Code Generation for Machines with Multiregister Operations

EXTENDED ABSTRACT

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

J. D. Ullman†
Princeton University
Princeton, New Jersey

Summary

Previous work on optimal code generation has usually assumed that the underlying machine has identical registers and that all operands fit in a single register or memory location. This paper considers the more realistic problem of generating optimal code for expressions involving single and double length operands, using several models of register-pair machines permitting both single and double word instructions. With register-pair machines a new phenomenom arises that is not present in optimal code generation for single register machines: In an optimal evaluation of an expression it may be necessary to oscillate back and forth between evaluating subexpressions of the expression.

A linear-time optimal code generation algorithm is derived for a register-pair machine in which all registers are interchangeable. The algorithm is based on showing that for this model there is an optimal evaluation sequence with limited oscillation between the subtrees dominated by the children of a given node. For other machine models including the familiar even-odd register-pair machine, optimal evaluation sequences can always require unlimited oscillation.

1. Introduction

In programming languages rich in expressions, like BLISS [W] or C [RKL], we can find expressions containing many operators. In generating good object code for these languages, it is desirable to use registers to hold frequently-used data and addresses. Consequently, without exercising care in the way available registers are used, there may not always be enough registers available to evaluate expressions efficiently.

Previous theoretical work on optimal code generation for expressions (e.g., [AJ], [B], [BL], [N], [R], [SU]) has assumed that the underlying machine has N identical registers, that each data object fits in a single register or memory location and that all operators produce a result in a single register. Unfortunately, few real machines have such a simple register architecture. Many have special purpose registers for handling floating point operations, integer multiplication and division, addressing, condition codes, etc. In this paper we make a modest sally into this world of reality by considering the influence of operand size and register structure on the complexity of code generation.

We relax the condition that data occupies one register or memory location by allowing operands and results of instructions to have size *single* or *double*, that is, taking one or two registers or memory locations. We shall call machines in which all instructions operate on single-sized operands *single-register machines*. Machines with both single and double instructions will be called *register-pair machines*. A precise definition of these machine models is contained in Section 2 of this paper.

This paper considers optimal code generation algorithms for register-pair machines. In particular we are interested in the optimal evaluation of expressions having no common subexpressions. (When the expressions are represented graphically, they are trees rather than dags [AU].) We exclude expressions with common subexpressions in this paper because in this case optimal code generation is already difficult even on single register machines [BS, AJU].

With register-pair machines a new phenomenom arises that is not present in optimal code generation for single-register machines. Section 2 shows that in an optimal evaluation of an expression on a register-pair machine it

†Work partially supported by NSF grant DCR-74-15255.

may always be necessary to oscillate back and forth between subexpressions of the expression. This oscillation is in direct contrast to the "contiguity property" of optimal programs for single-register machines [AJ].

For certain types of register-pair machines Sections 3 and 4 show that we can bound this oscillation and thus derive a dynamic programming algorithm that generates optimal code for expressions in time that is linear in the input size of the expression. For other types of register-pair machines, such as the even-odd register-pair machines, Section 5 shows that this oscillation can become arbitrarily large.

2. The Machine Model

Computers in existence today present a bewildering diversity of instruction repertoires. Certain operations such as multiplication and division often require double length data. The precise details of these operations, however, vary widely from machine to machine.

The multiplication instruction, for example, may take as operands registers r and r' and deliver the result in the register pair $(r, r+1)$ or in the register pair $(r-1, r)$ or possibly in register r with the contents of register $r+1$ destroyed. Rather than considering an intricate model which has sufficient complexity to mimic every real machine, we shall adopt a rather simple register-pair machine model in the hope that we can gain insight into the effects of the architecture of the underlying machine on the complexity of code generation.

We assume a machine has N registers $r_0, r_1, \ldots, r_{N-1}$ which are interchangeable with respect to all instructions involving single-length operands. The single-length instructions have the usual form

$r \leftarrow m$ [load]

$m \leftarrow r$ [store]

$$\left.\begin{array}{l} r \leftarrow r \textbf{ op } r' \\ r \leftarrow r \textbf{ op } m \end{array}\right\} \text{[operations]}$$

Here r and r' refer to registers, and m to a memory location.

We also assume that there are instructions involving double-length operands. These instructions use two registers or two memory locations to hold an operand. The register pairs that can be used to hold an operand are specified by a set P of permissible register pairs. For example, P might consist of all pairs (r_i, r_{i+1}) such that i is even. The double-length instructions have the following form:

$(r, r') \leftarrow (m_i, m_{i+1})$ [double load]

$(m_i, m_{i+1}) \leftarrow (r, r')$ [double store]

$(r, r') \leftarrow (r, r') \textbf{ op } (s, s')$

$(r, r') \leftarrow (r, r') \textbf{ op } (m_i, m_{i+1})$

$(r, r') \leftarrow (r, r') \textbf{ op } s \quad s \neq r, r'$

$(r, r') \leftarrow (r, r') \textbf{ op } m$

$r \leftarrow r \textbf{ op } (s, s') \quad r \neq s, s'$

$r \leftarrow r \textbf{ op } (m_i, m_{i+1})$

In these instructions (r, r') and (s, s') can be any register pairs in P and (m_i, m_{i+1}) is a pair of consecutive memory locations used to hold a double-length operand. Note that in the operations, the destination is always the same as the left operand. If (r, r') is a register pair in P, we shall refer to r as the *left* register of the pair and r' as the *right* register.

To model real computers more closely we shall also permit classes of *extend* and *shorten* operations of the following form:

$(r, r') \leftarrow \text{EL } r'$

$(r, r') \leftarrow \text{ER } r$

$r \leftarrow \text{SL } (r, r')$

$r' \leftarrow \text{SR } (r, r')$

where (r, r') is any register pair in P. Finally we assume "strong typing" of operands; we do not permit the use of a double load or double store to move two single values.

We shall consider three classes of register-pair machines in this paper. These classes are distinguished by the set P of register pairs usable in double-length instructions.

1. *Unrestricted Model*

   $P = \{(r_i, r_j) \mid 0 \leqslant i, j \leqslant N-1; i \neq j\}$

2. *Adjacent Model*

   $P = \{(r_i, r_{i+1}) \mid 0 \leqslant i < N-1\}$

3. *Even-Odd Model*

   $P = \{(r_{2i}, r_{2i+1}) \mid 0 \leqslant i \leqslant \lfloor (N-2)/2 \rfloor\}$

Although many other models are also possible, these three serve to illustrate salient aspects of existing computers with double-length instructions. Results pertaining to the unrestricted model can be used for machines where register-to-register moves and exchanges are much faster than other operations. The adjacent and even-odd models approximate machines such as the IBM System/370 and the PDP-11 where somewhat similar restrictions are placed on what registers can be used to hold the operands of multiply and divide instructions.

Example 1. Let us consider how an expression might be evaluated on an unrestricted machine with four registers r_0 through r_3. The tree for an expression with which we shall deal is shown in Figure 1. We use circles to represent single values and rectangles for doubles. A sequence of machine instructions evaluating this expression tree with no stores is shown in Figure 2. □

It should be observed that the code of Figure 2 meets the restrictions of the adjacent machine but not the even-odd machine (because E is loaded into an odd-even pair rather than an even-odd pair). In fact there is no way to evaluate Figure 1 on a four-register even-odd machine without using a store instruction.

Another important observation about the program of Figure 2 is that it begins working on the left subtree of the root then moves to the right subtree, then back and

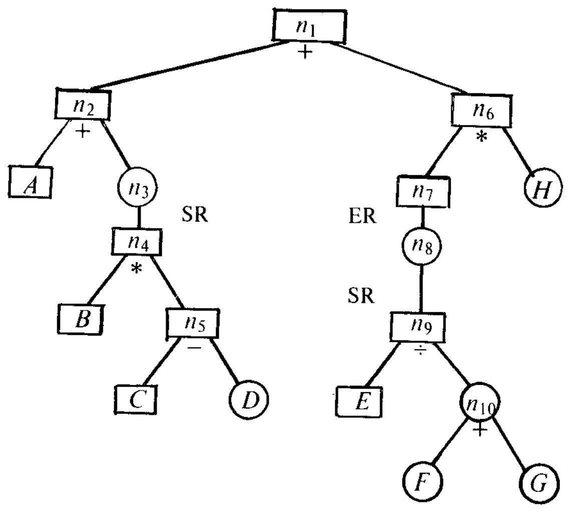

Figure 1. Expression tree.

$(r_0, r_1) \leftarrow C$ /* load C into (r_0, r_1) */
$(r_0, r_1) \leftarrow (r_0, r_1)-D$ /* compute n_5 */
$(r_2, r_3) \leftarrow B$
$(r_2, r_3) \leftarrow (r_2, r_3)*(r_0, r_1)$ /* compute n_4 */
$r_3 \leftarrow \mathrm{SR}(r_2, r_3)$ /* compute n_3 */
$r_0 \leftarrow F$
$r_0 \leftarrow r_0+G$ /* compute n_{10} */
$(r_1, r_2) \leftarrow E$
$(r_1, r_2) \leftarrow (r_1, r_2)/r_0$ /* compute n_9 */
$r_2 \leftarrow \mathrm{SR}(r_1, r_2)$ /* compute n_8 */
$(r_0, r_1) \leftarrow A$
$(r_0, r_1) \leftarrow (r_0, r_1)+r_3$ /* compute n_2 */
$(r_2, r_3) \leftarrow \mathrm{EL}\ r_2$ /* compute n_7 */
$(r_2, r_3) \leftarrow (r_2, r_3)*H$ /* compute n_6 */
$(r_0, r_1) \leftarrow (r_0, r_1)+(r_2, r_3)$ /* compute n_1 */

Figure 2. Machine code to evaluate Figure 1.

forth once more before evaluating the root. This oscillation is necessary because if no STORE's are used, n_3 requires all four registers, and n_8 requires three. In fact, all optimal programs evaluating Figure 1 on unrestricted register machines exhibit this oscillatory behavior, thus showing that this model does not satisfy the conditions for the contiguity theorem of [AJ].

It should be noted that the use of the SHORTEN operators is not essential for this example. We could replace the subtrees of n_3 and n_8 by trees of singles requiring 4 and 3 registers, respectively, for evaluation with no STORE's and the same oscillation would appear in any optimal program using four registers.

3. Limited Oscillation in Unrestricted Machines

This section shows that the oscillations required to evaluate an expression tree for an unrestricted machine can be confined to two bounces between the subtrees of the root. This result is used in the next section to derive a linear-time dynamic programming algorithm for the generation of optimal code for an unrestricted machine.

The *value* of an instruction in a program is the expression which that instruction computes. The *scope* of an instruction I is the subsequence of instructions from the instruction following I, up to the last instruction using the value of I. As a special case, if I computes the root and is not redefined, then the scope of I is deemed to include the "end" of the program. A value is *live* at any instruction within its scope. If the value of an instruction is never used, that instruction is *useless* and its scope is empty. The *width* of a program is the maximum over all instructions of the number of live values held in registers after that instruction, counting doubles as 2 and singles as 1. Except for doubles, all these notions are as in [AJ], where more formal definitions appear.

There are two properties of optimal programs for single-register machines that were used by [AJ] to obtain a dynamic programming code generation algorithm.

(1) *Renaming*, which allows us to take any optimal program of width w and execute it using a fixed set of w registers. That is, registers can be "renamed" at will.

(2) *Rearrangement* which allows us to take any optimal program for an expression tree T and rearrange it into a sequence of subprograms $P_1P_2 \cdots P_rP_{r+1}$ such that each P_i, $1 \leqslant i \leqslant r$, evaluates and stores into memory the value of some node of T and P_{r+1} evaluates the root of T. Except for the last instruction, there are no other stores in P_i, $1 \leqslant i \leqslant r$. Moreover, for every node n of T, the portion of the subtree of T with root n that omits proper descendants of stored nodes is evaluated contiguously by a consecutive sequence of instructions in one of the P_i's.

In the unrestricted machine model, the renaming condition continues to hold, as all registers are equivalent in their capabilities. The rearrangement condition, however, fails to hold, as we saw from Example 1. There is no optimal program which evaluates each subtree of the root of Fig. 1 contiguously. Nevertheless, we can prove the following modified version for programs with no STORE's; an obvious generalization analogous to the rearrangement theorem of [AJ] holds for programs with STORE's.

Consider an expression tree T with root n, having two subtrees T_1 and T_2. Roughly speaking, the following rearrangement theorem states that if T can be evaluated optimally with no stores, then T can be evaluated optimally by a program which has one of the following forms:

$$P_1I$$
$$P_1P_2I$$
$$P_2P_1I$$
$$P_1'P_2P_1I$$
$$P_2'P_1P_2I$$
$$P_1'P_2'P_1P_2I$$
$$P_2'P_1'P_2P_1I$$

where P_1' and P_2' evaluate subtrees T_1 and T_2, respectively, leaving a result in a register, and P_1 and P_2 evaluate the remainder of T_1 and T_2, respectively. I is the instruction evaluating the root n. Thus an optimal program never needs more than two bounces between T_1 and T_2.

Theorem 1. Let P be a program with no STORE's and no useless instructions, computing some expression tree T on an N-register unrestricted machine. Suppose the root of T has children c_1 and c_2, where c_2 may not exist. Then for each of c_1 and c_2 which is double and has a descendant whose size is single, we can optionally select a node m_i which is a proper descendant of c_i, $i=1,2$, such that:

(1) m_i is single.

(2) No proper ancestor of m_i except possibly the root of T is single.

(3) We can find a program computing T with the same number of instructions as P, of the same or smaller width as P, and having the form $P_1P_2 \cdots P_sI$, where P_j, $1 \leqslant j \leqslant s$, computes m_i or c_i into a register or registers, and I computes the root.

Moreover, P_j is computed using at most $N - \sum_{k=1}^{j-1} x(k)$ registers, where $x(k)$ is 2 if P_k computes some c_l and there is no m_l, and $x(k)$ is 1 otherwise. That is, $x(j)$ is the increase in the number of registers needed to hold the results of $P_1, P_2, \ldots, P_j$ over that needed to hold the results of $P_1, P_2, \ldots, P_{j-1}$.

Proof. We shall actually prove a somewhat stronger result by induction on the length of P. The strengthening of the theorem necessary to form the inductive hypothesis is to permit P to compute T starting with the values of some leaves in registers, provided these leaves each meet the above conditions (1)-(2) on the m_i's. P may also start with the values of some of the c_i's in registers even if they are doubles.

Suppose the first instruction of P computes or loads a node n (perhaps a leaf) of the subtree of c_i. The value of n is a double and at no time during the execution of P are fewer than two registers used for values in the subtree of c_i until the root is evaluated. Then, using all N registers compute the value of c_i using the instructions of P that serve that purpose. Call this sequence of instructions P_1 and suppose without loss of generality that P_1 leaves its result in the last two registers, r_{N-2} and r_{N-1}. Rename the registers in the remaining instructions of P so that only registers r_0 through r_{N-3} are used. Since at least two of the N registers are always devoted to the subtree of c_i, this renaming can be done. Let the resulting program be Q. By the inductive hypothesis, we can put Q in the form $P_2P_3 \cdots P_sI$ satisfying the conditions of the theorem and using $N-2$ registers. We may construct the desired program $P_1P_2 \cdots P_{k+s}I$ by prepending P_1 to $P_2P_3 \cdots P_sI$. At some time during the computation of P only one register is devoted to values of descendants of c_i. Let m_i be the descendant of c_i whose value is in a register the last time in P that one register is devoted to descendants of c_i. It is not possible that some descendant of c_i was initially in a register when P began, or else condition (2) in the theorem statement would be violated. Thus we may let P_1 be the subsequence of P's instructions computing m_i into some register, say $N-1$. Rename the registers used by the remaining instructions of P to avoid register $N-1$ until the value of m_i is used. Since at least one register is utilized for a descendant of m_i until the value of m_i is used, this renaming can be done. Apply the algorithm recursively to the remaining program and prepend P_1 as before.

Example 2. Consider the expression tree T in Figure 3. Theorem 1 implies that if T can be evaluated without STORES, then in the worst (most oscillatory) case we can find singles m_1 and m_2, no proper ancestor of which is single, such that an optimal program to evaluate T is never more complex than $P_1'P_2'P_1P_2I$ or $P_2'P_1'P_2P_1$, where P_1' and P_2' are optimal programs to evaluate the subtrees of T with roots m_1 and m_2. P_1 and P_2 are optimal programs evaluating the left and right subtrees of the root r, assuming m_1 and m_2 have been evaluated and left in registers. I is the instruction evaluating the root r. □

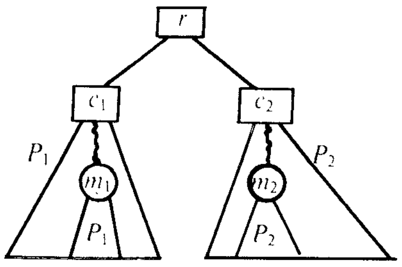

Figure 3. Expression tree T.

4. Algorithm for Unrestricted Machines

In this paper we measure the cost of a program in terms of the number of instructions used. (The results can be generalized to apply to more general additive cost functions, if necessary.) Theorem 1 tells us that an optimal way to compute a tree involves working on each subtree of the root in turn, breaking off work on a given subtree at most once, and leaving a single register holding a value for that subtree if and when we do. The dynamic programming approach of [AJ] can now be applied, provided we compute the following cost vectors for each node n.

(1) $\alpha_{ij}(n)$ = the cost of computing some single-valued descendant m of n using i registers and later computing the balance of the tree with root n using j registers, including the register used to hold the value of m. We can assume $j \leqslant i$, since otherwise we could evaluate the entire tree with root n without oscillation using j registers. If the tree with root n has no singles, then $\alpha_{ij}(n)$ can be taken to be infinite.

(2) $\beta_i(n)$ = the cost of computing the tree with root n using i registers. If n is a leaf, $\beta_i(n) = 0$ for all i.

To compute these values for a node n, suppose the α and β values are available for its children. We consider a list $c_1, c_2,\ldots$ of children of the node n, such that each single-sized child appears at most once, and each double-sized child appears at most twice. The list corresponds to an evaluation order in which

(1) the children which do not appear are assumed to have been computed and stored,

(2) the children which appear once are assumed to be completely computed at that time, and

(3) the children which appear twice are assumed to have some single-valued subtree computed, and then later the entire subtree computed.

For each list, we shall describe the contribution to $\alpha_{ij}(n)$ and $\beta_i(n)$ which might arise from computing the node n in the way described in the list. The actual costs $\alpha_{ij}(n)$ and $\beta_i(n)$ will be the minimum, over all lists, of the contributed costs. The contribution to each cost of a non-leaf child c not appearing in the list is the cost of precomputing it and storing it; this is $\beta_N(c)$ plus the cost of a single or double store. If c is the left descendant of n, it must also be loaded, so this cost must be included.

The contributions to $\alpha_{ij}(n)$ of a child c depend on whether it appears once or twice on the list and whether its sibling is a single or a double. A few cases should illustrate what is involved.

1. The list is $c_1c_2c_1$ where c_2 is a single. The cost of this list is $\alpha_{i,j-1}(c_1) + \beta_{j-1}(c_2)$ + cost of the instruction for n. This cost reflects a computation in which we first evaluate a subtree of c_1 (whose root is single) with i register available, then compute the entire right subtree (whose root is c_2) with $j-1$ registers available (one being busy with the descendant of c_1), then return to evaluate the remainder of the left subtree with $j-1$ registers, and finally evaluate n.

If c_2 is a double in this case, then the first term in the cost is $\alpha_{i,j-2}(c_1)$.

2. The list is $c_1c_2c_1c_2$. The cost of this list is $\alpha_{i,j-1}(c_1)$ + $\alpha_{j-1,j-2}(c_2)$ + cost of the instruction for n. Here we evaluate a subtree of c_1 with i available registers, evaluate a subtree of c_2 with $j-1$ available registers, complete the evaluation of c_1 with $j-1$ available registers, then complete the evaluation of c_2 with $j-2$ available registers, and finally evaluate n.

Similar formulas can be derived for the other cases.

The cost $\beta_i(n)$ is the minimum taken over $\alpha_{ii}(n)$ and the costs of the lists representing the ways of computing the tree with root n without oscillation. For example, the list c_1c_2, where both c_1 and c_2 are singles is $\beta_i(c_1)$ + $\beta_{i-1}(c_2)$ + cost of the instruction for n.

Clearly, given a list, the time required to compute α_{ij} and β_i for a given i and j is bounded above by a constant. For binary operators, the number of lists is finite (in fact, lists have length at most 4). Applying the same kind of dynamic programming considerations as [AJ], we can justify the following theorem:

Theorem 2. There is an algorithm to generate optimal code for an unrestricted machine that is linear in the size of the expression. □

Note that the above algorithm is quadratic in the number of registers. If we permit operators of arbitrary arity k, the complexity grows as $O((2k)!)$. (In effect, this is the number of lists which must be considered.)

5. Models in Which Limited Contiguity Fails to Hold

The result of Theorem 1 implies a modified version of the contiguity theorem of [AJ], which in turn implies a polynomial algorithm for optimal code generation. However, there are a number of machine models in which Theorem 1 can be shown not to hold. That is, arbitrary oscillation between two subtrees may be necessary for optimal evaluation of an expression. For example, consider the even-odd model.

Lemma. For all integers a and b with $2a+b \leqslant N$, we can construct a tree $S_{a,b}$ which can be evaluated with no STORE's into a single register if and only if at least a even-odd pairs and b additional registers are available.

Proof. The case $a=0$ follows from the labeling algorithm of [E,N]. The case $b=0$ follows similarly, since a tree with single-valued operands requiring a registers can be made to require a register pairs by making all its operands and operators double, and then shortening the result. If $a \neq 0$ and $b \neq 0$, consider the tree of Figure 4. Since $b>0$, we can evaluate the right side and store the result in a register which is not part of the a pairs, then evaluate the left. If we could evaluate Figure 4 with no stores using fewer than a register-pairs we would violate what we know about $S_{a,0}$, and if we used fewer than $2a+b$ registers in total, we would violate what we know of $S_{0,2a+b}$. □

Theorem 3. For every N there is an expression tree T with a node n with two children c_1 and c_2 having the property that any optimal program for T on an even-odd machine with $2N$ registers has a subsequence of instructions $I_1, I_2, \ldots, I_{2N-1}$ such that $I_1, I_3, \cdots$ evaluate descendants of c_1 and $I_2, I_4, \cdots$ evaluate descendants of c_2. That is, $2N-2$ oscillations occur in the subtree of n.

Proof. Figure 5 shows the tree T. Consider a program for T which does not store a result or move a value from one register to another. Surely any such program must be optimal. We must first evaluate $S_{N,0}$, or we shall never be able to do so without STORE's. $S_{N,0}$ must be computed into an even register, say register 0, for we shall eventually extend its value into the paired odd register. If we do the extension now, we can never evaluate $S_{N-1,1}$, since at least two registers will be tied up with the resulting value. Furthermore, the result of $S_{N-1,1}$ must appear in register 1, the other half of the above pair, else there will not be sufficient register-pairs to compute $S_{N-1,0}$.

We are now in a situation where we cannot extend the value of $S_{N,0}$ in register 0 until we have computed the entire right side of Figure 5. Continuing in this way, by symmetry of register-pairs, we must compute $S_{i,0}$ in register $2(N-i)$ and then $S_{i-1,1}$ in register $2(N-i)+1$, for $i=N-1, N-2, \ldots, 3$. Then we may compute $S_{2,0}$ in register $2N-4$. We now use the remaining 3 registers for $S_{1,1}$, arranging that the result end up in register $2N-2$. We then extend that value into register $2N-1$ and evaluate the right side of Figure 5 in registers $2N-2$ and $2N-1$. Finally we may extend $S_{i,0}$ into register $2(N-i)+1$ for $2 \leqslant i \leqslant N$ and evaluate the left side of Figure 5 and the root in registers 0 and 1.

Thus every optimal program for T must oscillate $2N-2$ times back and forth, proving the theorem. □

It is interesting to observe that even with the adjacent-register constraint we can produce subtrees to play the role of the S_i's in Theorem 3. For example, if there are N registers, the tree of Figure 6 can be evaluated with no stores only if the result winds up in register 0. Thus, if we permit trees like Figure 6, we can prove:

Theorem 4. Theorem 3 also holds for adjacent register-pair machines. □

6. Summary and Suggestions for Further Work

We have considered several models of register pair machines and shown that there is an oscillatory behavior to the optimal evaluation of expressions on these machines that is not present on single register machines. For the unrestricted model we were able to bound the oscillations to two and thus derive a linear-time dynamic programming algorithm. While the dynamic programming algorithm is too expensive to implement in most compiler applications, its existence suggests a more economical specialized linear-time algorithm could be constructed for particular machines in this class.

For the even-odd and adjacent register-pair machines we showed the oscillations can be at least proportional to the number of registers. This result suggests that efficient optimal algorithms, if they exist, are unlikely to evolve from any dynamic programming considerations for these classes of register-pair machines.

There are a number of additional problems that would be worth solving to add to our understanding of code generation for register-pair machines.

(1) Is the optimal code generation problem polynomial or exponential for the (a) even-odd (b) adjacent register-pair machines?

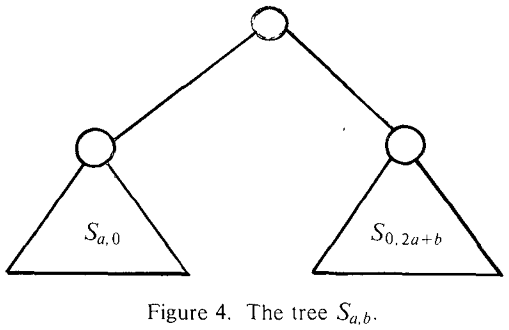

Figure 4. The tree $S_{a,b}$.

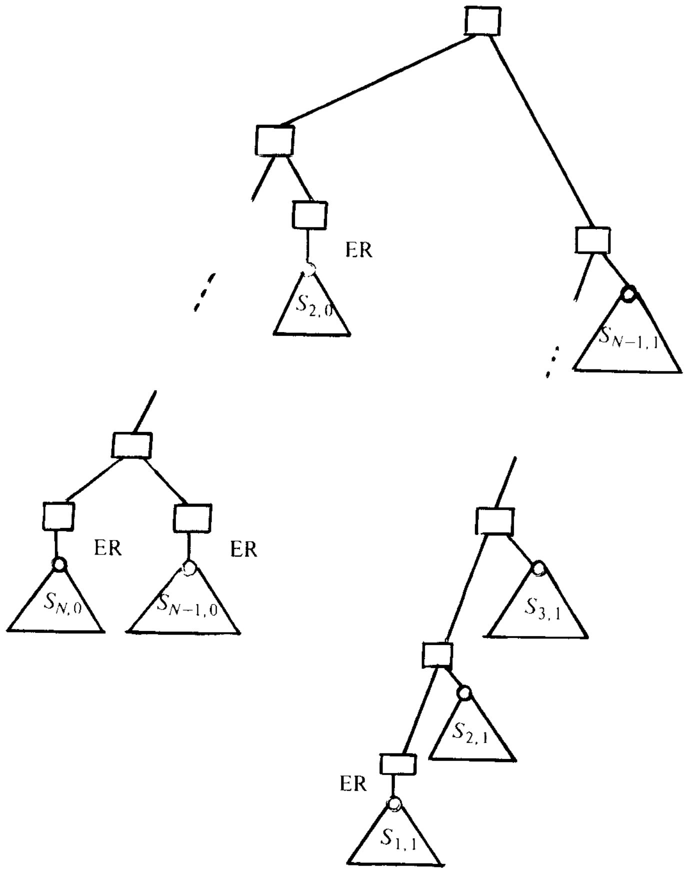

Figure 5. The tree T.

(2) For what other sets of register pairs P, smaller than the unrestricted model, does the limited contiguity theorem hold. An interesting observation along these lines is that if there is one additional register r_a which can only be used as the left pair of a double, i.e., $P = \{(r_a, r_0), (r_a, r_1), \ldots, (r_a, r_{N-1})\}$, then unlimited bouncing is still required for optimal evaluation of expressions.

(3) The trees exhibiting oscillatory behavior for Theorems 4 and 5 produce a number of bounces bounded by the number of registers. Is this the worst possible case? If so, it appears that optimal code generation could be done in time polynomial in the tree size but exponential in the number of registers.

(4) If we could bound the number of oscillations by some polynomial in the height of the tree, the algorithm of Section 4 can be generalized to yield a polynomial algorithm. Is there such a bound for the even-odd or adjacent register-pair machines?

(5) How closely can optimality be approximated by a linear- or polynomial- algorithm for the even-odd or adjacent register-pair machines?

(6) How well can we generate code for register-pair machines when expressions contain common subexpressions?

Bibliography

[AJ] A. V. Aho and S. C. Johnson, "Optimal Code Generation for Expression Trees," *JACM* 23:3 (July 1976), 488-501.

[AJU] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code Generation for Expressions with Common Subexpressions," *JACM* 24,1 (January 1977), to appear.

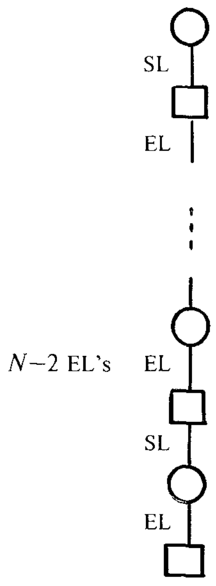

Figure 6. Tree requiring N registers with the result in register 0.

[AU] A. V. Aho and J. D. Ullman, "Optimization of Straight Line Code," *SIAM J. Computing* 1,1 (1972), 1-19.

[B] J. C. Beatty, "An Axiomatic Approach to Code Optimization for Expressions," *JACM* 19:4 (October 1972), 613-640.

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

[BS] J. Bruno and R. Sethi, "Code Generation for a One-Register Machine," *JACM* 23:3 (July 1976), 502-510.

[E] A. P. Ershov, "On Programming of Arithmetic Operations," *Dokl. A. N. USSR* 118:3 (1958), 427-430. (English translation in *Comm. ACM* 1:8 (1958), 3-6.)

[N] I. Nakata, "On Compiling Algorithms for Arithmetic Expressions," *Comm. ACM* 18:8 (August 1967), 492-494.

[R] R. R. Redziejowski, "On Arithmetic Expressions and Trees," *Comm. ACM* 12:2 (February 1969), 81-84.

[RKL] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, "The C Programming Language," CSTR #31, Bell Laboratories, Murray Hill, N. J., 1975.

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

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