Generating Optimal Contiguous Evaluations for Expression DAGs

C. W. Kessler* Thomas Rauber †
Computer Science Department
Universität des Saarlandes
Postfach 151150
66041 Saarbrücken, Germany
+49–681–302–4130
FAX 49–681–302–4290
{kessler,rauber}@cs.uni–sb.de

Abstract

We consider the NP–complete problem of generating contiguous evaluations for expression DAGs with a minimal number of registers. We present two algorithms that generate optimal contiguous evaluation for a given DAG. The first is a modification of a complete search algorithm that omits the generation of redundant evaluations. The second algorithm generates only the most promising evaluations by splitting the DAG into trees with import and export nodes and evaluating the trees with a modified labeling scheme. Experiments with randomly generated DAGs and large DAGs from real application programs confirm that the new algorithms generate optimal contiguous evaluations quite fast.

Key words: program optimization, basic block, expression DAG, contiguous evaluation, register allocation, code generation

1 Introduction

Register allocation is one of the most important problems in compiler optimization. Using fewer registers is important if the target machine has not enough registers to evaluate an expression without storing intermediate results in the main memory (*spilling*). This is especially important for vector processors that are often used in parallel computers. Vector processors usually have a small number

*Research partly supported by DFG, SFB 124, TP D4. Present address: FB 4 Informatik, Universität Trier, D-54286 Trier, Germany

†supported by DFG, SFB 124, TP D4

of vector registers (e.g., the CRAY vector computers have 8 vector register of 64×64 bit) or a register file that can be partitioned into a number of vector registers of a certain length (e.g., the vector acceleration units of the CM5 have register files of length 128×32 bit that can be partitioned into 1, 2, 4 or 8 vector registers, see [1]). A vector operation is evaluated by splitting it into stripes that have the length of the vector registers and computing the stripes one after another. If the register file is partitioned into a small number of vector registers, each of these can hold more elements and the vector operation to be evaluated is split into fewer stripes. This saves initialization costs and results in a faster computation [2].

Scientific programs often contain large basic blocks. Large basic blocks can also result from the application of compiler techniques like loop unrolling [3] and trace scheduling [4]. Therefore, it is important to derive register allocation techniques that cope with large basic blocks [5].

Among the numerous register allocation schemes, register allocation and spilling via graph coloring [6, 7] is generally accepted to yield good results. But register allocation via graph coloring uses a fixed evaluation order within a given basic block B. This is the evaluation order specified in the input program. Often there exists an evaluation order for B that allows to use fewer registers. By using this order, the global register allocation generated via graph coloring could be improved.

The reordering of the operations within a basic block can be arranged by representing the basic block by a number of directed acyclic graphs (DAGs). An algorithm to build the DAGs for a given basic block can be found in [8]. A basic block is evaluated by evaluating the corresponding DAGs. For the evaluation of a DAG G the following results are known:

(1) If G is a tree, the algorithm of *Sethi* and *Ullman* [9] generates an optimal evaluation in linear time. (In this paper, optimal always means: uses as few registers as possible. Recomputations are not allowed.)

(2) The problem of generating an optimal evaluation for G is NP–complete, if G is not restricted [10].

In this paper, we restrict the attention to contiguous evaluations. Experiments with randomly generated DAGs and with DAGs that are derived from real programs show that for nearly all DAGs, there exist a contiguous evaluation that is optimal. This leads to an algorithm that computes an optimal contiguous evaluation for a given DAG in time $O(n \cdot 2^d)$ where d is the number of decision nodes [11]. Decision nodes are binary nodes on paths from the root of the DAG to a node with more than one father.

This paper improves this simple $O(n \cdot 2^d)$ algorithm that performs a rather inefficient complete search, by identifying and eliminating redundant evaluations. It also presents a new algorithm that splits the given DAG into a number of trees with import and export nodes and evaluates the trees with a modified labeling scheme. Import and export nodes constitute the connection between the generated trees: when evaluating the tree, export nodes are nodes that remain into registers because they are used later by neighboring trees. On the other hand, import nodes need not to be loaded into a register because they have been left there by the evaluation of a neighboring tree. To find an optimal contiguous evaluation, the new algorithm considers all possibilities to split the given DAG into trees and selects the splitting that uses the fewest registers. Experiments with DAGs from real applications show that the number of generated evaluations is quite small even for large DAGs. Therefore, the running time of the algorithm remains reasonable.

After giving some basic definitions in Section 2, we describe in Section 3 how the running time of the algorithm from [11] can be reduced by generating each evaluation only once. In Section 4, we show how the running time can be further reduced by splitting the DAG in several trees with import and export nodes and applying a modified labeling scheme to the trees. Section 5 describes the splitting procedure, Section 6 presents the modified labeling scheme for trees with import and export nodes and proves that the generated evaluations are optimal. Section 7 shows the experimental results that confirm that the described method can be used in practice to generate optimal contiguous evaluations even for large DAGs.

2 Evaluating DAGs

2.1 Expression DAGs

We assume that we are generating code for a single processor machine with general–purpose registers $\mathcal{R} = \{R_0, R_1, R_2, \ldots\}$ and a countable sequence of memory locations. The arithmetic machine operations are three-address instructions of the following types:

$R_k \leftarrow R_i \; op \; R_j$		binary operation, $op \in \{+, -, \times, \ldots\}$,
$R_k \leftarrow op \; R_i$		unary operation,
$R_k \leftarrow \mathtt{Load}(a)$		load register k with the value in memory location a
$\mathtt{Store}(a) \leftarrow R_k$		store the value in register k into memory location a,

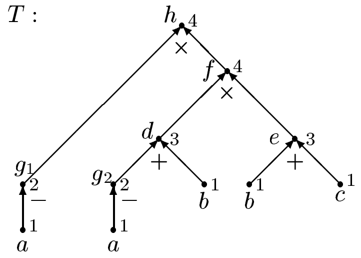
In the following, we assume $i \neq j \neq k \neq i$, for $R_k, R_i, R_j \in \mathcal{R}$ to facilitate the description. Note that the following considerations are also applicable, if $k = i$ or $k = j$.

Each input program can be partitioned into a number of basic blocks.

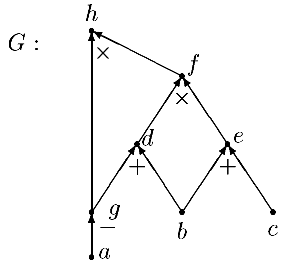
A *directed graph* is a pair $G = (V, E)$, where V is a finite set of nodes and $E \subseteq V \times V$ is a set of edges. In the following, $n = |V|$ always stands for the number of nodes in the graph. A node w is called *operand* or *son* of a node v, if $(w, v) \in E$. v is called *result* or *father* of w, i.e., the edge is directed from the son to the father. A node with no father is called a *root* of G. A node which has no sons is a *leaf*, otherwise it is an *inner node*. We call a node with two sons *binary* and a node with only one son *unary*. In the following, we suppose for simplicity that the DAGs contain only unary and binary inner nodes.

The *outdegree* $outdeg(w)$ is the number of edges leaving w, i.e., the number of its fathers.

The data dependencies in a basic block can be described by a *directed acyclic graph (DAG)*. The leaves of the DAG are the variables and constants occurring as operands in the basic block; the inner nodes represent intermediate results. An example is given in Figure 1.



a	$R_1 \leftarrow \texttt{Load}(a)$
g_2	$R_2 \leftarrow -R_1$
b	$R_1 \leftarrow \texttt{Load}(b)$
d	$R_3 \leftarrow R_2 + R_1$
b	$R_1 \leftarrow \texttt{Load}(b)$
c	$R_2 \leftarrow \texttt{Load}(c)$
e	$R_4 \leftarrow R_1 + R_2$
f	$R_1 \leftarrow R_3 \times R_4$
a	$R_2 \leftarrow \texttt{Load}(a)$
g_1	$R_3 \leftarrow -R_2$
h	$R_2 \leftarrow R_3 \times R_1$



A_0 :

a	$R_1 \leftarrow \texttt{Load}(a)$	c	$R_1 \leftarrow \texttt{Load}(c)$
g	$R_2 \leftarrow -R_1$	b	$R_2 \leftarrow \texttt{Load}(b)$
b	$R_1 \leftarrow \texttt{Load}(b)$	e	$R_3 \leftarrow R_1 + R_2$
d	$R_3 \leftarrow R_2 + R_1$	a	$R_1 \leftarrow \texttt{Load}(a)$
c	$R_4 \leftarrow \texttt{Load}(c)$	g	$R_4 \leftarrow -R_1$
e	$R_5 \leftarrow R_1 + R_4$	d	$R_1 \leftarrow R_4 + R_2$
f	$R_1 \leftarrow R_3 \times R_5$	f	$R_2 \leftarrow R_1 \times R_3$
h	$R_3 \leftarrow R_2 \times R_1$	h	$R_1 \leftarrow R_4 \times R_2$

Figure 1: Example: The expression $(-a) \times ((-a+b) \times (b+c))$ can be represented by the tree T shown above. The tree can be evaluated by the labeling algorithm of *Sethi/Ullman* with 4 registers as shown to the right. (The labels are printed at the right–hand side of the nodes). By eliminating common subexpressions, a DAG G can be constructed. Evaluating G in the original order results in an evaluation A_0 that reduces the number of instructions and hence the computation time of the basic block, but uses 5 instead of 4 registers. By reordering A_0 as shown to the right, we get an evaluation that needs only 4 registers.

Definition 1 *(*subDAG*) Let $G = (V, E)$ be a DAG. A DAG $S = (V', E')$ is called* subDAG *of G, if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. A subDAG $S = (V', E')$ of $G = (V, E)$ with root w is called* complete*, if:*

$V' \;=\; \{v \in V : \; \exists$ *path from v to w* $\}$ *and*

$E' \;=\; \{e \in E : \; e$ *is an edge on a path from a node $v \in V'$ to w* $\}$.

2.2 DAG Evaluations

We now consider the evaluation of DAGs. Let $G = (V, E)$ be a directed graph with n nodes. A mapping *ord:* $V \to \{1, 2, \ldots, n\}$ with

$$\forall (w, v) \in E : \quad ord(w) < ord(v)$$

is called a *topologic order* of the nodes of G. It is well–known that for a directed graph G a topological order exists iff G is acyclic (e. g., [12]).

Definition 2 *(*evaluation of a DAG*) An evaluation A of a DAG G is a permutation of the nodes in V such that for all nodes $v \in V$ the following holds: If v is an inner node with sons $v_1, \ldots, v_k$, then v occurs in A behind v_i, $i = 1, ..., k$.*

This implies that the evaluation A is *complete* and contains *no recomputations*, i. e., each node of the DAG appears exactly once in A. Moreover, the evaluation is *consistent*, because no node is evaluated before all of its sons are evaluated. Thus, each topological order of G represents an evaluation, and vice versa.

Definition 3 *(*contiguous evaluation*) An evaluation A (represented by the topological order ord) of a DAG $G = (V, E)$ is called* contiguous*, if for each node $v \in V$ with children v_1 and v_2 the following is true: if w_i is a predecessor of v_i, $i = 1, 2$, and $ord(v_1) < ord(v_2)$, then $ord(w_1) < ord(w_2)$.*

A contiguous evaluation of a node v first evaluates the complete subDAG with one of the children of v as root before evaluating any part of the remaining subDAG with root v.
While general evaluations can be generated by variants of topological–sort[1], contiguous evaluations are generated by variants of *depth–first search* (*dfs*). From now on, we restrict our attention to contiguous evaluations to reduce the number of generated evaluations. By doing so, we may not always get the evaluation with the least register need. There are some DAGs for which a general evaluation exists that uses fewer registers than every contiguous evaluation. However, these DAGs are usually quite large and do very rarely occur in real programs. The smallest DAG of this kind that we could construct so far has 14 nodes and is given in Figure 2. Note that for larger DAGs, it is quite difficult to decide whether there exists a general evaluation that uses fewer registers than every contiguous evaluation. This is because of the enormous running time of the algorithms that generate general evaluations.

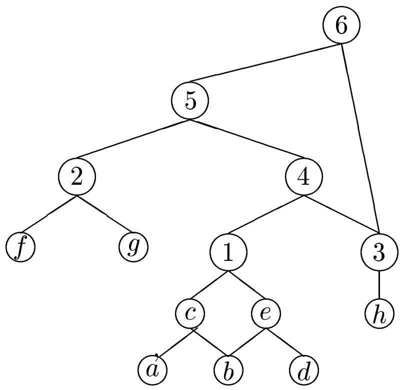


Figure 2: For this DAG, the noncontiguous evaluation (a, b, c, d, e, 1, f, g, 2, h, 3, 4, 5, 6) uses 4 registers. There is no contiguous evaluation that uses fewer than 5 registers. To use 4 registers, a contiguous evaluation would have to evaluate the subDAG with root 1 first. A contiguous evaluation could do this only by first evaluating the left son of node 6, then the right son of node 5 and then the left son of node 4. In order this evaluation to be contiguous, nodes h and 3 must be evaluated after 1 and thus, the values of nodes 3 and 4 must be held in two registers. But in order to evaluate nodes f, g and 2, three more registers would be required. Thus, the contiguous evaluation would use at least five registers altogether.

Definition 4 *(*register allocation, register need, optimal evaluation*) (cf. [10]) Let $num : \mathcal{R} \to \{0, 1, 2 \ldots\}$, $num(R_i) = i$ be a function that assigns a number to each register. A mapping $reg : V \to \mathcal{R}$ is called a (consistent)* register allocation *for A, if for all nodes u, v, $w \in V$ the following holds: If u is a son of w, and v appears in A between u and w, then $reg(u) \neq reg(v)$.*

$$m(A) = \min_{reg \text{ is reg. alloc. for } A} \{ \max_{v \text{ appears in } A} \{ num(reg(v)) + 1 \} \} \tag{1}$$

[1] See [12] for a summary on topological sorting.

is called the register need *of the evaluation A. An evaluation A for a DAG G is called* optimal *if for all evaluations A' of G holds $m(A') \geq m(A)$.*

Sethi proved in 1975 [10] that the problem of computing an optimal evaluation for a given DAG is NP–complete. Assuming $\mathbf{P} \neq \mathbf{NP}$, we expect an optimal algorithm to require nonpolynomial time.

3 Counting Evaluations

In [11], we give the following definitions and prove the following lemmata:

Definition 5 *(*tree node*)*

(1) Each leaf is a tree node.

(2) An inner node is a tree node iff all its sons are tree nodes and none of them has outdegree > 1.

Definition 6 *(label)*

(1) For every leaf v, $label(v) = 1$.

(2) For a unary node v, $label(v) = \max\{label(son(v)), 2\}$.

(3) For a binary node v, $label(v) = max\{3,\ \max\{label(lson(v)), label(rson(v))\} +\ q\}$ where $q = 1$, if $label(lson(v)) = label(rson(v))$, and 0 otherwise.

Let *new_reg*() be a function that returns an available register and marks it to be busy. Let *regfree*(*reg*) be a function that marks the register *reg* to be free again. A possible implementation is given in Section 8. The Labeling–algorithm *labelfs* of *Sethi* and *Ullman* (see [9]) generates optimal evaluations for a tree with labels by first evaluating the son with the greater label value for each binary node.

```
(1) function  labelfs(node v)
    // generates an optimal evaluation for the subtree with root v //
(2) if v is not a leaf
(3) then if label(lson(v)) > label(rson(v))
(4)          then labelfs(lson(v)); labelfs(rson(v))
(5)          else  labelfs(rson(v)); labelfs(lson(v))
             fi
    fi
(6) reg(v)  ←  new_reg(); print(v, reg(v));
(7) if v is not a leaf then regfree(reg(lson(v))); regfree(reg(rson(v))) fi
    end  labelfs;
```

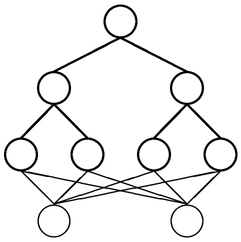


Figure 3: A DAG with $n-2$ decision nodes.

Definition 7 *(*decision node*) A* decision node *is a binary node which is not a tree node.*

Thus, all binary nodes that have at least one predecessor with more than one father are decision nodes. In a tree, there are no decision nodes. For a general DAG let d be the number of decision nodes and b be the number of *binary* tree nodes. Then $k = b + d$ is the number of binary nodes of the DAG. A DAG may have up to $d = n - 2$ decision nodes, see Figure 3.

Lemma 1 *For a tree T with one root and b binary nodes, there exist exactly 2^b different contiguous evaluations.*

Lemma 2 *For a DAG with one root and k binary nodes, there exist at most 2^k different contiguous evaluations.*

Lemma 3 *Let G be a DAG with d decision nodes and b binary tree nodes which form t (disjoint) subtrees $T_1, \ldots, T_t$. Let b_i be the number of binary tree nodes in T_i, $i = 1 \ldots t$, with $\sum_{i=1}^{t} b_i = b$. Then the following is true: If we fix an evaluation A_i for T_i, then there remain at most 2^d different contiguous evaluations for G.*

Corollary 4 *If we evaluate all the tree nodes in a DAG G with d decision nodes by labelfs(), there remain at most 2^d different contiguous evaluations for G.*

The following simple algorithm performs a complete search to create all 2^d contiguous evaluations for G, provided that a fixed contiguous evaluation for the tree nodes of G is used:

---

algorithm *complete_search*
Let $v_1, \ldots, v_d$ be the decision nodes of a DAG G, and
let $\beta = (\beta_1, \ldots, \beta_d) \in \{0,1\}^d$ be a bitvector.
forall 2^d different $\beta \in \{0,1\}^d$ **do**
    start *dfs*(*root*) with each β, such that for $1 \leq i \leq d$
      if $\beta_i = 0$ in the call *dfs*(v_i),
      then the left son of v_i is evaluated first
      else the right son of v_i is evaluated first **fi**
od
end *complete_search*;

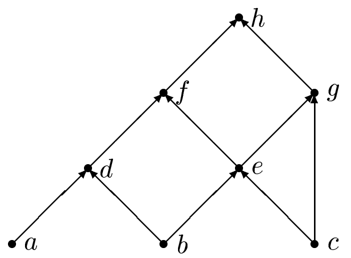


Figure 4: Example DAG.

This algorithm has exponential running time, since a DAG with n nodes can have up to $d = n - 2$ decision nodes, see Figure 3. The running time of the algorithm can be reduced by exploiting the following observation (consider the example DAG in Figure 4): Assume that the algorithm to generate a contiguous evaluation decides to evaluate the left son f of the root h first (i.e., the decision bit of h is set to zero). Then node e appears in the evaluation before g, since e is in the subDAG of f, but g is not. Therefore, there is no real decision necessary when node g is evaluated, because the son e of g is already evaluated. But because g is a decision node, the algorithm generates bitvectors containing 0s and 1s for the decision bit of g, although bitvectors that only differ in the decision bit for g describe the same evaluation.

We say that g is *excluded* from the decision by setting the decision bit of h to 0, because the son e (and c) are already evaluated when the evaluation of g starts. We call the decision bit of g *redundant* and mark it by an asterisk ($*$).

The following algorithm computes only those bitvectors that yield different evaluations. We suppose again that tree nodes are evaluated by the labeling algorithm *labelfs*:

Let $v_1, \ldots, v_d$ be the decision nodes in reverse topological order (i.e., the root comes first)

We call the following function *descend*$(\Theta, 1)$ where Θ is a bitvector that contains d 0's.

function *descend* (**bitvector** β, **int** *pos*)
while $\beta_{pos} = *$ **and** $pos < d$ **do** $pos \leftarrow pos + 1$ **od**
if $pos \geq d$
then if $\beta_{pos} = *$
    then print β // new evaluation found //
    else // β_{pos} is empty: //
        $\beta_d = 0$; **print** β; // new evaluation found //
        $\beta_d = 1$; **print** β; // new evaluation found //
    fi
else $\beta_{pos} = 0$;
    mark exclusions of nodes v_j, $j \in \{pos + 1, ..., d\}$ through $lson(v_{pos})$ by $\beta_j \leftarrow *$;
    descend(β, $pos + 1$);
    $\beta_{pos} = 1$;
    mark exclusions of nodes v_j, $j \in \{pos + 1, ..., d\}$ through $rson(v_{pos})$ by $\beta_j \leftarrow *$;
    descend(β, $pos + 1$);

decision nodes $v_1, v_2, \ldots, v_5$:	h	f	g	d	e	
start at the root: preset first bit:	**0**		$*$			
propagate bits and asterisks to next stage:	0	**0**	$*$		$*$	
all bits set: first evaluation found:	0	0	$*$	**0**	$*$	A_1
	0	0	$*$	**1**	$*$	A_2
'backtrack':	0	**1**	$*$	$*$		
	0	1	$*$	$*$	**0**	A_3
	0	1	$*$	$*$	**1**	A_4
'backtrack':	**1**	$*$		$*$		
	1	$*$	**0**	$*$		
	1	$*$	0	$*$	**0**	A_5
	1	$*$	0	$*$	**1**	A_6
'backtrack':	1	$*$	**1**	$*$	$*$	A_7

Table 1: For the example DAG of Figure 4, the algorithm *descend* executes the above evaluation steps. Only 7 instead of $2^5 = 32$ contiguous evaluations are generated.

fi

end *descend*;

Table 1 shows the application of *descend* to the example DAG of Figure 4.

Lemma 5 *For a DAG G without unary nodes, the algorithm* descend *generates at most 2^{d-1} different contiguous evaluations.*

Proof: If there are only binary inner nodes, there must exist a DAG node v that has at least two different fathers w_1 and w_2. Suppose w_1 is evaluated first. Then the decision bit of w_2 is redundant and is set to $*$. $\square$

Let N be the number of different contiguous evaluations returned by the algorithm *descend*. We have $N = 7$ for the example DAG of Figure 4. We call $d_{\mathit{eff}} = \log N$ the *effective number of decision nodes* of G. It is $d_{\mathit{eff}} \leq d - 1$ because of Lemma 3 and Lemma 6.

Furthermore, we can show the following *lower bound*:

Lemma 6 $d_{\mathit{eff}} \geq \min_{P \text{ path from the root to some leaf}} \#\text{decision nodes on } P$

Proof: There must be at least as many bits set to 0 or 1 in each final bitvector as there are decision nodes on an arbitrary path from some leaf to the root, because no exclusion is possible on the path to the node being evaluated first. The bitvector describing the path with the smallest number of decision nodes is enumerated by the algorithm, so the lower bound follows. $\square$

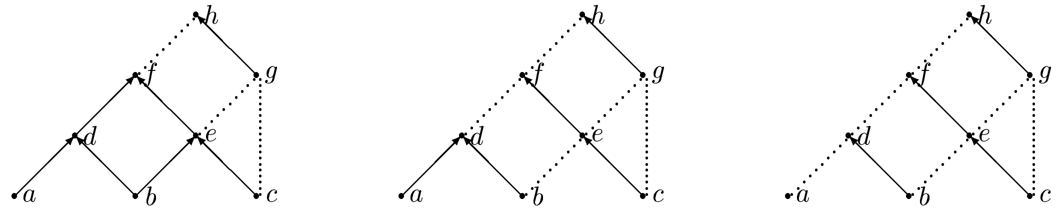


Figure 5: The example DAG is split in three steps by setting $\beta_1 = 0$, $\beta_2 = 0$, $\beta_4 = 0$. The edges between the generated subtrees are shown as dotted lines.

In the example above, the lower bound for d_{eff} is 2, since the path with the least number of decision nodes is (h, g, c) which has two decision nodes.
This lower bound may be used to get a lower bound $(2^{d_{eff}})$ for the run time of the algorithm *descend.*

4 Reducing the Number of Evaluations

We now construct an algorithm that reduces the number of generated evaluations further. The reduction is based on the following observation: Let v be a decision node with two children v_1 and v_2. Let $G(v) = (V(v), E(v))$ be a DAG with root v, $G(v_i)$ the complete subDAG with root v_i, $i = 1, 2$. By deciding to evaluate v_1 before v_2, we decide to evaluate all nodes of $G(v_1)$ before the nodes in $G_{rest} = (V_{rest}, E_{rest})$ with $V_{rest} = V(v) - V(v_1), E_{rest} = E(v) \cap (V_{rest} \times V_{rest})$. Let $e = (u, w) \in E(v)$ be an edge with $u \in V(v_1), w \in V_{rest}$. The function *descend* marks w with a $*$. This can be considered as eliminating e: at decision node w, we do not have the choice to evaluate the son u first, because u has already been evaluated and will be held in a register until w is evaluated. Therefore, *descend* can be considered as splitting the DAG G into smaller subDAGs. We will see later that these subDAGs are trees after the splitting has been completed. The root of each of these trees is a decision node.[2]
The trees are evaluated in reverse of the order in which they are generated. For the example DAG of Figure 4, there are 7 possible ways of carrying out the splitting. The splitting steps that correspond to evaluation A_1 from Table 1 are shown in Figure 5.
If we look at the subDAGs that are generated during the splitting operation, we observe that even some of the intermediate subDAGs are trees which could be evaluated without a further splitting. E.g., after the second splitting step ($\beta_2 = 0$) in Figure 5, there is a subtree with nodes a, b, d which does not need to be split further, because an optimal contiguous evaluation for the subtree can be found by a variant of *labelfs.*. By stopping the splitting operations in these cases, the number of generated evaluations can be reduced from 7 to 3 for the example DAG.
Depending on the structure of the DAG, the number of generated evaluations may be reduced dramatically when splitting the DAG into trees. An example is given in Figure 6. To evaluate the generated trees we need a modified labeling algorithm that is able to cope with the fact that some nodes of

[2]As we will see later, the root of the last generated tree is not a decision node.

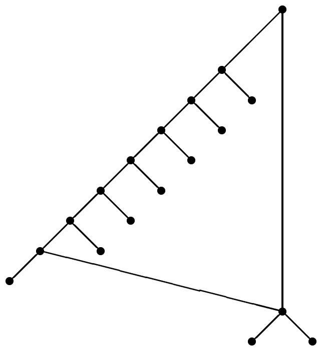


Figure 6: The DAG to the left has 8 decision nodes. When using the function *descend*, only one node gets an asterisk, i.e. 2^7 evaluations are generated. When using the labeling version, only 2 evaluations are generated: the first one evaluates the left son of the root first, the second one evaluates the right son first.

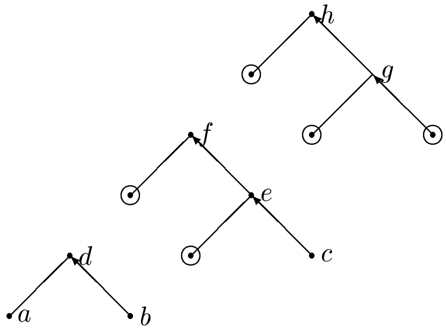


Figure 7: The example DAG is split into 3 subtrees by setting $\beta_1 = 0$, $\beta_2 = 0$, $\beta_4 = 0$. The newly introduced import nodes are marked with a circle. They are all non–permanent.

the trees must be held in a register until the last reference from any other tree is resolved. Such an algorithm is given in Section 6. Before applying the new labeling algorithm, we explicitly split the DAG in subtrees $T_1 = (V_1, E_1), \ldots, T_k = (V_k, E_k)$. We suppose that these subtrees must be evaluated in this order. The splitting procedure is described in detail in the next section. After the splitting, we introduce additional import nodes which establish the communication between the trees. The resulting trees to the second DAG in Figure 5 are given in Figure 7.

We present the labeling algorithm in Section 6 with the notion of import and export nodes: An *export node* of a tree T_i is a node which has to be left in a register because another tree $T_j (j > i)$ has a reference to v, i.e., T_j has an import node which corresponds to v. An *import node* of T_i is a leaf which is already in a register R because another tree $T_j (j < i)$ that has been evaluated earlier has left the corresponding export node in R. Therefore, an import node need not to be loaded in a register and does not appear again in the evaluation. For each import node, there exists a corresponding export node. Two import nodes $v_1 \neq v_2$ may have the same corresponding export node.

We distinguish two types of import nodes:

- A *permanent* input node v can be evaluated without being loaded in a register. v can*not* be removed from the register after the father of v is evaluated, because there is another import node of T_i or of another tree T_j that has the same corresponding export node as v and that has not been evaluated yet.

- A *non–permanent* input node v can also be evaluated without being loaded into a register. But the register that contains v can be *freed* after the father of v has been evaluated, because all other import nodes that have the same corresponding export node as v are already evaluated.[3]

Let the DAG nodes be $V = V_1 \cup \ldots \cup V_k$. We describe the import and export nodes by the following characteristic functions:

$$exp : V \to \{0,1\} \quad with \quad exp(v) = \begin{cases} 1 & \text{if } v \text{ is an export node} \\ 0 & \text{otherwise} \end{cases}$$

$$imp_p : V \to \{0,1\} \qquad with \qquad imp_p(v) = \begin{cases} 1 & \text{if } v \text{ is a permanent import node} \\ 0 & \text{otherwise} \end{cases}$$

$$imp_{np} : V \to \{0,1\} \qquad with \qquad imp_{np}(v) = \begin{cases} 1 & \text{if } v \text{ is a non–permanent import node} \\ 0 & \text{otherwise} \end{cases}$$

$$corr : V \to V \qquad with \qquad corr(v) = u, \text{ if } u \text{ is the corresponding export node to } v$$

The definition of import and export nodes implies

$$exp(v) + imp_p(v) + imp_{np}(v) \leq 1 \text{ for each } v \in V_i$$

5 Splitting the DAG into subtrees

We now describe how the DAGs are split into subtrees and how the import and export nodes are determined. We derive a recursive procedure *descend2* that is a modification of *descend*. *descend2* generates a number of evaluations for a given DAG G by splitting G into subtrees and evaluating the subtrees with a modified labeling scheme. Among the generated evaluations are all optimal evaluations. We first describe how the splitting is executed.

Let d be the number of decision nodes. The given DAG is split into at most d subtrees to generate an evaluation. After each split operation, export nodes are determined and corresponding import nodes are introduced as follows: Let $v = v_{pos}$ be a decision node with children v_1 and v_2 and let $G(v), G(v_1)$ and G_{rest} be defined as in the previous section. We consider the case that v_1 is evaluated before v_2 ($\beta_{pos} = 0$). Let $u \in V(v_1)$ be a node for which an edge $(u, w) \in E(v)$ with $w \in V_{rest}$ exists. Then u is an export node in $G(v_1)$. A new import node u' is added to G_{rest} by setting $V_{rest} = V_{rest} \cup \{u'\}$ and $E_{rest} = E_{rest} \cup \{(u', w)\}$. u' is the corresponding import node to u. If u has already been marked in $G(v_1)$ as export node, then u' is a permanent import node, because there is another reference to u (from another tree) that is evaluated later. Otherwise, u' is a non–permanent import node. If there are other edges $e_i = (u, w_i) \in E(v)$ with $i = 1, \ldots, k$ and $w_i \in V_{rest}$, then new edges $e'_i = (u', w_i)$ are added to E_{rest}. If $k \geq 1$, G_{rest} is not a tree and will be split later on.

A difficulty arises if $u = v_1$ is a leaf in $G(v)$ and there is a node $w \neq v$ in V_{rest} with $e = (v_1, w) \in E(v)$, see Figure 8. Then $G(v_1) = (\{v_1\}, \emptyset)$. w is a decision node that gets a $*$, v is a decision node for

[3] This partitioning of the import nodes is well defined, since the order of the T_i is fixed.

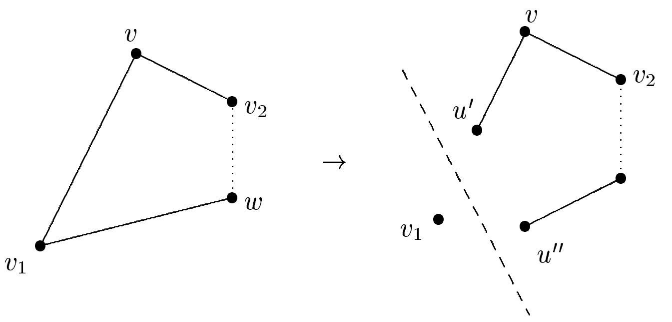


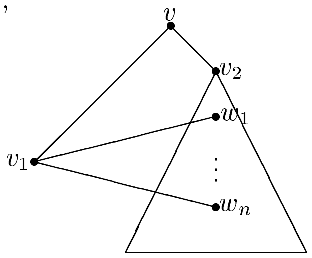
Figure 8: v has two children v_1 and v_2. v_1 is a leaf, w is a predecessor of v_2.

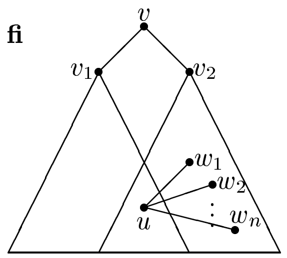
which the decision has been chosen. If G_{rest} contains no other decision node except v and w, we have the situation that G_{rest} is not split further, but is still a DAG after $e' = (u', w)$ and $e'' = (u', v)$ are added to E_{rest}. We solve the problem by introducing another node $u'' \neq u'$ by setting

$$V_{rest} = V_{rest} \cup \{u', u''\} \quad \text{and} \quad E_{rest} = E_{rest} \cup \{(u', v), (u'', w)\}$$

The corresponding export node to u'' is u in $G(v_1)$. So v_1 in $G(v_1)$ is the corresponding export node to two import nodes in G_{rest}. u'' is a permanent import node, because the value of the corresponding export node is still needed to evaluate v. If $u = v_1$ has not been marked as export node before, then u' is a non–permanent import node, because the register containing u can be freed after v is evaluated. One splitting step is executed by the following function *split_dag*:

function *split_dag*(**node** v, v_1, v_2, **dag** $G = (V, E)$) : **dag**;
// v is a decision node with children v_1 and v_2 //
u_1 = *new_node*();
$V = V \cup \{u_1\}; E = E \cup \{(u_1, v)\}$;
if $exp(v_1) == 0$ **then** $imp_{np}(u_1) = 1$ **else** $imp_p(u_1) = 1$; **fi**;
$exp(v_1) = 1; corr(u_1) = v_1$;
delete (v_1, v) from E;
for each edge $e = (v_1, w) \in E$ **do**
u_1 = *new_node*();
$V = V \cup \{u_1\}; E = E \cup \{(u_1, w)\}$;
$imp_p(u_1) = 1; corr(u_1) = v_1$;
delete e from E;
od;
Let $G(v) = (V(v), E(v))$ be the subDAG of G with root v,
let $G(v_1) = (V(v_1), E(v_1))$ be the subDAG of G with root v_1
build $G_{rest} = (V_{rest}, E_{rest})$ with $V_{rest} = V(v) - V(v_1)$, $E_{rest} = E(v) \cap (V_{rest} \times V_{rest})$;
for each $u \in V(v_1)$ **do**
if $\exists w_1, \ldots, w_n \in V_{rest}$ with $(u, w_i) \in E(v)$
then u_1 = *new_node*();
$V = V \cup \{u_1\}; E = E \cup \{(u_1, w_i), 1 \leq i \leq n\}$;
if $exp(u) == 0$ **then** $imp_{np}(u_1) = 1$ **else** $imp_p(u_1) = 1$; **fi**
$exp(u) = 1; corr(u_1) = u$;
delete (u, w_i) from E, $1 \leq i \leq n$;
fi;
od;
Let G_{ret} be the subDAG of G with root v;
return G_{ret};
end *split_dag*;





new_node is a function that returns a new node x and sets $exp(x), imp_p(x)$ and $imp_{np}(x)$ to 0. *split_dag* is called by the recursive procedure *descend2* that visits the decision nodes in reverse topological order (in the same way as *descend*). For each decision node v with children v_1 and v_2, *descend2* executes two possible split operations by using the complete subDAGs with roots v_1 and v_2. For each split operation, two subDAGs G_{left} and G_{right} are built. If one of these subDAGs is a tree, all decision nodes in the tree are marked with a $*$ so that no further split is executed for these decision nodes. The root of the tree is stored in *roots*. *roots* is a set of nodes that is empty at the beginning. If all decision nodes are computed, the trees that have their roots in *roots* are evaluated according to *ord* with the modified labeling scheme *labelfs2* presented in the next section.

To evaluate a DAG G, we start *descend2*$(\Theta, 1, G)$ where Θ is a bitvector with 0's at all positions. The decision nodes $v_1, \ldots, v_d$ are supposed to be sorted in reversed topological order (the root first).

```
function descend2 ( bitvector β, int pos, dag G )
while β_pos = * and pos ≤ d do pos = pos + 1 od;
if pos == d + 1
then ord = top_sort(roots);
        for i = 1 to d do labelfs2(ord(i)) od;
else β_pos = 0; G_1 = copy(G);
        mark exclusions of nodes v_j, j ∈ {pos + 1, ..., d} through lson(v_pos) with β_j = *;
        G_left = complete subDAG of G_1 with root lson(v_pos)
        if is_tree(G_left)
        then mark all decision nodes in G_left with a *; roots = roots ∪ {lson(v_pos)} fi;
        G_right = split_dag(v_pos, lson(v_pos), rson(v_pos), G_1);
        if is_tree(G_right)
        then mark all decision nodes in G_right with a *; roots = roots ∪ {v_pos} fi;
        descend2( β, pos + 1, G_1);
        β_pos = 1; G_2 = copy(G);
        mark exclusions of nodes v_j, j ∈ {pos + 1, ..., d} through rson(v_pos) with β_j = *;
        G_right = complete subDAG of G_2 with root rson(v_pos)
        if is_tree(G_right)
        then mark all decision nodes in G_right with a *; roots = roots ∪ {rson(v_pos)} fi;
        G_left = split_dag(v_pos, rson(v_pos), lson(v_pos), G_2);
        if is_tree(G_left)
        then mark all decision nodes in G_left with a *; roots = roots ∪ {v_pos} fi;
        descend2( β, pos + 1, G_2);
fi
end descend2;
```

---

top_sort is a function that sorts the nodes in its argument set in topological order according to the global DAG. If there are nodes v, v_1, v_2, w_1, w_2 where $v = v_{pos}$ is a decision node with $\beta_{pos} = 0$ and $(v_1, v), (v_2, v) \in E$ and w_1 is a predecessor of v_1 and w_2 is a predecessor of v_2, then $ord(w_1) < ord(w_2)$. If $\beta_{pos} = 1$, then $ord(w_2) < ord(w_1)$. *copy* is a function that yields a copy of the argument DAG. *is_tree(G)* returns true, if G is a tree.

By fixing the evaluation order of the trees, we also determine the type of the import nodes[4] and thus which import nodes return a free register after their evaluation. An import node is non–permanent

[4] If two import nodes v_1 and v_2 of the same tree T_i have the same corresponding export node, then the type is determined according to the evaluation order of T_i as described in the nect section. For the moment we suppose that both nodes are permanent.

if it is the last reference to the corresponding export node. Otherwise it is permanent: The register cannot be freed until the last referencing import node is computed.

6 Evaluating trees with import and export nodes

We suppose that we have a number of trees $T_1 = (V_1, E_1), \ldots, T_k = (V_k, E_k)$ with import and export nodes after the split operation executed by *descend2*. In this section, we describe how an optimal evaluation is generated for these trees. With the definitions from Section 4 we define the following two functions *occ* and *freed*:

$$occ : V \to \{0,1\} \qquad with \qquad occ(v) = \sum_{w \text{ is a proper predecessor of } v} exp(w)$$

counts the number of export nodes in the subtree $T(v)$ with root v (excluding v), i.e. the number of registers that remain occupied after $T(v)$ has been evaluated.

$$freed : V \to \{0,1\} \qquad with \qquad freed(v) = \sum_{w \text{ is a proper predecessor of } v} imp_{np}(w)$$

counts the number of import nodes of the second type in $T(v)$, i.e. the number of registers that are freed after $T(v)$ has been evaluated.

We now define for each node v of a tree $T_i (1 \leq i \leq k)$ a label *label*(v) which specifies the number of registers required to evaluate v as follows:

If v is a leaf, then $label(v) = 2 - 2 \cdot (imp_p(v) + imp_{np}(v))$. Let v be an inner node with two children v_1 and v_2. Let S_i be the subtree with root $v_i, i = 1, 2$. We have two possibilities to evaluate v, when we use contiguous evaluations: If we evaluate S_1 before S_2, we use

$$m_1 = \max(label(v_1), label(v_2) + occ(v_1) + 1 - freed(v_1))$$

registers, provided that v_1 (v_2) can be evaluated with label(v_1) (label(v_2)) registers. After S_1 is evaluated, we need $occ(v_1)$ registers to hold the export nodes of S_1 and one register to hold v_1. On the other hand, we free $freed(v_1)$ registers, when evaluating S_1. If we evaluate S_2 before S_1, we use

$$m_2 = \max(label(v_2), label(v_1) + occ(v_2) + 1 - freed(v_2))$$

registers. We suppose that the best evaluation order is chosen and set

$$label(v) = \min(m_1, m_2)$$

The following algorithm generates an evaluation for a labeled tree T with root v:

---

(1) **function** *labelfs2* (**node** v)
(2) **if** v is a leaf
(3) **then** **if** $imp_p(v) + imp_{np}(v) == 0$ **then** $reg(v) = new\_reg()$; **print** $(v, reg(v))$ **fi**
(4) **else if** v is an inner node with $lson(v) = v_1$ and $rson(v) = v_2$

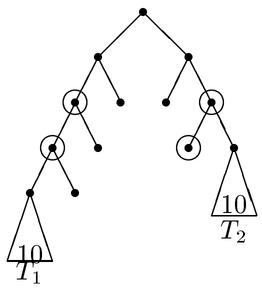


Figure 9: T_1 and T_2 are complete binary trees with height 9, so we need 10 registers to evaluate each of them. The export nodes are marked with a circle. If a contiguous evaluation is used, we must first evaluate the left or the right subtree of the root completely before starting the evaluation of the other subtree. We need at least 13 registers, because 3 registers are required to hold the export nodes and the root node of the other subtree. A non–contiguous evaluation can evaluate the tree with 11 registers by first evaluating T_1, then T_2 and then the rest of the tree.

```
        then if  label(v1) + occ(v2) − freed(v2) ≥ label(v2) + occ(v1) − freed(v1)
             then labelfs2(v1); labelfs2(v2);
             else labelfs2(v2); labelfs2(v1);
             fi
             reg(v) = new_reg(); print  (v, reg(v)); fi
        if  exp(v1) == 0 then  regfree(reg(v1)); fi
        if  exp(v2) == 0 then  regfree(reg(v2)); fi
fi;     end labelfs2;
```

Now we will prove that the call *labelfs2*(v) generates an optimal contiguous evaluation of v and uses *label*(v) registers. We prove this by two lemmata:

Lemma 7 *Let $T = (V, E)$ be a tree and $v \in V$ be an arbitrary inner node of T. labelfs2 generates an evaluation for v that uses label(v) registers.*

Lemma 8 *Let $T = (V, E)$ be a tree and $v \in V$ be an arbitrary inner node of T. label(v) is a lower bound for the minimal number of registers needed by a contiguous evaluation for v.*

Lemma 7 and Lemma 8 result in the following theorem:

Theorem 9 *The presented algorithm generates a contiguous evaluation that uses no more registers than any other contiguous evaluation.*

However, there may be a non–contiguous evaluation that needs fewer registers than the generated contiguous one. An example is given in Figure 9.

Until now, we have assumed that two different import nodes of a tree T_i have different corresponding export nodes. We now explain what has to be done if this is not true. Let $A = \{w_1, \ldots, w_n\} \subseteq V_i$ be a set of import nodes of T_i with the same corresponding export node that is stored in a register r. As described above we have set

$$imp_p(w_1) = \ldots = imp_p(w_n) = 1 \text{ and } imp_{np}(w_1) = \ldots = imp_{np}(w_n) = 0$$

But r can be freed, after the last node of A is evaluated. By choosing an appropriate node $w \in A$ to be evaluated last, T_i eventually can be evaluated with one register less than the label of the root specifies. We determine w by a top–down traversal of T_i. Let v be an inner node of T_i with children v_1 and v_2. Let S_j be the subtree with root v_j, $j = 1, 2$. If only one of S_1 and S_2 contains nodes of A, we descend to the root of this tree. If both S_1 and S_2 contain nodes of A, we examine, whether we can decrease the label value of v by choosing S_1 or S_2. Let be $a = \mathit{label}(v_1) + \mathit{occ}(v_2) - \mathit{freed}(v_2)$ and $b = \mathit{label}(v_2) + \mathit{occ}(v_1) - \mathit{freed}(v_1)$ If $a > b$, this can only be achieved by searching w in S_1. If $a < b$, this can only be achieved by searching w in S_2. If $a = b$, we cannot decrease the register need and can search in S_1 or S_2.
We repeat this process until we reach a leaf $w \in A$. We set $imp_p(w) = 0$, $imp_{np}(w) = 1$.

7 Experimental Results

We have implemented *descend* and *descend2* and have applied them to a great variety of randomly generated test DAGs with up to 150 nodes and to large DAGs taken from real application programs, see Tables 2 and 3. The random DAGs are generated by initializing a predefined number of nodes and by selecting a certain number of leaf nodes. Then, the children of inner nodes are selected randomly. The following observations can be made:

- *descend* reduces the number of different contiguous evaluations considerably.
- *descend2* often leads to a large additional improvement over *descend*, especially for DAGs where *descend* is not so successful in reducing the number of different contiguous evaluations.
- *descend2* works even better for DAGs from real application programs than for random DAGs.
- Only one of the considered DAGs with $n \leq 25$ nodes has a non–contiguous evaluation that uses fewer registers than the computed contiguous evaluation.[5]
- In almost all cases, the computational effort of *descend2* seems to be justified. This means that, *in practice*, an *optimal* contiguous evaluation (and thus, contiguous register allocation) can be computed in acceptable time even for large DAGs.

8 Register Allocation

After the evaluation order is determined, we can compute the register allocation.

[5] For a subDAG of MDG with $n = 24$ nodes, there is a non–contiguous that uses 6 registers. The computed contiguous evaluation takes 7 registers. The program to compute the non–contiguous evaluation has run for about 7 days, the corresponding program for the contiguous evaluation took less than 0.1 seconds. For DAGs with $n > 25$ nodes it is not possible to compute the best non–contiguous evaluation because of the runtime of the program that computes them is growing too fast.

n	d	N_{simple}	$N_{descend}$	$N_{descend2}$
24	12	4096	146	5
25	14	16384	1248	3
28	16	65536	748	22
27	17	131072	744	15
28	19	524288	630	32
33	21	2097152	1148	98
36	24	16777216	2677	312
38	26	67108864	6128	408
39	27	134217728	1280	358
42	29	536870912	6072	64
42	31	2^{31}	2454	152
46	34	2^{34}	4902	707
54	39	2^{39}	30456	592
56	43	2^{43}	21048	4421

n	d	N_{simple}	$N_{descend}$	$N_{descend2}$
20	14	16384	160	10
28	16	65536	784	8
29	18	262144	938	32
30	21	2097152	1040	64
37	23	8388608	13072	24
38	24	16777216	11924	56
45	27	134217728	100800	18
41	29	536870912	74016	364
41	31	2^{31}	3032	142
41	31	2^{31}	3128	180
44	33	2^{33}	40288	435
46	34	2^{34}	40244	1008
48	37	2^{37}	21488	1508
53	42	2^{42}	79872	3576

Table 2: Some examples from a test series for large random DAGs. The number of contiguous evaluations generated by the algorithms *simple*, *descend* and *descend2* are given for typical examples. The tests confirm the large improvements of *descend* and *descend2*.

Source	DAG	n	d	N_{simple}	$N_{descend}$	$N_{descend2}$	$T_{descend}$	$T_{descend2}$
LL 14	second loop	19	10	1024	432	18	0.1 sec.	<0.1 sec.
LL 20	inner loop	23	14	16384	992	6	0.2 sec.	<0.1 sec.
MDG	calc. $\cos(\theta), \sin(\theta), \ldots$	26	15	32768	192	96	<0.1 sec.	<0.1 sec.
MDG	calc. forces, first part	81	59	2^{59}	—	7168	—	13.6 sec.
	subDAG of this	65	45	2^{45}	—	532	—	0.9 sec.
	subDAG of this	52	35	2^{35}	284672	272	70.2 sec.	0.8 sec.
	subDAG of this	44	30	2^{30}	172032	72	42.9 sec.	0.3 sec.
	subDAG of this	24	12	4096	105	8	<0.1 sec.	<0.1 sec.
SPEC77	mult. FFT analysis	49	30	2^{30}	131072	32768	20.05 sec.	21.1 sec.

Table 3: Some measurements for DAGs taken from real programs (LL = Livermore Loop Kernels; MDG = Molecular Dynamics, and SPEC77 = atmospheric flow simulation, both from the Perfect Club Benchmark Suite). The table also gives the run times of the algorithms *descend* and *descend2*, implemented on a SUN SPARC station SLC. The tests show that for large DAGs *descend* is too slow. but the run times required by *descend2* remain really acceptable.

Our experiments have shown that the reordering of large basic blocks according to an optimal contiguous evaluation saves about 30% of the required registers on the average (see [11]).

We use a register allocation scheme called *first_free_reg* that allocates, for each node, the free register with the smallest number. Since a new register is allocated only if there is no other free register left, the generated register allocation is optimal and the number of allocated registers is equal to the register need of the evaluation.

The register allocation scheme uses a binary tree with the register 1,...,n as leaves. In each node, there is a flag *free* that indicates, whether the subtree of this node contains a free register. In order to allocate a free register, we walk along a path from the root to a free register by turning at each node to its leftmost son with a `TRUE` *free* flag. After switching the flag of the leaf found to `FALSE`, we traverse the path back to the root in order to update the flags. For each node on the path we set *free* to `FALSE` iff its two sons have *free* = `FALSE`.

If a register is marked free again, we must restore the *free* flags on the path from this register back to the root in the same way by setting for each node *free* to `TRUE` if at least one son has a true *free* flag. The run time is $O(\log n)$ for allocating or freeing a register, thus the total run time is $O(n \log n)$ for the evaluation of a DAG with n nodes.

The advantage of this allocation method is that the allocated registers usually have rather different access rates since, in general, registers with a low number are used more often than registers with a high number. That results in an allocation scheme that is well suited for spilling registers. If we have fewer registers available in the target machine than the evaluation requires, then we are forced to spill those registers with the least usage. The spill cost are at a minimum, if usage is distributed as unequally as possible over the allocated registers. The proposed heuristic *first_free_reg* fulfills this condition quite well. The actual spilling algorithm is described in [2] for basic blocks of vector instructions and may easily be adapted for the scalar case.

The general problem of computing an evaluation that is *optimal* with respect to spill cost seems to be a hard problem in terms of computational complexity, but that does not really matter in practice because a possible further gain in execution time compared to *first_free_reg* appears to be marginal for real DAGs.

9 Conclusions

We have presented two variants of the simple algorithm that evaluates only the tree nodes by a labeling algorithm and generates 2^d contiguous evaluations where d is the number of decision nodes of the DAG. The first variant is the exclusion of redundant decision nodes as performed by procedure *descend*. The second variant is the splitting of the DAG in subtrees (performed by *descend2*) and the evaluation of these by the modified labeling algorithm *labelfs2*. The experimental results in Section 7 confirm that this variant generates only a small number of contiguous evaluations, even for large DAGs. Among the generated evaluations are all evaluations with the least register need. Therefore, by using *descend2* we find the optimal contiguous evaluation in a reasonable time even for large DAGs. The dramatic reduction in evaluations generated makes *descend2* suitable for the use in optimizing compilers, especially for time–critical regions of the source program.

10 Acknowledgements

The authors would like to thank Prof. Dr. R. Wilhelm and Prof. Dr. W.J. Paul for their helpful support.

References

[1] *The Connection Machine CM–5 Technical Summary*, Thinking Machines Corporation, Cambridge, MA, 1991

[2] Keßler, C.W., Paul, W.J., Rauber, T.: *Scheduling Vector Straight Line Code on Vector Processors.* in: R. Giegerich, S.L. Graham (Ed.): Code Generation — Concepts, Tools, Techniques. Springer Workshops in Computing Series (WICS), 1992.

[3] Dongarra, J.J., Jinds, A.R.: *Unrolling Loops in Fortran*, Software Practice and Experience, **9**:3, 219–226 (1979)

[4] Fisher, J,: *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Transactions on Computers, **C–30**:7 (1981)

[5] Goodman J.R., Hsu Wei–Chung: *Code Scheduling and Register Allocation in Large Basic Blocks*, ACM International Conference on Supercomputing, 1988, 442–452

[6] Chaitin, G.J., Auslander M.A., Chandra A.K., Cocke J., Hopkins M.E., Markstein P.W.: *Register allocation via coloring.* Computer Languages Vol. **6**, 47–57 (1981)

[7] Chaitin, G.J.: *Register allocation & spilling via graph coloring.* ACM SIGPLAN Notices **17:6**, 201–207 (1982)

[8] Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools.* Addison–Wesley (1986)

[9] Sethi, R., Ullman, J.D.: *The generation of optimal code for arithmetic expressions.* Journal of the ACM, Vol. **17**, 715–728 (1970)

[10] Sethi, R.: *Complete register allocation problems.* SIAM Journal of Computing **4**, 226–248 (1975)

[11] Keßler, C.W., Paul, W.J., Rauber, T.: *A Randomized Heuristic Approach to Register Allocation.* Proceedings of PLILP'91 Third International Symposium on Programming Language Implementation and Logic Programming, Aug. 26–28, 1991, Passau, Germany. Springer LNCS Vol. 528, 195–206.

[12] Mehlhorn, K.: *Data Structures and Algorithms 2: Graph Algorithms and NP–Completeness.* (1984)

[13] Aho A.V., Johnson S.C.: *Optimal Code Generation for Expression Trees*, Journal of the ACM **23:3** (1976), pages 488-501

[14] Rauber, Thomas: *An Optimizing Compiler for Vector Processors.* Proc. ISMM International Conference on Parallel and Distributed Computing and Systems, New York 1990, Acta press, 97–103