Accepted for publication in *Computer Languages*.

Scheduling Expression DAGs for Minimal Register Need

Christoph W. Keßler

Universität Trier, FB 4 – Informatik, D-54286 Trier, Germany.
email: kessler@psi.uni-trier.de

Abstract

Generating schedules for expression DAGs that use a minimal number of registers is a classical NP–complete optimization problem. Up to now an exact solution could only be computed for small DAGs (with up to 20 nodes), using a trivial $O(n!)$ enumeration algorithm. We present a new algorithm with worst–case complexity $O(n2^{2n})$ and very good average behaviour. Applying a dynamic programming scheme and reordering techniques, our algorithm is able to defer the combinatorial explosion and to generate an *optimal* schedule not only for small DAGs but also for medium–sized ones with up to 50 nodes, a class that contains nearly all DAGs encountered in typical application programs. Experiments with randomly generated DAGs and large DAGs from real application programs confirm that the new algorithm generates optimal schedules quite fast. We extend our algorithm to cope with delay slots and multiple functional units, two common features of modern superscalar processors.

Key words: program optimization, basic block, expression DAG, instruction scheduling, register allocation, code generation

1 Introduction

Register allocation and instruction scheduling are the most important optimization problems in the code generation phase of a compiler.

Global register allocation by coloring a register interference graph [1, 2] is still the state of the art and has been refined by several heuristic techniques [3, 4, 5]. Spill code is automatically generated until the register need of the code does not exceed the number of physical registers. The graph coloring approach works on a given fixed schedule of the instructions. This typically results in a suboptimal usage of registers and increased register need. In earlier work [6] we have shown that for large randomly generated basic blocks the register need could be reduced by approximately 30 percent on the average by suitably reordering the instructions within a range permitted by the data dependences among them.

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 which usually have a small number 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 [7]). A vector operation is evaluated by splitting it into appropriate blocks and computing the blocks 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 operations can be split into fewer blocks. This saves startup times and results in a faster computation. On the other hand, if such a small number of vector registers has to be enforced by holding some intermediate results not in vector registers but in the main memory, the overall execution time increases again due to the additional memory references. By balancing this trade–off, an optimal number of vector registers can be found [8]. — Using fewer registers also pays off if less registers are to be saved and restored at procedure calls.

Basic blocks are typically rather small, up to 20 instructions. Nevertheless, scientific programs often contain also larger basic blocks, due to e.g. complex arithmetic expressions and array indexing. Larger basic blocks can also be produced by compiler techniques such as loop unrolling [9] and trace scheduling [10]. Therefore, it is important to derive register allocation techniques that cope with rather large basic blocks [11].

First, we consider the problem of reordering the instructions of a basic block whose dependence graph is a DAG, in order to just minimize the register need. For the simplest case where instructions are not delayed (i.e., take unit time to complete and do not overlap) and partial recomputations are not admitted, this is a classical NP–complete problem [12]. Some heuristic methods (e.g., [13, 14]) have been proposed. Only if the DAG is a tree, the labeling algorithm of Sethi and Ullman [15] yields an optimal solution; its run time is linear in the number of instructions; spilling of registers is not considered. [16] proposes a dynamic programming algorithm which computes an optimal schedule for *trees* in linear time that uses a given, fixed number of registers and contains spill code where required.

Modern microprocessors have sophisticated hardware features like pipelining and multiple functional units that exhibit further challenges for scheduling. Currently, register allocation is generally subordinated to scheduling because both optimization goals often conflict with each other. Typically there are enough registers available such that remarkable speedup can be obtained from scheduling improvements rather than from better register usage.

In the presence of delayed instructions or multiple functional units, the register requirements are either completely ignored for scheduling (e.g., [17]), or the DAG is assumed to be a tree [18, 19, 20], or heuristic techniques are applied [18, 21, 22, 23, 24]. In the presence of a loop, the completion time of a schedule may be improved by suitably overlapping subsequent iterations: Software pipelining (e.g., [25, 26, 27]) transforms a loop such that independent instructions from several consecutive iterations may be executed concurrently. Nevertheless, this technique tends to increase operand live ranges, and thus, the register need. [21] shows that separating register allocation and instruction scheduling produces inefficient code. But even for basic blocks, we are currently not aware of any method that integrates register allocation and scheduling that is guaranteed to find an *optimal* solution for register allocation or both register need and completion time when the dependence graph is a DAG. In fact, nothing better than $O(n!)$ [14] was known up to now, making the problem intractable for basic blocks of more than 15–20 instructions.

In this paper we present a new algorithm based on dynamic programming that reduces the worst–case bound to $O(n2^{2n})$. Tests for large basic blocks from real scientific programs and for suites of randomly generated basic blocks have shown that the algorithm makes the problem generally solvable for the first time also for medium–sized basic blocks with up to 40–50 instructions, a class that contains nearly all basic blocks encountered in real programs.

The time complexity can be further reduced because we can exploit massive parallelism present in the algorithm.

We further show how the method can be adapted to scheduling for delayed instructions and multiple functional units. As soon as such constraints on the time consumption of the schedule are added, the two goals of scheduling, namely register space optimality and time optimality, typically conflict with each other. With our framework it is possible to optimize either time or space consumption as the first priority, or one may specify a mixed optimization goal.

2 DAG schedules

We assume that we are generating code for a RISC–style single processor machine with general–purpose registers numbered consecutively (1, 2, 3, ...) and a countable sequence of memory locations. The arithmetic machine operations are three–address instructions (unary and binary arithmetic operations on registers, loading a register from memory, and storing a register's value to memory). For the first part of this paper, we assume that each instruction takes one unit of time to execute, i.e. the result of an instruction can be used by the next instruction in the next time slot. The extension of our work for delayed instructions is discussed in section 7.

Each input program can be partitioned into a number of basic blocks. On the machine instruction level, a *basic block* is a maximum–length sequence of three–address–instructions that can only be entered via the first instruction and only be left via the last one. The data dependencies in a basic block can be described by a *directed acyclic graph (DAG)*. The leaves of the DAG are the variables occurring as operands in the basic block; the inner nodes represent intermediate results. An example is given in Fig. 1.

Let $G = (V, E)$ be such a DAG with n nodes. A schedule of G is a total order of the nodes in V such that the partial order implied by the dependence arcs in E is preserved:

Definition 1 *A* schedule S *of a DAG* G *is a bijective mapping of the nodes in* V *to the set of time slots* $\{1, ..., n\}$ *such that for all inner nodes* $v \in V$ *with children* $v_1, ..., v_k$ *holds* $S(v) > S(v_i)$, $i = 1, ..., k$, *i.e.* v *is computed only after all its children* v_1,...,v_k *have been computed.*

This implies that a schedule is *complete*, i.e. all nodes are scheduled at some time slot between 1 and n, and the schedule contains *no recomputations*, i.e., each node of the DAG is scheduled only once. Moreover, the schedule is *consistent* because the precedence constraints are preserved. Thus, a schedule is just a topological order of G.

A *register allocation reg* for a given schedule maps the DAG nodes, in the order defined by that schedule, to a set of virtual machine registers such that the values computed by the DAG nodes are kept in registers from their definition up to their last use:

Definition 2 *A mapping reg:* $V \rightarrow \{1, 2, \ldots\}$ *is called a (consistent)* register allocation *for* S*, if for all nodes* $u, v, w \in V$ *the following holds: If* u *is a child of* w*, and* $S(u) < S(v) < S(w)$*, i.e.* v *is scheduled by* S *between* u *and* w*, then* $reg(w) \neq reg(u) \neq reg(v)$*.*

An *optimal register allocation* for a given schedule S uses the minimal number of registers, $m(S)$, that is still possible with that schedule:

Definition 3 *Let* R_S *denote the set of register allocations for a schedule* S*. Then,*

$$m(S) = \min_{reg \in R_S} \left\{ \max_{v} \; reg(v) \right\}$$

is called the register need *of the schedule* S*.*

Computing an optimal register allocation for a given schedule is straightforward. The technique of usage counts was first proposed in [28]: Let *new_reg*() be a function which returns an unoccupied register and marks it 'occupied'. Let *free_reg*(r) be a function that marks register r 'unoccupied' again. *new_reg*() is called each time a node is scheduled. *free_reg*($reg(u)$) is called immediately after[1] the last parent of node u has been scheduled. To determine this event, we keep a usage counter for each node u which counts the number of u's parent nodes already scheduled. Obviously, a register allocation for a given schedule can be computed with run time linear in the number of DAG edges, which is at most twice the number of instructions. — If one desires to keep register usage as *unequal* as possible (in order to obtain good candidates for a posteriori spilling of registers), sorting the unoccupied registers requires an additional factor of $O(\log n)$, as proposed in [14].

A schedule is called *space–optimal* (or just *optimal* for short, as long as time constraints are not considered) if an optimal register allocation for this schedule uses not more registers than any other schedule for the DAG:

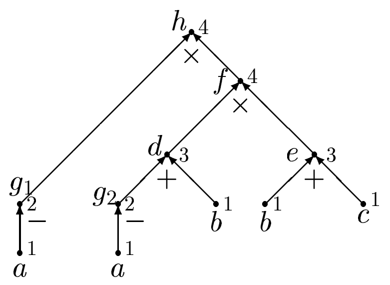
Definition 4 *A schedule* S *for a DAG* G *is called* (space–)optimal *if for all possible schedules* S' *of* G *holds* $m(S') \geq m(S)$*.*

In general, there will exist several optimal schedules for a given DAG. The problem of computing an optimal schedule for a DAG is NP–complete [12].

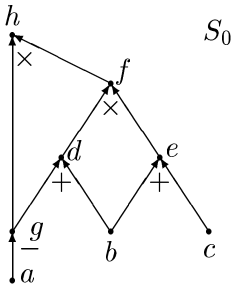
Let N_G denote the number of different schedules S for a DAG G with n nodes. N_G obviously depends on the structure of the DAG. Clearly N_G is less than $n!$, the number of permutations of the instructions. Our hope is that this is a very coarse upper bound and that N_G is not too high for small and medium–sized DAGs. Then we could just generate *all* N_G possible schedules for a given DAG G and select an optimal one among them.

Let us begin with a naive approach using an algorithm for topological sorting to generate these schedules. Topological sorting performs n steps. Each step selects one of the nodes, say v, with indegree zero, schedules it, and deletes all edges from v to its parents. Note that some of v's parents may get indegree 0, thus they become selectable in the following step.

[1] On most target architectures, the target register of an instruction may also be identical to one of the source registers (i.e., $reg(w) = reg(u)$ may be possible in Def. 2). Then, the register of u can be freed immediately *before* issuing the last parent of u. In the context of (strip–mined) vector instructions, however, the architecture may require registers to be mutually different.



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	$R_2 \leftarrow R_3 \times R_1$



S_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: Suppose the expression tree for $(-a) \times ((-a+b) \times (b+c))$ has been scheduled using the Labeling algorithm of *Sethi/Ullman* with minimal register need 4. — Common subexpression elimination results in a DAG G, reducing the number of instructions to be executed. If G is scheduled according to S_0, now 5 registers are required. The better schedule on the right obtained by reordering the instructions needs only 4 registers, which is optimal.

We use an array `S[1:n]` of nodes of G to store the current (partial) schedule S. Let z_0 be the set of all leaves of G, i.e. the nodes with initial indegree zero, and let $INDEG_0$ be an array containing the initial indegree of every node. The call $nce(z_0, INDEG_0, 1)$ of the recursive algorithm *nce* given in Figure 2 yields all possible schedules of G.

It is easy to see that the algorithm has a recursive nesting depth of n. For each recursive call, a copy of the current set z and of the current array $INDEG$ must be generated, so the required space is $O(n^2)$. The iterations of the **for all** loop over z could be executed in parallel. If this loop is executed in sequential, copying can be avoided; thus space $O(n)$ will suffice. The run time of *nce* depends on the structure of the given DAG G and of the number of possibilities for the choice of the next node for S, i.e. the cardinality of z in each step of the algorithm.

The total run time of *nce* is proportional to the total number of generated schedules. Thus, the run time is bounded by $O(n! \cdot n)$, because for each call we either update the $INDEG$ array *or* print a schedule.[2] The *real* number of schedules generated will be considerably smaller than the worst case bound $n!$ because only a small fraction of the permutations are valid schedules. Nevertheless, algorithm *nce* is impractical for DAGs with more than 15 nodes; our prototype implementation of *nce* already needs several days for DAGs of size around 20.

How can we reduce this complexity? One approach, following those in [13] and [14], restricts the set of schedules considered to *contiguous schedules*:

[2] We can easily eliminate a factor of n if the DAG has only one root r: This node r must always be the last node in a schedule, i.e. $S(r) = n$, terminating the recursion at nesting level $n-1$.

```
function nce (z, INDEG, pos)
 if pos = n + 1   // S[1:n] is a complete schedule of G
 then print S; compute register allocation for S and print it;
 else for all v ∈ z do   // select DAG node v to be scheduled next in current S:
            let P be the set of all parent nodes of v in G
            for all w ∈ P do INDEG'[w] ← INDEG[w] − 1 od;
            for all u ∉ P do INDEG'[u] ← INDEG[u] od;
            let P' be the subset of those nodes of P which have indegree 0;
            z' ← z − {v} ∪ P';
            S[pos] ← v;
            nce (z', INDEG', pos + 1);
      od
 fi
end nce
```

Figure 2: Algorithm *nce*.

Definition 5 *A schedule S 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 is a predecessor of v_2 but not of v_1, and $S(v_1) < S(v_2)$, then $S(v_1) < S(w)$.*

A contiguous schedule has thus the property that for each binary node v, all nodes of the complete subDAG of one of v's children are scheduled first, before any node belonging to the remaining subDAG rooted at v is scheduled. — While general schedules can be generated by variants of topological–sort, contiguous schedules are generated by variants of *depth–first search*.

Restriction to contiguous schedules degrades the worst-case complexity from $O(n!)$ to $O(2^n)$. While this is still exponential, for most practical cases the run time appears to be much smaller after clever pruning of the search space [14]. However, the heuristic may not find an optimal schedule because it might be non-contiguous. There are DAGs for which a general schedule exists that uses fewer registers than every contiguous schedule (see [14]). — Moreover, if delayed instructions or multiple functional units are to be considered (see sections 7 and 8), contiguous schedules tend to be inferior to noncontiguous ones, because at contiguous schedules it is much more typical that a parent node is scheduled directly after its child, which may cause a structural hazard.

3 The new algorithm

How can the exhaustive search performed by *nce* be improved? *nce* builds a decision tree, called the *selection tree*, $T = (Z', H')$. The set Z' of nodes, called the *selection tree nodes*, contains all the *instances* z' of the zero–indegree sets of DAG nodes that occur as the first parameter z in a call to *nce* during the execution of the algorithm. A directed edge $h = ({z'}_1, {z'}_2) \in H'$, called *selection tree edge*, connects two selection tree nodes ${z'}_1, {z'}_2 \in Z'$ iff there is a step in *nce* that directly generates ${z'}_2$ from ${z'}_1$ by selecting a DAG node v from ${z'}_1$ and scheduling it. The selection tree edge h is labeled $\ell(h) = v$ by the DAG node v selected. Clearly, the selection tree has one root that corresponds to the initial set z_0 of zero–indegree DAG nodes. All leaves of the selection tree are instances of the empty set of DAG nodes. The number of selection tree edges leaving a selection tree node $z \in Z'$ is equal

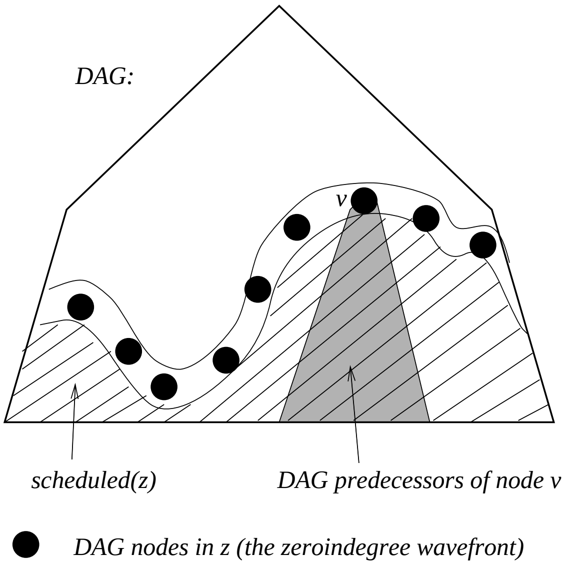


Figure 3: A snapshot of a DAG being processed by algorithm *nce*.

to the cardinality of z. According to the algorithm *nce*, the order of the labels $\ell((z_{j-1}, z_j))$ of the edges on any path $\pi = (z_0, ..., z_n)$, read from the root z_0 to a leaf z_n, corresponds one–to–one to a valid schedule $S_\pi : V \to \{1, ..., n\}$ of the DAG G with $S_\pi(\ell((z_{j-1}, z_j))) = j$, $j = 1, ..., n$; and for each schedule S of G there exists a unique corresponding path π_S in T from z_0 to some leaf.

Let *scheduled*(z) denote the set of DAG nodes that have already been scheduled when function *nce* is called with first parameter z. In particular, all DAG predecessors of the DAG nodes in z belong to *scheduled*(z), see Fig. 3. Furthermore, we denote by $L(z)$ the length of the path from the root z_0 to z, i.e. $L(z) = |\mathit{scheduled}(z)|$.

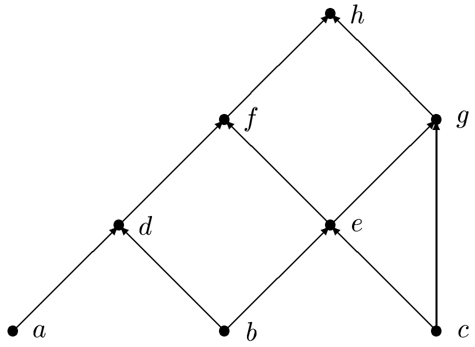
Also, we denote by *occ*(z) the number of nodes in *scheduled*(z) which are operands of nodes not in *scheduled*(z), i.e. the number of values which are currently to be held in registers. Note that *occ*(z) is the same for all schedules of *scheduled*(z).

As example, for the DAG in Figure 4 we have
$\mathit{scheduled}(\{c, d\}) = \{a, b\}$, $L(\{c, d\}) = 2$, $\mathit{occ}(\{c, d\}) = 2$;
$\mathit{scheduled}(\{a, b, c\}) = \emptyset$, $L(\{a, b, c\}) = 0$, $\mathit{occ}(\{a, b, c\}) = 0$;
$\mathit{scheduled}(\emptyset) = V = \{a, b, c, d, e, f, g, h\}$, $L(\emptyset) = n = 8$, $\mathit{occ}(\emptyset) = 0$.

We have observed that in such a selection tree, there occur generally several different instances $z' \in Z'$ of the same set z. We will modify the algorithm *nce* in such a way that these multiple occurrences of z are eliminated and replaced by a single node named z. Thus, the computation of the modified algorithm will result in a *selection DAG* rather than a selection tree. Let us first introduce some notation.

Definition 6 *A* selection DAG $D = (Z, H)$ *is a directed acyclic graph. The set* Z *of* selection nodes *contains all* different *sets* z *generated by nce. A* selection edge $h = (z_1, z_2) \in H$ *connects two selection nodes* $z_1, z_2 \in Z$ *iff there is a step in nce that directly generates* z_2 *from* z_1 *by selecting a DAG node* $v \in z_1$.

Note that a selection DAG has a single root, namely z_0, and a single sink, the empty set $\emptyset$.



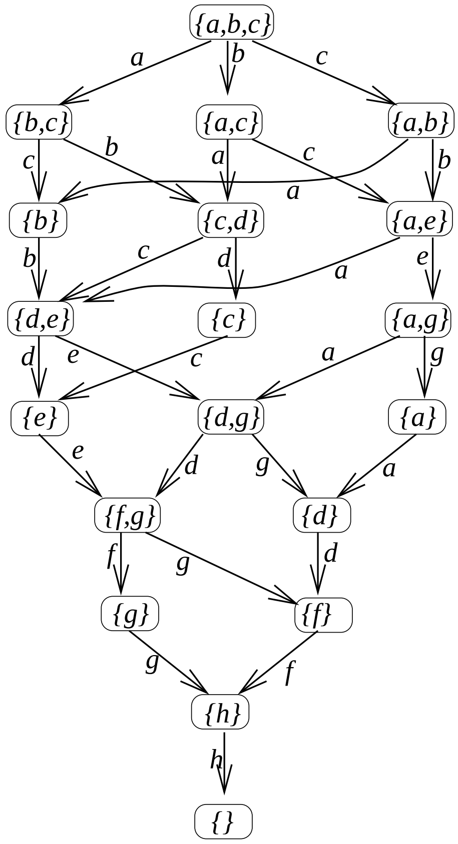


Figure 4: An example DAG and its selection DAG

Example: Fig. 4 shows an example DAG G and the selection DAG computed for it.

Each selection edge (z_1, z_2) is annotated with the DAG node $v = \ell((z_1, z_2))$ that is selected when going from z_1 to z_2.

For each $z \in Z$, we define by $G_z = (\mathit{scheduled}(z),\ E \cap (\mathit{scheduled}(z) \times \mathit{scheduled}(z)))$ the subgraph of G induced by $\mathit{scheduled}(z)$. Note that $G = G_\emptyset$.

Lemma 1 *Each path $\pi = (z_0, z_1, ..., z_{n-1}, z_n = z)$ of length n in the selection DAG $D = (Z, H)$ from the root z_0 to a selection node $z \in Z$ corresponds one-to-one to a schedule of $\mathit{scheduled}(z)$ which is given by the mapping $S : V \to \{1, ..., n\}$ with $S(\ell((z_{j-1}, z_j))) = j$ for $j = 1, ..., n$.*

Proof: by induction on the length n of π. Initially, the path $\pi = (z_0)$ consists only of the root z_0 and has length $n = 0$, with $\mathit{scheduled}(z_0) = \emptyset$. Clearly, π corresponds one-to-one to the empty schedule.

Now assume $n > 0$.

$\Rightarrow$: Let z be an inner node of D. z has some direct predecessor y in D. In y a DAG node $\ell((y, x)) = v \in y$ was selected by nce. Thus, $\mathit{scheduled}(y) = \mathit{scheduled}(x) - \{v\}$ and $L(y) = L(x) - 1 = n - 1$. Consider the prefix path $\pi' = (z_0, ..., z_{n-1} = y)$ of π to y. We apply the induction hypothesis to π', yielding a schedule $S_{\pi'}$ for $\mathit{scheduled}(y)$. We generate a schedule S_π for $\mathit{scheduled}(z)$ by just adding $S_\pi(v) = n$ to $S_{\pi'}$.

$\Leftarrow$: Consider a schedule S of n DAG nodes. Let $v = S^{-1}(n)$. We define $V_z = \{S^{-1}(1), ..., S^{-1}(n)\}$ and $V_y = V_z - \{v\}$. Let S_y be some arbitrary schedule for V_y. By the induction hypothesis there is a path $\pi_y = (z_0, ..., z_{n-1})$ in D to some selection node $z_{n-1} = y$ with $scheduled(y) = V_y$. Since v can be scheduled next (i.e. at position n), there must exist a selection edge (y, z) to some selection node z with $\ell((y, z)) = v$, $scheduled(z) = V_z = scheduled(y) \cup \{v\}$ and $L(z) = n$. Thus, $\pi_S = (z_0, ..., y, z)$ is the path corresponding to schedule S. $\square$

This means that subpaths corresponding to different partial schedules of the same subsets of DAG nodes end up in the same selection node, e.g. the paths inscripted a, b, c and b, c, a in Fig. 4 both end up in selection node $\{d, e\}$. If we read the edge labels along any path from the root to the sink of the selection DAG, we obtain a valid schedule of G.

Corollary 2 *All paths π in the selection DAG $D = (Z, H)$ from the root to a selection node $z \in Z$ have equal length $L(z)$.*

Corollary 3 *D is leveled, i.e. the nodes z of D can be arranged in $n+1$ levels $L_0, L_1, ..., L_n$ such that the selection edges in D connect only selection nodes in neighbored levels L_{i-1}, L_i, $i = 1, ..., n$.*

Proof: Set $i = L(z)$ in corollary 2. $\square$

Now we show that it suffices to keep *one* arbitrary optimal schedule of *scheduled*(z) for each selection node z:

Lemma 4 *Consider all direct predecessors $y_1, ..., y_q \in L_{N-1}$ of a selection node $z \in L_N$. Let $v_j = \ell((y_j, z))$, $j = 1, ..., q$, and let S_{y_j} denote an arbitrary optimal schedule for scheduled(y_j), $j = 1, ..., q$. We construct schedules S_j for scheduled(z) by just adding $S_j(v_j) = N$ to schedule S_{y_j}, $j = 1, ..., q$. Then, one of these schedules, $S_{j'}$, with $m(S_{j'}) \leq m(S_j)$ for all $j = 1, ..., q$, is an optimal schedule of scheduled(z).*

Proof: For each optimal schedule S of the DAG nodes $v_1, ..., v_N$ there is a unique path π_S in the selection DAG to z with *scheduled*$(z) = \{v_1, ..., v_N\}$. This path passes through an edge (y_j, z) for some j. S_j is generated by adding $S(v_j) = N$ to the existing schedule S_{y_j} (which is optimal for *scheduled*(y_j)). Then the register need of S_j will either remain the same or be increased by 1. If it remains the same, then $m(S_{y_j}) = m(S_j)$ and, since S is optimal, $m(S) = m(S_{y_j})$ (otherwise S_{y_j} would not have been optimal), i.e. S_j is also optimal. In the other case, $m(S_j) = m(S_{y_j} + 1)$. In this case, v_j must be located in a new register as all other registers are occupied, i.e. $occ(y_j) = m(S_{y_j})$. But this situation is independent of the order of the DAG nodes in *scheduled*(y_j), i.e., any optimal schedule for *scheduled*(y_j) would lead to the same result. Thus, as S is optimal, S_j is also optimal. $\square$

For each selection node z we further define:
Σ_z as the set of all schedules for G_z;
S_z as one of the schedules in Σ_z with minimal register need, and
$m_z = m(S_z)$ as this minimal register need. Note that $m_\emptyset$ is then the register need of an optimal schedule for G.

The definition of Σ_z and the selection of S_z from Σ_z is well–defined due to Lemma 1.

The new algorithm *ncc* (see Figure 5) constructs the selection DAG instead of the selection

```
function ncc ( DAG G with n nodes and set z_0 of leaves)
S_{z_0} = ∅; L_0 = {z_0}; L_1 = L_2 = ... = L_n = ∅;
for level L_i from L_0 to L_{n-1} do
    for all z ∈ L_i do
        for all v ∈ z do
            selecting v produces (as in nce) a selection node z'
             and a schedule S of scheduled(z') from S_z by adding S(v) = i + 1
            if  z' already exists in L_{i+1}
            then compare S and S_{z'}:
                  if  m(S) < m(S_{z'}) then  S_{z'} ← S else  forget S fi
            else  L_{i+1} ← L_{i+1} ∪ z'; S_{z'} ← S
            fi
        od
    od
od
return S_∅  (located in L_n)
end ncc
```

Figure 5: Algorithm *ncc*.

tree. *ncc* generates the selection nodes level–wise. At any creation of a new zero–indegree set z, it inspects whether the same DAG node set has already been created before. Because D is leveled (corollary 3), it needs only to inspect already generated selection nodes in the current level L_{i+1}. If such a selection node z' does not yet exist, a new selection node z' is created and $S_{z'}$ is initialized to the current schedule S. Otherwise, if the selection node z' already exists, *ncc* computes whether the current schedule S or the schedule $S_{z'}$ uses fewer registers, and sets $S_{z'}$ to the better schedule of these two.

Finally, each selection node z is annotated by an optimal schedule S_z of *scheduled*(z), and consequently $S_\emptyset$ is an optimal schedule for the DAG G.

From lemma 4 the correctness of *ncc* follows by induction:

Theorem 5 *Algorithm ncc computes an optimal schedule for the DAG G.*

Theorem 6 *The worst–case time complexity of algorithm ncc is $O(n2^{2n})$.*

Proof: The number of selection nodes is bound by the number of subsets of V which is 2^n. In each selection step we have to inspect one level of selection nodes which has less than 2^n nodes. Each comparison of two z sets costs at most $O(n)$. □

4 Improvement

We improve the algorithm *ncc* by the following modification: Instead of entering all selection nodes of the same level into the same set L_i, we subdivide L_i into sets L_i^0, L_i^1, ..., L_i^K where

K is some upper bound of the minimal number of registers required for G.[3] Now we store in L_i^k exactly those selection nodes z of L_i that have register need $m_z = k$. Now, we no longer have to look up all L_i^k, $k = 1, ..., K$ when looking for equality of z sets because of data dependencies:

Lemma 7 *The predecessors of a selection node in L_{i+1}^k are located in either L_i^{k-1} or L_i^k.*

Proof: Appending a DAG node v to an existing schedule S with register need $m(S)$ giving a new schedule S' yields $m(S) \leq m(S') \leq m(S)+1$ since we could be required to allocate a new register for v that was previously unused, and a composed schedule S' can never take less registers than any of its sub-schedules. □

This leads to the data dependencies in the selection graph illustrated in Fig. 7.

We use this theorem to free storage for levels as soon as they are not required any more.

The data dependencies impose a partial order of the selection steps. Hence, we can change our algorithm's order of constructing selection nodes as long as there is no conflict with these data dependencies. For instance, we can change the priority of the time slot axis over the register need axis, which corresponds, roughly speaking, just to a loop interchange. The resulting algorithm *ncv* (see Figure 6) can finish computing selection nodes as soon as the (first) selection node is entered into level L_n, because it represents a valid schedule with minimal register need — just what we want! Thus, expensive computation of later unused, inefficient partial schedules is deferred as far as possible — if we are lucky, to a point that needs not be considered any more because a solution has been found meanwhile.

5 Parallelization

There are several possibilities to exploit parallelism in *ncv*:

- exploiting the structure of the data dependencies among the lists L_i^k (see Fig. 7): all lists L_i^{j-i}, $i = 1, 2, ...$ (the lists on the jth "wavefront" diagonal) can be computed concurrently in step j, for $j = 1, ..., n + K$.
- further subdivision of a list L_i^k into sublists of roughly equal size (hashing) and expanding the selection nodes in these sublists concurrently. Note that this requires a parallel list data structure that guarantees that the same selection node is not erroneously inserted twice at the same time.
- parallelization of the lookup routine (according to previous item).
- parallelization of the expansion step itself.

Particularly, the second and third item promise the exploitation of massive parallelism.

We are currently implementing the parallel version of *ncv* on a massively parallel shared memory multiprocessor, the SB-PRAM [29], using the parallel programming language Fork95 [30].

---

[3] K could be a priori determined by using contiguous schedule techniques or by just choosing a random schedule of G. If storage is not too scarce, we can conservatively choose $K = n$.

```
function ncv ( DAG G with n nodes and set z_0 of leaves)
S_{z_0} = ∅; L_i^k = ∅ for all i and k; L_0^0 = {z_0};
for k = 0,...,K-1 do // outer loop: over space axis
    for level L_i^k from L_0^k to L_{n-1}^k do
        for all z ∈ L_i^k do
            for all v ∈ z do
                selecting v produces (as in nce) a selection node z'
                 and a schedule S of scheduled(z') from S_z by adding S(v) = i+1
                if z' already exists in L_{i+1}^k or L_{i+1}^{k+1}
                then compare S and S_{z'}:
                    if m(S) < m(S_{z'}) then S_{z'} ← S; move z' from L_{i+1}^{k+1} to L_{i+1}^k;
                    else forget S fi
                else L_{i+1}^{m(S_{z'})} ← L_{i+1}^{m(S_{z'})} ∪ z'; S_{z'} ← S
                fi
                if L_n^k ≠ ∅ then return the schedule S_z for some z ∈ L_n^k fi
            od
        od
    od
od
end ncv
```

Figure 6: Algorithm *ncv*.

6 Experimental Results

We have implemented *nce*, *ncc* and *ncv*, and have applied them to hundreds of randomly generated test DAGs and to larger DAGs taken from real application programs. Some of the experimental results are shown in Tables 1 and 2. All measurements were taken on a SUN SPARC-20.

The random DAGs are generated by initializing a predefined number of nodes and by selecting a certain number of them as leaf nodes. Then, the children of inner nodes are selected randomly. We observed:

- *nce* is not practical for DAGs with more than 20 nodes and often takes inacceptable time for DAGs with 15 to 20 nodes.
- *ncc* reduces the number of different contiguous schedules considerably. It is, roughly spoken, practical for DAGs with up to 40 nodes, sometimes also for larger DAGs, very depending on the DAG's structure. A timeout feature, controlled e.g. by a compiler option, should be provided for practical use that switches to a heuristic method if the deadline is exceeded..
- *ncc* is particularly suitable for "slim" DAGs because the zero–indegree sets z always remain small.
- *ncv* defers the combinatorial explosion to a later point of time but, of course, cannot always avoid it for larger and more complex DAGs. For some DAGs in the range of 41...50 nodes and for nearly all DAGs with more than 50 nodes *ncv* runs out of space and time.

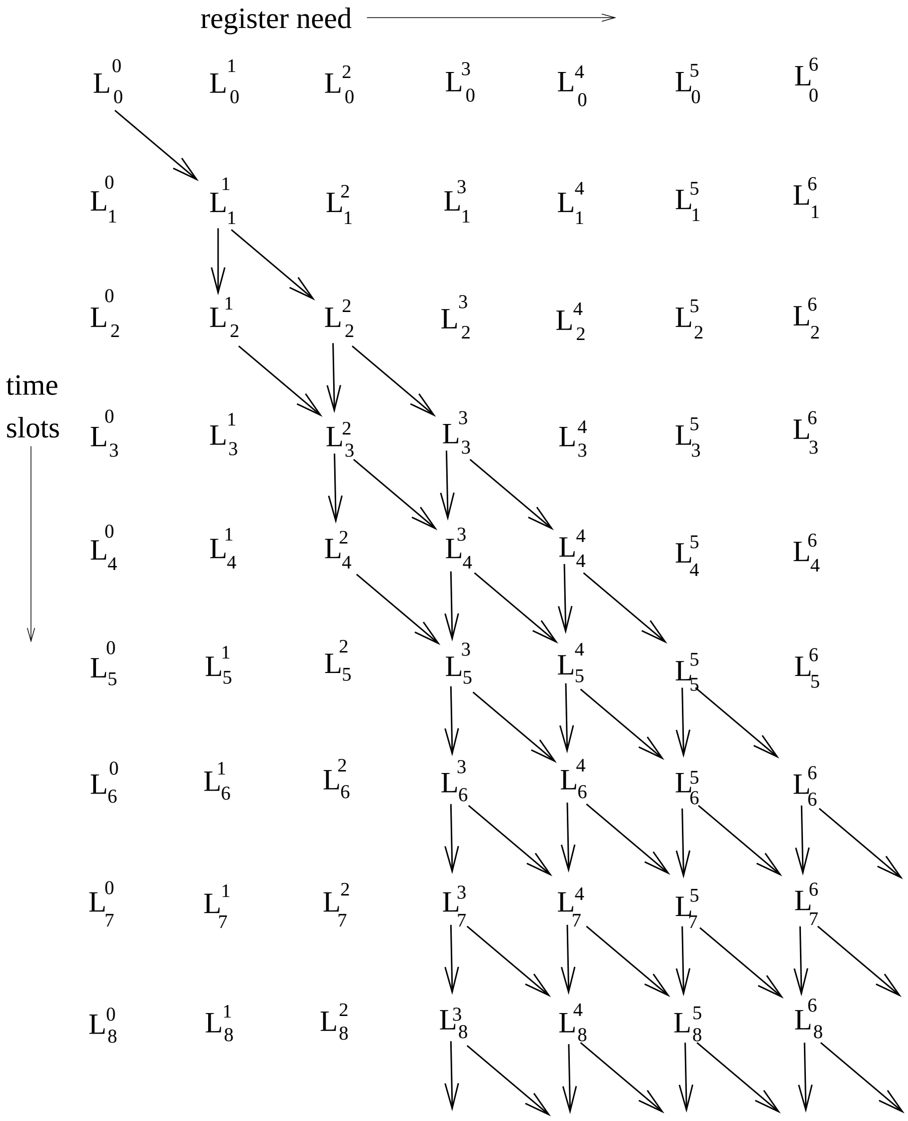


Figure 7: Data dependencies among the sets L_i^k of selection nodes

7 Filling delay slots

Up to now, we assumed that there are no delay slots to be filled since the result of an operation was available at the beginning of the next machine cycle. However, modern processors often have operations with one or more delay slots. These delay slots may be filled by subsequent operations that do not use the result of the delayed operation. If there are no such operations available, a `NOP` instruction has to be inserted to fill the delay slot. We will show how our technique can be easily extended to cope with delay slots.

For each DAG node v, let $delay(v)$ denote the number of delay slots required by the operation performed at v. For a given schedule S, we compute a mapping T of the DAG nodes to the set $\{1, 2, ...\}$ of physical time slots as given in Figure 8. The algorithm returns the time which is required to execute the basic block reordered according to schedule S.

n	T_{ncv}	$m_\emptyset$
24	0.8 sec	8
26	1.5 sec	7
26	0.8 sec	7
27	0.1 sec	8
28	1.4 sec	8
30	4.6 sec	8
31	22.5 sec	7
31	48.6 sec	7
32	17.5 sec	8
32	1:14.9 sec	8
33	4.6 sec	10
34	25.3 sec	7
34	54.3 sec	9
35	34.4 sec	10
36	14.9 sec	10

n	T_{ncv}	$m_\emptyset$
36	32.4 sec	10
37	3:00.8 sec	9
38	56.1 sec	10
38	2:38.8 sec	9
38	28.8 sec	9
38	7.0 sec	10
38	3:35.8 sec	9
39	10:01.7 sec	10
39	34.4 sec	9
39	43.1 sec	10
39	3:39.6 sec	9
39	1:44.8 sec	10
41	14:28.1 sec	9
41	1:48.6 sec	11
41	1:09.3 sec	10

n	T_{ncv}	$m_\emptyset$
42	23.4 sec	10
42	46.9 sec	11
43	43:30.6 sec	9
43	23:51.7 sec	9
44	3:44.6 sec	11
44	5:20.7 sec	12
44	18.9 sec	11
44	10:30.9 sec	10
45	58:01.7 sec	10
46	36:46.0 sec	12
46	55:59.8 sec	10
47	22:30.0 sec	11
49	1:09:37.2 sec	10
50	32:58,1 sec	12
51	19:01.2 sec	13

Table 1: *ncv* applied to some random DAGs. For all these DAGs, *nce* failed because of running out of space and time.

A schedule is called *time–optimal* if its execution on a given processor takes not more cycles than any other schedule for the DAG.

We are faced with the problem that we now get a second optimization goal: minimizing $time(S)$, i.e. the number of `NOP`s, in addition to minimizing the number $m(S)$ of registers. Comparing two schedules S_1 and S_2 becomes more difficult. Clearly, if $time(S_1) < time(S_2)$ and $m(S_1) \leq m(S_2)$ or if $time(S_1) \leq time(S_2)$ and $m(S_1) < m(S_2)$ we prefer S_1 over S_2. In the context of our previous algorithm, this means that S_1 would be kept in the lists of schedules, and S_2 would be thrown away. But what happens if S_1 and S_2 have both equal register need and equal number of `NOP`s, or, even worse, if $time(S_1) > time(S_2)$ but $m(S_1) < m(S_2)$ (or vice versa)? The only safe method is to keep both S_1 and S_2 in the lists for later expansion. This clearly increases the computational work.

But we apply the same trick as above to defer computationally expensive expansions to a later point in time. In addition to the previous partition of the selection nodes into lists L_i^k (cf. Fig. 7), we add a third dimension to the space of lists of execution nodes z, namely execution time $time(S_z)$. Specifically, a list $L_i^{k,d}$ contains the selection nodes z with $|scheduled(z)| = i$ and $m_z = k$ and $time(S_z) = d$. The structure of data dependencies among the $L_i^{k,d}$ results from

Lemma 8 *Let $maxdelay = \max_v delay(v)$. The predecessors of a selection node in $L_{i+1}^{k,d}$ are located in $L_i^{k-1,D}$ or $L_i^{k,D}$, with D ranging from $d - maxdelay$ to d.*

Proof: Evidence regarding k was already given in Lemma 7. Appending a DAG node v to an existing schedule S with execution time $time(S)$, giving a new schedule S', yields $time(S) \leq time(S') \leq time(S) + maxdelay$ since v can be placed safely $maxdelay$ time slots after the last time slot occupied by S. □

Another difficulty arises. For the previous algorithm, *ncv*, all optimal schedules, with respect to register need, were equivalent also when later used as sub-schedules in subsequent stages of the selection DAG (see Lemma 4). Thus, *ncv* needed to keep only one optimal schedule for each $z \in L_i^k$. But, as we outlined above, this is now no longer true because we have now

Source	DAG	n	T_{ncv}	$m_\emptyset$
LL 14	second loop	19	0.12 sec	4
LL 20	inner loop	23	0.48 sec	6
MDG	calculation of cos, sin,...	26	0.44 sec	7
SPEC77	spherical flow	27	0.93 sec	7
SPEC77	multiple FFT analysis	49	23:26.0 sec	7

Table 2: *ncv* applied to some DAGs taken from real programs (LL = Livermore Loop Kernels; MDG = molecular dynamics, and SPEC77 = atmospheric flow simulation; the last two are taken from the Perfect Club Benchmark Suite). *nce* failed for all DAGs because of running out of space and time.

more than one optimization direction. Thus, *ncv* may keep a "wrong" optimal schedule and may throw away the "right" ones.

This is illustrated in the following example (cf. Fig. 4): When selection node $\{d, e\}$ has been completed, *ncv* keeps only one optimal schedule for *scheduled* $(\{d, e\}) = \{a, b, c\}$, e.g. (b, c, a). The optimal solution found by *ncv* may follow a path through $\{d, e\}$. But if the final schedule should still require only 4 registers, *ncv must* continue with node d. But the schedule (b, c, a, d) enforces a `NOP` after issuing a if the DAG leaves have one delay slot (delayed `Load`). Thus the final schedule computed may be suboptimal w.r.t. execution time because the decision to keep only (b, c, a) at $\{d, e\}$ was made too early.

To repair this deficiency, we must keep *all* optimal schedules for the $L_i^{k,d}$ rather than only one. Clearly, this increases optimization time (and space) considerably, since all optimal schedules for *scheduled*(z) have to be taken into consideration now when expanding selection node z.

As we have two different optimization goals (number of registers and number of `NOP`s) we have to consider trade-offs between them. We can select one of them as primary goal of optimization and the other one as a secondary goal, which determines the order of stepping through the search space $\{L_i^{k,d}\}_{i,k,d}$. Or we formulate a mixed criterion (e.g., a weighted sum of k and d) that results in the algorithm traversing the search space in a wavefront manner. It is up to the user to formulate his favorite optimization goal.

We implemented algorithm *ncn*, the modification of *ncv* for delay slots, for the processor of the SB-PRAM [29] that has a delayed `Load` with one delay slot; all other operations perform in one cycle. We decided to minimize execution time as a primary goal and register need as a secondary goal. It is very easy now to backtrack the algorithm if one is willing to trade more `NOP`s for decreased register space. The results for several DAGs at different delay configurations are given in Tab. 3. It appears that the program runs quickly out of space for $n > 25$ in the case of delayed `Load`s of 1 delay slot, and for $n > 20$ if more delay slots are assumed. This is due to the unavoidable (but perhaps optimizable) replication of schedules described above. Clearly, our test implementation of *ncn* still wastes a lot of space. Further research has to be done here. In any case *ncn* appears to be feasible at least for small DAGs.

In general, optimization seems to be the faster, the more limitations and dependencies there are. Thus it is a straightforward idea to extend our method for processors with multiple functional units.

function *time* (**schedule** S of a **DAG** G with n nodes)
Initially we set $t = 1$. Then,
for the DAG nodes u_i in the order induced by S, i.e., $u_1 = S^{-1}(1),\ u_2 = S^{-1}(2),\ \ldots$
do if u_i has a left child l **then** $t = \max(t, T(l) + \mathit{delay}(l) + 1)$ **fi**
  if u_i has a right child r **then** $t = \max(t, T(r) + \mathit{delay}(r) + 1)$ **fi**
  $T(u_i) = t;\ \ t = t + 1;$ **od**
return $\max_i(T(u_i) + \mathit{delay}(u_i))$;

Figure 8: Algorithm *time* computes the time slots and the completion time of a schedule.

8 Multiple functional units

In the presence of multiple functional units it becomes possible to exploit fine grained parallelism. Consider a schedule S and two DAG nodes u, v such that v is scheduled directly after u. If u is not a DAG predecessor of v, u and v could be executed in parallel, provided that both are ready to be issued and functional units are available for the operations to be performed at u and v.

In contrast to VLIW machines, today's superscalar microprocessors do not rely on compilers to schedule operations to functional units. Scheduling is done on-line in hardware by a dispatcher unit. The dispatcher is able to issue several subsequent instructions from the instruction stream in the same clock cycle as long as there are (1) enough functional units available to execute them, and (2) there are no structural hazards such as control dependencies or data dependencies that would prohibit parallel execution These dependences are determined on-line by the dispatcher. If a subsequent instruction v is control or data dependent on a previous one, u, that has just been issued in this cycle, or if all suitable functional units are busy, the later instruction has to be delayed.

We see that this on–line scheduling performed by the dispatcher is indeed bounded by the rather small lookahead. Thus, it is a reasonable idea to reorder the instruction stream at compile time to facilitate the dispatcher's job. In order to optimize the schedule in this context, we need exact knowledge of the hardware features such as number, type, and speed of functional units, and the dispatcher's functionality which is then just simulated during optimization, as we already did for the pipeline behaviour in the previous section.

We can extend algorithm *ncn* for this case. The result is an optimal schedule S with issuing–time slots T and a mapping F of the DAG nodes to the functional units, such that the overall time required by S is minimized. T and F are computed as side–effects of the modified function *time* which just imitates the dispatcher's functionality.

9 Related Work

Vegdahl [31] applies a similar idea to instruction scheduling only. He first constructs the entire selection DAG, which is not leveled there, and then applies a shortest path algorithm. In contrast, we compute also partial costs and thus construct only those nodes of the selection DAG which may belong to an optimal path (schedule). His method directly generalizes to Software Pipelining. However, register requirements are not considered at all.

DAG	n	delay slots for `Loads`	avg. delay slots for compute	T_{ncn}	min. reg. need	#nops at min. reg. need	reg. need at min. #nops	min. #nops
LL 14	19	1	0	2.2 sec	4	4	5	0
		2	0.5	15.2 sec	4	11	6	1
LL 20	23	1	0	18.5 sec	6	0	6	0
random	17	1	0	0.7 sec	5	1	6	0
random	19	1	0	6.4 sec	5	2	6	0
random	19	1	0	14.8 sec	5	4	6	0
random	20	1	0	30.3 sec	6	2	7	0
random	21	1	0	11.8 sec	8	0	8	0
random	25	1	0	10.4 sec	7	0	7	0
random	15	2	0.5	0.1 sec	5	3	6	1
random	16	2	0.5	4.1 sec	5	3	6	1
random	16	2	0.5	1.1 sec	6	1	7	0
random	17	2	0.5	10.2 sec	5	7	6	1
random	19	2	0.5	62.5 sec	5	9	8	0
random	21	2	0.5	5.5 sec	6	2	7	0
random	22	2	0.5	45.3 sec	6	5	7	0

Table 3: Real and random DAGs, submitted to *ncn* with two target machine constellations: (1) delayed `Load` with one delay slot, and (2) delayed `Load` with 2 delay slots and delayed `Compute` with 0 or 1 delay slots (randomly chosen with probability 0.5). Columns 6 and 7 show register need and number of `NOP`s if register need is minimized; columns 8 and 9 show the results if the number of `NOP`s is minimized. For DAGs of more than 25 resp. 22 nodes our *ncn* implementation ran out of space. Interesting tradeoffs between register need and execution time occur mainly for small DAGs.

10 Conclusion

We have presented a new algorithm *ncv* that computes a schedule of a given expression DAG which uses a minimal number of registers. *ncv* makes it possible, for the first time, to solve this classical optimization problem also for medium-sized DAGs. Based upon the experimental results we conclude that *ncv* is feasible for usage in optimizing compiler passes; regarding DAGs of medium size, a timeout should be specified, depending on program size and the importance of this particular DAG for the overall execution time of the program. If *ncv* reaches this deadline without the solution being completed, we should switch to contiguous schedules, as proposed in [14]. For DAGs with more than 50 nodes, we should immediately restrict ourselves to contiguous schedules since it is not very probable that the solution can be completed before the deadline. But nearly all the DAGs occurring in real application programs are of small or medium size and can thus be processed by *ncv*.

We have proposed several ideas how to exploit massive parallelism in *ncv*.

We have shown how to adapt the algorithm to scheduling in the presence of delayed instructions and multiple functional units. First results for *ncn* show that an optimal solution for any optimization direction can be computed in a reasonable amount of time for small DAGs with up to 20–25 nodes.

Note also that for test compilations, any legal schedule is sufficient, which can be determined in linear time. The optimization algorithms presented here need only be run in the final, optimizing compilation of an application program. Moreover, they need only be applied to the most time–critical parts of a program, e.g. the bodies of innermost loops.

In principle it is also possible to integrate spilling of register values into our framework. Nevertheless the straightforward generalization of our method to computing optimal schedules including optimal spill code blows up the time and space requirements dramatically; further research is necessary here.

Future research will address techniques, other than trace scheduling [10], that extend our method beyond basic block boundaries. For instance, it may be interesting to combine our algorithms with software pipelining techniques in the presence of loops, maybe in a way similar to [31].

Acknowledgements

The author thanks Dr. Anton M. Ertl (Technical University of Vienna), Dr. Arno Formella (University of Saarbrücken), and Prof. Dr. Helmut Seidl (University of Trier) for valuable comments on an earlier version of this paper.

References

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

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

[3] D. Bernstein, M.C. Golumbic, Y. Mansour, R.Y. Pinter, D.Q. Goldin, H. Krawczyk, and I. Nahshon. Spill code minimization techniques for optimizing compilers. In *Proc. ACM SIGPLAN Programming Language Design and Implementation*, pages 258–263, 1989.

[4] Preston Briggs, Keith Cooper, Ken Kennedy, and Linda Torczon. Coloring heuristics for register allocation. In *Proc. ACM SIGPLAN Programming Language Design and Implementation*, 1989.

[5] Fred C. Chow and John L. Hennessy. Register allocation by priority-based coloring. *ACM SIGPLAN Notices*, 19(6):222–232, 1984.

[6] Christoph W. Keßler. Code–Optimierung quasiskalarer vektorieller Grundblöcke. Diploma thesis, University of Saarbrücken (Germany), 1990.

[7] Thinking Machines Corp. Connection Machine Model CM-5. Technical Summary. TMC, Cambridge, MA, Nov. 1992.

[8] Christoph W. Keßler, Wolfgang J. Paul, and Thomas Rauber. Scheduling Vector Straight Line Code on Vector Processors. In R. Giegerich and S.L. Graham, editors, *Code Generation – Concepts, Tools, Techniques*, pages 77–91. Springer Worksh. in Computing Series, 1992.

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

[10] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C–30(7):478–490, July 1981.

[11] James R. Goodman and Wei-Chung Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. ACM Int. Conf. on Supercomputing*, pages 442–452, July 1988.

[12] Ravi Sethi. Complete register allocation problems. *SIAM J. Comput.*, 4:226–248, 1975.

[13] Christoph W. Keßler, Wolfgang J. Paul, and Thomas Rauber. A Randomized Heuristic Approach to Register Allocation. In *Proc. 3rd Symp. on Programming Language Implementation and Logic Programming*, pages 195–206. Springer LNCS vol. 528, Aug. 1991.

[14] Christoph W. Keßler and Thomas Rauber. Generating optimal contiguous evaluations for expression DAGs. *Computer Languages*, 21(2):113–127, 1996.

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

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

[17] David Bernstein and Izidor Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Trans. on Progr. Lang. and Systems*, 11(1):57–67, Jan. 1989.

[18] David Bernstein, J.M. Jaffe, and Michael Rodeh. Scheduling arithmetic and load operations in parallel with no spilling. *SIAM J. Comput.*, 18:1098–1127, 1989.

[19] Todd A. Proebsting and Charles N. Fischer. Linear–time, optimal code scheduling for delayed–load architectures. In *Proc. ACM SIGPLAN Programming Language Design and Implementation*, pages 256–267, June 1991.

[20] R. Venugopal and Y.N. Srikant. Scheduling expression trees with reusable registers on delayed-load architectures. *Computer Languages*, 21(1):49–65, 1995.

[21] David G. Bradlee, Susan J. Eggers, and Robert R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, Apr. 1991.

[22] P.B. Gibbons and S.S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. SIGPLAN Symp. on Compiler Construction*, pages 11–16, July 1986.

[23] John Hennessy and Thomas Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[24] Rajeev Motwani, Krishna V. Palem, Vivek Sarkar, and Salem Reyen. Combining Register Allocation and Instruction Scheduling (Technical Summary). Technical Report TR 698, Courant Institute of Mathematical Sciences, New York, July 1995.

[25] C. Eisenbeis, S. Lelait, and B. Marmol. The meeting graph: a new model for loop cyclic register allocation. In *Proc. 5th Workshop on Compilers for Parallel Computers*, pages 503–516. Dept. of Computer Architecture, University of Malaga, Spain. Report No. UMA-DAC-95/09, June 28–30 1995.

[26] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. SIGPLAN Symp. on Compiler Construction*, pages 318–328, July 1988.

[27] J. Llosa, M. Valero, and E. Ayguade. Bidirectional scheduling to minimize register requirements. In *Proc. 5th Workshop on Compilers for Parallel Computers*, pages 534–554. Dept. of Computer Architecture, University of Malaga, Spain. Report No. UMA-DAC-95/09, June 28–30 1995.

[28] R.A. Freiburghouse. Register allocation via usage counts. *Comm. ACM*, 17(11), 1974.

[29] Jörg Keller, Wolfgang J. Paul, and Dieter Scheerer. Realization of PRAMs: Processor Design. In *Proc. WDAG94, 8th Int. Workshop on Distributed Algorithms, Springer Lecture Notes in Computer Science vol. 857*, pages 17–27, 1994.

[30] Christoph W. Keßler and Helmut Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *International Journal on Parallel Programming* **25**(1), pages 17–50, Feb. 1997.

[31] Steven R. Vegdahl. A Dynamic-Programming Technique for Compacting Loops. In *Proc. 25th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 180–188. Los Alamitos: IEEE Computer Society Press, 1992.

About the Author—CHRISTOPH W. KESSLER received a diploma degree in Computer Science in 1990 and a Ph.D. degree in computer science in 1994 at the University of Saarbrücken, Germany. Since 1995 Dr Keßler is an assistant professor at the Computer Science department of the University of Trier, Germany. His research interests include the design and implementation of parallel programming languages, parallel programming, automatic program comprehension, parallelizing compilers, and optimization problems in compilers such as instruction scheduling.