Generalizations of the Sethi-Ullman algorithm for register allocation

Andrew W. Appel
Kenneth J. Supowit†

Department of Computer Science
Princeton University
Princeton, NJ 08544

March 31, 1986
revised Sept 24, 1986

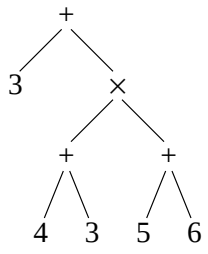
ABSTRACT

The Sethi-Ullman algorithm for register allocation finds an optimal ordering of a computation tree. Two simple generalizations of the algorithm increase its applicability without significantly increasing its cost.

Keywords: Register allocation, Code generation, Code optimization, Expression trees

Introduction

Certain computations can be appropriately modelled as computation trees. A computation tree has constant values as its leaves, and the internal nodes are given values inductively as specified arithmetic functions of the values of their children. For example, the computation tree for the expression $(3+(4+3)\times(5+6))$ is



On a computer, such a computation tree may be evaluated by keeping the values of previously computed nodes in registers. When the value of a previously computed child node (stored in a register) is

† The work of this author was supported in part by NSF grant number DMC-8451214 and by a grant from IBM.

used to compute the value of its parent node, then the register associated with the child may be re-used for another computation.

The Sethi-Ullman algorithm[3] determines an order of computation of the nodes of the tree that uses the fewest possible registers, subject to these assumptions:

1. The properties of the arithmetic operators are not considered; that is, no arithmetic identities are used.
2. All registers are equivalent; there are no operations that can produce results only in certain registers.
3. The tree is a binary tree: each internal node has exactly two children.
4. The value of each node will fit in one register.

Their algorithm relies on the observation (which they proved) that once the computation at a subtree is begun, it is always better to complete that computation before moving to a disjoint subtree. Thus, the algorithm need only decide, for each node, which of its two children to evaluate first.

Suppose the optimally ordered computation of the left subtree requires n registers, and the computation of the right subtree requires m registers. To determine the number of registers needed to compute the parent's value, there are three cases to consider:

$n > m$ In this case, we first compute the left child, using n registers, but finally freeing all registers except the one used to hold the value of the left-child node. Then we compute the right child, using m registers (in addition to the one holding the left-child's value). The parent may then be computed; this process has used at any time no more than n registers.

$n < m$ This case is similar to the first case; the right child should be computed first.

$n = m$ Either child may be computed first; then the other child must be computed. This case uses $n + 1$ registers, since the computation of the second child must take place while there is also a register being used to hold the value of the first child.

These cases are considered for each node in a bottom-up fashion. Each node is labelled with a ''Sethi-Ullman number'' (the number of registers required during its computation), which is equal to the maximum of the numbers of its children (if these are unequal) or to the number of either child plus one (if the children have equal numbers). The optimal ordering of the entire tree, and the number of registers required, is thus determined in $O(n)$ time.

The four conditions listed above can be overly restrictive in real compilers. This paper presents a generalized algorithm that is still very simple to implement.

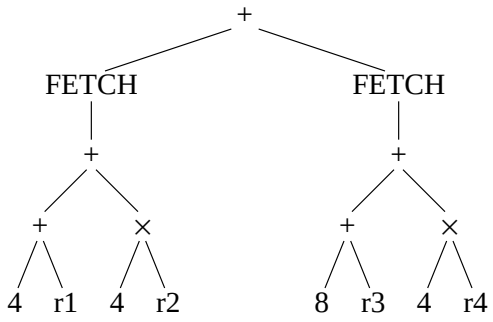
Two generalizations

In writing a code generator using the Twig tree-pattern-matching/dynamic-programming code-generator-generator[2], two generalizations of the model proved to be useful. In this context it is often possible to match (as machine-instructions) much larger tree-patterns than the simple case of an arithmetic operator with two children.

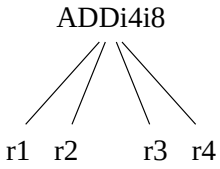
The first generalization is to remove the restriction on the degree of the nodes. There are machine instructions (on the VAX, for example) that compute a value from the values of several other registers and constants, for example the instruction:

```
addl3        4(r1)[r2],  8(r3)[r4],  r5
```

computes in register 5 the value of the tree



This instruction could be modelled as a 4-ary node:



The second generalization is to remove the restriction on the size of the computed result. On many computers, there are operands (like double-precision floating-point numbers) that require two registers to hold them. Each node in the tree, then, should be annotated with the number of registers occupied by the computed value of that node.

A more interesting example of a multiregister value comes from the addressing modes of the VAX. The operands of an **add** instruction may be complicated addressing modes that use more than one register, as illustrated by this partial grammar:

reg : PLUS(operand,operand)

operand : reg

operand : FETCH(reg)

operand : FETCH(PLUS(reg,CONST))

operand : FETCH(PLUS(PLUS(reg,CONST),MUL(CONST,reg)))

The operand types correspond to register mode, register-deferred mode, displacement-deferred mode, and displacement-deferred-indexed mode, respectively.

The value of an **operand** node may require two registers to hold. The last of the **operand** patterns above specifies the displacement-deferred-indexed mode; an example of this mode (in assembler syntax) is

4(r1)[r5]

To incorporate this into an instruction, other operands must be accumulated. In the meantime, the values of registers 1 and 5 must be held.

The addressing mode works with any pair of registers; in particular, they need not be adjacent. This turns out to be important; allocation for multi-register values turns out to be much easier when the multiple-registers may be any subset of the register bank[1].

In this way, we extend the model to include trees with more than two children, and nodes requiring more than one register to hold the result. However, the generalization includes simpler cases as well. For example, the pattern

operand: CONST

corresponding to an immediate-mode operand, requires zero registers to hold its value (since it will be compiled directly into an instruction), and has zero subtrees.

An evaluation algorithm for generalized Sethi-Ullman numbers

We thus define a generalized computation tree as one in which each node has an arbitrary number of children and is labelled with a number specifying how many registers are required to hold its value.

For machines that operate on multiregister values that must be kept in adjacent registers (like the double-precision floating-point numbers on the VAX), it is not always optimal to generate each subtree contiguously[1]. It may be necessary to partially evaluate one subtree, partially evaluate another subtree, evaluate more of the first subtree, and so on.

When the registers in a multiregister operation need not be adjacent (as in the index-mode operand example given above) — or when the cost of a register-register move is trivial — there is always an optimal solution that evaluates entire subtrees contiguously. Furthermore, in a compiler that is already structured to generate machine code bottom-up from expression trees, it is convenient to use a simple bottom-up register-allocation algorithm. Thus, we restrict our attention to algorithms that evaluate subtrees contigously, even though they may perform suboptimally in some cases.

We seek an algorithm that minimizes the total number of registers needed, subject to the constraint that the evaluation of one subtree is completed before the evaluation of a disjoint subtree is begun. As before, it suffices to order the evaluation of the children of each node.

For each node *t*, we are given as input the number b_t of registers required to hold *t*'s result. We compute the number a_t of registers required in the evaluation of all of the children of node *t*.

Assume we are at a given node *t* with *k* children, and we have computed for each child *i* the number a_i. We must find a permutation $\pi_{\min}$ such that the evaluation of the children in the order $\pi_{\min}(1), \pi_{\min}(2), \cdots, \pi_{\min}(k)$ takes a minimum number of registers.

The number of registers required for a given permutation π is easily computed; it is given by

$$R_\pi = \max(a_{\pi(1)}, b_{\pi(1)} + \max(a_{\pi(2)}, b_{\pi(2)} + \max(\cdots, b_{\pi(k-1)} + \max(a_{\pi(k)}, b_{\pi(k)}))\cdots)$$

where the rightmost "$\cdots$" indicates a sequence of closing parentheses needed to balance the elided opening parentheses.

This equation is developed by the following reasoning: Subtree $\pi(1)$ will be evaluated first. This will require $a_{\pi(1)}$ registers. Then the rest of the subtrees must be evaluated while saving the result of subtree $\pi(1)$ in $b_\pi(1)$ registers. This will require a number of registers equal to the sum of $b_{\pi(1)}$ and the number of registers required to compute the rest of the children. The number of registers required to compute the parent is the larger of $a_{\pi(1)}$ and the specified sum. Thus, we compute a_t as the minimum of R_π over all permutations π.

Finding an optimal permutation turns out to be no more difficult than sorting *k* numbers. In particular, we sort the ordered pairs (a_i, b_i) by their differences $a_i - b_i$, thereby obtaining an optimal ordering:

THEOREM: Each permutation $\pi_{\min}$ satisfying

$$a_{\pi_{\min}(1)} - b_{\pi_{\min}(1)} \geq a_{\pi_{\min}(2)} - b_{\pi_{\min}(2)} \geq \cdots \geq a_{\pi_{\min}(k)} - b_{\pi_{\min}(k)}$$

minimizes *R*.

PROOF: By induction on k. The claim is immediate for $k=1$; assume it for each set of fewer than k pairs. Assume $a_1 - b_1 \geq a_2 - b_2 \geq \cdots \geq a_k - b_k$ and let π be a permutation on $\{1, 2, ..., k\}$. Then

$$R_\pi = \max(a_{\pi(1)}, b_{\pi(1)} + \max(a_{\pi(2)}, b_{\pi(2)} + \max(\cdots, b_{\pi(k-1)} + \max(a_{\pi(k)}, b_{\pi(k)}))\cdots)$$

$$\geq \max(a_r, b_r + \max(a_1, b_1 + \max(a_2, b_2 +$$

$$\max(\cdots, b_{r-1} + \max(a_{r+1}, b_{r+1} + \max(\cdots, b_{k-1} + \max(a_k, b_k))\cdots)$$

(1)

by the inductive hypothesis applied to the $k-1$ pairs (a_1, b_1), (a_2, b_2), ..., (a_{r-1}, b_{r-1}), (a_{r+1}, b_{r+1}), ..., (a_k, b_k), where $r = \pi(1)$. Letting

$$c = \max(a_2, b_2 + \max(\cdots, b_{r-1} + \max(a_{r+1}, b_{r+1} + \max(\cdots, b_{k-1} + \max(a_k, b_k))\cdots),$$

we prove that

$$E_1 = \max(a_r, b_r + \max(a_1, b_1 + c)) \geq \max(a_1, b_1 + \max(a_r, b_r + c)) = E_2 \qquad (2)$$

by considering cases. Note that $c, a_i, b_i \geq 0$ for all $1 \leq i \leq k$.

Case 1: $a_r \geq b_r + c$.

Then $a_1 - b_1 \geq a_r - b_r \geq c$. Therefore

$$E_1 = \max(a_r, b_r + a_1) \quad \text{and} \quad E_2 = \max(a_1, b_1 + a_r)$$

If $a_1 \geq b_1 + a_r$ then

$$E_2 = a_1 \leq \max(a_r, b_r + a_1) = E_1 .$$

Otherwise ($a_1 < b_1 + a_r$) we have

$$E_2 = b_1 + a_r \leq b_r + a_1 \leq \max(a_r, b_r + a_1) = E_1$$

(the first inequality follows from $a_1 - b_1 \geq a_r - b_r$).

Case 2: $a_r < b_r + c$.

Then $E_2 = \max(a_1, b_1 + b_r + c)$. If $a_1 \geq b_1 + b_r + c$ then

$$E_2 = a_1 \leq \max(a_1, b_1 + c) \leq \max(a_r, b_r + \max(a_1, b_1 + c)) = E_1 .$$

Otherwise ($a_1 < b_1 + b_r + c$) we have

$$E_2 = b_1 + b_r + c \le b_r + \max(a_1, b_1 + c) \le \max(a_r, b_r + \max(a_1, b_1 + c)) = E_1 .$$

Putting together (1) and (2) gives

$$R_\pi \ge \max(a_1, b_1 + \max(a_r, b_r + c))$$

$$\ge \max(a_1, b_1 + \max(a_2, b_2 + \max(\cdots, b_{k-1} + \max(a_k, b_k)) \cdots)$$

by applying the inductive hypothesis to the pairs (a_2, b_2), (a_3, b_3), ..., (a_k, b_k). □

Spilling Registers

The Sethi-Ullman algorithm also solves the problem of what to do when there are not enough registers. That is, when the number of registers required (as computed by the bottom-up algorithm) exceeds the number of registers on the machine, some of the values in registers will have to be "spilled" into memory locations.

The original algorithm does spilling in the evaluation of any node whose label is greater than *r*, the number of registers available. As before, let *n* be the number of registers required in computing the left subtree, and *m* be number required for the right subtree:

$m < n \le r$	Compute the left child first (as in the simple algorithm).
$n < m \le r$	Compute the right child first.
otherwise	the computation cannot be done in registers. In this case, evaluate either subtree first; save its value in memory; evaluate the other subtree; fetch the saved value; and continue the computation.

This spilling algorithm generalizes nicely. First, the subtrees of a node must be sorted (as described above, by $a_i - b_i$) into a list of nodes n_i. Then, after each node on the list is evaluated, enough registers are saved in memory to enable evaluation of the rest of the list. When all of the nodes n_i are evaluated, the saved values are fetched before doing the operation at the root node.

This requires, of course, that the total of the b_i (the number of registers required to hold all the subtrees' values) is not more than *r*. If this were the case, then there must be some machine operation that requires more registers then exist on the machine — a highly unlikely architectural feature.

Implementation

The generalized algorithm is of particular use with in an automatic code-generator generator system, for two reasons. Such a system is more likely to have to process nodes of varying degree (not just the binary tree nodes for which the original algorithm was designed). Secondly, automatically generated code generators can more easily take advantage of the index modes, which lead to nodes with multiregister values.

The algorithm has been used to improve register-allocation in a code generator produced using the Twig system. The procedure **reorder** (written in **C**) is shown in Figure 1. It computes the optimal ordering, and the number of registers required, for the evaluation of the children of a node. The code for spilling into memory locations is not shown.

```
struct node {int hold;
             int maxregs;
             int sideEffect;
              . . . other fields
            };

int reorder(nodes)
   struct node *nodes[];
{int i,j; struct node *temp;
 for(j=0; nodes[j]!=NULL; j++)
   for(i=j; i>0; i--)
     if (   nodes[i]->maxregs - nodes[i]->hold
          > nodes[i-1]->maxregs - nodes[i-1]->hold
         && !(nodes[i]->sideEffect || nodes[i-1]->sideEffect) )
           {temp=nodes[i]; nodes[i]=nodes[i-1]; nodes[i-1]=temp;}
 maxregs=0;
 for(i=j-1; i>=0; i--)
  {maxregs += nodes[i]->hold;
   if (maxregs < nodes[i]->maxregs)
        maxregs = nodes[i]->maxregs;
  }
 return maxregs;
}
```

Figure 1. The generalized Sethi-Ullman ordering procedure.

The **hold** field of a node specifies the number of registers needed to hold its computed result. The **maxregs** field indicates the number of registers required at some point during the computation of the node. The **sideEffect** field is a boolean flag indicating whether the subtree rooted at this node has any side effects (like assignment statements); if it does, then it is unsafe to move the evaluation of this node from its original position.

The parameter **nodes** is a NULL-terminated vector of node pointers. The procedure sorts this list according to the value **maxregs**−**hold**, using an insertion sort; except that it won't make any exchange

involving a node with side effects.

After the nodes are sorted into the optimal evaluation order, the parent node's **maxregs** value is computed and returned as the procedure's result.

The register-allocation algorithm works bottom-up on trees, so it fits naturally into a code generator that traverses trees bottom-up; Twig produces such code generators. It was therefore easy to incorporate the generalized Sethi-Ullman numbering into the cost-computation part of Twig.

Because of the generality of the **reorder** procedure, it can be called by the code generator for each node — even those with no children or one child. (Actually, Twig calls **reorder** for each pattern it matches, not for each node.) There is no need for a special case for each kind of node (match). To convert the original code generator — which had no register-allocation optimization — to one which does generalized Sethi-Ullman allocation, it was necessary to make the following changes: The **reorder** procedure was added; About 5 lines of ''glue'' were added to call **reorder,** and 20 lines to handle spilling; and to each pattern in the specification, a token was added specifying the number of registers required to **hold** the result. The time taken in **reorder** is about 1.5% of the total time spent compiling.

Conclusion

Real compilers must deal with expression trees containing nodes of varying degree, and multi-word values. By generalizing the Sethi-Ullman register allocation scheme, we make it possible to write one simple procedure that does the tree reordering, rather than many special cases and heuristics spread throughout the code generator.

Incorporating the generalized algorithm into a compiler (that generates code from trees) adds only a few tens of lines to the size of the compiler, and just over a percent to its execution time.

References

1. A. V. Aho, S. C. Johnson, and J. D. Ullman, ''Code generation for machines with multiregister operations,'' *Fourth ACM Symp. on Principles of Programming Languages*, pp. 21-28, ACM, 1977.
2. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, ''Code generation using tree matching and dynamic programming,'' *ACM Trans. Prog. Lang. and Systems*, vol. (to appear), 1989.
3. R. Sethi and J. D. Ullman, ''The generation of optimal code for arithmetic expressions,'' *J. Assoc. Computing Machinery*, pp. 715-728, ACM, 1970.