Code Generation for Expressions with Common Subexpressions

EXTENDED ABSTRACT

A. V. Aho and S. C. Johnson
Bell Laboratories,
Murray Hill, New Jersey 07974

J. D. Ullman[*]
Princeton University
Princeton, New Jersey 08540

1. Introduction

Easy as the task may seem, many compilers generate rather inefficient code. Some of the difficulty of generating good code may arise from the lack of realistic models for programming language and machine semantics. In this paper we show that the computational complexity of generating efficient code in realistic situations may also be a major cause of difficulty in the design of good compilers.

We consider the problem of generating optimal code for a set of expressions. If the set of expressions has no common subexpressions, then a number of efficient optimal code generation algorithms are known for wide classes of machines [SU, AJ, BL].

In the presence of common subexpressions, however, Bruno and Sethi have shown that the problem of producing optimal code for a set of expressions is NP-complete, even on a single register machine [BS, S1]. However, Bruno and Sethi's proof of NP-completeness uses rather complex expressions, so it leaves some hope of being able to find efficient algorithms for generating optimal code for restricted classes of expressions with common subexpressions. Unfortunately, we show in this paper that the problem of optimal code generation remains NP-complete even for expressions in which no shared term is a subexpression of any other shared term. We also show that the optimal code generation problem is NP-complete for these expressions on two-address machines, even when the number of registers is unlimited.

Faced with these negative results, we consider both heuristic and exact solutions for generating code. First, we investigate the worst case performance of a collection of fast heuristics for single and multiregister machines. One seemingly reasonable heuristic is shown to produce code that is in the worst case three times as long as optimal; other heuristics are given which have a worst case of 3/2 for one-register machines.

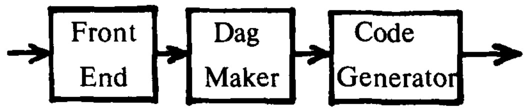
Then, we present for the single register machine an algorithm which generates optimal code and whose time complexity is linear in the size of an expression and exponential only in the amount of sharing. Since the number of common subexpressions in expressions tends to be limited in practical situations, this approach appears attractive. Finally, after discussing code generation for commutative machines, we conclude with a list of open problems.

* Work partially supported by NSF grant DCR74-15255.

2. Background and Definitions

2.1 Dags

For the purposes of this paper we can assume we have a compiler of the form



The front end translates a source program into a sequence of straight-line intermediate code segments, called basic blocks, along with information that identifies the flow of control among basic blocks. Within a basic block the flow of control is sequential.

Each basic block consists of a sequence of assignment statements of the form

$$a \leftarrow b \textbf{ op } c$$

where a, b, and c are distinct variables and **op** is any binary operator for which there exists a corresponding machine operation. Since we shall concentrate on generating code for basic blocks in this paper, the flow of control information will not be mentioned further.

The dag maker transforms the basic blocks into a directed acyclic graph (dag, for short) that represents the computations of the expressions in the block. (See [AU] or [CS] for algorithms to construct a dag from a basic block.) We shall not consider the use of algebraic identities to transform dags to make them easier to compute; this has been discussed in [Br], [Fa].

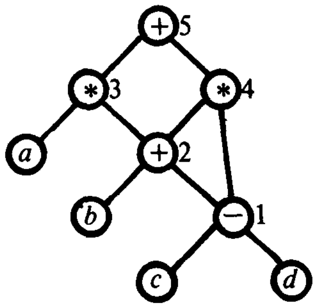
Fig. 1 shows a basic block and its corresponding dag.

We call node 2 a *right child* of node 3, and a *left child* of node 4. For symmetry, we call node 4 a *right parent* of node 2, and node 3 a *left parent* of node 2.

We say node x *uses* a node y if y is either a left or right child of x. We say x *left uses* (resp. *right uses*) y if y is a left (resp. right) child of x.

$$\begin{aligned}
u_1 &\leftarrow c - d\\
u_2 &\leftarrow b + u_1\\
u_3 &\leftarrow a * u_2\\
u_4 &\leftarrow u_2 * u_1\\
u_5 &\leftarrow u_3 + u_4
\end{aligned}$$

Basic Block



Dag

Fig. 1 Basic Block and its Dag

A node with no children is called a *leaf.* A node with no parents is called a *root.* Nodes that are not leaves are often referred to as *interior* nodes.

We assume for simplicity that all operations are binary. We ignore constraints that may be introduced into the dag because of side effects. For example, suppose two nodes of a dag represent the operation of indirect assignment through a pointer. If the two pointers could point to the same datum, and the source code specified an order for the two assignments, then an edge in the dag connecting the two nodes in the proper order must be introduced.

2.2 The Machine Model

We assume the code generator is to produce code for a multiregister two-address machine. The instructions of the machine are of the form

(1) $r_i \leftarrow r_i \textbf{ op } r_j$	/* op-instruction */
(2) $r_i \leftarrow r_i \textbf{ op } m$	/* op-instruction */
(3) $r_i \leftarrow r_j$	/* register copy */
(4) $r_i \leftarrow m$	/* load */

(5) $m \leftarrow r_i$ /* store */

Here r_i and r_j are any of $N \geqslant 1$ registers, and m is any memory location; **op** stands for any machine operation. When a machine has only one register (i.e., $N = 1$), then there are only type (2), type (4), and type (5) instructions.

A *machine program* is a sequence of instructions. The *length* of a program is the number of instructions it contains.

Definition: The *optimal code generation problem* (OCG) is to produce from a dag a shortest machine program that evaluates and stores all roots of the dag.

When discussing single and multiregister machines, we assume the leaves of the dag are labeled by memory locations and the interior nodes (non-leaves) by machine operations. We also assume for convenience that a dag has no root that is a leaf.

Example 1: The dag of Figure 1 can be evaluated on a two-register machine and stored in memory location m by the sequence of instructions:

```
r1 ← c              /* load c */
r1 ← r1 - d         /* evaluate node 1 */
r2 ← b              /* load b */
r2 ← r2 + r1        /* evaluate 2 */
 t ← r1             /* store 1 */
r1 ← a              /* load a */
r1 ← r1 * r2        /* evaluate 4 */
r2 ← r2 * t         /* evaluate 3 */
r1 ← r1 + r2        /* evaluate 5 */
 m ← r1             /* store root */  □
```

In Section 6, we shall discuss code generation for *commutative* machines; i.e., machines in which for every type (1) and (2) op-instruction above, there is also an instruction of the form

(1') $r_i \leftarrow r_j$ **op** r_i
(2') $r_i \leftarrow m$ **op** r_i

Even for noncommutative machines, OCG is a very difficult problem; the next section discusses why.

3. Why OCG is Hard

Bruno and Sethi have shown that OCG is NP-complete even for one-register machines. Their proof technique was to polynomially transform the satisfiability problem with three literals per clause (see [AHU], e.g.) to OCG. Their technique, however, resulted in rather complex dags. We begin by showing that OCG is NP-complete for one-register machines even on a rather simple class of dags.

A node both of whose children are leaves is called a *level-one node.* A *shared node* in a dag is a node with more than one parent. A *level-one* dag is a dag in which every shared node is a level-one node. A *leaf* dag is a dag in which every shared node is a leaf [C].

Several code generation algorithms for one-register machines make use of the notion of a *left chain,* that is, a sequence of interior nodes $n_1, n_2, \ldots, n_k$ such that n_i is the left child of n_{i+1} for $1 \leqslant i < k$. For example, in the dag of Figure 1, 3-5 and 2-4 are the only nontrivial left chains. The first (lowest) node on a left chain is called its *tail,* and the last (highest) node its *head.*

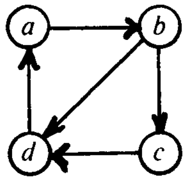
Definition: The *feedback node set problem* (FNS) is: Given a directed graph G, find a smallest set of nodes F *(a feedback node set)* such that removing F from G eliminates all cycles. FNS is a well-known NP-complete problem (see [AHU], e.g.).

Theorem 1: OCG for level-one dags on a one-register machine is NP-complete.

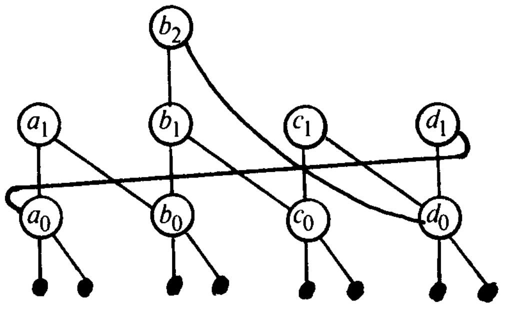
Proof: We show how to polynomially transform an instance of FNS to OCG. Let G be the directed graph in the feedback node set problem. From G construct a dag D as follows. For each node n in G of out-degree $d \geqslant 0$ create a corresponding left chain of $d+1$ interior nodes $n_0, n_1, \ldots, n_d$ in D; n_0 is the tail and n_d the head of the chain. Make n_0 a level-one node by giving it left and right children labeled by memory locations.

The remaining edges of D are determined as follows. Suppose the out-edges of n are directed to nodes $m_1, m_2, \ldots, m_d$ in G. Make the tails of the left chains corresponding to $m_1, m_2, \ldots, m_d$ right children of $n_1, n_2, \ldots, n_d$, respectively, in D.

Fig. 2(a) shows a directed graph G and Fig. 2(b) the dag resulting from G using the construction above.



(a) Directed Graph G



(b) Resulting Dag D

Fig. 2 Graph and Corresponding Dag

We must now show that we can construct a minimal feedback node set F for G from an optimal program P for D and conversely. It can be shown that an optimal program for a one-register machine does not store and load any uniquely left-used interior nodes of a dag (interior nodes with exactly one left use and no right uses). Thus except for the loads of leaves, the only loads in P are loads of some level-one nodes. It can be seen that these level-one nodes identify a feedback node set F in G. Conversely, given a minimal feedback node set F of G we can construct an optimal program for D by first evaluating and storing the tails in D of the left chains corresponding to the nodes in F.

For example, $\{d\}$ is a minimal feedback node set for the directed graph G of Fig. 2(a). The optimal program P corresponding to this feedback node set first computes node d_0 of D in Fig. 2(b). Then P evaluates $c_0, c_1, b_0, b_1, b_2, a_0, a_1, d_1$. To evaluate d_1, P needs to load d_0; this is the only level-one node loaded by P. □

To appreciate the difficulty of generating optimal code for dags, let us assume we are generating code for an *infinite* register machine, a machine in which the number of registers is unbounded. To eliminate the problem of deciding what to store or load, let us further assume the leaves of the dag are labeled by register names rather than by memory locations; similarly, let us assume we need not store the roots. The relevant instructions of the infinite register machine then become

$$r_i \leftarrow r_i \textbf{ op } r_j$$
$$r_i \leftarrow r_j$$

Even in this highly simplified environment, the optimal code generation problem is NP-complete.

Theorem 2: OCG for leaf dags on an infinite register machine is NP-complete.

Proof: Similar to that of Theorem 1. □

Thus, even if the problems of code selection and storage of intermediate values are made trivial, just finding an optimal evaluation order for the nodes of a dag is an NP-complete problem. On the other hand, if we perturb the infinite register machine architecture by permitting arbitrary three-address instructions of the form $r_i \leftarrow r_j \textbf{ op } r_k$, then we can generate optimal code for arbitrary dags in linear time. We simply evaluate the dag bottom up, level by level, assigning a distinct register to each node.

The three main problems in code generation are what instructions to use, in what order to do the computations, and what values to keep in registers. The results of this section indicate that, for two-address machines, just deciding the order in which instructions are to be executed is an NP-complete problem.

4. Heuristic Techniques

Since even simple versions of the optimal code generation problem are NP-complete, it is not surprising that in the past code generation algorithms for dags usually have made several restricting assumptions. One approach has been to ignore sharing by representing a set of expressions as a forest of trees. For this case a number of optimal code generation algorithms have been developed [SU], [AJ], [BL], [Was]. Another approach has been to avoid the problem of finding an optimal evaluation order by taking some fixed order for the nodes of a dag and then concentrating on optimal utilization of registers ([Bea], [HKMW], [WJWHG], [Fr], e.g.). Optimal code cannot be guaranteed, however, without considering sharing and retaining the freedom to reorder code that is inherent in the source program.

When faced with an NP-complete problem, there are two standard approaches:

(1) develop and analyze heuristics, and

(2) look for useful special cases that have polynomial time algorithms.

We shall consider both approaches here. For the analysis of heuristics we use the worst case measure, a time-honored way of measuring the goodness of a heuristic which may be applied to various populations of data. For our purposes, we define the *worst case* of an algorithm to be the supremum over all dags of the ratio of the length of the code produced by that algorithm to the length of the optimal code for the dag.

4.1 Two Methods of Accounting for Costs

There is a usual way of charging the interior nodes of a dag for the cost of their evaluation. Charge a node one unit of cost for each of: (1) performing its operation, (2) storing its value, and (3) loading its left child or copying its left child from another register.

We call a program *semi-intelligent* if

(1) it performs no useless instructions, i.e., instructions which can be deleted without changing the value of the program,

(2) it never moves (via loads or register-to-register transfers) a value into a register without subsequently left using that value, and

(3) it never stores the same value more than once.

It is easy to check that a semi-intelligent program will have each of its instructions assigned to some one node by the above scheme. Formally, we may show the following.

Theorem 3: Let P^* be an optimal program for some dag D, and let P be any semi-intelligent program for D. Then the ratio of the length of P to that of P^* is at most 3.

Proof: Using the above costing scheme, every interior node of D is assigned at least one instruction, but never more than three instructions (a load, an operation, and a store). □

Obviously there are an infinite number of programs to evaluate any dag. We shall restrict ourselves to semi-intelligent programs. Doing so serves only to rule out blatantly inefficient programs. There is little loss of generality since we can construct from any program P an equivalent semi-intelligent program in time proportional to the length of P.

There is a second cost accounting scheme which we find quite useful. This scheme gives the same overall cost as the scheme above but the cost units are apportioned differently amongst the nodes. We now charge each node for every instruction that affects its value, i. e., for the operation that computes its value, and any loads, stores, or register-to-register copies of that value. We call this cost accounting scheme the *use-cost.*

If n is a node of a dag, let $l(n)$ and $r(n)$ be the number of times n is used as a left and right child, respectively, of some other node. In the dag of Figure 1, for example, $l(1) = 0$ and $r(1) = 2$. Also, $l(2) = r(2) = 1$.

Lemma 1: The following costs are upper and lower bounds on the use-cost of a node n with respect to any semi-intelligent program on a one-register machine.

case	*lower*	*upper*
(1) n is a leaf	$l(n)$	$l(n)$
n is not a leaf and:		
(2) $l(n) = 0$	2	2
(3) $l(n) > 1$	$l(n)+1$	$l(n)+2$
(4) $l(n) = 1$ and $r(n) = 0$	1	3
(5) $l(n) = 1$ and $r(n) > 0$	2	3

Proof: Let us consider case (3) as an example. Surely an op-instruction evaluating n is necessary. Since n is used more than once and there is only one register, one store instruction is also necessary. Every time n is left used, its value in the register is destroyed, and it must be reloaded the next time it is left used. Thus, $l(n) - 1$ load instructions are necessary, and $l(n)$ load instructions are sufficient by condition (2) in the definition of semi-intelligent. Therefore, a total of $l(n)+1$ or $l(n)+2$ instructions is needed. □

Lemma 2: For a multiregister machine, Lemma 1 holds with the lower bound set to 1 in cases (2) and (5) and to $l(n)$ in case (3).

Proof: Omitted. □

We observe from Lemma 1 that it is only interior nodes with one left parent and no right parent that could give us a worst case ratio of 3 for the one-register machine. Interestingly, it is quite easy to handle such cases. If node n is uniquely left-used, we can always arrange to have n evaluated immediately before its parent. Thus it is unnecessary to load or store n, and it achieves a use-cost of 1. Therefore, we can state the following.

Theorem 4: Any algorithm for a one-register machine which generates semi-intelligent programs that avoid storing uniquely left-used nodes has a worst case no greater than 3/2.

Note that Theorem 4 fails to hold in the multiregister case, since $l(n) = 1$, $r(n) > 0$ is another case that can yield a worst case ratio of 3, and some other cases yield a ratio of 2.

We must also point out that the use-cost can, in some cases, underestimate the cost of an optimal program on a one-register machine by a factor of 3/2. Fig. 3, for example, shows a dag whose lower bound use-cost is $6+2p$. This lower bound, however, is not achievable since any program evaluating this dag must store and subsequently load each of the +-nodes. The cost of an optimal program for the dag is $6+3p$.

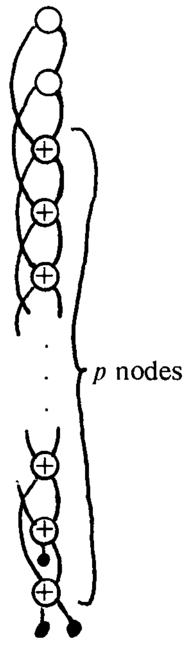


Fig. 3 Underestimated Dag

Thus, if we found an algorithm with a worst case ratio less than 3/2, the proof of that worst case bound must use a more sophisticated cost analysis than the above.

4.2 Heuristics for One Register Machines

An *evaluation order* for a dag is any topological sort of the interior nodes of the dag. For a one-register machine, from an evaluation order we can easily construct a program that is as short as any other which computes the dag in that order. Thus an algorithm that produces an evaluation order for a dag is in effect a code generation algorithm for a single register machine. In this section we analyze the performance of some simple heuristics for creating evaluation orders for single register machines.

Definition: The *Bottom-up Greedy Algorithm (BUG)* creates an evaluation order for a dag by repeatedly listing the nodes of a longest left chain that can be currently evaluated. To evaluate a node, both its children must have been previously evaluated. Note that it is permissible for the right child of a node in a chain to be an unevaluated node which is lower on the same chain.

Example 3: Return to the dag of Figure 1. The only left chain that can be listed initially is 1. Then the chain 2-4 can be listed, and finally the chain 3-5, giving an evaluation order of 12435. In this case the code is:

$$
\begin{aligned}
r_1 &\leftarrow c\\
r_1 &\leftarrow r_1 - d\\
t &\leftarrow r_1\\
r_1 &\leftarrow b\\
r_1 &\leftarrow r_1 + t\\
u &\leftarrow r_1\\
r_1 &\leftarrow r_1 * t\\
v &\leftarrow r_1\\
r_1 &\leftarrow a\\
r_1 &\leftarrow r_1 * u\\
r_1 &\leftarrow r_1 + v\\
m &\leftarrow r_1
\end{aligned}
$$

By Lemma 1, this program is optimal. □

Theorem 5: BUG has a worst case ratio of 3.

Proof: Figure 4 shows how the worst case of 3 can be approached arbitrarily closely. At the right we show p nodes $c_1, c_2, \ldots, c_p$, called "controllers," whose initial evaluation is necessary for optimal code. If we evaluate the controllers first and store them, at a cost of $3p$, we can go up each of the p left chains with a cost of $p + 2$ per chain, for a total of $p^2 + 5p$.

However, BUG could select c_1, then the bottom node of each chain, then c_2, then the next node of each chain, then c_3, and so on, using three instructions per interior node. The worst case ratio for this example is $3(p + 1)/(p + 5)$. □

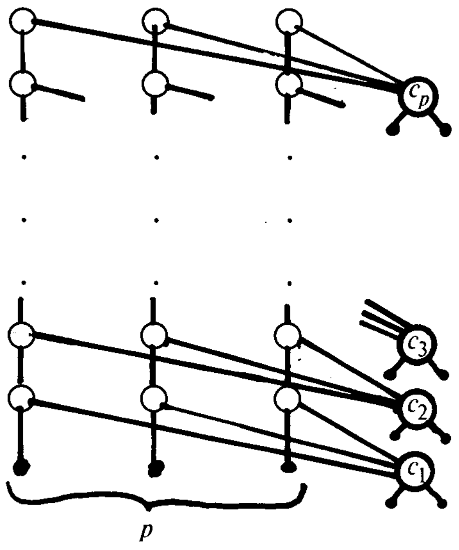


Fig. 4 BUG Buggerer

Definition: The *Top-down Greedy Algorithm* (TDG) ([AU], p. 866; see also [Wai]) works by listing left chains in the reverse order of their evaluation. Repeatedly select a node n all of whose parents, if any, have already been listed. Then list n and as many nodes of the left chain with head n as may be listed. Note that a node may be listed only if all of its parents have already been listed, as we are generating the evaluation sequence in reverse. Also, once an unlistable node is encountered, we do not proceed further down the chain. After sequencing all interior nodes, reverse the list to get the evaluation order.

Example 4: In the dag of Figure 1, we would select the root 5 first and find we may list it and its left child 3. Then we could select 4, since its parent, 5, has already been listed. We could proceed to 2, the left child of 4, since its parents, namely 3 and 4, have been listed. Finally we list 1. Reversing the list gives 12435 again, so the same code as for BUG is produced in this case. □

TDG and BUG, however, are quite different in their worst case performance. TDG produces optimal code for Fig. 2 for which BUG produced the worst case code.

Theorem 6: TDG has a worst case ratio of 3/2.

Proof: To see that it is no worse than 3/2, note that if n is uniquely left-used by m, then n must be listed immediately after m, and therefore n is evaluated immediately before m. Thus case (4) of Lemma 1 is always handled correctly; the lower bound of one instruction charged to n is attained. All other cases in Lemma 1 have a ratio of 3/2 or less.

For the proof that 3/2 can be approached from below, consider the grid of Figure 5. The optimal sequence of evaluation goes up (the slanted) left chains, starting at the bottom right, storing each value with the exception of those on the leftmost chain. Roughly two instructions per node are used in this evaluation sequence. On the other hand, TDG could list nodes row by row, from the right, taking three instructions per node. □

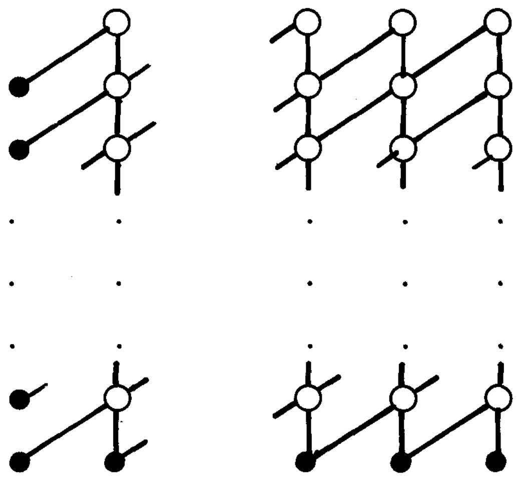


Fig. 5 Grid

It is also worth noting that TDG always handles the case $r(n) = 0, l(n) > 1$ correctly, since n will be listed immediately after the last of its left parents to be listed. Thus the generated code will have only $l(n) - 1$ loads of n.

In truth there is no magic about "top down" vs. "bottom up" algorithms; by exercising care in the selection of chains it is easy to construct a modification of BUG that has the same performance as TDG.

Another 3/2 worst case algorithm can be obtained using depth first search.

Definition: The *Depth-first Search Algorithm (DFS)* performs a depth-first search (see [AHU], e.g.) of the dag, preferring to move to the right child rather than the left when there is a choice. Nodes are then evaluated in order of last visit.

Theorem 7: The worst case ratio for DFS is 3/2.

4.3 Heuristics for Multiregister Machines

The Top-down Greedy Algorithm can be generalized to the case of an N register machine. For a multiregister machine, however, it is no longer sufficient to specify only an evaluation order; we must also specify in which register a computation is to be done. The following procedure lists the interior nodes of a dag in reverse evaluation order. The register assigned to a node is the register in which that node is to be computed. Stores and loads of registers are performed as needed.

```
**procedure** TDG($n$, $k$); /* $n$ is a node, $k$ is the number of registers available */
**if** $k \geqslant 1$ and $n$ is an interior node all of whose parents have been listed **then**
    **begin**
        list $n$ and assign it register $k$;
        TDG(right child of $n$, $k-1$);
        TDG(left child of $n$, $k$)
    **end**;
/* main program */
**while** not all interior nodes have been listed **do**
    select an interior node $n$, all of whose parents have been listed, and perform TDG($n$, $N$)  .
```

Although TDG performs well in many cases, the worst case performance of TDG approaches 3 as N gets large.

Theorem 8: The worst case ratio for the TDG Algorithm with N registers is no less than $3N/(N + 1)$.

Proof: The grid of Figure 5 provides the essence of the proof. □

Demers [D] has considered a generalization of DFS to multiregister machines in which a depth-first search is used to obtain an ordering of the nodes, and then Beladay's algorithm [Bel] is used to allocate registers.

5. An Optimal Algorithm for the One Register Machine

We define $s(n)$, the *sharing* of a node n in a dag, to be

$$
\begin{array}{ll}
s(n) = l(n) + 1 & \text{if } l(n) > 1 \\
s(n) = 2 & \text{if } l(n) = 1 \text{ and} \\
 & \quad r(n) > 0 \\
s(n) = 0 & \text{otherwise}
\end{array}
$$

The sharing, s, for a dag is the sum of the $s(n)$ over all interior nodes n in the dag.

We shall now present an algorithm for the one register case that is optimal and is of time complexity $O(p2^s)$, where p is the number of nodes in the dag and s is the amount of sharing. To introduce this algorithm, we note that any dag can be partitioned in various ways into left chains. Given a program for a dag, we can create a partition by looking for maximal sequences of one or more consecutive operations uninterrupted by loads. Each such consecutive sequence forms a left chain, and the set of left chains so formed partitions the interior nodes of a dag.

We can obtain a partial converse of the above. We say that a partition of the interior vertices of a dag D is *legal* if the following holds: Form a graph G whose nodes correspond to the left chains of D and with an edge from c_1 to c_2 whenever there is a path in D from some node of c_1 to some node of c_2. Then there must be no cycles in G. We may thus state the following.

Lemma 3: There is an evaluation order producing a given partition of a dag into left chains if and only if that partition is legal.

More importantly, we can relate the cost of evaluation sequences to the costs of the heads of the left chains.

Theorem 9: Let D be a dag. Then there is a constant c_D with the following property: Suppose P is any semi-intelligent program evaluating D. Let the partition induced by P have k_1 chains whose heads are left used at least once. Let k_2 of these be uniquely left-used. Then the length of P is $c_D + k_1 + k_2$. Conversely, if there is a program for D of cost c, then we can find a legal partition into left chains with parameters k_1 and k_2 such that $c = c_D + k_1 + k_2$.

Proof: c_D is the sum over all interior nodes of D of the lower bound on cost given in Lemma 1. k_1 accounts for excess loads of left-used nodes, that is, the lth load of a node left used l times. Note that a node will be loaded l rather than $l-1$ times if and only if it is the head of a chain. k_2 accounts for stores of uniquely left-used nodes. □

We intend to reduce the problem of finding an optimal program to that of finding a set of heads of chains. Clearly any node with $l(n) = 0$ must be the head of a chain. A node with $r(n) = 0$ and $l(n) = 1$ can always be attached to its left parent in a chain, as claimed in the following lemma.

Lemma 4: If π is a legal partition of dag D, and there is a uniquely left-used node n which heads a chain, then the partition formed from π by removing n from its current chain and attaching it to the chain of its parent is also legal.

Proof: If there is a path to n in D, it must go through the unique parent of n. □

We thus see that the question of whether or not a node is the head of a chain in an optimal program is only unresolved for those interior nodes with more than one parent, at least one of which is a left parent. We may thus try all subsets of these nodes, selecting those which are *not* the head of a chain. For each selected node with more than one left parent, we must also select the left parent to whose chain it is attached. The number of these selections can be shown not to exceed 2^s, where s is the sharing. We may thus test each selection, in order of lowest cost, until we find one that is legal. The test for legality is seen to be $O(p)$ on a p node dag. Thus:

Theorem 10: There is an $O(p2^s)$ algorithm for obtaining an optimal program for a dag.

6. Commutative Machines

A *commutative machine* is one in which for each instruction of the form $r_i \leftarrow r_i$ **op** r_j or $r_i \leftarrow r_i$ **op** m there is also an instruction of the form $r_i \leftarrow r_j$ **op** r_i or $r_i \leftarrow m$ **op** r_i. That is, the register used to hold the result can be the same as as either the left or the right register operand. This arrangement allows us to think of the order of the children of any node as being permutable, as far as code generation is concerned, although the operator itself may not be commutative.

For one-register commutative machines the analog of a left-chain is a *worm.* A worm is any path in a dag, excluding the leaves. We define a partition of a dag D into worms to be *legal* if the graph whose nodes are the worms of D, with an edge from w_1 to w_2 if and only if some node of worm w_1 has a path in D to some node of worm w_2, is acyclic. The following is an analog to Theorem 9.

Theorem 11: Let D be a dag. Then there is a constant c_D with the following property. Suppose D has a legal partition into worms such that there are k_1 worms whose heads are used (either left- or right-used), k_2 of which are uniquely used. Then there is a program for D on a commutative machine of cost $c_D + k_1 + k_2$. Conversely, if there is a program for D with cost c, then we can find a legal worm partition with parameters k_1 and k_2 as above, such that $c = c_D + k_1 + k_2$.

We can generalize the top-down greedy algorithm to commutative machines by listing worms in reverse evaluation order. If we always list a uniquely used child immediately after its parent, then we can show that 3/2 is a worst case ratio for this algorithm for the commutative one-register machine.

We can also produce an analog of the optimal algorithm of Section 5 which is polynomial in the size of the dag and exponential only in the sharing. The following lemma is needed; it is not as strong a result as could be proved about worm partitions.

Lemma 5: Let W be a legal worm partition of a dag. Suppose node n_1 is the head of a worm $w = n_1, n_2, \ldots, n_k$ in which $n_1, n_2, \ldots, n_k$ are all uniquely used. Then there is another worm partition W' of D with a worm of which $n_1, n_2, \ldots, n_k$ is a proper tail (that is, the new worm includes at least the parent of n_1) such that the cost of the program induced by W' is no greater than the cost of the program induced by W.

Proof: Fig. 6 shows a fragment of a dag in which node m is on some worm; perhaps n, m's other child, is on the same worm.

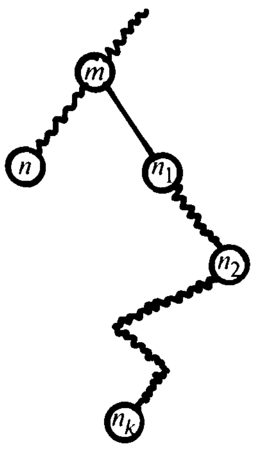
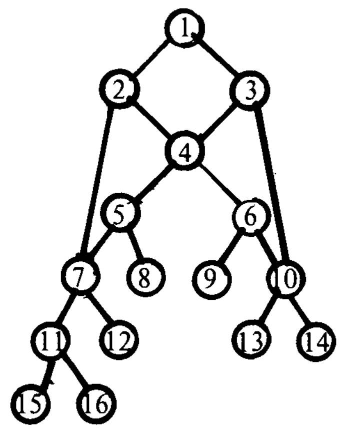


Fig. 6 Worm Construction

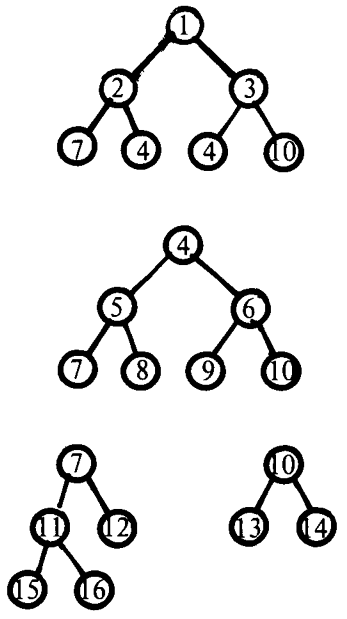
Remove edge $m \rightarrow n$ from that worm if it is there, and add the edge $m \rightarrow n_1$. This may make node n a head, but n_1 will no longer be a head. It is easy to show that the cost of the partition is not increased, and the fact that $n_1, n_2, \ldots, n_k$ are uniquely used makes a proof of legality for the new partition easy. □

Theorem 12: There is an $O(n6^s)$ algorithm for finding an optimal program on a commutative machine for an n-node dag with sharing s.

Proof: The first step is to take the dag D and divide it into trees, an idea discussed by [Wai]. For each root or shared node n, we do the following. Find the maximal subtree with n as root which includes no other shared nodes, except as leaves. An example is shown in Fig. 7.



(a) Dag



(b) Trees

Fig. 7 Division of a Dag into Trees

Note that each multiply-used node appears as a leaf in at most one tree per use. If it is used by two nodes in the same tree, divide the shared node into as many leaves as necessary so that no leaf is used more than once. Note also that no non-shared node appears in more than one tree.

The algorithm begins by determining which edges of the various trees belong to worms. Later we shall combine the trees to allow worms to cross tree boundaries. Imagine worms proceeding upward from all leaves in a given subtree. The fact that worms do not actually reach the leaves can be accounted for later by removing edges from worms at the bottom. At each interior node *n* we must decide which, if any, of the worms reaching children of *n* include the node *n*. If either or both of the children of *n* are part of worms that begin at tree leaves of *D* or interior nodes of *D* that are not shared (recall some leaves of a tree here may actually be shared nodes of *D*), then Lemma 5 assures us that we may allow any such worm to continue up to *n*. The only uncertainty occurs when both worms reaching the children of *n* originate at shared nodes of *D*. Then we must try all three possibilities - that either worm or neither incorporates *n*. In the last case, *n* begins a new worm, a situation which could be necessary to achieve a legal worm partition.

We see from the above that the only worms where there is uncertainty regarding how far up to proceed are those which originate at shared nodes of *D*. If tree T_i has k_i leaves which are shared nodes of *D*, then there are at most 3^{k_i} outcomes for the "contests" regarding which worm proceeds upwards. Moreover, for each shared node *n* that is a leaf of T_i, *n*'s worm in T_i may or may not connect with that worm which includes *n* in the tree of which *n* is the root. (Recall the edges from leaves to their parents are not really parts of worms.) Thus, we may either include or exclude the edges from node *n* to its parent in T_i from the set of edges of *D* comprising worms.

Thus for tree T_i there are at most 6^{k_i} subsets of edges which could possibly be the edges used in an optimal worm partition of *D*. Not all subsets of edges need be consistent. For example, some node could be connected to two or more of its parents by selected edges. The number of possible sets of selected edges is no more than $\Pi 6^{k_i}$, which is no greater than 6^s, where *s* is the total sharing of *D*. Each such set of selected edges can be checked for consistency and legality in $O(n)$ time, where n is the number of nodes of D. If legal, the cost of the partition can easily be computed in $O(n)$ time by inspecting the worm heads. We can therefore run through all the candidate worm partitions in $O(n6^s)$ time and select that one with the least cost. □

7. Summary and Open Questions

We have analyzed several simple heuristics for generating code for a one-register machine, showing them to have a worst case of 3/2 or, in one case, 3. We have also shown for the one-register machine, both in the commutative and non-commutative cases, that there are optimal code generation algorithms which are linear in the size of the dag and exponential only in the sharing. Additionally, we have shown that even some very simple code generation problems are NP-complete.

We feel that this work only scratches the surface of what can be learned about the important area of code generation algorithms. We therefore propose the following questions as potentially fruitful areas for future research.

1. Is there an optimal algorithm for multiregister machines which is polynomial in the number of nodes and registers, and exponential only in the amount of sharing?

2. How closely can the optimal code generation problem be approximated by a polynomial time heuristic on (a) single register, (b) multiregister, and (c) infinite register machines? In particular, can we, for all $\epsilon > 0$, develop polynomial time algorithms with a worst case ratio of $1 + \epsilon$? What about the same problems for commutative machines?

3. On some machines certain operations such as multiplication require an even-odd register pair. How do machine anomalies such as these affect the computational complexity of code generation? Is optimal code generation polynomial, even for trees?

4. How difficult is it to generate code for a tree in which some of the leaves are labeled by registers rather than memory locations? The leaves whose values are in registers cause a register to be freed when they are used. Sethi [S2] has shown that we can without loss of generality evaluate any subtree containing a leaf in a register, provided we can do so with no stores, replacing that subtree by a leaf in a register. The problem of what to do when no such reductions are possible appears NP-complete. Is it?

Acknowledgements

The authors wish to thank Brenda Baker, Brian Kernighan, Doug McIlroy, and Elliot Pinson for their helpful comments on the manuscript.

References

[AHU] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison Wesley, 1974.

[AJ] A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *Proc. Seventh Annual ACM Symposium on Theory of Computing,* May 1975, pp. 207-217.

[AU] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling,* Vol. II, *Compiling,* Prentice Hall, 1973.

[Bea] J. C. Beatty, "A Register Assignment Algorithm for Generation of Highly Optimised Object Code," *IBM J. Res. Dev. 18,1* (January 1974), 20-39.

[Bel] L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Syst. J.,* 5:2 (1966) 78-101.

[Br] M. A. Breuer, "Generation of Optimal Code for Expressions via Factorization," *Comm. ACM 12,6* (June 1969), 333-340.

[BL] J. L. Bruno and T. Lassagne, "The Generation of Optimal Code for Stack Machines," *J. ACM 22,3* (July 1975), 382-397.

[BS] J. L. Bruno and R. Sethi, "Register Allocation for a One-Register Machine," TR-157, Computer Science Dept., Penn State Univ., University Park, Pa., Oct., 1974.

[C] S. Chen, "On the Sethi-Ullman Algoport #11, Bell Laboratories, Holmdel, N. J., May, 1973.

[CS] J. Cocke and J. T. Schwartz, *Programming Languages and their Compilers* (second edition) Courant Institute, NYU, New York, 1970.

[D] A. Demers, private communication.

[Fa] R. J. Fateman, "Optimal Code for Serial and Parallel Computation," *Comm. ACM 12,12* (December 1969), 694-695.

[Fr] R. A. Freiburghouse, "Register Allocation Via Usage Counts," *Comm. ACM 17,11* (November 1974), 638-642.

[HKMW] L. P. Horowitz, R. M. Karp, R. E. Miller and S. Winograd, "Index Register Allocation," *J. ACM 13,1* (January 1966), 43-61.

[S1] R. Sethi, "Complete Register Allocation Problems," *SIAM J. Computing 4,3* (September 1975), 226-248.

[S2] R. Sethi, private communication.

[SU] R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *J. ACM 17,4* (October 1970), 715-728.

[Wai] W. M. Waite, "Optimization," In *Compiler Construction: An Advanced Course,* F. L. Bauer and J. Eickel, eds., Springer-Verlag, 1974, pp. 549-602.

[Was] S. G. Wasilew, "A Compiler Writing System with Optimization Capabilities for Complex Order Structures," Ph. D. Thesis, Northwestern Univ., Evanston, Ill., 1971.

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