Appeared in: Proceedings of PLILP 91 Third International Symposium
on Programming Language Implementation and Logic Programming,
Aug. 26–28, 1991, Passau, Germany.

Springer LNCS vol. 528, pp. 195–206.

# A randomized heuristic approach to register allocation

C. W. Keßler[*], W. J. Paul, T. Rauber[†]
Computer Science Department, University Saarbrücken, Germany

**Abstract**

We present a randomized algorithm to generate contiguous evaluations for expression DAGs representing basic blocks of straight line code with nearly minimal register need. This heuristic may be used to reorder the statements in a basic block before applying a global register allocation scheme like Graph Coloring. Experiments have shown that the new heuristic produces results which are about 30% better on the average than without reordering.

## 1 Introduction

Register allocation is one of the most important problems in compiler optimizations. Among the numerous register allocation schemes proposed register allocation and spilling via graph coloring is generally accepted to give good results. But register allocation via graph coloring has the disadvantage of using a fixed evaluation order within a given basic block. This is the evaluation order given by the source program. But often there exists an evaluation order for the basic block that uses less registers. By using this order the global register allocation generated via graph coloring could be improved. The aim of this article is to achieve such an improvement by improving the evaluation order within the basic blocks.
We can represent a basic block by a directed acyclic graph (DAG); see Fig. 1 for an example. An algorithm that constructs a DAG for a given basic block is given in [1]. For the evaluation of DAGs the following results are known:

(1) If the DAG is a tree, the well–known algorithm of *Sethi* and *Ullman* (see [8]) generates an optimal evaluation in linear time (optimal means: uses as few registers as possible).
(2) The problem of generating an optimal evaluation for a given DAG is NP–complete (see [9]).

To generate a good evaluation order for a DAG that is not a tree, we have to find an heuristic to do this task. We present such an heuristic in the following sections.
The new heuristic uses a mix of several simple evaluation strategies that also include a randomized evaluation selection. These simple evaluation strategies are applied concurrently and the best evaluation generated is selected.

[*]research partially funded by the Leibniz program of the DFG
[†]research partially funded by DFG, SFB 124

The idea behind this approach is that there exists no uniform heuristic that generates good evaluations for every possible DAG. But most of the DAGs encountered in real programs belong to one of a few simple classes. For each of these classes there exists a simple algorithm that generates good, often optimal evaluations. By running these simple algorithms "in parallel" and choosing the best result we obtain a heuristic that copes with most of the DAGs encountered in real programs.
In section 2 we introduce the basic formalism and two simple depth–first–search (*dfs*) variants as evaluation strategies. In section 3 we present the randomized evaluation strategy, in section 4 a variant of the Labeling Algorithm (see [8]). In section 5 we put the pieces together and discuss the performance improvement reached by the reordering with respect to the original basic block.

## 2 Evaluating DAGs

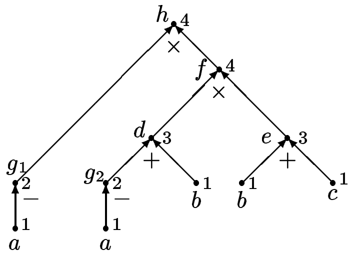
We assume that we are generating code for a single processor machine with general purpose registers $\mathcal{R} = \{R_0, R_1, R_2 \ldots\}$ and a countable sequence of memory locations. The arithmetic machine operations are three-address-instructions of the following types:

| | |
|---|---|
| $R_k \leftarrow R_i \; op \; R_j$ | binary operation, $op \in \{+, -, \times, \ldots\}$, |
| $R_k \leftarrow op \; R_i$ | unary operation, |
| $R_k \leftarrow$ `Load`$(a)$ | load register $k$ from the memory location $a$, or |
| `Store`$(a) \leftarrow R_k$ | store the contents of register $k$ into the memory location $a$, |

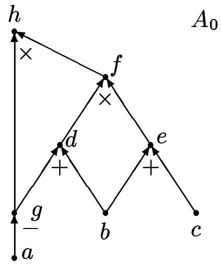
where $i \neq j \neq k \neq i$, $R_k, R_i, R_j \in \mathcal{R}$.
The following considerations are also applicable to the case $k = i$ or $k = j$. Our claim that the registers used by an operation must be mutually different makes the handling partially easier, but does not affect the validity of our results.

**Definition:** A *basic block* is a sequence of three–address–instructions that can only be entered via the first and only be left via the last statement.
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 let $n = |V|$ denote the number of nodes of the graph.
A sequence of vertices $v_0, v_1, \ldots v_k$ with $(v_i, v_{i+1}) \in E$ is called a *path of length $k$ from $v_0$ to $v_k$*. A *cycle* is a path from $v$ to $v$. A directed graph is called *acyclic* if it contains no cycle (of length $\geq 1$).
If $(w, v) \in E$, then $w$ is called *operand* or *son* of $v$; $v$ is called *result* or *father* of $w$, i. e. the edge is directed from the son to the father. A node which has no sons is a *leaf*, otherwise it is an *inner node*; in particular we call a node with two sons *binary* and a node with only one son *unary*. A node with no father is called *root* of $G$.
The *outdegree* $outdeg(w)$ of a node $w \in V$ 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.
Let $G = (V, E)$ be a directed graph with $n$ nodes. A mapping *ord:* $V \rightarrow \{1, 2, \ldots, n\}$ with

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



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



$A_0$ :

| | | | |
|---|---|---|---|
| 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: At first the front end generates an expression tree for $(-a) \times ((-a+b) \times (b+c))$ and evaluates it according to the Labeling algorithm of *Sethi/ Ullman* with minimal register need 4. (The labels are printed at the right hand side of the nodes). The optimizer recognizes common subexpressions, constructs the DAG $G$ and evaluates $G$ in the original order resulting in an evaluation $A_0$. This reduces the number of instructions and hence the completion time of the basic block. However, now 5 instead of 4 registers are required for the new basic block since $A_0$ is not optimal. The better evaluation on the right hand side obtained by reordering $A_0$ needs only 4 registers, and that is optimal.

is called a *topologic order* of the nodes of $G$. It is well–known that for a directed graph a topological order exists iff it is acyclic. (see e. g. [5]).

**Definition:** (*Evaluation of a DAG $G$*) An evaluation $A$ of $G$ 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 $v$ appears in $A$ after $v_i$, $i = 1, ..., k$.
This means 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 can be evaluated before all of his sons are. Thus each topological order of $G$ is an evaluation, and vice versa.

**Definitions:** 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 $w$ $\}$.

**Definition:** (*contiguous evaluation of a DAG $G$*)
(1) Let $G = (V, E)$ be a DAG with $V = \{v\}$, $E = \emptyset$, i. e. $v$ is the only node, root and

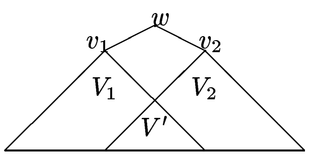


Figure 2: Example to the definition of a contiguous evaluation

leaf. Then $A = (v)$ is a contiguous evaluation of $G$.

(2) Let $G = (V, E)$ be a DAG with root $w$, let $w$ be an inner node with sons $v_1, \dots, v_p$ ($p \geq 1$). Let $S_1 = (V_1, E_1), \dots, S_p = (V_p, E_p)$ be the complete subDAGs of $G$ with the roots $v_1, \dots, v_p$. Let $\pi : \{1,...,p\} \to \{1,...,p\}$ be an arbitrary permutation. Furthermore let $A$ be an evaluation of $G$ in the form
$A = (A_{\pi(1)},\ A_{\pi(2)},\ \dots\ ,\ A_{\pi(p)},\ w)$,
where the following holds for all $j \in \{1, \dots, p\}$: $A_{\pi(j)}$ contains exactly the nodes of

$$\tilde{V}_{\pi(j)} = V_{\pi(j)} - \bigcup_{i=1}^{j-1} V_{\pi(i)}$$

and $A_j$ is a contiguous evaluation of $\tilde{G_j} = (\tilde{V_j}, E_j \cap (\tilde{V_j} \times \tilde{V_j}))$. Then $A$ is a contiguous evaluation of $G$.

(3) Those are all contiguous evaluations of $G$.

¿From now on we suppose $p \leq 2$ because no node has more than two sons in our machine model.

**Example:** Consider Fig. 2: Let $G = (V, E)$ be a DAG with root $w$. $w$ has the sons $v_1$ und $v_2$. Let $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ be the complete subDAGs of $G$ with the roots $v_1$ resp. $v_2$. Let $V' = V_1 \cap V_2$. Let $\pi(1) = 1$ and $\pi(2) = 2$. Then $\tilde{V_1} = V_1$ and $\tilde{V_2} = V_2 - V'$. Let $A_1$ be a contiguous evaluation of $\tilde{G_1}$, and let $A_2$ be a contiguous evaluation of $\tilde{G_2}$. Then $A = (A_1, A_2, w)$ is a contiguous evaluation of $G$.

The advantage of a contiguous evaluation is the fact that it can be generated by simple algorithms (variations of *depth–first–search* (*dfs*) ). In this paper we will restrict our attention to contiguous evaluations. This is already a heuristic because there are some DAGs for which a noncontiguous evaluation exists that uses less registers than every contiguous evaluation. However, in practice these cases seem to be rare. The smallest DAG of this kind we found so far has 14 nodes and is printed in Fig. 3.

**Definitions:** (cf. [9]) Let $num : \mathcal{R} \to \{0, 1, 2 \dots\}$, $num(R_i) = i$ 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 $reg(u) \neq reg(v)$.

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

is called the *register need* of the evaluation $A$. — An evaluation $A$ for a DAG $G$ is called *optimal* (w. r. to its register need $m = m(A)$) if for all evaluations $A'$ of $Gm(A') \geq m(A)$. In general there will exist several optimal evaluations for a given DAG. — In this paper we always use the word "optimal" with respect to the register need.

*Sethi* proved in 1975 ([9]) that the problem of computing an optimal evaluation for a given DAG is NP–complete. Assuming $\mathbf{P} \neq \mathbf{NP}$ we expect an algorithm with nonpolynomial run time. Unfortunately, this problem often occurs in compiler construction and should be solved fast. We will present a heuristic to produce fairly good evaluations in linear time.

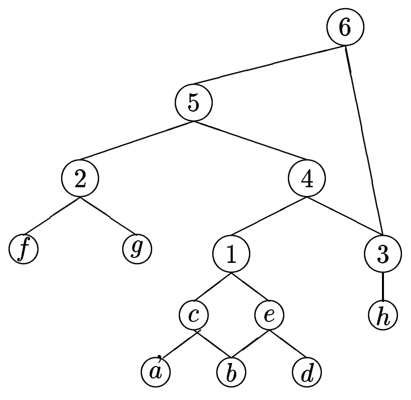


Figure 3: This DAG can be evaluated noncontiguously (in the order $a, b, c, d, e$, 1, $f, g$, 2, $h$, 3, 4, 5, 6) using 4 registers, less than each contiguous evaluation: An evaluation using 4 registers must evaluate the subDAG with root 1 first. A contiguous evaluation can do this only by evaluating the left son of node 6, the right son of node 5 and the left son of node 4 first. In order the 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 are required, thus the contiguous evaluation uses at least five registers altogether.

Let *new_reg*() be a function which returns an available register and marks it busy, and let *regfree*(*reg*) be a function which marks the register *reg* free again. (For more details see [3]).
We apply *dfs*–variations to evaluate the given DAG contiguously. The crucial point in *dfs* is the order in which the sons of the binary nodes are visited. If we always visit the left son first, we obtain *left first search* (*lfs*):

```
Let $G = (V, E)$ be a DAG.
Set $visited(v) =$ FALSE for all $v \in V$.
(1) function lfs(node v)
    (* evaluate the subDAG induced by node $v \in V$ *)
    begin
(2) $visited(v) \leftarrow$ TRUE;
(3) if $v$ is not a leaf
(4) then if not $visited(lson(v))$ then $lfs(lson(v))$ fi
(5)        if not $visited(rson(v))$ then $lfs(rson(v))$ fi
    fi
(6) $reg(v) \leftarrow new\_reg()$; print$(v, reg(v))$;
(7) if $v$ is not a leaf
(8) then if $lson(v)$ will not be used any more then $regfree(reg(lson(v)))$ fi
(9)        if $rson(v)$ will not be used any more then $regfree(reg(rson(v)))$ fi
    fi
    end lfs;
```

We get the information whether a node will be used later from a reference counter *ref*($v$) for each node $v \in V$, which is initialized with *outdeg*($v$) at the beginning of *lfs* and decremented when using $v$ as operand. If *ref*$(v) = 0$, $v$ will not be needed any more; the register containing the result of $v$ can be marked free (line 8/9).
We observe that each node $v$ will be held in a register from its evaluation point until the last reference to $v$ is reached. Thus, the register need $m$ results from the highest marked register number plus one, as in the definition above.

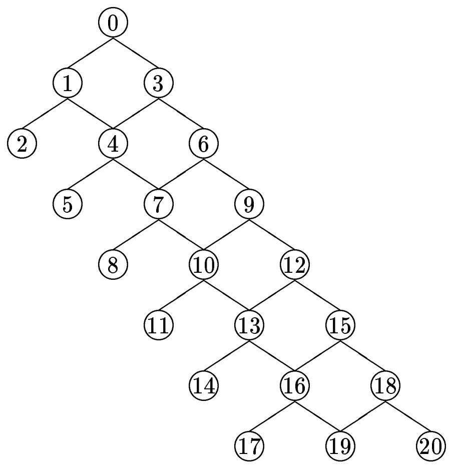


Figure 4: *lfs*(0) needs 9 registers, *rfs*(0) only 4.

***lfs*–evaluation of the DAG in Fig. 4:**

| node | 2 | 5 | 8 | 11 | 14 | 17 | 19 | 16 | 13 | 10 | 7 | 4 | 1 | 20 | 18 | 15 | 12 | 9 | 6 | 3 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| in reg. | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 5 | 4 | 3 | 2 | 1 | 0 | **8** | 6 | 7 | 5 | 4 | 3 | 2 |

***rfs*–evaluation of the DAG in Fig. 4:**

| node | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| in reg. | 0 | 1 | 2 | **3** | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 |

Table 1: Two examples for an evaluation of the DAG in Fig. 4

We obtain *rfs*() by swapping lines (4) and (5).
*new_reg*() and *regfree*() have constant run time. So *lfs*() has run time $O(n)$ since *lfs*($v$) is called exactly once for each node $v \in V$.
*lfs*() (or *rfs*()) might return a very bad evaluation when the DAG has a certain structure (see Fig. 4 and Tab. 1). For this reason we try to modify these algorithms in the next section.

## 3 Random first search

**Lemma 1** *A unary node has no influence on a contiguous evaluation generated by a dfs–variation.*

That is evident since for a unary node $u$ with son $v$, *dfs*($u$) has no other choice than to call *dfs*($v$).
**Definition:** (*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 $> 1$.
(3) Those are all tree–nodes.

**Definition:** (*label*)

(1) For any leaf $v$ is $label(v) = 1$.
(2) For a unary node $v$ is $label(v) = \max\{label(son(v)), 2\}$.
(3) For a binary node $v$ is
$label(v) = max\{3,\ \max\{label(lson(v)), label(rson(v))\} +\ q\}$
where $q = 1$ if $label(lson(v)) = label(rson(v))$, and 0 otherwise.

The Labeling–algorithm of *Sethi* and *Ullman* (see [8]) generates optimal evaluations for a tree with labels:

```
(1) function  labelfs(node v)
    (* generates an optimal evaluation for the subtree with root v *)
    begin
(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;
```

**Definition:** A *decision node* is a binary node which is not a tree–node.
It is clear that in a tree there are no decision nodes. In general, for a DAG let $d$ be the number of decision nodes and $b$ be the number of *binary* tree–nodes. Then $k = b + d$ denotes the number of all binary nodes of the DAG.

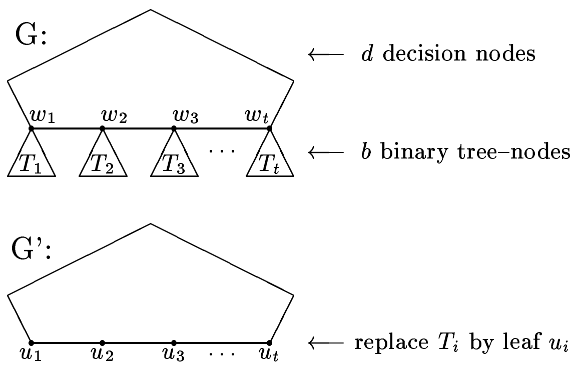
**Lemma 2** *For a tree $T$ with one root and $b$ binary nodes there exist exactly $2^b$ different contiguous evaluations.*

**Proof:** For each binary node there are 2 possibilities to select the order in which the subtrees are evaluated. □

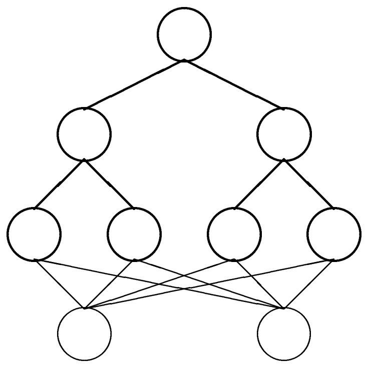
**Lemma 3** *For a DAG with one root and $k$ binary nodes there exist at most $2^k$ different contiguous evaluations.*

**Proof:** Each contiguous evaluation of a DAG with one root is characterized by the permutations $\pi$ applied at each node (see the definition of a contiguous evaluation). Thus, if the inner nodes $v_1, v_2, \ldots$ of the DAG have indegrees (the number of sons) $d_1, d_2, \ldots$, then there are at most $\prod_i (d_i!)$ contiguous evaluations. Since $d_i \leq 2$ for all $i$ the lemma follows. □

**Lemma 4** *Let $G$ be a DAG with $d$ decision nodes and $b$ binary tree–nodes which form $t$ (disjoint) subtrees $T_1, \ldots, T_t$; in $T_i$ there are $b_i$ binary tree–nodes, $i = 1 \ldots t$, 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 $G$.*


G:
$\longleftarrow$ $d$ decision nodes
$w_1$ $w_2$ $w_3$ $w_t$
$T_1$ $T_2$ $T_3$ $\cdots$ $T_t$
$\longleftarrow$ $b$ binary tree–nodes
G':
$\longleftarrow$ replace $T_i$ by leaf $u_i$
$u_1$ $u_2$ $u_3$ $\cdots$ $u_t$

To the proof of lemma 4



A DAG with $n-2$ decision nodes

Figure 5:

**Proof:** If we replace in $G$ each subtree $T_i$ with root $w_i$ (see Fig. 5) by a leaf $u_i$, $i = 1, \ldots, t$, we obtain a reduced DAG $G'$ with $d$ decision nodes and 0 binary tree–nodes. According to lemma 3 there are at most $2^d$ different contiguous evaluations $A'$ for $G'$. By replacing $u_i$ in $A'$ by the fixed evaluation $A_i$ we get for each contiguous evaluation $A'$ of $G'$ exactly one contiguous evaluation $A$ of $G$. $\square$

**Corollary 5** *If we evaluate all the tree–nodes in the DAG $G$ by labelfs(), there remain at most $2^d$ different contiguous evaluations for $G$.*

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. Now we enumerate the $2^d$ different $\beta \in \{0,1\}^d$ and start *dfs*(*root*) with each $\beta$ such that the following holds:
$\beta_i = 0$ iff in the call *dfs*($v_i$) the left son of $v_i$ should be evaluated first, $1 \leq i \leq d$.
By doing this we obtain *all* (up to $2^d$) possible contiguous evaluations for $G$ provided that we use a fixed contiguous evaluation for the tree–nodes of $G$.
Unfortunately, the algorithm induced by that still might have exponential run time since a DAG with $n$ nodes can have up to $d = n-2$ decision nodes (e. g. consider a binary tree with $n-2$ nodes; by adding two new nodes and $n-1$ edges as given in Fig. 5, we get a DAG with $n-2$ decision nodes).
Of course we do not want to invest exponential run time if we have a lot of decision nodes. Often a heuristic solution suffices[1]. This suggests to throw coins in order to generate several random bitvectors $\beta$ and to hope that at least one of the evaluations computed by this procedure has a register need close to the optimum.

**Algorithm** *randomfs*: We generate a fixed number $zv$ of random bitvectors (with $prob(\beta_i = 1) = 1/2$) and apply *dfs*() to each $\beta$. Among the computed evaluations we select one with the least register need.
The run time of *randomfs* is $O(zv \cdot n)$ according to the discussion of *lfs*.
Of course, if $zv \geq 2^d$ we have enough time to enumerate all possible $2^d$ bitvectors, i. e. we simulate a binary counter on the $\beta = (0...0000), (0...0001), \ldots, (1...1111)$. This procedure surely gives an optimal contiguous evaluation for $G$.

[1]In particular if we would like to evaluate vector DAGs (the nodes represent vectors of a certain length $L$) by vector processors, one or two additional registers do not matter very much, see [3]. The details will be given in [4].

The advantage of this method lies in the fact that the quality of the generated solution can be controlled by $zv$ ($zv$ may be passed as parameter to the compiler). That is why we are interested in the questions how good the computed evaluation is on the average and what size $zv$ should have in order to get sufficiently good results.
We want to illustrate this problem for a special example: Consider the DAG of Fig. 4. It is easy to see that *randomfs* can generate an optimal contiguous evaluation with a register need of 4 only if at the decision nodes 0, 3, 6, 9 and 12 the right son is always visited first. The probability for the subDAG with the root 15 being evaluated first is $p = 1/2^5 = 1/32$, about 3%. The probability to find at least one optimal evaluation among $zv$ possibilities is

$$1 - \left(\frac{31}{32}\right)^{zv}$$

in this example. If we wish that probability being over 90%, we conclude

$$zv \geq \frac{\log 0.1}{\log 31 - \log 32} \approx 72.5,$$

for a probability of 50% we need $zv \geq 22$, and so on.
Of course we might be satisfied if the generated evaluation would require five instead of four registers. For the average register need with given $zv$ we have found the following results for our example by experiments:

| *zv* | 0 | 1 | 3 | 5 | 6 | 7 | 8 | 10 | 12 | 15 | 18 | 20 | 30 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *reg* | 9 | 8.4 | 7.7 | 6.7 | 6.3 | 5.6 | 5.6 | 5.6 | 5.6 | 5.6 | 5.0 | 5.0 | 4.9 | 4.4 |

We can see that already for a relatively small size of $zv$, e.g. 10, a fairly good average register need is scored. Of course the improvement of the evaluation quality decreases for increasing $zv$, and the probability for the same bitvector being chosen twice certainly increases for increasing $zv$, e. g. for our example DAG with $d = 13$ decision nodes (thus 8192 possible bitvectors) the probability of at least one bitvector occuring several times is over 50% already for $zv = 107$.
Certainly these computations are limited to our example DAG above; a more general discussion of $zv$ may be a subject of further research. For the present we will choose $zv$ with respect to the run time of *randomfs*.

## 4 *labelfs* — another heuristic

It is possible to compute labels for all nodes of the DAG according to the formula of *Sethi/Ullman* for trees given above. In general a label–controlled evaluation of a DAG which is not a tree will not be optimal (otherwise **P** = **NP**). However, *labelfs* often scores fairly good results even for DAGs:

```
Start with $visited(v)$ = FALSE for all $v \in V$.
(1) function labelfs(node $v$)
      (* evaluate the subDAG induced by node $v$ *)
      begin
(2)   $visited(v) \leftarrow$ TRUE;
```

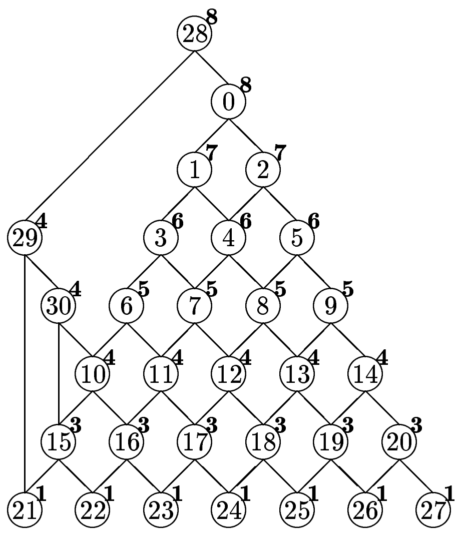


Figure 6: An extended 7–pyramid as counterexample to the assumption of *labelfs* giving always optimal evaluations: *labelfs*(28) allocates 11 registers, *rfs*(28) eight, *lfs*(28) nine registers. The label values are printed on the right top at each node.

```
(3)  if $v$ is not a leaf
(4)  then if $label(lson(v)) > label(rson(v))$
(5a)      then if not $visited(lson(v))$ then $labelfs(lson(v))$ fi;
(5b)           if not $visited(rson(v))$ then $labelfs(rson(v))$ fi
(6a)      else if not $visited(rson(v))$ then $labelfs(rson(v))$ fi;
(6b)           if not $visited(lson(v))$ then $labelfs(lson(v))$ fi
          fi
     fi
(7)  $reg(v) \leftarrow new\_reg()$; print$(v, reg(v))$;
(8)  if $v$ is not a leaf
(9)  then if $lson(v)$ will no longer be used then $regfree(reg(lson(v)))$ fi
(10)      if $rson(v)$ will no longer be used then $regfree(reg(rson(v)))$ fi
     fi
     end;
```

For the DAG of Fig. 4 *labelfs* gives an optimal evaluation (4 registers).
But we give a counterexample where *labelfs* does not the best (Fig. 6, Tab. 2).
So it seems sensible to unify all heuristics considered so far in a combination called V4 which applies all algorithms one after another and chooses the best evaluation generated.

## 5 V4 and its performance

V4 starts successively *lfs*, *rfs*, *labelfs* and $zv$ instances of *randomfs*.
For test compilations (set $zv = 0$) we can stop after *lfs* since for testing programs any correct evaluation is sufficient. For the final optimizing compilation of a program we choose a great value of $zv$. The run time is $O((zv + 3) \cdot n)$ according to the discussions above.
We use V4 to improve the register need of Graph Coloring. Graph Coloring (*Chaitin*, 1981, see [2]) allocates the registers by coloring the nodes of a so–called register inter-

***rfs*–evaluation of the counterexample:**

| node | 27 | 26 | 20 | 25 | 19 | 14 | 24 | 18 | 13 | 9 | 23 | 17 | 12 | 8 | 5 | 22 | 16 | 11 | 7 | 4 | 2 | 21 | 15 | 10 | 6 | 3 | 1 | 0 | 30 | 29 | 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| in reg. | 0 | 1 | 2 | 0 | 3 | 1 | 2 | 4 | 0 | 3 | 1 | 5 | 2 | 4 | 0 | 3 | 6 | 1 | 5 | 2 | 4 | 0 | **7** | 3 | 6 | 1 | 5 | 2 | 1 | 3 | 0 |

***labelfs*–evaluation of the counterexample:**

| node | 21 | 22 | 15 | 23 | 16 | 10 | 24 | 17 | 11 | 6 | 25 | 18 | 12 | 7 | 3 | 26 | 19 | 13 | 8 | 4 | 1 | 27 | 20 | 14 | 9 | 5 | 2 | 0 | 30 | 29 | 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| in reg. | 0 | 1 | 2 | 3 | 4 | 1 | 5 | 6 | 3 | 4 | 7 | 8 | 5 | 6 | 3 | 4 | 9 | 7 | 8 | 5 | 6 | 3 | **10** | 3 | 4 | 3 | 4 | 3 | 4 | 1 | 0 |

Table 2: Two examples for evaluating the DAG of Fig. 6: The decisive difference between the *labelfs*– and the *rfs*–evaluation in this case is the fact that *labelfs* visits the left son first if the label values of both sons are equal. Of course we may formulate *labelfs* such that with equal labels always the right son is preferred, but then the reflection of the DAG of Fig. 6 at the vertical axis would be a counterexample to *labelfs* too.

| random DAG No. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| number of nodes | 128 | 80 | 43 | 135 | 100 | 53 | 32 | 96 | 98 | 115 | 130 | 55 | 125 | 32 | 81 | 5 | 97 | 79 | 50 | 119 |
| reg. need with V4 | 27 | 16 | 8 | 27 | 21 | 11 | 7 | 23 | 22 | 25 | 28 | 11 | 25 | 8 | 21 | 3 | 18 | 18 | 12 | 25 |
| reg. need with GC | 31 | 20 | 15 | 37 | 23 | 16 | 12 | 32 | 26 | 32 | 35 | 16 | 36 | 14 | 26 | 4 | 25 | 25 | 15 | 34 |

Table 3: A series of tests with 20 randomly constructed DAGs: V4 always improved the register need of the original evaluation for the basic block (GC stands for "Graph Coloring without reordering by V4"); here we obtained the average ratio GC / V4 $\approx$ 1.38.

ference graph (RIG) which must be constructed from a fixed evaluation $A$ (this is the evaluation given in the original basic block). Two nodes (symbolic registers, here identical with the DAG nodes) interfere (thus they are connected in the RIG by an edge) if they are active simultaneously in $A$, i.e. they cannot be assigned to the same physical register (same color). The coloring can be computed by a linear–time heuristic (applied here) or via *backtracking* (where exponential run time is possible). The number of different colors (*chromatic number*) needed for $A$ corresponds to the register need $m$.
In order to show the advantages of the new heuristic we apply V4 with $zv = 10$ to randomly constructed DAGs with 30 to 150 nodes (average ca. 80), see Tab. 3. The result is surprisingly clear:
For the original evaluation of the basic block about 1/3 more registers are needed on the average than for the evaluation returned by V4. The improvement achieved by V4 might even be increased by choosing a greater $zv$.

This observation can only be explained by the fact that before the reordering we have *one* fixed evaluation $A_0$ (just the one which is given by the random construction of the test DAG), and in general this $A_0$ is noncontiguous. The probability for exactly this evaluation having a very low register need is rather small. On the other hand, V4 examines here $zv + 3 = 13$ (mostly) different evaluations, and only one of them must have a lower register need than $A_0$ to improve the result.

## 6 Final remarks

We are rather pleased with the results returned by V4, so we use it for the code optimizer in the implementation of a compiler for vector PASCAL which is being developed at our institute. For more details see [7]. The next step in that optimizer, the adaption of a computed evaluation of a vector DAG to a special vector processor, is described in [3] and will be presented in a later paper.

## References

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

[2] Chaitin, G.J. et al.: *Register allocation via coloring.* Computer Languages Vol. **6**, 47–57 (1981)

[3] Keßler, C.W.: *Code–Optimierung quasiskalarer vektorieller Grundblöcke für Vektorrechner.* Master thesis (1990), Universität Saarbrücken.

[4] Keßler, C.W., Paul, W.J., Rauber, T.: *Scheduling Vector Straight Line Code on Vector Processors.* Submitted to: First International Conference of the Austrian Center for Parallel Computation, Sept. 30 – Oct. 2, 1991, Salzburg (Austria).

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

[6] Paul, W.J., Tarjan, R.E., Celoni, J.R.: *Space bounds for a game on graphs.* Math. Systems Theory **10**, 239–251 (1977)

[7] 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

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

[9] Sethi, R.: *Complete register allocation problems.* SIAM J. Comput. **4**, 226–248 (1975)