SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 13, 745–763 (1983)

Simple Code Optimizations*

DAVID R. HANSON
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.

SUMMARY

Program optimization has received a great deal of attention for many years, which has resulted in numerous advances in compiler technology. The effectiveness of various *simple* optimizations has received comparably little attention during the same time period. The simplicity of most programs suggests that straightforward optimizations pay the greatest dividends. This paper describes three such optimizations suitable for one-pass compilers. The optimizations involve expression rearrangement, instruction selection, and the use of a cache for the allocation of resources. The cost of these optimizations is low; none require major changes to the size or structure of the compiler or reduce compilation speed by more than 10%. The benefits are high; each optimization results in at least a 10% average reduction in object code size and a corresponding reduction in execution time. Examples and implementation details are also described.

KEY WORDS code optimization register allocation caching instruction selection

INTRODUCTION

There has been a great deal of research in program optimization over the last ten years, which as lead to theoretical and practical advances in loop optimization, data flow analysis, and associated techniques.[1] Recent work has concentrated on interprocedural data flow analysis,[2,3] optimization in the presence of other constraints (viz., Reference 4), and the automation of code generation[5,6] and peephole optimization.[7]

Much of the work in optimization has been directed toward obtaining the best code possible according to some quality measure, such as space or execution time. The simplicity of most programs and consequent effectiveness of straightforward optimizations has long been recognized[8] and persists.[9-12] This suggests that relatively simple optimizations deserve more attention than they have received.

Optimization cost is typically measured by complexity, the implementation effort required, and the space and time impact on the compiler. The benefits of optimizations are measured by the reduction in code size or execution time.[13] Peephole optimization, which can greatly improve local code, is an example of a relatively simple optimization that is very effective. It is one of the few optimizations that is frequently cited as generally worthwhile, and it often accounts for significant improvement in optimizing compilers.[14] Similar experience with other, more complicated optimizations, especially those based on data flow analysis, is rare and makes balancing the complexity of an optimization with its effectiveness a difficult design decision. For example, including an optimization that would require an additional pass during compilation in exchange for a 10% reduction in code size is questionable. As a result, optimization is often confined to *ad hoc* techniques or methods such as peephole optimization.

*This work was supported by the National Science Foundation under Grant MCS-7802545.

0038–0644/83/080745–19$01.90

Received 29 November 1982
Revised 9 May 1983

This paper describes three optimizations used in a one-pass compiler for the Y programming language,[15] which is syntactically similar to Ratfor[16,17] and semantically similar to C.[18] The optimizations are simple and effective. Optimizations are simple if they require no major changes to data structures or algorithms of the compiler, result in no more than a 10% increase in space or running time required by the compiler, and do not require additional passes. (This last constraint rules out optimizations that require a backward scan over the generated code, such as those that employ 'next-use' information.[19]) Optimizations are effective if they result in at least a 10% average reduction in object code size. While compactness was considered more important than speed, the optimizations described here also reduced execution time.

The Y compiler is typical of simple, one-pass compilers. It parses its input using a recursive descent parser, emitting assembly language along the way. The code generator is a separate module with an interface consisting of 18 procedures, many of which are trivial (e.g. generate a jump instruction). Expressions are compiled into directed acyclic graphs (DAGs), which are traversed by the code generator to produce the appropriate code. Some aspects of expressions, such as short-circuit evaluation of conditional expressions,[20] are handled prior to code generation, which simplifies code generation for expressions. Producing a naive code generator for Y requires writing the 18 interface procedures. This task is straightforward and a naive code generator can be written in a few days. Code generators have been written for the DEC-10, PDP-11, the Motorola 6502 and several abstract machines.

Of course, naive code generators produce poor code. The three optimizations described in the subsequent sections are simple, and they yield good code. Indeed, for many 'average' programs, the code is comparable to or better than that produced by optimizing compilers set at mid-range optimization levels and by some recent automated approaches.[7] Although these optimizations were designed for use in one-pass compilers, they can be applied equally well in multi-pass compilers. The three optimizations are, in order of increasing machine dependence, expression rearrangement, cache-based resource allocation, and lazy instruction selection.

Expression rearrangement simplifies expressions, reorders computations according to arithmetic laws, folds constant expressions, reduces the strength of certain operations, and recognizes idioms. Lazy instruction selection defers the selection of specific instructions until the last possible point in order to exploit instruction variants, such as 'immediate' mode on the DEC-10. Both of these optimizations are straightforward and are similar to other techniques.

Cache-based resource allocation employs a cache to track the use of resources, such as registers. While the other two optimizations produce local improvements, this one yields some global improvements, including common subexpression elimination and register allocation. It is similar to, but more general than, the techniques used in the CDC 6600 Pascal compiler for tracking the register contents.[21]

EXPRESSION REARRANGEMENT

Expressions are compiled into DAGs in a bottom-up fashion. Using DAGs saves space and permits common subexpressions to be recognized, and they are nearly as easy to construct as trees. Nodes in the DAGs are built as the expression is parsed. Nodes correspond to operators appearing explicitly in expressions and to other implicit operators, such as addressing operators, introduced to facilitate semantic checking and code generation. For example, the Y expression

```
c = (a + b) - b/(a + b)
```

is compiled into the tree

```
(= (& c) (- (+ a b) (/ b (+ a b))))
```

The duplication of the common subexpression a + b is avoided in the DAG, whose nodes, listed in prefix order, are

```
1.  =   9   2
2.  -   4   3
3.  /   6   4
4.  +   5   6
5.  *   7
6.  *   8
7.  &   a
8.  &   b
9.  &   c
```

Operands of interior nodes are indicated by their node number, unary & and unary * denote the 'address of' operator and indirection, respectively, and generic operators, such as binary +, are used in place of their type-specific counterparts to simplify the examples.

Straightforward construction of the nodes produces DAGs from which naive code generators produce poor or awkward code sequences. Improvements can be obtained by rearranging the DAGs, which is similar to making source-to-source transformations.[22] Expression rearrangment differs from the latter technique in that it can introduce constructs that have no source language counterpart. It evaluates constant expressions, recognizes arithmetic identities and expression idioms, and factors additive components of addressing expressions, all of which result in better code.

Expression rearrangement is implemented by recognizing certain patterns as each node is built. Since DAGs are constructed from the bottom up, complete tree pattern matching is unnecessary; only the current node and the roots of its offspring must be examined for most patterns. Consequently, a very simple implementation suffices. In the Y compiler, for example, the patterns are encoded in a series of conditional statements.

The simplest patterns involve the arithmetic identities and the reduction in strength of certain operators:

$$
\begin{array}{ll}
(+\ x\ 0) & \rightarrow\ x \\
(-\ x\ 0) & \rightarrow\ x \\
(*\ x\ 0) & \rightarrow\ 0 \\
(*\ x\ 1) & \rightarrow\ x \\
(*\ x\ 2) & \rightarrow\ (+\ x\ x) \\
(*\ x\ 2^n) & \rightarrow\ (\ll\ x\ n) \\
(*\ x\ 0) & \rightarrow\ 0 \\
(/\ x\ 1) & \rightarrow\ x \\
(/\ x\ 2^n) & \rightarrow\ (\gg\ x\ n) \\
(/\ x\ x) & \rightarrow\ 1
\end{array}
$$

where x represents any expression and $\ll$ and $\gg$ denote left and right shift, respectively. Constant expressions are also easily recognized:

$$
\begin{array}{ll}
(op\ c) & \rightarrow\ op\ c \\
(op\ c_1\ c_2) & \rightarrow\ c_1\ op\ c_2
\end{array}
$$

where *op* indicates any unary or binary operator and c and c_i indicate constants.

The arithmetic laws can be used to reduce the number of patterns by placing some expressions in a 'canonical form'. This reduction simplifies case analysis during code generation.

$$
\begin{array}{ll}
(+\ c\ x) & \rightarrow\ (+\ x\ c) \\
(*\ c\ x) & \rightarrow\ (*\ x\ c) \\
(+\ (+\ x\ c)\ y) & \rightarrow\ (+\ (+\ x\ y)\ c) \\
(+\ (-\ x\ c)\ y) & \rightarrow\ (+\ (+\ x\ y)\ -c) \\
(+\ (+\ x\ c_1)\ (+\ y\ c_2)) & \rightarrow\ (+\ (+\ x\ y)\ c_1{+}c_2)) \\
(+\ (+\ x\ c_1)\ (-\ y\ c_2)) & \rightarrow\ (+\ (+\ x\ y)\ c_1{-}c_2)) \\
(-\ (+\ x\ c_1)\ (-\ y\ c_2)) & \rightarrow\ (+\ (-\ x\ y)\ c_1{+}c_2))
\end{array}
$$

The omitted cases are similar.

Other patterns rearrange addressing expressions to expose common factors. Examples include:

$$
\begin{array}{ll}
([]\ x\ (+\ y\ c)) & \rightarrow\ (++\ ([]\ x\ y)\ s{*}c) \\
([]\ x\ c) & \rightarrow\ (++\ (\&\ x)\ s{*}c) \\
(++\ (\&\ x)\ 0) & \rightarrow\ (\&\ x) \\
(++\ (++\ (\&\ x)\ c_1)\ c_2) & \rightarrow\ (++\ (\&\ x)\ c_1{+}c_2) \\
(++\ (\&\ x)\ (+\ y\ c)) & \rightarrow\ (++\ ([]\ x\ (/\ y\ s))\ c)
\end{array}
$$

where [] and ++ denote indexing and address adjustment, respectively, and s is the size of an element of the array x. The operation $([]\ x\ y)$ is defined as

$$(++\ (\&\ x)\ (*\ s\ y))$$

and is included to avoid proliferation of ++ and & operators. These patterns often permit the 'constant part' of an addressing computation to performed by the addressing hardware.

The preponderance of simple, idiomatic assignments in most programs is well known.[8,10,23] Those that occur frequently can be recognized and tagged during DAG construction. The following patterns are examples.

$$
\begin{array}{ll}
(=\ x\ (+\ x\ 1) & \rightarrow\ (=+\ x\ (+\ x\ 1)) \\
(=\ x\ (-\ x\ 1) & \rightarrow\ (=-\ x\ (-\ x\ 1)) \\
(=\ x\ (op\ x\ c) & \rightarrow\ (=op\ x\ (op\ x\ 1))
\end{array}
$$

where x is an expression without side effects, =+ and =− denote increment and decrement operations, respectively, and $=op$ denotes assignment 'augmented' with operation op. Note that the replacement DAGs have the same form as the originals; the only changes are the augmented assignment operators. This approach permits augmented assignment to be handled exactly like assignment in code generators that cannot make use of the special operators. The appropriate portions of the subtree can be ignored when these operators are supported by hardware.

These patterns permit a simple code generator to produce substantially better code. For example, expressions such as

```
6 + 1*x - 5
x | 0400 | 1<<10
```

where 0400 is an octal constant, | is bitwise inclusive OR, and 1<<10 is 1 shifted left 10 places, occur frequently when compile-time parameters are used in place of explicit constants. These are compiled into, respectively,

```
(+ x 1)
(| x 1280)
```

where x denotes (* (& x)). The addressing patterns and the definition of [] permit expressions such as

```
x[j+4].link
x[10].table[i].count
```

to be compiled into

```
(++ ([] x j) 614)
(++ (++ (& x) (* 12 i)) 1797)
```

assuming widths of the elements of x and table are 200 and 12, respectively, that the offsets to the link, table, and count fields are 14, 2, and 7, respectively, and array indexing begins at 1.† On the DEC-10, these references can be accomplished in two instructions. Likewise, the idiom patterns cause expressions such as

```
f = f | 010
x[p+4].count = 6 + 1*x[p+4].count - 5
```

to be compiled into

```
(=| f (| f 8))
(=+ (++ ([] x p) 607) (++ ([] x p) 607) 1)
```

which, on the DEC-10, yields

```
f = f | 010
        movei   2,8             ; 010 to register 2
        iorb    2,f             ; f | 010 to register 2 and f

x[p+4].count = 6 + 1*x[p+4].count - 5
        move    3,p             ; p to register 3
        aos     x+607(3)        ; increment x[p+4].count
```

The improvements described above can be done, often more throughly, by other kinds of optimizations or during code generation, but usually at the cost of additional passes. The advantages of expression rearrangement are that it is machine-independent, amenable to one-pass compilation, simple to implement as a part of DAG construction, and it yields substantial improvement. In addition, it enhances the effect of the other optimizations described below.

LAZY INSTRUCTION SELECTION

Instruction selection is complicated by the idiosyncrasies of the instruction sets of most computers. Good selection can significantly improve the quality of the resulting code,[14] but typically requires tedious case analysis. Automating this case analysis and insuring its correctness motivates recent table-driven approaches to code generation. While these techniques do automate case analysis and produce good code, they currently require large tables and a separate processor to construct the tables.[24]

† Y does not have records or structures, but the code generator interface supports the operations necessary to implement them.

Lazy instruction selection[25] is a simpler, procedural approach to instruction selection and, consequently, does not produce code of quality comparable to the table-driven methods. For example, lazy selection deals only with selecting a single instruction instead of sequences of instructions. It does, however, result in improvements of 10 to 20%, is easy to implement, and has almost no impact on compiler speed or size.

Lazy instruction selection effectively generalizes the instruction set and defers selection of specific instructions until the last possible moment. Hence the label 'lazy', which is taken from a similar concept in programming languages.[26] For example, for a computer with two-operand instructions, it would be assumed that *all* instructions can accept two operands of *any* kind, and the details would be handled as instructions are output. A similar generalization can be made for computers that have a variety of 'modes', indicated by different address specifications (e.g. the PDP-11) or different instructions (e.g. the DEC-10).

Contrast this 'optimistic' approach to the pessimistic approach, typically made in naive code generators, in which operands must appear in registers. On the DEC-10, for example, these two approaches would compile the statement

```
a = b*c + 5
```

into the code

pessimistic

```
move   2,b
move   3,c
imul   2,3
move   3,[5]
add    2,3
movem  2,a
```

optimistic

```
move   2,b      ; b to register 2
                ; c to register 3
imul   2,c      ; b*c to register 2
                ; 5 to register 3
addi   2,5      ; b*c + 5 to register 2
movem  2,a      ; b*c + 5 to a
```

where brackets indicate the address of a word containing what they enclose. The optimistic sequence, which requires 4 words of memory instead of 7, references c directly in the imul instruction and uses the immediate mode form of the add instruction in which the address, 5, *is* the operand.

Implementing lazy instruction selection is straightforward. Code generation for the leaves of a DAG, e.g. references to variables and constants, result in data structures, often called 'suspensions', that represent the references. Code generation for interior nodes, which represent operations, consists of calling the output routine with the appropriate generalized instruction and the suspensions returned from generating code for the operands. When an actual instruction must be emitted, the operand suspensions are examined and the appropriate selection is made.

For example, assuming operands can be either registers, constants, or addresses, code generation for the example given above begins by constructing suspensions for the references to a, b, and c: {*address*, a}, {*address*, b}, and {*address*, c} where braces enclose the operand type and operand of a suspension. Attempting to output a generic multiply instruction with operands {*address*, b} and {*address*, c} causes b to be loaded:

```
move   2,b
```

then multiplied by c:

```
imul   2,c
```

The resulting suspension is {*register*, 2}. Similarly, the reference to 5 produces the suspension {*constant*, 5}, and the generic addition of {*register*, 2} and {*constant*, 5} emits

```
addi    2,5
```

and produces the suspension {*register*, 2}. Finally, the emission of the generic store with operands {*address*, a} and {*register*, 2} emits

```
movem 2,a
```

Other aspects of code generation can be simplified by generalizing the instruction set further and adding more information to suspensions. For example, adding data type information to suspensions permits some implicit type conversions, which are usually included in the DAGs, to be done during instruction selection. Assuming a and b are integers and c is real, code generation for the example given above results in the following instruction sequence.

```
fltr    2,b      ; b to register 2 and convert to real
fmpr    2,c      ; b*c to register 2
fadri   2,(5.0)  ; b*c + 5.0 to register 2
fixr    2,2      ; convert b*c + 5.0 to integer
movem   2,a      ; b*c + 5.0 to a
```

The generic multiplication of {*integer*, *address*, b} and {*real*, *address*, c} causes the implicit conversion of b as it is loaded. Likewise, the generic addition of {*real*, *register*, 2} and {*integer*, *constant*, 5} causes compile-time conversion of 5 to 5.0 and the use of the immediate mode of fadr instruction since the rightmost 18 bits of 5.0 on the DEC-10 are zero.

Lazy instruction selection divides the case analysis required during code generation into two parts. Analysis that depends the semantics of the language, such as data type considerations, can be performed in the 'main part' of the code generator. Analysis that depends on the low level details of instruction selection can be performed by the output routines. This separation simplifies both components of the code generator.

CACHE-BASED RESOURCE ALLOCATION

Another reason that naive code generators produce poor code is because they do not 'track' the contents of registers, temporaries, and other scarce resources, and they do not eliminate common subexpressions. These omissions lead to redundant and awkward instruction sequences. Peephole optimization can solve the first of these problems, but often introduces an additional pass unless the instruction stream is buffered, which complicates the implementation of the output routines. More recent approaches to object code optimization can also eliminate some common subexpressions.[27]

The remainder of this section describes a simple cache-based method for tracking resource use and detecting some common subexpressions. The algorithm is explained by considering a cache model more general than that actually implemented. In addition, some of the searching implied by the model used in the presentation is avoided in the actual implementation, as described below.

Expressions in Basic Blocks

Given the DAG framework described above, the basic idea is to maintain during compilation a cache of associations (r, t), where r describes a *resource*, typically a register, a temporary, or a suspension, and t is the root of a *subtree* in the set of DAGs for the program. r contains the result of executing the code for subtree t. Whenever code for computing t is to be emitted, the

cache is examined. If an entry (r, t) exists, r is returned and no code is emitted. If there is no entry for t, the code is emitted as usual and a cache entry indicating that the resource r contains the result of executing t is made.

For example, the Y sequence

```
x[a] = b
b = a + 10
c = x[a]
```

results in the following code from a naive code generator that produces low-level quadruples.[19]

```
x[a] = b
 1.     T1 = &a          address of a to T1
 2.     T2 = *T1         value of a to T2
 3.     T3 = &x [] T2    address of x[a+1] to T3
 4.     T4 = T3 ++ -1    address of x[a] to T4
 5.     T5 = &b          address of b to T5
 6.     T6 = *T5         value of b to T6
 7.     *T4 = T6         value of b to x[a]
b = a + 10
 8.     T7 = &b          address of b to T7
 9.     T8 = &a          address of a to T8
10.     T9 = *T8         value of a to T9
11.     T10 = 10         constant 10 to T10
12.     T11 = T9 + T10   a + 10 to T11
13.     *T7 = T11        a + 10 to b
c = x[a]
14.     T12 = &c         address of c to T12
15.     T13 = &a         address of a to T13
16.     T14 = *T13       value of a to T14
17.     T15 = &x [] T14  address of x[a+1] to T15
18.     T16 = T15 ++ -1  address of x[a] to T16
19.     T17 = *T16       value of x[a] to T17
20.     *T12 = T17       value of x[a] to c
```

Using the cache eliminates quadruples 8-10 and 15-19 and produces the following code. Note the elimination of the common subexpression x[a].

```
x[a] = b
 1.     T1 = &a          address of a to T1
 2.     T2 = *T1         value of a to T2
 3.     T3 = &x [] T2    address of x[a+1] to T3
 4.     T4 = T3 ++ -1    address of x[a] to T4
 5.     T5 = &b          address of b to T5
 6.     T6 = *T5         value of b to T6
 7.     *T4 = T6         value of b to x[a]
b = a + 10
11.     T10 = 10         constant 10 to T10
12.     T11 = T2 + T10   a + 10 to T11
13.     *T5 = T11        a + 10 to b
```

```
c = x[a]
14.     T12 = &c          address of c to T12
20.     *T12 = T6         value of x[a] to c
```

Translating these quadruples into the final code is straightforward. Indeed, the DEC-10 compiler emits assembly language directly bypassing the quadruple representation. It produces the following unoptimized and optimized code for this example. (Both sequences include the effects of expression rearrangement and lazy instruction selection).

```
                unoptimized                optimized
x[a] = b
        move    2,a                move    2,a        ; a to register 2
        move    3,b                move    3,b        ; b to register 3
        movem   3,x-1(2)           movem   3,x-1(2)   ; b to x[a]
b = a + 10
        move    2,a                                   ; a to register 2
        addi    2,10               addi    2,10       ; a + 10 to register 2
        movem   2,b                movem   2,b        ; a + 10 to b
c = x[a]
        move    2,a                                   ; a to register 2
        move    3,x-1(2)                              ; x[a] to register 3
        movem   3,c                movem   3,c        ; x[a] to c
```

Addressing operations are a particularly important source of common subexpressions. For example, array indexing is usually decomposed into individual addressing operations, which explains the adjustment of T3 shown above. On many computers, some addressing operations (e.g. indexing, indirection) can be performed by the addressing hardware. Using the cache eliminates recurrent addressing operations as illustrated by the Y sequence

```
x[a] = b
x[a+3] = x[a-4]
```

yields the following unoptimized and optimized sequences.

```
                unoptimized          optimized
x[a] = b
        T1 = &a                 T1 = &a
        T2 = *T1                T2 = *T1
        T3 = &x [] T2           T3 = &x [] T2       address of x[a+1] to T3
        T4 = T3 ++ -1           T4 = T3 ++ -1       address of x[a] to T4
        T5 = &b                 T5 = &b
        T6 = *T5                T6 = *T5
        *T4 = T6                *T4 = T6
x[a+3] = x[a-4]
        T7 = &a
        T8 = *T7
        T9 = &x [] T8                               address of x[a+1] to T9
        T10 = T9 ++ 2           T10 = T3 ++ 2       address of x[a+3] to T10
        T11 = &a
        T12 = *T11
        T13 = &x [] T12                             address of x[a+1] to T13
```

```
T14 = T13 ++ -5     T14 = T3 ++ -5      address of x[a-4] to T14
T15 = *T14          T15 = *T14
*T10 = T15          *T10 = T15
```

The corresponding DEC-10 code is

```
                 unoptimized                optimized
x[a] = b
        move     2,a                 move     2,a
        move     3,b                 move     3,b
        movem    3,x-1(2)            movem    3,x-1(2)     ; b to x[a]
x[a+3] = x[a-4]
        move     2,a
        addi     2,3                                       ; a + 3 to register 2
        move     3,a
        subi     3,4                                       ; a - 4 to register 3
        move     4,x-1(3)            move     4,x-5(2)     ; x[a-4] to register 4
        movem    4,x-1(2)            movem    4,x+2(2)     ; x[a-4] to x[a+3]
```

As shown by the DEC-10 code for this example, some details of addressing expressions, such as negative offsets, are machine-dependent. Such variations in addressing manifest themselves as variations or additions in the operators that appear in the DAGs, but do *not* affect the caching mechanism.

These examples show that the cache represented by the values of r during execution is a 'write-through' cache; values associated with permanent memory locations (e.g. variables) are also stored in memory. This property permits 'old' entries to be discarded when the cache becomes full. An aging algorithm, such as least recently used, determines which entry is removed. Note that better algorithms, such as farthest distance to the 'next use', cannot be used in a single pass environment.

Cache entries are also removed when they are invalidated by side effects and, as described below, flow of control. While language semantics dictate the details of this action, any operation that invalidates an association (r, t) results in the removal of that entry. In Y, the cache must be examined after each assignment and procedure call.

Assignment to a scalar variable invalidates all cache entries that depend on that variable, and assignment to an array element invalidates all entries that depend on any element of the array. Specifically, upon assignment to v all entries (r, t) in which t depends on v must be discarded. Since t is the root of an arbitrary subtree in the set of DAGs, each t is traversed recursively to determine its dependence on v. This potentially expensive operation could be avoided by propagating dependency information to the roots of the subtrees when the DAGs are constructed. In practice, however, the vast majority of subtrees are quite shallow and the traversal cost is within reasonable bounds.

Procedure calls present three situations in which cache entries might be removed. First, entries that depend on global variables must be discarded. Second, since the callee can change the values of reference parameters, entries that depend on those variables must be discarded. Finally, entries for reference parameters to the caller, which might refer to global variables, must be discarded. The only reference parameters in Y are arrays. Thus, following the call to g in

```
int x[10], y
f(a)
   int a[], b[20], c, d[5], e
   ...
   g(x, b, c)
   ...
end
```

entries that depend on x, y, a, and b must be discarded. Entries for c remain valid because it is a scalar local to f and is passed by value. Entries for d and e remain valid because they are local to f and cannot be affected by g.

Although Y lacks pointers, they are handled much like array references and have the well known effect on the outcome of optimization.[28] Adding pointers (and records) to Y would require substantial modifications to syntax analysis and symbol table modules of the compiler, but few changes to the code generator.

Assignment via a pointer p or passing p to a procedure invalidates entries that contain variables to which p might refer. Entries that depend on pointers passed to the caller must also be discarded. Likewise, assignment to a non-pointer variable invalidates entries for those pointers that could potentially point to it. Note that the number of entries invalidated by these operations is potentially much less in a language, such as Pascal, that has only type-specific pointers as opposed to a language, such as C, that permits type-free pointers.

Control Flow Constructs

Flow of control also invalidates cache entries. An unnecessarily conservative approach would be to clear the entire cache at every label. Consider, however, the Y sequence

```
b = x[a]
if (b > max)
   max = b
else
   max = a
c = x[a] + b
```

which produces code of the form

```
        code for b = x[a]
        if (b <= max) goto L1
        code for max = b
        goto L2
L1      code for max = a
L2      code for c = x[a] + b
```

Clearing the cache at L1 removes the entry for a, which is still valid at this point. Similarly, clearing the cache at L2 removes the valid entries for a, b, and x[a].

Although solving this problem in general requires data flow analysis, a good approximation can be achieved by generalizing the cache mechanism. Instead of a single cache, a *stack* of caches is maintained. To search the entire cache for subtree t, each cache on the stack is examined, beginning at the top of the stack. Removing entries involves a similar traversal. The stack is pushed and popped in accordance with the semantics of the various control flow constructs. In addition, a *searchdepth* parameter, which limits the number of caches examined during a search, is required to handle loops. For example, if *searchdepth* is 1, only the cache at

the top of the cache is examined. Initially, *searchdepth* is infinite, indicating that the entire stack is to examined.

To explicate this mechanism, the stack is manipulated by the following push and pop operations. *push*(*n*) pushes the current value of *searchdepth* and a new, empty cache onto the stack and sets *searchdepth* to *n*. *pop*() discards the cache at the top of the stack, removing all of its entries, and restores the value of *searchdepth* saved by the corresponding *push*.

Code is generated for each control flow construct by generating code for the components of the construct interleaved with the appropriate *push* and *pop* calls. For example, the code generation template for

```
if (e) S1 else S2
```

is

```
emit(if (~e) goto L1)
push(stackdepth+1)
emit(code for S1)
pop()
emit(goto L2)
emitlabel(L1)
push(stackdepth+1)
emit(code for S2)
pop()
emitlabel(L2)
```

The effect of the stack is to 'restore' the state of computation at L2 that before the if statement, except for those entries removed during code generation for each of the 'arms'. In addition, entries made before the if statement are available in both arms despite the intervening label (L1) due to the search strategy. New entries made in either of the arms, however, are discarded when they are no longer valid. Applying this template to the Y sequence given above yields the following quadruples.

```
                unoptimized                 optimized
b = x[a]
                T1 = &b                     T1 = &b
                T2 = &a                     T2 = &a
                T3 = *T2                    T3 = *T2
                T4 = &x [] T3               T4 = &x [] T3
                T5 = T4 ++ -1               T5 = T4 ++ -1
                T6 = *T5                    T6 = *T5
                *T1 = T6                    *T1 = T6
if (b > max)
                T7 = &b
                T8 = *T7
                T9 = &max                   T9 = &max
                T10 = *T9                   T10 = *T9
                if (T8 <= T10) goto L1      if (T6 <= T10) goto L1
max = b
                T11 = &max
                T12 = &b
                T13 = *T12
```

```
        *T11 = T13                      *T9 = T6
        goto L2                         goto L2
L1:                             L1:
max = a
        T14 = &max
        T15 = &a
        T16 = *T15
        *T14 = T16                      *T9 = T3
L2:                             L2:
c = x[a] + b
        T17 = &c                        T17 = &c
        T18 = &a
        T19 = *T18
        T20 = &x [] T19
        T21 = T20 ++ -1
        T22 = *T21
        T23 = &b
        T24 = *T23
        T25 = T22 + T24                 T25 = T6 + T6
        *T17 = T25                      *T17 = T25
```

Code generation templates for other branching control constructs, such as the **switch** or **case** statement, are similar. The resulting code is good, but not as good as can be achieved using more advanced data flow techniques. For example, in

```
a = ...
if (...)
   code that modifies a
else
   code that uses a
```

the modification of a in the *then* arm results in the removal of the cache entry for the value of a, causing it to be unnecessarily reloaded in the *else* arm. More sophisticated—and costly—techniques can detect this case and avoid the redundant load.

Code generation templates for loops is complicated because there is more than one path to the beginning of the loop. To handle these constructs, the search must be restricted to the cache at the top of the stack. For example, the code generation template for

```
while (e) S
```

is

```
push(1)
emitlabel(L1)
emit(if (~e) goto L2)
emit(code for S)
emit(goto L1)
emitlabel(L2)
pop()
```

The C-style for statement

```
for (e1 ; e2 ; e3) S
```

is equivalent to

```
e1
while (e2) {
   S
   e3
   }
```

Code generation for **for** statements is complicated by **next** statements, which transfer control to the 'next iteration' point—just before e_3—of the loop. Thus, cache entries made during the compilation of S may be invalid in e_3. The template for the **for** statement is

```
emit(code for e1)
push(1)
emitlabel(L1)
emit(if (~e2) goto L3)
emit(code for S)
if (number of next statements > 0) {
   pop()
   push(1)
   }
emitlabel(L2)
emit(code for e3)
emit(goto L1)
emitlabel(L3)
pop()
```

Alternative code templates for loops also cause problems similar those caused by **next** statements. For example, **while** loops are sometimes compiled into

```
        goto L2
L1      code for S
L2      if (e) goto L1
```

which avoids n-1 jump instructions when the loop is executed n times.[9,29] The corresponding code generation template is

```
push(1)
emit(goto L2)
emitlabel(L1)
emit(code for S)
pop()
push(1)
emitlabel(L2)
emit(if (e) goto L1)
pop()
```

Since there are two execution paths to L2, the cache must be cleared at that point. Consequently, in loops where e and S share common subexpressions, this template may yield poorer code than the previous **while** template.

Implementation

Clearly, the searching and maintenance of multiple caches is too inefficient to be used in practice, especially if the goal is to minimize cost. There are two simplifications to the scheme that bring its cost within acceptable bounds without sacrificing its advantages.

A single cache can be used in place of the stack by augmenting each entry. The cache consists of entries (r, t, l) where l is a 'level' number that corresponds to the stack height of the entry. Searching is restricted to those entries for which l is greater than *level - searchdepth*, where *level* is the current stack height and is incremented and decremented by *push* and *pop*, respectively.

Searching can be avoided altogether by permitting only one cache association for any subtree t. In this case, r and l can be included as fields in the nodes of the DAGs. Code generation for subtree t reduces to using r if it is present and l satisfies the condition given above, or generating the appropriate code and annotating t with the resulting r and l.

Another, less obvious, aspect of the cache technique is that subtrees can arise from other than syntactic sources. For example, on the DEC-10 integer division also computes the remainder. When the DEC-10 code generator generates an integer division, it creates a subtree and cache entry for the corresponding remainder. If the remainder later appears, its computation may be avoided. The typical decimal output procedure is an example:

```
1.      putd(n)
2.          integer n
3.          if (n/10)
4.              putd(n/10)
5.          putc('0' + n%10)
6.      end
```

(putc outputs the character given by its ASCII code argument). The division at line 3 causes a subtree and cache entry for n%10 to be created. When this expression is encountered in line 5, the previously computed value is used. The resulting DEC-10 code for the body of putd is

```
        move    2,n     ; n to register 2
        idivi   2,10    ; n/10  to register 2, n%10 to register 3
        jumpe   2,L1    ; jump to L1 if n/10 is 0
        push    15,2    ; call putd(n/10)
        pushj   15,putd
        adjsp   15,-1
L1:     addi    3,48    ; n%10 + '0' to register 3
        push    15,3    ; call putd(n%10 + '0')
        pushj   15,putc
        adjsp   15,-1
```

The combined effect of this simplified implementation of the cache system and the other two optimizations yields good code for a large percentage of programs. The following procedure, which is from an eight queens program,[30] demonstrates many of the benefits.

```
queens(c)
    int r, c
    for (r = 1; r <= 8; r = r + 1)
        if (rows[r] & up[r-c+8] & down[r+c-1]) {
            rows[r] = up[r-c+8] = down[r+c-1] = 0
            x[c] = r
```

```
        if (c == 8)
            print()
        else
            queens(c + 1)
        rows[r] = up[r-c+8] = down[r+c-1] = 1
        }
end
```

Omitting the procedure entry and exit sequences, the optimized DEC-10 code is

```
 1.          movei   2,1             ; r = 1
 2.          movem   2,r
 3.  L1:     move    3,r             ; r <= 8
 4.          caile   3,8
 5.          jrst    L2
 6.          move    4,rows-1(3)     ; test rows[r]
 7.          jumpe   4,L3
 8.          sub     3,c             ; test up[r-c+8]
 9.          move    5,up+7(3)
10.          jumpe   5,L3
11.          move    6,r             ; test down[r+c-1]
12.          add     6,c
13.          move    7,down-2(6)
14.          jumpe   7,L3
15.          move    8,r
16.          setzb   9,down-2(6)     ; down[r+c-1] = 0
17.          movem   9,up+7(3)       ; up[r-c+8] = 0
18.          movem   9,rows-1(8)     ; rows[r] = 0
19.          move    4,c             ; x[c] = r
20.          movem   8,x-1(4)
21.          caie    4,8             ; test c == 8
22.          jrst    L4
23.          pushj   15,print
24.          jrst    L5
25.  L4:     addi    4,1             ; call queens(c + 1)
26.          push    15,4
27.          pushj   15,queens
28.          adjsp   15,-1
29.  L5:     movei   4,1
30.          movem   4,down-2(6)     ; down[r+c-1] = 1
31.          movem   4,up+7(3)       ; up[r-c+8] = 1
32.          movem   4,rows-1(8)     ; rows[r] = 1
33.  L3:     aos     r               ; r = r + 1
34.          jrst    L1
35.  L2:
```

The effects of lazy instruction selection appear in the use of immediate mode instructions in lines 1, 4, 21, 25, and 29 and in the use of the `setzb` instruction in line 16 to clear both `down[r+c-1]` and register 9. The addressing offsets in lines 9, 13, 16-18, 20, and 30-32 and the increment instruction (`aos`) in line 33 illustrate the effects of expression rearrangement. The

use of the cache to track resources is shown in lines 6 and 8 where register 3 contains r, in lines 21 and 25 where c is in register 4, and in lines 16-18 and 29-32 where the righthand side of the multiple assignments is reused. Significant improvement appears in the addressing of the arrays rows, down, and up. Repetition of the computation of the subscripts for up and down done in lines 8-12 is avoided in the assignment code in lines 16-18, and these subscript computations are still available for the subsequent assignment in lines 30-32. For this example, use of the cache alone accounted for a 24% improvement.

This example also illustrates the effect of the 'one-pass' nature of the cache-based approach. The value of r is loaded into register 3 in line 3 and used in lines 4, 6, and 8. Line 8, however, changes register 3 to r-c, thereby 'losing' r. Lines 11 and 12 show a similar effect in computing r+c. Since r is used again, it must be reloaded (lines 11 and 15). More sophisticated algorithms might move r to other registers to compute r-c and r+c to avoid the additional loads, if that proved cost effective (saving 1 instruction in this example). Loop optimization algorithms would go one step further and dedicate a register to r for the entire loop, which would also permit the instructions in lines 33 and 34 to be replaced by one instruction.

CONCLUSIONS

Despite the approximate nature of the optimizations described in the previous sections, they are very effective in practice. This experience substantiates claims that simple improvements tend to pay the greatest dividends.[8,11,14] The success of using a cache to track subtree computations is another indication of the general effectiveness and importance of caching.

The implementation of the optimizations is relatively straightforward and has little impact on the performance and size of the compiler. Expression rearrangement and lazy instruction selection were the easiest to implement. The cache mechanism was not much more difficult, but the unpredictable lifetimes of resources, such as registers, required additional code to keep track of them properly. In addition, relating the cache operations (*push* and *pop*) to source language constructs requires care; an error can cause unpredictable results.

All three optimizations must be implemented to achieve their maximum effect. Expression rearrangement enhances the effectiveness of cache-based resource allocation by exposing common subexpressions, especially those that deal with addressing. Lazy instruction selection adds a final touch, polishing the rough edges off the instruction sequence.

The obvious disadvantages of these optimizations are the lack of completeness, retargetability, and guaranteed correctness of the automated approaches to code generation. When available, tools using the automated approaches are to be preferred, assuming their cost is bearable. The quality of the code produced by such tools is comparable to, or better than, that produced by the three optimizations. Their cost, in terms of compiler size and additional passes, remain considerably higher, however. If such tools are unavailable or inapplicable, these three optimizations are a cost-effective alternative, especially for one-pass compilers.

This disadvantage can be overcome by automating the optimizations described in this paper and other similar improvements. For example, expression rearrangement and lazy instruction selection are similar to the pattern matching and replacement schemes used in some peephole optimizers[31] and may be amenable to similar automation. Likewise, it may be possible to include the specification of the use of the cache in the grammar or semantic specification[32] for the language and have the appropriate actions generated automatically. Current research efforts are directed towards such automation, emphasizing the application of the improvements to one-pass compilers without sacrificing their simplicity and effectiveness.

ACKNOWLEDGEMENTS

Conversations and competition with Chris Fraser and Jack Davidson contributed to this work and to its explanation.

REFERENCES

1. S. S. Muchnick and N. D. Jones, eds., *Program Flow Analysis: Theory and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1981.
2. J. P. Banning, 'An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables', *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, TX, 29-41 (1979).
3. E. W. Myers, Jr., 'A Precise Interprocedural Data Flow Algorithm', *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, Williamsburg, VA, 219-230 (1981).
4. J. L. Hennessy and T. R. Gross, 'Code Generation and Reorganization in the Presence of Pipeline Constraints', *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, 120-127 (1982).
5. M. Ganapathi and C. N. Fischer, 'Description-Driven Code Generation using Attribute Grammars', *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, 108-119 (1982).
6. R. S. Glanville and S. L. Graham, 'A New Method for Compiler Code Generation', *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ, 231-240 (1978).
7. J. W. Davidson and C. W. Fraser, 'The Design and Application of a Retargetable Peephole Optimizer', *ACM Trans. Prog. Lang. and Systems*, **2**, 191-202 (1980).
8. D. E. Knuth, 'An Empirical Study of FORTRAN Programs', *Software—Practice & Experience*, **1**, 105-133 (1971).
9. J. L. Bentley, *Writing Efficient Programs*, Prentice Hall, Englewood Cliffs, NJ, 1982.
10. R. P. Cook and I. Lee, 'A Contextual Analysis of Pascal Programs', *Software—Practice & Experience*, **12**, 195-203 (1982).
11. S. C. Johnson, 'A Portable Compiler: Theory and Practice', *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ, 97-104 (1978).
12. R. L. Sites, 'Programming Tools: Statement Counts and Procedure Timings', *SIGPLAN Notices*, **13**, 98-101 (1978).
13. J. Cocke and P. Markstein, 'Measurement of Program Improvement Algorithms', Tech. Rep. RC-8111, IBM Research, Yorktown Heights, NY, 1980.
14. W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke, *The Design of an Optimizing Compiler*, American Elsevier, New York, NY, 1975.
15. D. R. Hanson, 'The Y Programming Language', *SIGPLAN Notices*, **16**, 59-68 (1981).
16. B. W. Kernighan, 'RATFOR—A Preprocessor for a Rational Fortran', *Software—Practice & Experience*, **5**, 395-406 (1975).
17. B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison Wesley, Reading, MA, 1976.
18. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ, 1978.
19. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison Wesley, Reading, MA, 1977.
20. G. Logothetis and P. Mishra, 'Compiling Short-Circuit Boolean Expressions in One Pass', *Software—Practice & Experience*, **11**, 1197-1214 (1981).
21. U. Ammann, 'On Code Generation in a Pascal Compiler', *Software—Practice & Experience*, **7**, 391-423 (1977).
22. J. J. Arsac, 'Syntactic Source to Source Transforms and Program Manipulation', *Comm. ACM*, **22**, 43-54 (1979).

23. A. S. Tanenbaum, 'Implications of Structured Programming for Machine Architecture', *Comm. ACM*, **21**, 237-246 (1978).
24. S. L. Graham, R. R. Henry and R. A. Schulman, 'An Experiment in Table Driven Code Generation', *Proceedings SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, 32-42 (1982).
25. D. R. Hanson, 'Code Improvement via Lazy Evaluation', *Inf. Proc. Letters*, **11**, 163-167 (1980).
26. P. Henderson and J. H. Morris, 'A Lazy Evaluator', *Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages*, Atlanta, GA, 95-103 (1976).
27. C. W. Fraser and J. W. Davidson, 'Eliminating Redundant Object Code', *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, 128-132 (1982).
28. W. E. Weihl, 'Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables', *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, Las Vegas, NV, 83-94 (1980).
29. F. Baskett, 'The Best Simple Code Generation Technique for While, For, and Do Loops', *SIGPLAN Notices*, **13**, 31-32 (1978).
30. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, Englewood Cliffs, NJ, 1976.
31. D. A. Lamb, 'Construction of a Peephole Optimizer', *Software—Practice & Experience*, **11**, 639-648 (1981).
32. R. Sethi, 'Control Flow Aspects of Semantics Directed Compiling', *Proceedings SIGPLAN '82 Symposium on Compiler Construction*, Boston, MA, 245-260 (1982).