Optimal Scheduling of Arithmetic Operations in Parallel with Memory Access

(Preliminary Version)

David Bernstein

Dept. of Electrical Engineering
Technion - Israel Institute of Technology
Haifa 32000, ISRAEL

Ron Y. Pinter
Michael Rodeh

IBM Israel Scientific Center
Technion City
Haifa 32000, ISRAEL

ABSTRACT

We propose a new machine model in which load operations can be performed in parallel with arithmetic operations by two separate functional units. For this model, the evaluation of expression trees is considered. An efficient algorithm to produce an optimal order of evaluation is described and analyzed. For a tree with n vertices the algorithm runs in time $O(n\log^2 n)$. If the arithmetic operations have at most two arguments, the complexity goes down to $O(n\log n)$.

1. Introduction

By and large, modern computers are not sequential any more. Their designs usually include a certain degree of parallelism, most commonly by offering a number of functional units that can operate simultaneously - some units handle memory access while others manipulate (in registers) the data thus retrieved. So far, this trend in computer architecture has only partially been matched by efforts to devise efficient code generation algorithms that will utilize such capabilities in order to reduce the running time of compiled programs. Most work was devoted to highly structured designs, e.g. vector machines, but not parallel machines in general.

The difficulties are not easily dismissed. On one hand, the paradigms used for efficient code generation on sequential machines (e.g. as found in [2,7]) do not generalize to handling the intricate phenomena arising in a parallel machine as the cost function is not super additive. On the other hand, Li [5] proved that the problem of finding an optimal (fastest) schedule for a set of tasks whose precedence graph forms a tree on a machine with even two dedicated functional units is NP-complete.

In this paper we make a contribution towards filling the gap between these two extremes. We provide an efficient algorithm that produces optimal (i.e. fastest possible) schedules for evaluating a set of expression-trees on a machine that has *one* memory access unit and *one* arithmetic-logic unit that operates on data in registers. This case is special in that the precedence relations between memory access and arithmetic operations are limited in nature; for example, no arithmetic operation is ever required to precede any load command. A number of variations on this architecture have been proposed and even built (e.g. [8]), but they all use generic schemes for information transfer between the two units rather than providing a tight, customized scheduling algorithm for code generation.

We assume that the number of registers at our disposal is unbounded, thus concentrating on the issue of how to alleviate the memory-to-ALU bottle-neck rather than the constraints caused by an inadequate number of registers. In this situation, the schedule provided by our algorithm is all a compiler needs to generate the object code.

The algorithm proposed here processes the vertices of expression trees in that order which least impedes the processing of arithmetic vertices due to yet unloaded arguments; this order, in general, is quite different from the standard way of enumerating tree vertices, as used in [2]. The algorithm is easy to program. It performs one pass on the expression tree during which it produces the optimal schedule of operations for

both the memory-access and the arithmetic units. The algorithm runs in time $O(n\log^2 n)$, where n is the number of vertices of the tree. If no vertex has more than two immediate descendants then the complexity is reduced to $O(n\log n)$ without change to the algorithm.

We start in the next section with definitions of our machine model and tree evaluations for it, then we formulate an abstract representation of such evaluations and state a number of key properties on which the algorithm relies. The algorithm itself is presented in Section 3, followed by the analyses of its optimality (Section 4) and complexity (Section 5). We conclude with extensions and open problems.

2. Preliminaries

2.1. The machine

Our machine has an unbounded number of general purpose registers $r_1, r_2, \ldots$ and memory cells $mem_1, mem_2, \ldots$. It supports the following operations:

a) Load operation: $(mem_i) \rightarrow (r_j)$

b) Store operation: $(r_i) \rightarrow (mem_j)$

c) Arithmetic operations of the kind $A((r_{i_1}),\ldots,(r_{i_k})) \rightarrow (r_j)$ where $1 \le k \le d$ (d is a global bound on the number of arguments (arity) that an arithmetic operation may have).

The major parallel feature of our machine is that it can execute either a load or a store operation concurrently with an arithmetic operation. For the purposes of presentation, the execution times of all the three types of operations are assumed to be equal; we also deal with load operations alone, for the time being. Section 6 explains how to relax these two restrictions.

2.2. Computation forests and their evaluation

Following [2,7], computations are represented by rooted forests. An example of a forest with one tree in it is given in Figure 1.

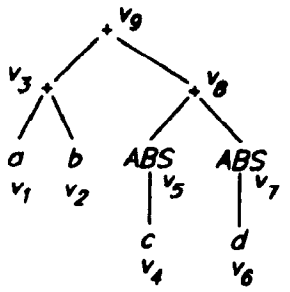

Figure 1. An expression tree T for $(a+b)+(ABS(c)+ABS(d))$

The vertices of a computation forest F that are associated with load operations are denoted by $L(F)$, and those associated with arithmetic operations are denoted by $A(F)$. Notice that all elements in $L(F)$ are leaves of F; those leaves that are not in $L(F)$ are associated with arithmetic operations whose operands are in registers. Let $l(F) = |L(F)|$, $a(F) = |A(F)|$, $d(v)$ be the number of immediate descendants of the internal vertex v, and $d(T)=\max_{v\in T} d(v)$. Given a machine with a bound d on the arity of its arithmetic operations, we will only consider trees T such that $d(T)\le d$.

The evaluation of a computation forest may be viewed as a proper scheduling [6] of two processes. One process loads values from the memory into registers, and the other carries out the arithmetic operations. A proper scheduling (hereafter called *legal evaluation*) $E(F)$ of a forest F comprises two one-to-one mappings into the set N of the positive integers:

$$\lambda: L(F)\rightarrow N$$

$$\alpha: A(F)\rightarrow N$$

such that for every vertex $v\in F$ other than the root, if $v\in L(F)$ then $\lambda(v)<\alpha(parent(v))$. Otherwise $\alpha(v)<\alpha(parent(v))$.

The ranges of α and λ are interpreted as the *time slots* (of unit length each) at which instructions are performed. Clearly a load instruction can be executed simultaneously with an arithmetic operation as long as the precedence relation defined by the forest is obeyed. Figure 2 provides three legal evaluations of the tree from Figure 1.

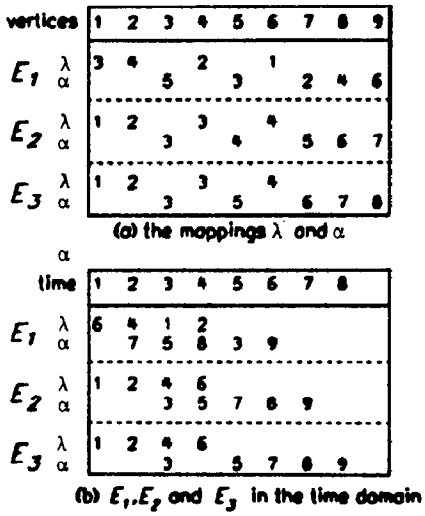

Figure 2. Three legal evaluations of the tree from Figure 1

The cost $c(E(F))$ of the evaluation $E(F)$ of F is defined as:

$$\max(\max_{v\in L(F)} \lambda(v), \max_{v\in A(F)} \alpha(v))$$

This measures the time (or number of slots) it takes to evaluate the expression represented by F using $E(F)$. For example, in Figure 2 $c(E_1)=6$, $c(E_2)=7$, and $c(E_3)=8$.

An *optimal evaluation* E is a legal evaluation for which the cost is smallest. The cost of an optimal evaluation of a forest F is denoted by $C(F)$. We shall concentrate on the problem of finding $C(F)$. In Section 6 an algorithm to find optimal evaluations is derived.

2.3. Compressed evaluations

In the sequel the parameters E and/or F are omitted when no ambiguity results. An evaluation $E=(\lambda,\alpha)$ is *compressed* if the following three conditions are satisfied:

a. *Range*$(\lambda) = \{1,\ldots,l\}$, i.e the loads are consecutive beginning at time slot 1.

b. *Range*$(\alpha) = \{c-a+1,\ldots,c\}$, i.e. the arithmetic operations are consecutive at the end of the evaluation.

c. The evaluation $E'=(\lambda,\alpha')$ obtained by setting (for all v) $\alpha'(v)=\alpha(v)-1$ is illegal, i.e. it either violates the precedence conditions or sets $\alpha(v)=0$ for some v.

For example, the evaluations E_1 and E_2 in Figure 2 are compressed, whereas E_3 is not.

LEMMA 2.1. For every evaluation E there exists a compressed evaluation E' such that $c(E')\le c(E)$.

Proof. To compress E, we squeeze the loads and arithmetic operations separately, and then advance the arithmetic block as far to the left as possible. This transformation can only reduce the cost of E. ▪

Hereafter, all evaluations will be assumed to be compressed.

2.4. Schematic representation of compressed evaluations

Compressed evaluations may be described schematically as in Figure 3. In such a representation each 'A' stands for some arithmetic operation, and each 'L' represents a load operation.

As we shall see in Section 3, once an optimal compressed evaluation for a subforest F of a tree T has been determined, only its schematic representation is necessary to compute $C(T)$. Therefore, we shall only be interested in schematic representations.

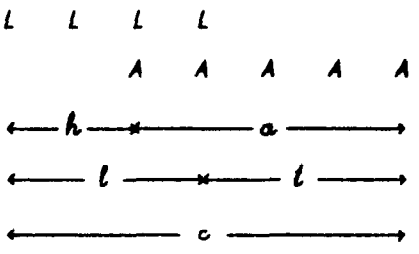

Figure 3. A schematic representation of E_2 from Figure 2

Schematic descriptions are fully characterized by triples of the form $<c,h,t>$ (c for cost, h for head, and t for tail). The three components are interrelated as follows:

$$h=c-a \quad (1)$$

$$t=c-l \quad (2)$$

In particular, for a given forest F, anyone of h, t, and c determines the others. Still, for our purposes it is easier to keep track of all three parameters. Thus, we define $S(F)$ to be the set of all different schematic representations of evaluations of F. Also, let $\mu(F)$ be the schematic representation for which $c(\mu(F))=C(F)$.

Another important parameter of a schematic representation is its *balance* b defined by

$$b=t-h. \quad (3)$$

To rate the quality of evaluations with respect to their cost as well as potential parallelism, we impose two disjoint relations on $S\times S$. The first denoted by $(<)$ is a strict order that is special in the sense that the order between two elements depends on their balance as well as on the values of h or t. The second $(\simeq)$ is an equivalence relation. Formally, let $f,g\in S$. We say that $f<g$ if one of the following conditions holds:

a. $b(f)>0$ and $b(g)\le 0$

b. $b(f),b(g)>0$ and $h(f)<h(g)$

c. $b(f),b(g)\le 0$ and $t(f)>t(g)$

Also, $f\simeq g$ if $b(f),b(g)>0$ and $h(f)=h(g)$, or $b(f),b(g)\le 0$ and $t(f)=t(g)$.

We say that $f\le g$ if $f\simeq g$ or $f<g$. The relation $\le$ is a total order modulo $\simeq$.

An *increasing* sequence $f_1,\ldots,f_k$ of elements from S is one in which for every two consecutive elements $f_i<f_{i+1}$. A *non-decreasing* sequence (as above) is one such that $f_i\le f_{i+1}$.

2.5. Concatenation of schematic representations

The *concatenation* (|) of two elements $f_1, f_2 \in S$ is the compressed evaluation obtained by performing the load operations of f_2 immediately after those of f_1, and the arithmetic operations of f_2 immediately after those of f_1. This may introduce a certain delay to the arithmetic operations of f_1 or f_2. We use the notation $f_{1,2} = f_1 | f_2$.

LEMMA 2.2. Let $f_1 = \langle c_1, h_1, t_1 \rangle$, $f_2 = \langle c_2, h_2, t_2 \rangle$, and $f_{1,2} = \langle c, h, t \rangle$. Then

$$b = t - h = b_1 + b_2 \tag{4}$$

$$h = h_1 + \max(h_2 - t_1, 0) \tag{5}$$

$$t = t_2 + \max(t_1 - h_2, 0) \tag{6}$$

$$c = c_1 + c_2 + \max(h_2 - t_1, 0) - h_2 \tag{7}$$

Proof. By case analysis. ▪

The concatenation operator is associative but not commutative. For example, let $f_1 = \langle 5,2,1 \rangle$, $f_2 = \langle 4,2,3 \rangle$, and $f_3 = \langle 6,3,2 \rangle$. Then $f_{1,2} = \langle 8,3,3 \rangle$, but $f_{2,1} = \langle 7,2,2 \rangle$. Also,

$$(f_1 | f_2) | f_3 = \langle 11,3,2 \rangle = f_1 | (f_2 | f_3).$$

Concatenation has the following effect on the ordering relation $\leq$: If $f_1 \leq f_2$ then $h(f_{1,2}) \leq h(f_{2,1})$.

These and other properties of concatenation are derived in Section 4.1.

3. An Algorithm to Compute $C(T)$

In this section we describe an efficient algorithm to find $C(T)$ for a tree T. The algorithm uses the boolean function (predicate) TEST, and three subroutines: MERGE, JOIN, and ADDOP.

3.1. TEST(f,g)

f and g are schematic representations such that $f \leq g$. $TEST(f,g)$=**true** in the following cases:

a. $b(f) > 0$ and $b(g) \geq 0$ and $h(g) \leq t(f)$.

b. $b(f) \leq 0$ and $b(g) = 0$.

c. $b(f) \leq 0$ and $b(g) < 0$ and $h(g) \geq t(f)$.

TEST(f,g)=**false** otherwise.

TEST (f,g) determines whether the two schematic representations f and g can be combined into one without causing a potential loss of optimality to the overall result.

3.2. MERGE($\bar{f}_1, \ldots, \bar{f}_k$)

MERGE may have any number $1 \leq k \leq d$ of arguments, each being an increasing sequence of schematic representations. The output is a (single) non-decreasing sequence formed by merging the input sequences using the $\leq$ ordering (the order between equivalent elements is immaterial).

3.3. JOIN($\bar{f}$)

$\bar{f} = f_1, \ldots, f_k$ is a non-decreasing sequence of schematic representations. $\bar{g}$=JOIN($\bar{f}$) is an increasing sequence of i elements $g_1, \ldots, g_i$ where $1 \leq i \leq k$ defined by:

```
j=1
i=1
do while j≤k
   g_i=f_j
   j=j+1
   do while j≤k and TEST(g_i,f_j)
      g_i=g_i|f_j
      j=j+1
   end
   i=i+1
end
```

JOIN($\bar{f}$) iteratively concatenates consecutive elements of $\bar{f}$ as long as they meet the criterion of TEST. If concatenated, the resulting element is further tested against the next element, and so on.

3.4. ADDOP($\bar{f}$)

$\bar{f} = f_1, \ldots, f_k$ is the increasing sequence of schematic representations. Let $f_k = \langle c,h,t \rangle$, and $f'_k = \langle c+1,h,t+1 \rangle$. $\bar{g}$=ADDOP($\bar{f}$) is an increasing sequence consisting of k or $k-1$ elements defined as follows:

```
do i=1 to k-2
   g_i=f_i
end
if TEST(f_{k-1},f'_k) then g_{k-1}=f_{k-1}|f'_k
                      else do
                              g_{k-1}=f_{k-1}
                              g_k=f'_k
                           end
```

3.5. The top-level procedure

We visit the vertices of T in a *bottom-up* order and compute at each vertex the increasing sequence $\bar{f}_v$ of schematic representations using the following program:

```
if d(v)=0 then if v∈L(T) then f̄_v=(<1,1,0>)
                         else f̄_v=(<1,0,1>)
   else do
      let u,w,... be the children of v
      f̄_v = ADDOP(JOIN(MERGE(f̄_u,f̄_w,...)))
   end
```

Finally, once we found $\bar{f} = f_1, \ldots, f_k$ at the root, $\mu(T)$ can be computed as

$$\mu(T) = f_1 | \ldots | f_k.$$

$C(T)$ is easily derived since $C(T) = c(\mu(T))$.

4. The optimality of the algorithm

To prove that the suggested algorithm indeed computes $C(T)$, we first derive certain properties of the concatenation operation.

4.1. Properties of the concatenation operation

In the sequel, let $f_i=<c_i, h_i, t_i>$ be arbitrary schematic representations.

The results in Lemmas 4.1 and 4.2 have already been mentioned in Section 2.

LEMMA 4.1 (Associativity).

$$(f_1|f_2)|f_3=f_1|(f_2|f_3)$$

Proof. It suffices to prove that

$$h((f_1|f_2)|f_3)=h(f_1|(f_2|f_3)).$$

By (5) and (6) we get

$$h((f_1|f_2)|f_3)=h_1+\max(h_2-t_1,0)+$$
$$+\max(h_3-(t_2+\max(t_1-h_2,0)),0)$$
$$h(f_1|(f_2|f_3))=$$
$$=h_1+\max(h_2+\max(h_3-t_2,0)-t_1,0)$$

Now we proceed by case analysis.

Case 1. $h_2\geq t_1$. We get:

$$h((f_1|f_2)|f_3) = h_1+h_2-t_1+\max(h_3-t_2,0)$$
$$h(f_1|(f_2|f_3)) = h_1+h_2-t_1+\max(h_3-t_2,0)$$

Case 2. $h_2<t_1$.

$$h((f_1|f_2)|f_3)=h_1+\max(h_3-t_2-t_1+h_2,0)$$

Subcase 2.1. $h_3\geq t_2$. Then we get

$$h(f_1|(f_2|f_3))=h_1+\max(h_2+h_3-t_2-t_1,0)$$

and the lemma follows.

Subcase 2.2. $h_3<t_2$. Therefore:

$$h(f_1|(f_2|f_3))=h_1+\max(h_2-t_1,0)=h_1$$

However, $h_3<t_2$ and $h_2<t_1$ also imply:

$$\max(h_2+h_3-t_2-t_1,0)=0.$$

Therefore

$$h((f_1|f_2)|f_3)=h_1$$

and the proof is completed. ▪

Since the concatenation operation is associative we shall omit parenthesis wherever convenient.

LEMMA 4.2 (Ordering). If $f_1\leq f_2$, then $h(f_{1,2})\leq h(f_{2,1})$.

Proof. By (5)

$$h(f_{1,2}) = h_1+\max(h_2-t_1,0)$$
$$h(f_{2,1}) = h_2+\max(h_1-t_2,0)$$

Case 1. $b_1\leq 0$. Here we get $b_2\leq 0$, and $h_1\geq t_1\geq t_2$. Then

$$h(f_{2,1})=h_2+h_1-t_2$$
$$h(f_{1,2})\leq h_1+h_2-t_1\leq h(f_{2,1})$$

Case 2. $b_1>0$.

Subcase 2.1. $b_2>0$. Here $h_1\leq h_2<t_2$. Then

$$h(f_{2,1})=h_2$$

If $h_2\leq t_1$ then $h(f_{1,2})=h_1\leq h(f_{2,1})$, and if $h_2>t_1$ then $h(f_{1,2})=h_1+h_2-t_1$, and again $h(f_{1,2})\leq h_2=h(f_{2,1})$.

Subcase 2.2. $b_2\leq 0$. If $h_2\geq t_1$ then

$$h(f_{1,2})=h_1+h_2-t_1$$

If $h_1<t_2$ then $h(f_{2,1})=h_2>h(f_{1,2})$. Otherwise $h(f_{2,1})=h_2+h_1-t_2$, and because $t_1>h_1\geq t_2$ it follows that $h(f_{1,2})<h(f_{2,1})$. If $h_2<t_1$ then

$$h(f_{1,2})=h_1$$

If $h_1<t_2\leq h_2$ then $h(f_{2,1})=h_2>h(f_{1,2})$. Otherwise if $h_1\geq t_2$ and then

$$h(f_{2,1})=h_2+h_1-t_2\geq h_1=h(f_{1,2})$$ ▪

Remark 4.1. $f_1<f_2$ does not imply $h(f_{1,2})<h(f_{2,1})$.

Remark 4.2. $f_1\simeq f_2$ implies $h(f_{1,2})=h(f_{2,1})$.

LEMMA 4.3 (Monotonicity). If $b_2=b_4$ and $h_4\leq h_2$ then $h(f_1|f_4|f_3)\leq h(f_1|f_2|f_3)$.

Proof. By (5) we get:

$$h(f_1|f_4)\leq h(f_1|f_2)$$

Let $f_5=f_1|f_2$ and $f_6=f_1|f_4$. Thus, $h_6\leq h_5$. Also, by $b_2=b_4$ and (4) we get

$$b_6=b_5.$$

To complete the proof we have to show:

$$h(f_6|f_3)\leq h(f_5|f_3).$$

This also follows from (5). ▪

LEMMA 4.4. If $b_1=0$ and $b_2\leq 0$ then $h(f_{1,2})\leq h(f_{2,1})$.

Proof. By (5)

$$h(f_{1,2}) = \max(h_2+h_1-t_1,h_1)=\max(h_2,h_1)$$
$$h(f_{2,1}) = \max(h_1+h_2-t_2,h_2)\geq\max(h_1,h_2)$$ ▪

LEMMA 4.5. If $b_1,b_2\leq 0$, and $h_3\geq t_1,t_2$ then

$$h(f_1|f_2|f_3)=h(f_2|f_1|f_3)$$

Proof. By $h_3\geq t_1,t_2$ we get

$$h(f_2|f_3)=h_2+\max(h_3-t_2,0)=h_2+h_3-t_2$$
$$h(f_1|(f_2|f_3))=h_1+\max(h_2+h_3-t_2-t_1,0)$$

$b(f_2)\leq 0$ implies that $h_2\geq t_2$, and therefore:

$$h(f_1|f_2|f_3)=h_3-b_1-b_2$$

By symmetry of the above with respect f_1 and f_2

$$h(f_1|f_2|f_3)=h(f_2|f_1|f_3)\ \bullet$$

LEMMA 4.6. If $b_2,b_3\geq 0$ and $h_2,h_3\leq t_1$ then

$$h(f_1|f_2|f_3)=h(f_1|f_3|f_2)$$

Proof. By (5) we get

$$h(f_1|f_2|f_3)=h_1+$$
$$+\max(h_3+\max(h_2-t_3,0)-t_1,0).$$

However,

$$h_3+\max(h_2-t_3,0)=$$
$$=\max(h_2-(t_3-h_3),0)\leq h_2\leq t_1.$$

Therefore $h(f_1|f_2|f_3)=h_1$. By symmetry the same is true for $h(f_1|f_3|f_2)$. •

4.2. Increasing sequences of schematic representations

LEMMA 4.7. Let f,g be schematic representations such that $f\leq g$. If TEST(f,g)=**false** then $f<g$; otherwise $f\leq(f|g)\leq g$.

Proof. By case analysis.

Case 1. $b(f)>0$. TEST(f,g)=**false** implies either $b(g)<0$, or $b(g)\geq 0$ and $h(g)>t(f)>h(f)$. In both cases it follows that $f<g$. If TEST(f,g)=**true** then by (4)

$$b(f|g)=b(f)+b(g)>0$$

By (5)

$$h(f|g)=h(f)+\max(h(g)-t(f),0)=h(f)$$

It follows that $h(f|g)=h(f)$ and thus $f\simeq f|g$.

Case 2. $b(f)\leq 0$. If TEST(f,g)=**false** then $b(g)<0$ and $h(g)<t(f)$. Therefore $t(g)<h(g)<t(f)$, which implies $f<g$. If TEST(f,g)=**true** then by (4)

$$b(f|g)=b(f)+b(g)\leq 0$$

By (6)

$$t(f|g)=t(g)+\max(t(f)-h(g),0).$$

Then $t(f)\geq t(f|g)\geq t(g)$. •

LEMMA 4.8. Let $\bar{f}$ be non-decreasing. Then $\bar{g}$=JOIN$(\bar{f})$ is increasing.

Proof. By induction on the length of $\bar{f}$.

Basis. The lemma is vacuously true for $|\bar{f}|=1$.

Inductive step. JOIN$(\bar{f})$ is increasing for every $\bar{f}$ for which $|\bar{f}|<k$. Let $\bar{g}$ be of length k. Then for $\bar{f}'=(f_1,\ldots f_{k-1})$, $\bar{g}'$=JOIN $(\bar{f}')$ is increasing. All is needed to complete JOIN$(\bar{f})$ is to test f_k for the possibility to concatenate with g' the last element of $\bar{g}'$. If TEST(g',f_k)=**false** (notice that by Lemma 4.7 $g'\leq f_{k-1}<f_k$) then by Lemma 4.7, $g'<f_k$, and $\bar{g}$ is increasing, otherwise by the same lemma $g'|f_k\geq g'>g''$ (g'' is the element before the last in $\bar{g}'$) and again $\bar{g}$ is increasing. •

LEMMA 4.9. Let $\bar{f}$ be such that $\bar{f}$=JOIN$(\bar{e})$ and $\bar{e}$ is non-decreasing. Then $\bar{g}$=ADDOP$(\bar{f})$ is increasing.

Proof. If $|\bar{f}|=1$ then $\bar{g}$ consists of one element and is trivially an increasing sequence. If $|\bar{f}|=k\geq 2$, then by the definition of ADDOP the first $k-2$ elements of $\bar{g}$ are equal to the respective elements of $\bar{f}$, and their increasing order is preserved. ADDOP only appends the arithmetic operation at the end of f_k, resulting in the new schematic representation

$$f'_k=<c_k+1,h_k,t_k+1>.$$

Claim: $f'_k\geq f_{k-1}$. If this holds then we can use Lemma 4.7: If TEST(f_{k-1},f'_k)=**false** we get:

$$g_1<\ldots<g_{k-2}<g_{k-1}<g_k$$

otherwise by the same lemma

$$g_{k-1}=f_{k-1}|f'_k\geq f_{k-1}>f_{k-2}$$

yielding the same result.

To complete the proof of the lemma we must prove the claim. By Lemma 4.8 $f_{k-1}<f_k$. We proceed by case analysis.

Case 1. $b_{k-1}>0$ and $b_k\neq 0$. Then either $b_k<0$, or $b_k>0$ and $h_k>h_{k-1}$. In both cases it immediately follows that $f_{k-1}<f'_k$.

Case 2. $b_{k-1}>0$ and $b_k=0$. Here we claim that $h_k>t_{k-1}$.

Subcase 2.1. $f_k=e_{|\bar{e}|}$. TEST(f_{k-1},f_k)=**false**. Then by the definition of TEST the claim follows.

Subcase 2.2. $f_k=e_j|\cdots|e_{|\bar{e}|}$ for some j. This implies that TEST(f_{k-1},e_j)=**false**.

TEST has the following property: when applied to an increasing sequence $\bar{q}$ for which $b(q_1|\cdots|q_{|\bar{q}|})=0$ and TEST$(q_1|\cdots|q_i,q_{i+1})$=**true** for all i, $\bar{q}$ must satisfy $b(q_i)=0$ for all i.

When this observation is applied to $\bar{q}=e_j,\ldots,e_{|\bar{e}|}$ it shows that $b(e_i)=0$ for all $j\leq i\leq|\bar{e}|$. Now that we know that $b(e_j)=0$ and TEST(f_{k-1},e_j)=**false**, it follows that $h(e_j)>t_{k-1}$.

An increasing sequence of elements whose balance is 0 has a decreasing value of h. Thus, their concatenation inherits the h-value of the first element. In our case

$$h_k=h(e_j|\cdots|e_{|\bar{e}|})=h(e_j).$$

Therefore $h_k>t_{k-1}$.

Case 3. $b_{k-1}\leq 0$. Here $t_{k-1}>t_k$. If $b_k<0$ the proof is obvious. $b_k=0$ is impossible since by an argument similar to that of Subcase 2.2 TEST(f_{k-1},e_j)=**false** and therefore $b(e_j)$ is already negative. •

In the sequel T is a tree on which the algorithm is applied, and $\bar{f}_v$ is the sequence that it attaches to $v \in T$.

THEOREM 4.1. $\bar{f}_{root}$ is increasing.

Proof. By induction on $|T|$.

Basis. For a tree with one vertex, $f_{root}=<1,1,0>$, or $f_{root}=<1,0,1>$. In both cases f_{root} is an increasing sequence.

Inductive step. Let us assume that for all trees with less than n vertices $\bar{f}_{root}$ is increasing. Let $|T|=n$.

Let $u,w,...$ be the children of root. Then

$$\bar{f}_{root}=ADDOP(JOIN(MERGE(\bar{f}_u,\bar{f}_w,...)))$$

All of $u,w,...$ are the roots of subtrees with $n-1$ or less vertices and by induction hypothesis $\bar{f}_u,\bar{f}_w,...$ are increasing. Then clearly MERGE$(\bar{f}_u,\bar{f}_w,...)$ is non-decreasing, by Lemma 4.8 JOIN(MERGE$(\bar{f}_u,\bar{f}_w,...)$) is increasing, and finally by Lemma 4.9

$$\bar{f}_{root}=ADDOP(JOIN(MERGE(\bar{f}_u,\bar{f}_w,...)))$$

is increasing. ▪

4.3. Correctness and optimality of the algorithm

THEOREM 4.2 (Correctness). The concatenation of the elements comprising $\bar{f}_{root}$ represents a legal evaluation of T.

Proof. By induction on $|T|$.

Basis. For a tree with one vertex the claim is trivially true.

Inductive step. Let us assume that the lemma holds for all trees with less than n vertices. Let $u,w,...$ be the children of the root. Then the inductive hypothesis holds for $\bar{f}_u,\bar{f}_w,...$. To obtain $\bar{f}_{root}$ the algorithm computes:

$$\bar{f}_{root}=ADDOP(JOIN(MERGE(\bar{f}_u,\bar{f}_w,...)))$$

The ADDOP function only appends the arithmetic operation at the end of the evaluation in compliance with the precedence constraints, while JOIN never changes the order of evaluation. By Theorem 4.1 $\bar{f}_u,\bar{f}_w,...$ are increasing sequences, and therefore the order among their elements is preserved by MERGE. ▪

THEOREM 4.3 (Optimality). The concatenation of the elements of $\bar{f}_{root}$ is minimal.

Proof. The algorithm may be viewed as a routine which manipulates a bag P of schematic representations. Let

$$P_0=\{<1,1,0> : v \in L\} \cup \{<1,0,1> : v \in A\}.$$

We totally ignore the MERGE operation and concentrate on the most basic activity of the algorithm, namely the combination of two schematic representations into one. If the algorithm combines f and g at step m, then P_m is obtained from P_{m-1} as follows:

$$P_m=(P_{m-1}-\{f,g\}) \cup \{f|g\}.$$

Claim: For all m, the elements of P_m can be arranged into a sequence so that

$$f_1|f_2|\cdots|f_{|P|}=\mu(T). \qquad (8)$$

Once this claim has been proven, the bag which is the last to be obtained contains exactly the elements of $\bar{f}_{root}$. By Lemma 4.2 concatenating them in an increasing order is optimal.

Proof of claim: By induction on m.

Basis. $m=0$. Let (λ,α) be an optimal order of evaluation of T. To define the P_0 we first build a sequence of $c-a$ elements of the form $<1,1,0>$ where $\lambda(u_i)=i \le c-a$, then we add two elements at a time - $<1,0,1>$ and $<1,1,0>$ where $c-a<\lambda(u_i)=\alpha(v_i)=i \le l$ and finally we add $<1,0,1>$ where $l<\alpha(w_i)=i \le c$. It is easy to check that equation (8) holds for this construction.

Induction hypothesis: Let $P_{m-1}=\{\ f_i\ \}$ be such that (8) holds.

Induction step: Assume, that during step m the algorithm combines two schematic representations f_i and f_j, $f_i \le f_j$. We must prove the existence of an ordering of P_m which fulfills (8).

Case 1. $i<j$. Let $g=f_{i+1}|...|f_{j-1}$.

Subcase 1.1. $f_i \ge g$. Here we can make the following reordering of the elements of P_m, and then combine f_i and f_j as described by:

$$f_i,g,f_j \rightarrow g,(f_i|f_j)$$

and by Lemmas 4.2 and 4.3 the overall cost does not increase.

Subcase 1.2. $g \ge f_j$. Here the reordering is:

$$f_i,g,f_j \rightarrow (f_i|f_j),g$$

and again by Lemmas 4.2 and 4.3 the overall cost does not increase.

Subcase 1.3. $f_i<g<f_j$.

Subcase 1.3.1. $b(f_i)>0$. Here $b(f_j)>0$ and $h(f_j) \le t(f_i)$. Therefore, $b(g)>0$ and

$$h(f_i)<h(g)<h(f_j)$$

We reorder as follows:

$$f_i,g,f_j \rightarrow (f_i|f_j),g$$

and by Lemmas 4.6 and 4.3 the overall cost does not increase.

Subcase 1.3.2. $b(f_i) \le 0$. Here either $b(f_j)=0$ or $b(f_j)<0$ and $h(f_j) \ge t(f_i)$. Since $f_i<g$, $t(f_i)>t(g)$. In case $b(f_j)=0$ we can reorder:

$$f_i,g,f_j \rightarrow (f_i|f_j),g$$

and by Lemmas 4.4 and 4.3 the overall cost does not increase. If $b(f_j)<0$ then we make the reordering:

$$f_i, g, f_j \rightarrow g, (f_i | f_j)$$

and by Lemmas 4.5 and 4.3 the cost again does not increase.

Case 2. $i>j$. Let $g=f_{j+1}|\cdots|f_{i-1}$. It is clear that the order $f_j<g<f_i$ does not hold. If $g \leq f_j$ then we reorder:

$$f_j, g, f_i \rightarrow g, f_j, f_i \rightarrow g, (f_i | f_j)$$

otherwise we reorder:

$$f_j, g, f_i \rightarrow f_j, f_i, g \rightarrow (f_i | f_j), g$$

and in both cases by Lemmas 4.2 and 4.3 the cost does not increase. •

5. The Time Complexity of the Algorithm

ADDOP, MERGE and JOIN are called once for every internal vertex. ADDOP changes at most the last two elements of its input sequence, therefore it can be implemented in such a way that its running time is bounded by a constant.

JOIN is always applied to the result of MERGE, therefore the two procedures can be performed as one pass on the input of MERGE during which JOIN acts on the output of MERGE while it is being formed. Also, the MERGE operation at a vertex with more than 2 children can be effected by a series of binary MERGE operations applied to the subtrees in any order.

To process ordered sequences that are propagated from the leaves of a tree all the way up to its root, the following time-complexity recurrence relation holds at every (binary) internal vertex of the tree:

$$\tau(p+q)=\tau(p)+\tau(q)+r \cdot q$$

where p,q are the sizes of the two subtrees hanging off the vertex, $p \geq q$, and r is a global bound on the cost of the MERGE and JOIN operations per element in the sequence associated with the lighter tree.

Solving this recurrence (with $\tau(1)=1$) we obtain $\tau(n)=r \cdot n \cdot \log n$, since

$$\tau(p+q)=\tau(p)+\tau(q)+r \cdot q =$$
$$=r \cdot p \cdot \log p + r \cdot q \cdot \log q + r \cdot q =$$
$$=r \cdot p \cdot \log p + r \cdot q \cdot \log 2q \leq$$
$$\leq r \cdot p \cdot \log(p+q) + r \cdot q \cdot \log(p+q) =$$
$$=r \cdot (p+q) \cdot \log(p+q).$$

The bound r accounts for the need to insert elements from one sequence into the other in the right place, and also for the tests with its neighbours in order to decide whether they are to be concatenated or not (as part of JOIN). If the sequences are maintained as balanced trees, and if proper accounting of the effects of successful *vs.* unsuccessful TESTs is taken, careful analysis shows that $r=O(\log n)$ where n is the total number of vertices in the tree. All in all we obtain

THEOREM 5.1. The complexity of the algorithm is $O(n \log^2 n)$.

6. Extensions and Open Problems

Our results can be extended in a number of ways, but also leave a few problems open to further research.

6.1. Store operations

When results of computing subtrees need to be stored in memory, store operations are added to the evaluation in memory access slots. Clearly, grouping all stores after the last load has been completed, will not harm the optimality of the algorithm. Only when the number of stores exceeds t at the root, need the cost be increased. Still, in this case the computation is memory bound and its cost is the total number of memory access operations, which is optimal.

6.2. An algorithm for forests

If initially we are given a forest of expression trees rather than a single tree, we can construct a new tree by adding a "dummy" root that is the parent of all the roots of the forest. Then the algorithm is applied to the tree thus obtained, except for the last call to ADDOP.

6.3. Finding an optimal evaluation

Our algorithm can be easily augmented to provide an optimal evaluation of an expression tree T in addition to just computing $C(T)$. To do this a list is attached to every variable that contains a schematic representation. Each such list contains the vertices in the subtree corresponding to the schematic representation of the associated variable. Whenever two schematic representations f and g are concatenated, so are their respective vertex lists. Note that whenever a sequence of schematic representation is maintained, we must also keep a similar structure of lists; it is important not to confuse lists with sequences. With proper implementation, these list manipulation operations do not increase the complexity of the algorithm.

6.4. Non unitary execution times

Finally, we can relax our restriction that the execution time of memory access is equal to that of the arithmetic operations. Let τ_1 be the (integral) number of machine cycles required by a memory access, and τ_2 is how long an arithmetic operation takes. These numbers will be used now by the algorithm instead of 1 to charge for the appropriate operation; for example, a load leaf is represented by $<\tau_1,\tau_1,0>$. Since all the operations on schematic representations are defined in terms of numbers, this transition is rather smooth and natural.

6.5. Better complexity results

It seems that the complexity analysis of Section 5 can be tightened. In particular, we were able to prove that the time complexity goes down to $O(n\log n)$ when all the vertices of the forest have at most two children each (i.e. $d(T)\leq 2$). In the general case, the following observation might be of help in trying to tighten the analysis: for every vertex v, $|\bar{f}_v|=O(\sqrt{|T_v|})$ where T_v is the subtree rooted at v. Thus, the effort that goes into JOIN and MERGE is in fact smaller than what we charge for in our analysis.

6.6. Finite number of registers

The main problem left open by this work is the case when the number of registers in the machine is bounded. It seems that the purely binary case (i.e. $d(v)=2$ for all the internal vertices in T) could be easier, but the general case is evidently hard.

Acknowledgement. We would like to thank Jeff Jaffe for numerous suggestions that improved (and sometimes corrected) the presentation of our results.

References

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *Data structures and algorithms*, Addison-Wesley, Reading, MA, 1983.

[2] Aho, A.V., and Jonhson, S.C., "Optimal code generation for expression trees", *JACM 23, 3* (Jul. 1976), 488-501.

[3] Even, S., *Graph algorithms*, Computer Science Press, Rockville, MD, 1979.

[4] Knuth, D.E., *The art of computer programming: Sorting and Searching (Vol. 3)*, Addison-Wesley, Reading, MA, 1973.

[5] Li, H.F., "Scheduling trees in parallel/pipelined processing environments", *IEEE transactions on computers*, C-26, 11 (Nov. 1977), 1101-1112.

[6] Lloyd, E.L., *Scheduling task systems with resources*, Ph.D. dissertation, MIT-LCS-TR-236, May 1980.

[7] Sethi, R., and Ullman, J.D., "The generation of optimal code for arithmetic expressions", *JACM 17, 4* (Oct. 1970), 715-728.

[8] Smith, J. E., "Decoupled access/execute computer architectures", *Proceedings of the 9th Symposium on Computer Architecture*, (April 1982), 112-119.