from the first column. If the intermediate code uses a constant field — in the examples above, GLOBAL needs the name of a global variable and ILT needs a label number — the front end stores it in the appropriate pattern variable. The automatically generated code generators do the rest.

The compiler using these code generators is not yet complete, but it appears that a naive code generator for, say, ANSI C will require about three pages of lines like those above. The register allocator is retargeted by changing a table if the machine uses general registers; as with most retargetable code generators, machines with asymmetric register sets may require some recoding.

The Training Code Generator

The specification above is automatically compiled into a training code generator, whose general outlines appear below:

```
char *opcode[MAXOPS] = {
...
/* 36 */ "IADD",
/* 37 */ "addl3 r%1,r%0,r%2"
/* 38 */ "ISUB",
/* 39 */ "subl3 r%1,r%0,r%2"
...
}

rewrite(a)
register struct node *a;
{
   switch (a->op) {
   ...
   case 36: L36:  /* IADD */
      rewrite(a->kids[0]);
      rewrite(a->kids[1]);
      a->op = 37;
      goto L37;
   case 37: L37:  /* addl3 r%1,r%0,r%2 */
      (optimizing case analysis to go here)
      break;
   ...
   }
   combine(a);    (only in training version)
}
```

Initially, the code generator uses only those opcodes that appeared in the specification of the naive code generator, so the initial opcode list holds exactly the two columns from the specification.

The routine **rewrite** is the automatically generated, integrated code generator and optimizer. It accepts a pointer to a dag decorated with the simple intermediate code, and it rewrites the dag in place to represent optimized assembly code. The string opcodes are recoded as a range of contiguous integers primarily so that **rewrite** can decode them with an efficient **switch** statement. Each opcode has a distinct **case** that rewrites its particular opcode and jumps off to the case that handles the new opcode just introduced into the dag.

Cases for intermediate codes recursively rewrite any children, then change the node's opcode field to represent the specification's naive target instruction, and finally jump to the case for that target instruction. The training code generator has no compiled code to improve these instructions, so their cases break out of the switch and call **combine**, which is a retargetable peephole optimizer [2].

The production code generator replaces the call on **combine** with hard-coded case analysis in the cases for target instructions. This case analysis takes the form of an if-then-else chain that may edit the dag and jump off to the case that handles the new opcode. An example is presented in due course.

While the code is most easily introduced in the form above, it is actually optimized slightly. The code generator generator does not emit redundant branches, so some cases fall into their successor. (Recall that C cases exit only on an explicit **break**.) For example, the **goto L37** above is really omitted.

Also, the pattern above would have the production code generator's case analysis overwrite **a->op** (sometimes more than once) before leaving the **switch** statement. **rewrite** reads this field only upon entry, so it can be safely out-of-date until the **break**. Thus the code generator slides the assignment to **a->op** down just before the **break**, which guarantees that each invocation of **rewrite** sets it exactly once. In a sense, the program counter encodes the proper value for the opcode field while control remains inside the switch statement. This results in redundant assignments to the opcode field when **rewrite** re-encounters a multiply-referenced node that has been previously traversed and rewritten, but moving the assignment saves more than it sacrifices.

Two arrays not shown parallel the opcode array. They record for each opcode the number of children and the number of the pattern variable that denotes any result register. **rewrite** does not need these arrays because their values are compiled into the code; for example, the IADD case has the proper number of recursive calls compiled in, so it need examine no table to learn how many children it has. These arrays are needed by only the register allocator and output routine, which need to know where to store register names and how many children to traverse. Flags in the nodes (namely, zeros in the first unused slots in **kids** and **vars**) were used initially but rejected because maintaining them cost almost as much as maintaining the useful data.

The Peephole Optimizer and Trace

The training routine combine is a retargetable peephole optimizer. A programmer captures the semantics of the target machine's instructions in a bi-directional grammar for translation between assembly language and register transfers. A machine-independent optimizer uses this machine description to translate pairs and triples of assembler skeletons to register transfer skeletons, which it symbolically simulates to learn their combined effect. It then searches the machine description for an instruction with this combined effect. If it finds one whose cost does not exceed the cost of the original instructions, it rewrites the dag to use the new instruction. If the value produced by an instruction is used several times, its cost is divided equally between its users. A full review of this technique is beyond the scope of this paper, but Reference 2 elaborates. The current implementation adds instruction costs and machine descriptions re-engineered so that, for example, the current, nearly complete VAX description takes only 59 lines.

During training, the optimizer records every optimization. For example, when it replaces moval _%0,r%1 and movl (r%0),r%1 with movl _%0,r%1 (the moval is the first child of the movl, so the former's result register, r%1, is denoted by r%0 in the latter), the optimizer adds the following record to its growing optimization trace:

```
self==movl (r%0),r%1
kid0==moval _%0,r%1
new=movl _%0,r%1
refs<=1
a0=b0
a1=a1
result=1
```

The first three lines are self-explanatory. The fourth reports that, according to the cost metric in the machine description, the optimization pays off only if the child is referenced just once. The next two lines note that the new instruction's %0 is the old child's %0, and the new instruction's %1 is the old parent's %1. The last line above reports that the result register of the new instruction is to be bound to %1. The specification of the code generator names the pattern variable corresponding to the result register for each naive instruction, but the new instruction above has not been seen before, so the optimizer must infer and report the pattern variable corresponding to its result register.

The Production Code Generator

To produce the production system, the code generator generator accepts the trace above and the specification of the naive code generator. It produces an optimizing code generator that is like the naive one presented above, except the opcode list is extended to include all the new instruction variants generated during training, optimizing case analysis is inserted at the head of each case that handles a target instruction, and the call on combine is omitted. Here, for example, are the production versions of the cases presented above:

```
case 36: L36:  /* IADD */
  rewrite(a->kids[0]);
  rewrite(a->kids[1]);
case 37: L37:  /* addl3 r%1,r%0,r%2 */
  b = a->kids[0];
  if (
  b->op == 127  /* mull3 $%1,r%0,r%2 */
  && b->vars[0] == CON4
  ) {
    a->kids[0] = b->kids[0];
    goto L93;  /* moval (r%1)[r%0],r%2 */
  }
  if ( ...
  a->op = 37;
  break;
```

The conditional looks for a sequence that multiplies a register by four and adds it to another register. The expression b->vars[0] == CON4 compares the %1 from mull3 $%1,r%0,r%2 with the constant string "4". It uses b->vars[0] because %1 is the first pattern variable of b that requires local storage. Strings are stored uniquely in a constant table so that an address comparison can be substituted for what would otherwise be a character-by-character comparison. If the conditional succeeds, the dag is rewritten in place, so the "then" arm overwrites a's fields. In this case, the new values of %1 and %2 are the same as the old ones, so only the change to `%0` requires code, which promotes a grandchild.

If the conditional fails, the code generator looks for another pattern, at the point of the ellipsis above. If no optimization applies, control falls off the chain of `if`s into code that updates `a->op` and returns.

In the optimization above, the new instruction costs no more than the one originally pointed to by `a`, so the replacement pays off regardless of the number of uses of `b`. When the new instruction costs more than `a`, the replacement generally pays off when $a + b/n \geq c$, where n is the number of uses of `b`, and a, b, and c denote the costs of `a`, `b`, and the new instruction, respectively. All but n are known when the compiler is generated, so the code generator generator computes the largest n for which the replacement pays off and inserts a clause like `b->count <= 2` in the optimization's enabling condition (e.g. after the comparison with `CON4` above). Different cost metrics (like space, expected time, worst-case time) yield different comparands.

To support such comparisons, the code generator maintains reference counts as it edits the dag. Consider the example above. It edits the dag so that `a` references `b->kids[0]` instead of `b`. Thus it is necessary to decrement `b->count`. If the result is zero, then all reference counts are correct: node `b` is vanishing, but `a` inherits `b`'s references to its children, so these children have the same number of references before and after the edit. But if `--b->count` exceeds zero, then `b` is referenced elsewhere. It still references its children, and now `a` will too, so the reference counts for `b`'s children must be incremented. Thus the actual then-clause above is

```
if (--b->count)
  ++b->kids[0]->count;
a->kids[0] = b->kids[0];
goto L93;  /* moval (r%1)[r%0],r%2 */
```

In cases where `b` points to a leaf, the counts are maintained with just `--b->count`. And in cases where the optimization's enabling condition establishes that `b->count` was one, then even the `--b->count` is omitted.

Node storage is not reclaimed above because even the simplest implementation consumed almost as much time as the case analysis itself. The compiler thus allocates nodes from a fixed pool and then frees the entire pool at once at the end of the expression, block, or procedure. (All three of these compilation units have been used with this system.)

The case analysis above is close to typical. An "average" one has two comparisons, two assignments, and a simple `--b->count`. A few perform no assignments at all, because all important fields are already in the right place. Of course, an assignment to `a->op` occurs just before control leaves the switch.

The code generator is fast. `a` and `b` are in registers, so each line above takes just one or two VAX instructions, and the entire fragment takes just 17. It has not yet been possible to compile a thorough testbed, but it appears that a complete `rewrite` should not require more than 60kb.

It is also possible to eliminate most of the jumps above. Rather than ending a change with `goto L`*n*, the code generator generator could simply place `case` *n* and its code at the point of the `goto`. Since most labels are the target of exactly one `goto`, most of the branches would vanish. This optimization is performed by some existing compilers.

Case analysis like that above could be generated without training on a testbed. The trace encodes simple peephole optimization rules, and there exist mechanisms for enumerating such rules without training on a testbed [6, 7]. These mechanisms are immune to training failures, which can cause the production system to emit code that is sub-optimal (but never incorrect). Experiments have shown that training failures are rare [3], and training does have advantages. It allows the production system to test only rules known to have been useful, and it allows the code generator generator to sort if-then-else chains so that the most common patterns are tested first.

The compiler above gets all of its optimizations from a record of replacements made by a retargetable peephole optimizer, but it could easily accept rewriting rules from other sources as well. The system has already been adapted to accept hand-written optimization rules, and it is a natural client for rules discovered by exhaustive enumeration [8].

Discussion

Two emerging compilers use the techniques above. One uses a modified `pcc1` as a front end and has largely complete back ends for the VAX and the MC68020. The interface between its front end and generated code generators is somewhat less efficient than that shown above. At present, this compiler runs in about 55% of the time taken by `pcc1`. The other compiler uses a new front end and precisely the code generator shown above. This compiler runs in about 20% of the time of **pcc1**. In a typical run, its time was spent as follows:

17%	scanning
26%	parsing
37%	semantic analysis
4%	node allocation and assignment
4%	**rewrite**
4%	register allocation
8%	output

Thus **rewrite** currently takes less than 1% of the time taken by **pcc1**. This compiler does not yet accept full C, and it has thus not yet been possible to process more than a small testbed. Extending the language is likely to slow the front end somewhat, and incorporating the trace from a full testbed is likely to increase the length of the if-then-else chains in **rewrite**. On the other hand, only the scanner [10] and code generator have been extensively tuned, so improvements are also possible.

Pennello has described a technique for replacing an LR parsing table and its interpreter with equivalent optimized assembly code [9]. Such techniques could be applied to the LR parser used by Graham-Glanville code generators [1]. Like **rewrite**, the resulting code generator would be hard-coded, though differences between the two algorithms would complicate any assessment of their relative performance in the absence of measurements.

Acknowledgments

Dave Hanson, Robert Henry, Peter Kessler, and Peter Weinberger made several helpful suggestions. This work was supported in part by the National Science Foundation under Grant DCR-8320257.

References

1. 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.
2. J. W. Davidson and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, *ACM Trans. Prog. Lang. and Systems* **2**, 2 (Apr. 1980) 191–202.
3. J. W. Davidson and C. W. Fraser, Automatic Inference and Fast Interpretation of Peephole Optimization Rules, *Software—Practice & Experience* **17**, 11 (Nov. 1987) 801–812.
4. C. W. Fraser and A. L. Wendt, Integrating Code Generation and Optimization, *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices* **21**, 7 (July 1986) 242–248.
5. M. Ganapathi, C. N. Fischer, and J. L. Hennessy, Retargetable Compiler Code Generation, *Computing Surveys* **14**, 4 (Dec. 1982) 573–592.
6. R. R. Kessler, Peep—An Architectural Description Driven Peephole Optimizer, *SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices* **19**, 6 (June 1984) 106–110.
7. P. B. Kessler, Discovering Machine-Specific Code Improvements, *SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices* **21**, 7 (July 1986) 249–254.
8. H. Massalin, Superoptimizer — A Look at the Smallest Program, *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), SIGPLAN Notices* **22**, 10 (Oct. 1987) 122–126.
9. T. J. Pennello, Very Fast LR Parsing, *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices* **21**, 7 (July 1986) 145–151.
10. W. M. Waite, The Cost of Lexical Analysis, *Software—Practice & Experience* **16**, 5 (May 1986) 473–488.