Optimal Contiguous Expression DAG Evaluations

Christoph W. Keßler and Thomas Rauber*

Fachbereich 14 Informatik, Universität des Saarlandes, Postfach 151150,
D-66041 Saarbrücken, Germany. e-mail: {kessler,rauber}@cs.uni-sb.de

Abstract. Generating evaluations for expression DAGs with a minimal number of registers is NP–complete. 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 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.

1 Introduction

Register allocation is one of the most important problems in compiler optimization. Using fewer registers is essential 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 of vector registers (e.g., the CRAY 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 [5]). 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 them can hold more elements and the vector operation is split into fewer stripes. This saves initialization costs and results in a faster computation [10].

Scientific programs often contain large basic blocks. Large basic blocks can also result from the application of compiler techniques like loop unrolling [6] and trace scheduling [7]. Therefore, it is important to derive register allocation techniques that cope with large basic blocks [8].

Among the numerous register allocation schemes, register allocation and spilling via graph coloring [3] 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 as a set of directed acyclic graphs (DAGs). An

* supported by DFG, SFB 124, TP D4

algorithm to build the DAGs for a given basic block can be found in [2]. 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* [13] 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 [14].

In this paper, we only consider contiguous evaluations. Experiments with randomly generated DAGs and with DAGs that are derived from real programs show that for nearly all DAGs there exists a contiguous evaluation that is optimal. This leads to an algorithm [9] that computes an optimal contiguous evaluation for a given DAG in exponential time. — This paper improves this simple algorithm that performs a rather inefficient complete search by identifying and eliminating redundant evaluations. The second algorithm presented splits the given DAG into a series of trees with import and export nodes and evaluates the trees with a modified labeling scheme. 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.

2 Evaluating 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 (unary and binary arithmetic operations on registers[2], load and store instructions). Each input program can be partitioned into a number of basic blocks. On the machine instruction level, a *basic block* is a sequence of three–address–instructions that can only be entered via the first statement and only be left via the last statement. 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 [9].

Definition 1. Let $G = (V, E)$ be a DAG. A subDAG $S = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$ is called *complete subDAG* of G with root w, if $V' = \{v \in V : \exists \text{ path in } G \text{ from } v \text{ to } w\}$ and $E' = \{e \in E : e \text{ is an edge on a path in } G \text{ from a node } v \in V' \text{ to } w\}$.

Definition 2. An *evaluation* A of a DAG G is a permutation of the nodes in V such that for all inner nodes $v \in V$, v occurs in A behind all its children.

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, A is *consistent* because the precedence constraints are preserved. Thus, each topological order of G represents an evaluation, and vice versa.

Definition 3. 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

[2] To simplify the description, we assume w.l.o.g. that the registers participating in an operation are mutually different.

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[3], 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 [9]. 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.

Definition 4. [14] Let *num*: $\mathcal{R} \to \{0, 1, 2 \ldots\}$, $num(R_i) = i$ denote the numbering of registers. 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 child of w, and v appears in A between u and w, then $reg(u) \neq reg(v)$.

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

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 [14] 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 [9], we give the following definitions and prove the following lemmata:

Definition 5. Each leaf is a *tree node*. An inner node is a *tree node* iff all its children are tree nodes and none of them has outdegree > 1.

Definition 6. For a leaf v, $label(v) = 1$. For a unary node v, $label(v) = \max\{label(child(v)), 2\}$. For a binary node v, $label(v) = \max\{3, \max\{label(lchild(v)), label(rchild(v))\}+q\}$ where q=1 if $label(lchild(v)) = label(rchild(v))$ and q=0 otherwise.

Let *new_reg*() be a function that returns an available register and marks it 'busy'. Let *regfree*(*reg*) be a function that marks register *reg* 'free' again. The Labeling–algorithm *labelfs* of *Sethi* and *Ullman* [13] generates optimal evaluations for a tree with labels by first evaluating the child with the greater label value for each binary node.

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

[3] See [11] for a summary on topological sorting.

Thus, all binary nodes that have at least one predecessor with more than one parent are decision nodes. In a tree, there are no decision nodes.

Lemma 8. *For a tree T with one root and b binary nodes, there exist exactly 2^b different contiguous evaluations. For a DAG with one root and k binary nodes, there exist at most 2^k different contiguous evaluations.*

Lemma 9. *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 $\sum_{i=1}^{t} b_i = b$. If we fix an evaluation A_i for T_i, then 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 (supplied by *labelfs*) for the tree nodes of G is used:

```
algorithm simple_complete_search
Let v_1, ..., v_d be the decision nodes of a DAG G.
forall 2^d different β ∈ {0,1}^d do
        start dfs(root) with each β, such that for 1 ≤ i ≤ d
        if β_i = 0 in the call dfs(v_i),
        then the left child of v_i is evaluated first
        else the right child of v_i is evaluated first fi od
end simple_complete_search;
```

This algorithm has exponential running time, since a DAG with n nodes can have up to $d = n - 2$ decision nodes [9]. The running time of the algorithm can be reduced by exploiting the following observation (consider the example DAG in Fig. 1): Assume that the algorithm to generate a contiguous evaluation decides to evaluate the left child 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 child 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 child 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 *labelfs*. Fig. 1 shows the application of *descend* to an example DAG.

```
Let v_1, ..., v_d be the decision nodes in reverse topological order (the root comes first)
We call descend(Θ, 1) where Θ is a bitvector that contains d 0's.
function descend ( bitvector β, int pos )
while β_pos = * and pos < d do pos ← pos + 1 od
if pos ≥ d
then if β_pos = * then print β  // new evaluation found //
        else β_d = 0; print β;  β_d = 1; print β;  // 2 new evaluations found //  fi
else β_pos = 0;
        mark exclusions of nodes v_j, j ∈ {pos + 1, ..., d} through lchild(v_pos) by β_j ← *;
        descend( β, pos + 1);
        β_pos = 1;
        mark exclusions of nodes v_j, j ∈ {pos + 1, ..., d} through rchild(v_pos) by β_j ← *;
        descend( β, pos + 1);
fi end descend;
```

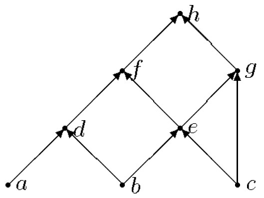

Fig. 1. Example DAG. For this DAG, algorithm *descend* executes the above evaluation steps. Only 7 instead of $2^5 = 32$ contiguous evaluations are generated.

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

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 child 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.The trees are evaluated in reverse of the order in which they are generated. For the example DAG of Fig. 1, there are 7 possible ways of carrying out the splitting. The splitting steps that correspond to evaluation A_1 from Fig. 1 are shown in Fig. 2.

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 Fig. 2, 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.

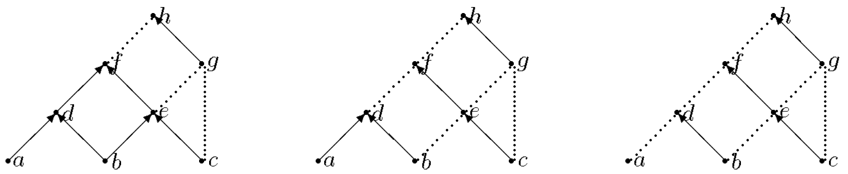

Fig. 2. The example DAG is split in three steps by setting $\beta_1 = 0$, $\beta_2 = 0$, $\beta_4 = 0$. The edges between the generated subtrees are shown as dotted lines.

Depending on the structure of the DAG, the number of generated evaluations may be reduced dramatically when splitting the DAG into trees. To evaluate the generated trees we need a modified labeling algorithm *labelfs2* that is able to cope with the fact that some nodes of the trees must be held in a register until the last reference from any other tree is resolved. 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 Fig. 2 are given in Fig. 3.

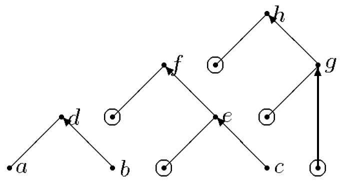

Fig. 3. The example DAG is split into 3 subtrees by setting $\beta_1 = 0$, $\beta_2 = 0$, $\beta_4 = 0$. The newly introduced import nodes are marked with a circle. They are all non–permanent.

We present *labelfs2* in Section 5 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 between permanent and non–permanent import nodes: A *permanent* import node v can be evaluated without being loaded in a register. v can*not* be removed from the register after the parent 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* import node v can also be evaluated without being loaded into a register. But the register that contains v can be *freed* after the parent of v has been evaluated, because all other import nodes that have the same corresponding export node as v are already evaluated.[4]

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:
$exp : V \longrightarrow \{0, 1\}$, $exp(v) = 1$ if v is an export node, and 0 otherwise;
$imp_p : V \longrightarrow \{0, 1\}$, $imp_p(v) = 1$ if v is a permanent import node, and 0 otherwise;
$imp_{np} : V \longrightarrow \{0, 1\}$, $imp_{np}(v) = 1$ if v is a non–permanent import node, and 0 otherwise;
$corr : V \longrightarrow V$, $corr(v) = u$ if u is the corresponding export node to v.
These definitions imply $exp(v) + imp_p(v) + imp_{np}(v) \leq 1$ for each $v \in V_i$.

The recursive procedure *descend2*, a modification of *descend*, generates a set of evaluations for a given DAG G by splitting G into subtrees and evaluating the subtrees with *labelfs2*. Among the generated evaluations are all optimal ones.

[4] This partitioning of the import nodes is well-defined as the order of the T_i is fixed.

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.

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, and
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**;
return the subDAG of G with root v; **end** *split_dag*;

new_node returns a new node x and sets $exp(x), imp_p(x)$ and $imp_{np}(x)$ to 0. *descend2* 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 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*, a set of nodes that is empty at the beginning. If all decision nodes are computed, the trees rooted in *roots* are evaluated according to *ord* with the modified labeling scheme *labelfs2*. — 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 $\beta_{pos} = *$ **and** $pos \leq d$ **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 $\beta_{pos} = 0$; $G_1 = \mathrm{copy}(G)$;
        mark exclusions of nodes v_j, $j \in \{pos + 1, \ldots, d\}$ through $lchild(v_{pos})$ with $\beta_j = *$;
        G_{left} = complete subDAG of G_1 with root $lchild(v_{pos})$
        if $is\_tree(G_{left})$
        then mark all decision nodes in G_{left} with a $*$; $roots = roots \cup \{lchild(v_{pos})\}$ **fi**;
        $G_{right} = split\_dag(v_{pos}, lchild(v_{pos}), rchild(v_{pos}), G_1)$;
        if $is\_tree(G_{right})$
        then mark all decision nodes in G_{right} with a $*$; $roots = roots \cup \{v_{pos}\}$ **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 rchild(v_pos) with β_j = *;
    G_right = complete subDAG of G_2 with root rchild(v_pos)
    if is_tree(G_right)
    then mark all decision nodes in G_right with a *; roots = roots ∪ {rchild(v_pos)} fi;
    G_left = split_dag(v_pos, rchild(v_pos), lchild(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;
```

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

5 Evaluating trees with import and export nodes

The split operation executed by *descend2* yields a series of trees $T_1 = (V_1, E_1)$, ..., $T_k = (V_k, E_k)$ with import and export nodes. Now we describe how an optimal evaluation is generated for these trees. We define two functions:

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

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.

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

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 $label(v) = 2 - 2 \cdot (imp_p(v) + imp_{np}(v))$. Otherwise, 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 by *labelfs2* and set $label(v) = \min(m_1, m_2)$.[5]

Theorem 10. *labelfs2 generates a contiguous evaluation that uses no more registers than any other contiguous evaluation.*

[5] 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 $imp_p(w_1) = \ldots = imp_p(w_n) = 1$ 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.

n	d	N_{simple}	$N_{descend}$	$N_{descend2}$
24	12	4096	146	5
25	14	16384	1248	3
27	17	131072	744	15
28	19	524288	630	32
33	21	2097152	1148	98
36	24	16777216	2677	312
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
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
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 1. 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*.

6 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 1 and 2. We observed:

- *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 one.[6]
- 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.

7 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

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 $a = label(v_1) + occ(v_2) - freed(v_2)$ and $b = label(v_2) + occ(v_1) - freed(v_1)$ 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 $imp_p(w) = 0$, $imp_{np}(w) = 1$.

[6] 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.

where d is the number of decision nodes of the DAG. The first variant excludes redundant decision nodes. The second variant splits the DAG into subtrees and evaluates these by a modified labeling algorithm. The experimental results confirm that this variant generates only a small number of contiguous evaluations; thus, this method finds the optimal contiguous evaluation in a reasonable time even for large DAGs. making it suitable for usage in optimizing compilers, especially for time–critical regions of the source program.

References

1. Aho, A.V., Johnson, S.C.: *Optimal Code Generation for Expression Trees*, J. ACM **23:3**, 1976, 488–501
2. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986
3. 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
4. Chaitin, G.J.: *Register allocation & spilling via graph coloring*. ACM SIGPLAN Notices **17:6**, 201–7, 1982
5. *The Connection Machine CM–5 Technical Summary*, Thinking Machines Corporation, Cambridge, MA, 1991
6. Dongarra, J.J., Jinds, A.R.: *Unrolling Loops in Fortran*, Software Practice and Experience, **9**:3, 219–26, 1979
7. Fisher, J,: *Trace Scheduling: A Technique for Global Microcode Compaction*, IEEE Transactions on Computers, **C–30**:7, 1981
8. Goodman J.R., Hsu Wei–Chung: *Code Scheduling and Register Allocation in Large Basic Blocks*, ACM Int. Conf. on Supercomputing, 1988, 442–52
9. Keßler, C.W., Paul, W.J., Rauber, T.: *A Randomized Heuristic Approach to Register Allocation*. Proc. of PLILP'91 3rd Int. Symp. on Programming Language Implem. and Logic Programming, Aug. 26–28, 1991. Springer LNCS 528, 195–206
10. 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
11. Mehlhorn, K.: *Data Structures and Algorithms 2*. Springer, 1984
12. Rauber, T.: *An Optimizing Compiler for Vector Processors*. Proc. ISMM Int. Conf. on Par. and Distr. Computing and Systems, New York 1990, Acta press, 97–103
13. Sethi, R., Ullman, J.D.: *The generation of optimal code for arithmetic expressions*. J. ACM, Vol. **17**, 715–28, 1970
14. Sethi, R.: *Complete register allocation problems*. SIAM J. Comput. **4**, 226–48, 1975

This article was processed using the LaTeX macro package with LLNCS style

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.
SPEC77	mult. FFT analysis	49	30	2^{30}	131072	32768	20.05 sec.	21.1 sec.

Table 2. 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 run times of the algorithms *descend* and *descend2*, implemented on a SUN SPARC station SLC, show that for large DAGs *descend* is too slow, but the run times required by *descend2* remain really acceptable.