A Dynamic Programming Approach to Optimal Integrated Code Generation

Christoph Keßler*
PELAB
Department of Computer Science
Linköping University
S-58183 Linköping, Sweden
chrke@ida.liu.se

Andrzej Bednarski
PELAB
Department of Computer Science
Linköping University
S-58183 Linköping, Sweden
andbe@ida.liu.se

ABSTRACT

Phase-decoupled methods for code generation are the state of the art in compilers for standard processors but generally produce code of poor quality for irregular target architectures such as many DSPs. In that case, the generation of efficient code requires the simultaneous solution of the main subproblems instruction selection, instruction scheduling, and register allocation, as an integrated optimization problem.

In contrast to compilers for standard processors, code generation for DSPs can afford to spend much higher resources in time and space on optimizations. Today, most approaches to *optimal* code generation are based on integer linear programming, but these are either not integrated or not able to produce optimal solutions except for very small problem instances.

We report on research in progress on a novel method for fully integrated code generation that is based on dynamic programming. In particular, we introduce the concept of a *time profile*. We focus on the basic block level where the data dependences among the instructions form a DAG. Our algorithm aims at combining time-optimal scheduling with optimal instruction selection, given a limited number of general-purpose registers. An extension for irregular register sets, spilling of register contents, and intricate structural constraints on code compaction based on register usage is currently under development, as well as a generalization for global code generation.

A prototype implementation is operational, and we present first experimental results that show that our algorithm is practical also for medium-size problem instances. Our implementation is intended to become the core of a future, retargetable code generation system.

*Research partially supported by the CENIIT program of Linköping University, project 01.06. Previous research in this area was supported in 1999/2000 by the Deutsche Forschungsgemeinschaft (DFG).

Keywords

Instruction scheduling, register allocation, instruction selection, integrated code generation, dynamic programming, time profile

1. INTRODUCTION

The impressive advances in the processing speed of current microprocessors are caused not only by progress in higher integration of silicon components, but also by exploiting an increasing degree of instruction-level parallelism in programs, technically realized in the form of deeper pipelines, more functional units, and a higher instruction dispatch rate. Generating efficient code for such processors is largely the job of the programmer or the compiler back-end. Even though most superscalar processors can, within a very narrow window of a few subsequent instructions in the code, analyze data dependences at runtime, reorder instructions, or rename registers internally, efficiency still depends on a suitable code sequence.

Digital Signal Processors (DSPs) became, in the last decades, the processor of choice for embedded systems. The high volume of the embedded processor market demands for high performance at low cost. In order to achieve high code quality, developers still write applications in assembly language. This is time consuming, and maintenance and updating are difficult. Traditional compiler optimizations for high-level languages still produce poor code for DSPs and thus do not meet the requirements [40]. Unfortunately, compiler researchers focused for a long time on general purpose processors, while DSP irregularities, such as dual memory banks and non-homogeneous register sets, have been largely ignored in the traditional compiler optimization phases. On the other hand, code generation itself is a very complex task.

Code generation consists of several subproblems. The most important ones are instruction selection, instruction scheduling, and register allocation (see Figure 1).

Instruction selection maps the abstract instructions given in a higher-level intermediate representation (IR) of the input program to machine-specific instructions of the target processor. For each instruction and addressing mode, the cost of that instruction (in CPU cycles) can be specified. Hence, instruction selection amounts to a pattern matching problem with the goal of determining a minimum cost cover of the IR with machine instructions. For treelike IR formats and most target instruction sets this problem can be solved efficiently.

Instruction scheduling is the task of mapping each instruction of a program to a point (or set of points) of time when it is to be executed, and (in the presence of multiple functional units) to the func-

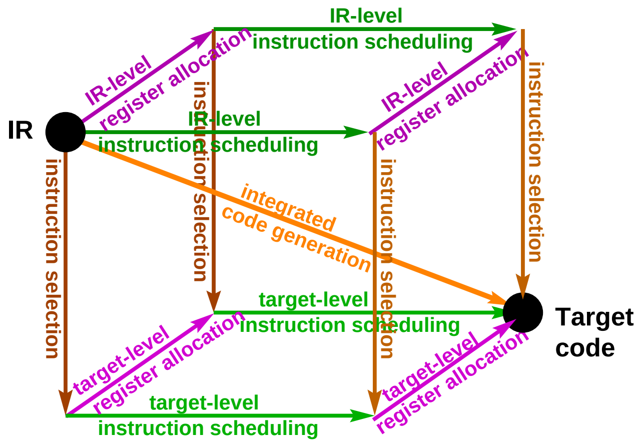


Figure 1: The task of generating target code from an intermediate program representation can be mainly decomposed into the interdependent subproblems instruction selection, instruction scheduling, and register allocation, which span a three-dimensional problem space. Phase-decoupled code generators proceed along the edges of this cube, while an integrated solution directly follows the diagonal, considering all subproblems simultaneously.

tional unit on which it is to be executed. For RISC and superscalar processors with dynamic instruction dispatch, it is sufficient if the schedule is given as a linear sequence of instructions, such that the information about the issue time slot and the functional unit can be inferred by simulating the dispatcher's behaviour. The goal is to minimize the execution time and avoiding severe constraints on register allocation.

Register allocation maps each value in a program that should reside in a register, thus also called a virtual register, to a physical register in the target processor, such that no value is overwritten before its last use. The goal is to use as few registers as possible. If there are not enough registers available from the compiler's point of view, the live ranges of the virtual registers must be modified, either by *coalescing*, that is, forcing multiple values to use the same register, or by *spilling* the register, that is, splitting the live range of a value by storing it back to the memory and reloading it later, such that a different value can reside in that register in the meantime. Using few registers is important, as generated spill code cannot be recognized as such and removed by the instruction dispatcher at runtime, even if there are internally enough free registers available. Also, spill code should be avoided especially for embedded processors because it increases power consumption.

During the last two decades there has been substantial progress in the development of new methods in code generation for scalar and instruction-level parallel processor architectures. New retargetable tools for instruction selection have appeared, such as `iburg`. New methods for fine-grain parallel loop scheduling have been developed, such as software pipelining. Global scheduling methods like trace scheduling, percolation scheduling, or region scheduling allow to move instructions across basic block boundaries. Also, techniques for speculative or predicated execution of conditional branches have been developed. Finally, high-level global code optimization techniques based on data flow frameworks, such as code motion, have been described.

On the other hand, most important problems in code generation have been found to be NP-complete. Hence, these problems are generally solved by heuristics. Instruction selection for basic blocks with a DAG-shaped data dependency structure is NP-complete [38]. Global register allocation is NP-complete, as it is isomorphic to coloring a live-range interference graph [10, 12] with a minimum number of colors [17]. Time-optimal local instruction scheduling for basic blocks is NP-complete for almost any non-trivial target architecture [1, 6, 17, 23, 29, 31, 34, 37] except for certain combinations of very simple target processor architectures and tree-shaped dependency structures [4, 5, 24, 28, 39]. Space-optimal local instruction scheduling for DAGs is NP-complete [9, 42], except for tree-shaped [43] or series-parallel [21] dependency structures.

For the general case of DAG-structured dependences, various algorithms for time-optimal local instruction scheduling have been proposed, based on integer linear programming (ILP) [45], branch-and-bound [22], and constraint logic programming [3]. Dynamic programming has been used for time-optimal [44] and space-optimal [26] local instruction scheduling.

In most compilers, the subproblems of code generation are treated separately in subsequent *phases* of the compiler backend. This is easier from a software engineering point of view, but often leads to suboptimal results because the strong interdependences between the subproblems are ignored [7]. For instance, early instruction scheduling determines the live ranges for a subsequent register allocator; where the number of physical registers is not sufficient, spill code must be inserted a-posteriori into the existing schedule, which may compromise the schedule's quality. Also, coalescing of virtual registers is not an option in that case. Conversely, early register allocation introduces additional ("false") data dependences and thus constrains the subsequent instruction scheduling phase. Moreover, there are interdependences between instruction scheduling and instruction selection. These interdependences can be quite intricate, caused by structural constraints on register usage and code compaction of the target architecture. Hence, the integration of these subproblems and solving them as a single optimization problem is a highly desirable goal, but unfortunately this increases the overall complexity of code generation considerably. Nevertheless, the user is in some cases willing to afford spending a significant amount of space and time in optimizing the code, such as in the final compilation of time-critical parts in application programs, or in code generation for DSPs.

There exist several heuristic approaches that aim at a better integration of instruction scheduling and register allocation [8, 16, 19, 27, 34]. However, there are only a few approaches that have the potential—given sufficient time and space resources—to compute an optimal solution to an integrated problem formulation, mostly combining local scheduling and register allocation [3, 25, 30, 32]. Some of these approaches are also able to partially integrate instruction selection problems, or to model certain hardware-structural constraints at a limited degree. Most of these approaches are based on ILP, which is again a NP-complete problem and can be solved optimally only for more or less trivial problem instances. Otherwise, integration must be abandoned and/or approximations and simplifications must be performed to obtain feasible optimization times, but then the method gives no guarantee how far away the reported solution is from the optimum.

Admittedly, ILP is a very general tool for solving scheduling problems that allows to model certain architectural constraints in a flexible way, which enhances the scope of retargetability of the system. However, we feel that integer linear programming should be rather considered as a fall-back method for the worst case where other algorithms are not available or not practical. Instead, we propose, for cases where the generality and the flexibility of the ILP approach is not really required, an integrative approach based on dynamic programing and problem-specific solution strategies, which can deliver optimal or at least good results with shorter optimization time. We have exemplified this for the case of space-optimal local scheduling in previous work [26], evidence for time-optimal local scheduling is given in this paper. We are now working on a generalization of our basic method to include more aggressive forms of instruction selection and to take also inhomogeneous register sets and intricate structural constraints on code compaction into account.

The remainder of this paper is organized as follows. Section 2 introduces basic notation, Section 3 explains the foundations of our dynamic programming method. Section 4 introduces time profiles and presents our algorithm for time-optimal scheduling. Section 5 presents the implementation and first results. Section 6 discusses possible extensions of our framework, Section 7 gives a systematic classification of related work, and Section 8 concludes.

2. BASIC TERMINOLOGY

In the following, we focus on scheduling basic blocks where the data dependences among the IR operations form a directed acyclic graph (DAG) $G = (V, E)$.

An *IR-schedule*, or simply *schedule*, of the basic block (DAG) is a bijective mapping $S : \{1, ..., n\} \mapsto V$ describing a linear sequence of the n IR operations in V that is compliant with the partial order defined by E, that is, $(u, v) \in E \Rightarrow S(u) < S(v)$. A *partial schedule* of G is a schedule of a subDAG $G' = (V', E \cap (V' \times V'))$ induced by a subset $V' \subseteq V$ where for each $v' \in V'$ holds that all predecessors of v' in G are also in V'. Note that a partial schedule of G can be extended to a (complete) schedule of G if it is prefixed to a schedule of the remaining DAG induced by $V - V'$.

We assume we are given a target processor with f functional units $U_1, ..., U_f$. The *unit occupation time* o_i of a functional unit U_i is the number of clock cycles that U_i is occupied with executing an instruction before a new instruction can be issued to U_i. The *latency* ℓ_i of a unit U_i is the number of clock cycles taken by an instruction on U_i before the result is available. We assume that $o_i \leq \ell_i$. The *issue width* ω is the maximum number of instructions that may be issued in the same clock cycle. For a single-issue processor, we have $\omega = 1$, while most superscalar processors are multi-issue architectures, that is, $\omega > 1$.

For simplicity, we consider for now only the case where each IR operation v is mapped to one of a set $\Psi(v)$ of equivalent, single target instructions: A *matching target instruction* $y \in \Psi(v)$ for a given IR operation v is a target processor instruction that performs the operation specified by v. An *instruction selection* Y for a DAG $G = (V, E)$ maps each IR operation $v \in V$ to a matching target instruction $y \in \Psi(v)$. Our framework can be easily generalized to the case of multiple IR operations matching a single target instruction (which is quite common for a low-level IR) as discussed in Section 6, while the case of a single IR operation corresponding to multiple target instructions requires either lowering the IR or scheduling on the target level only, rather than on the IR level (cf. Figure 1). We also assume that each target instruction y is executable by exactly one type of functional unit.

A *target-schedule* is a mapping s of the time slots in $(U_1, ..., U_f) \times \mathbb{N}_0$ to instructions such that $s_{i,j}$ denotes the instruction starting execution on unit U_i at time slot j. Where no instruction is started on U_i at time slot j, $s_{i,j}$ is defined as NOP. If an instruction $s_{i,j}$ produces a value that is used by an instruction $s_{i',j'}$, it must hold $j' \geq j + \ell_i$. Also, it must hold $j' \geq j'' + o_i$ where $s_{i',j''}$ is the latest instruction issued to $U_{i'}$ before $s_{i',j'}$. Finally, it must hold $|\{s_{i'',j'} \neq \texttt{NOP}, 1 \leq i'' \leq f\}| \leq \omega$.

For a given IR-schedule S and a given instruction selection Y, an optimal target-schedule s can be determined in linear time by a greedy method that just imitates the behavior of the target processor's instruction dispatcher when exposed to the instruction sequence given by $Y(S)$.

The *execution time* $\tau(s)$ of a target-schedule s is the number of clock cycles required for executing s, that is,

$$\tau(s) = \max_{i,j}\{j + \ell_i : \ s_{i,j} \neq \texttt{NOP}\}.$$

A target schedule s is *time-optimal* if it takes not more time that any other target schedule for the DAG.

A *register allocation* for a given target-schedule s of a DAG is a mapping r from the scheduled instructions $s_{i,j}$ to physical registers such that the value computed by $s_{i,j}$ resides in a register $r(s_{i,j})$ from time slot j and is not overwritten before its last use. (Spilling will be considered later.) For a particular register allocation, its register need is defined as the maximum number of registers that are in use at the same time. A register allocation r is optimal for a given target schedule s if its register need is not higher than that of any other register allocation r' for s. That register need is referred to as the *register need* of s. An optimal register allocation for a given target-schedule can be computed in linear time in a straightforward way [15].

A target-schedule is *space-optimal* if it uses no more registers that any other possible target-schedule of the DAG. For single-issue architectures with unit-time latencies, IR-schedules and target-schedules are more or less the same, hence the register allocation can be determined immediately from the IR schedule, such that space-optimality of a target-schedule also holds for the corresponding IR-schedule [26]. In all other cases, the register allocation depends on the target-schedule.

3. IR-LEVEL SCHEDULING AND DYNAMIC PROGRAMMING

The basic idea of our dynamic programming algorithm is to incrementally construct longer and longer partial schedules of the DAG and, wherever possible, apply local optimizations to prune the solution space of partial schedules. We start with a naive approach for generating optimal IR-schedules [26] that consists in the exhaustive enumeration of all possibilities for topological sorting of the DAG.

Topological sorting maintains a set of DAG nodes with indegree zero, the *zero-indegree set*, which is initialized to the set z_0 of DAG leaves. The algorithm repeatedly selects a DAG node v from the current zero-indegree set, appends it to the current schedule, and removes it from the DAG, which implies updating the indegrees of the parents of v. The zero-indegree set changes by removing v and adding those parents of v that now got indegree zero (see Figure 2). This process is continued until all DAG nodes have been scheduled. Most heuristic scheduling algorithms differ just in the way how they assign priorities to DAG nodes that control which node v is being selected from a given zero-indegree set. If these priorities always imply a strict linear ordering of nodes in the zero-indegree set, such a scheduling heuristic is also referred to as *list scheduling.*

A recursive enumeration of all possibilities for selecting the next node from a zero-indegree list generates all possible IR-schedules of the DAG. Of course, there are exponentially many different IR-schedules; an upper bound is $n!$, the number of permutations of n DAG nodes; the exact number of IR-schedules depends on the structure of the DAG. Hence, the run time of this enumeration method is exponential as well; our implementations [26] have shown

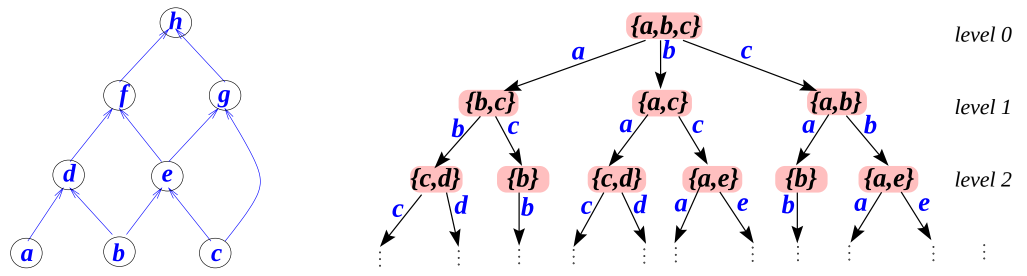
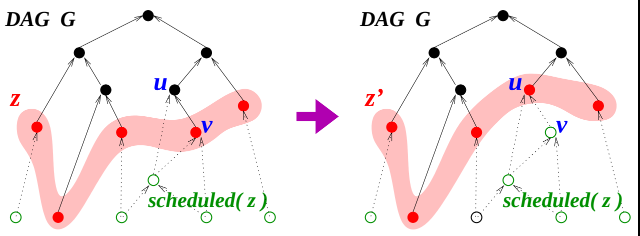


Figure 3: An example DAG (left hand side) and the resulting selection tree (first three levels, right hand side).

```
function selection( Set $z$, DAGnode $v \in z$, int[] $INDEG$)
 returns ( Set $z'$, int[] $INDEG'$ )
 // implicitly remove all edges $(v, u)$ $\forall u$:
 $INDEG'(u) = \begin{cases} INDEG(u) - 1 & where\ \exists(v,u) \\ INDEG(u) & elsewhere \end{cases}$
 // update zero-indegree set:
 $z' \leftarrow z - \{v\} \cup \{new\ leaves\} = \{u :\ INDEG(u) = 0\}$
 return ( $z'$, $INDEG'$ )
end selection
```



```
procedure top_sort( Set $z$, int[] $INDEG$, int $l$ )
 if $z \neq \emptyset$   // that is, $l \leq n$
      select an arbitrary node $v \in z$;
      $S[l] \leftarrow v$;   // append $v$ to partial schedule
      ( $z'$, $INDEG'$) $\leftarrow$ selection( $z$, $v$, $INDEG$);
      top_sort( $z'$, $INDEG'$, $l + 1$ );
 else output $S[1 : n]$ fi
end top_sort

Call top_sort( $z_0$, $INDEG_0$, 1 ) produces schedule $S[1 : n]$
```

Figure 2: Topological sorting, here a recursive formulation, generates a schedule of the DAG in n selection steps.

that it cannot be applied for basic blocks larger than about 15 instructions.

The exhaustive enumeration of topological sorts implicitly[1] builds a tree-like representation of all schedules of the DAG, called the *selection tree* (see Figure 3). Each node of the selection tree corresponds to an instance of a zero-indegree set of DAG nodes that occurs during topological sorting. A node z in the selection tree is connected by directed edges to the selection nodes z' corresponding to all zero-indegree set instances that can be produced by performing one *selection* step in the topological sort algorithm; the selection edges are labeled by the corresponding DAG node v chosen at that selection step. Each path in the selection tree, from the root (i.e., the set of all DAG leaves) to a leaf (i.e., an instance of the empty set), corresponds one-to-one to a valid IR-schedule of the DAG [26], which is given as the sequence of selection edge labels on that path. All paths in the selection tree that end up in instances of the same zero-indegree set have the same length, as the same subset of DAG nodes has been scheduled; hence the selection tree is leveled.

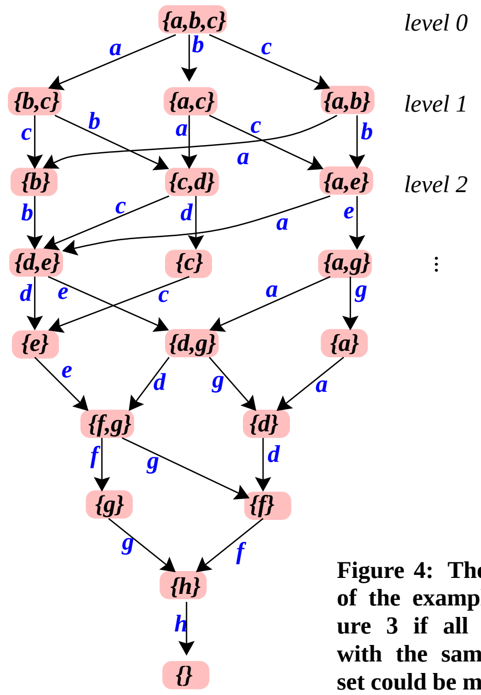


Figure 4: The selection DAG of the example DAG of Figure 3 if all selection nodes with the same zero-indegree set could be merged [26].

By looking closer at the selection tree, we can see that multiple instances of the same zero-indegree set may occur. For all these instances, the same set *scheduled*(z) of nodes in the same subDAG G_z of G below z has been scheduled. This leads to the idea that we could perhaps optimize locally among all the partial schedules corresponding to equal zero-indegree set instances, merge all these nodes to a single selection node and keep just one optimal partial schedule to be used as a prefix in future scheduling steps (see Figure 4). In previous work [26] we have proved that this aggressive merging is valid when computing a space-optimal schedule for a single-issue processor with unit-time latencies. When applying this idea consistently at each level, the selection tree becomes a *selection DAG*, which is leveled as well. By constructing the selection DAG level-wise, each zero-indegree set occurs at most once, thus at most 2^n selection nodes will be created. Additionally, many sub-

[1]The selection tree corresponds to the call tree of a recursive implementation of the naive enumeration algorithm.

sets of V may never occur as a zero-indegree set. Note also that, thanks to the leveling of the selection DAG, all selection nodes in level l can be released as soon as all selection nodes in level $l+1$ have been created.

This method leads to a considerable compression of the solution space. However, grouping of partial schedules by merging of selection nodes is only applicable to those schedules that are *comparable* with each other, where comparability depends on the target architecture and the optimization goal. For the space optimization mentioned above, *all* schedules ending up in an instance of the same zero-indegree set z are comparable, as the same subset of already scheduled DAG nodes

$$\begin{aligned} \textit{alive}(z) = \{ & u \in \textit{scheduled}(z): \\ & \exists (u,v) \in E, \;\; v \notin \textit{scheduled}(z) \;\} \end{aligned}$$

resides in registers. Together with other optimizations [26], this compression makes the algorithm practical also for medium-sized DAGs with up to 50 nodes.

This high degree of comparability is, however, no longer given if we are looking for a time-optimal schedule, because then it usually turns out only later whether one partial schedule is really better than another one if used as prefix in a schedule for all DAG nodes.

4. TIME PROFILES AND TIME-OPTIMAL SCHEDULING

A *time-profile* represents the occupation status of all units at a given time. It specifies for each functional unit the sequence of instructions that are currently under execution in that unit but not yet completed. In other words, the time-profile gives just that most recent part of a partial schedule that may still influence the issue time slot of the next instruction to be scheduled.

Formally, a *time profile* $P = (d,p)$ consists of a *issue horizon displacement* $d \in \{0,1\}$ and a *profile vector*

$$\begin{aligned} p \;\; = \;\; (& p_{1,1}, ..., p_{1,\ell_1+d-1}, \\ & p_{2,1}, ..., p_{2,\ell_2+d-1}, \\ & \cdots, \\ & p_{f,1}, ..., p_{f,\ell_f+d-1}) \end{aligned}$$

of $\sum_{i=1}^{f}(\ell_i + d - 1)$ entries $p_{i,j} \in V \cup \{\texttt{NOP}\}$. An entry $p_{i,j}$ of the profile vector denotes either the corresponding DAG node v for some instruction y issued to U_i, or a NOP (–) where no instruction is issued. Note that for a unit with unit-time latency there is no entry in p. The displacement d accounts for the possibility of issuing an instruction at a time slot where some other instruction has already been issued. For single-issue processors, d is always 0, thus we can omit d in P. For an in-order multi-issue architecture, we have $d \in \{0,1\}$ where for $d=1$ at least one and at most $\omega - 1$ of the entries corresponding to the most recent issue time, $p_{i,1}$ for $1 \leq i \leq f$, must be non-NOP. For out-of-order issue architectures, the displacement could be greater than one, but we need not consider this case as we aim at computing an optimal schedule statically. Of course, not all theoretically possible time profiles do really occur in practice.

The time profile $P = \textit{profile}(s)$ of a given target-schedule s is determined from s as follows: Let $t \geq 0$ denote the time slot where the last instruction is issued in s. Then $P = (d,p)$ is the time profile that is obtained by just concatenating (in reverse order to determine p) the DAG nodes (or NOPs) corresponding to the $\ell_i + d - 1$ latest entries $s_{i,t-d-\ell_i+2}, ..., s_{i,t}$ in s for the units U_i, $i = 1, ..., f$, where $d = 1$ if another instruction may still be issued at time t to a unit U_i with $s_{i,t} = p_{i,1}$ =NOP, and 0 otherwise. Entries $s_{i,j}$ with $j < 0$ are regarded as NOPs. t is called the *time reference point* of P in s. Note that t is always smaller than the execution time of s.

Hence, a time profile contains all the information required to decide about the earliest time slot where the node selected next can be scheduled.

THEOREM 1. *For determining a time-optimal schedule, it is sufficient to keep just one optimal target-schedule s among all those target-schedules s' for the same subDAG G_z that have the same time profile P, and to use s as a prefix for all target-schedules that could be created from these target-schedules s' by a subsequent selection step.*

Proof: All schedules of G_z contain the same set of nodes. We consider just the next node $v \in z$ that will be selected and appended to a schedule s' of G_z with time profile $P = (d,p)$ in a subsequent selection step; evidence for all subsequent selections will then follow by induction. For each s', let $t(s')$ denote the time reference point of P in s'. Let y be some matching target instruction for v to be chosen, and assume that y is to be scheduled on unit U_i. Let $s'_{i,j'}$ be the latest instruction that has been issued in s' to U_i. Assume that, if existing, the DAG predecessors w_1 and w_2 of v have been issued as instructions s'_{i_1,j_1} and s'_{i_2,j_2} respectively. Then y can be issued earliest at time $j = \max(t(s')+1-d,\; j'+o_i,\; j_1+\ell_{i_1},\; j_2+\ell_{i_2})$, provided that the issue width is not exceeded at time j. If $s'_{i,j'}$ is outside the time profile, that is, $j' \leq t(s') - d - \ell_i + 1$, then no instruction was issued to U_i within the time interval covered by the time profile P, thus $j' \leq t(s') + 1 - d - o_i$. A similar argument holds for the units U_{i_1} and U_{i_2}. In either case, the issue time j of y does not depend on instructions outside the profile window. As the profile P is supposed to be the same for all s', the difference $j - t(s')$ will be the same for all s'. Hence, one may take an arbitrary one of the s' as a prefix for the next selection of v. As the execution time should be minimized, one of the s' with minimum execution time must be kept. □

We extend the definition of selection nodes accordingly, as the zero-indegree set is no longer sufficient as a unique key:

An *extended selection node*, or *ESnode* for short, is identified by a triple $\eta = (z,P,t)$, consisting of a zero-indegree set z, a time profile P, and the time reference point t of P in the schedule s which is stored as an attribute η*.schedule* in that node. Technically, ESnodes can be retrieved efficiently, e.g. by hashing (as applied in the current implementation).

According to Theorem 1 it is sufficient to keep as the attribute η*.schedule* of an ESnode η, among all target-schedules with equal zero-indegree set and equal time-profiles, one with the shortest execution time.

Figure 5 illustrates the resulting extended selection DAG for the example DAG of Figure 3, applied to a single-issue target processor with two functional units, U_1 with latency $\ell_1 = 1$ and U_2 with latency $\ell_2 = 2$, hence the time profiles have a single entry. Nodes b and e are to be executed on unit U_2, the others on U_1.

An important optimization of this algorithm consists in the structuring of the space of all ESnodes as a two-dimensional grid of ESnode lists L_l^k, with one dimension for the number l of IR operations scheduled (i.e., the length of the IR-schedule), and one dimension for the execution time k. L_l^k contains only those ESnodes in level l whose associated partial schedules have execution time k (see Figure 6).

Let $D = \max_i \ell_i$ denote the maximum latency. Appending a node v to an existing schedule S will increase the execution time of any target schedule s derived from S by at most D cycles. Hence, if we perform a selection starting from an ESnode η in L_l^k, the resulting ESnode η' will be stored in exactly one of L_{l+1}^k,..., L_{l+1}^{k+D}.

This observation pays off in two aspects: First, it restricts the scope of searching for existing ESnodes with same time profile and

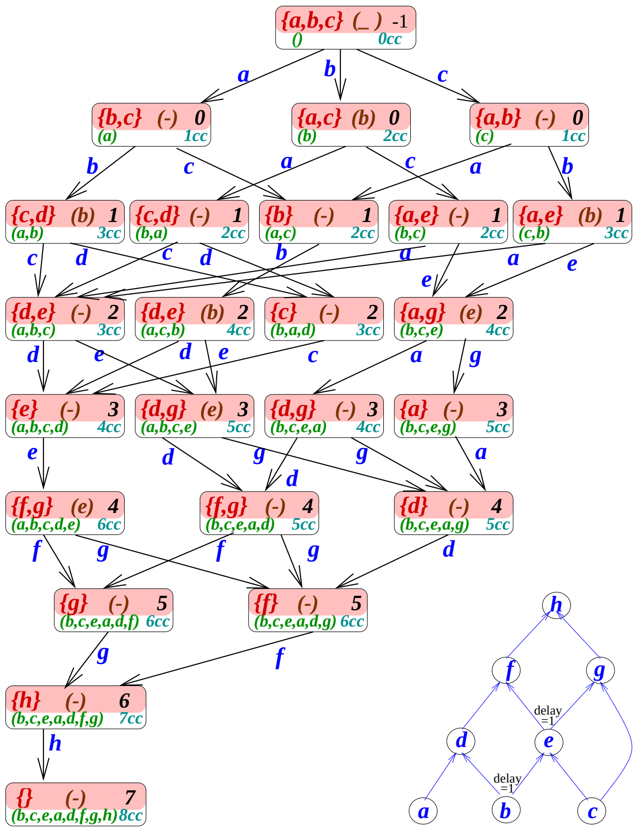


Figure 5: Example for an extended selection DAG.

same zero-indegree set to only $D+1$ lists. Second, it defines the precedence relations for the construction of ESnode lists L_l^k, see Figure 6. This knowledge allows us to construct the solution space of ESnodes in an order that is (1) compliant with the precedence relations among the selection nodes, and (2) most favorable with respect to the desired optimization goal, so that those selection nodes that look most promising with regard to execution time are considered first, while the less promising ones are set aside and reconsidered only if all the initially promising alternatives finally turn out to be suboptimal.

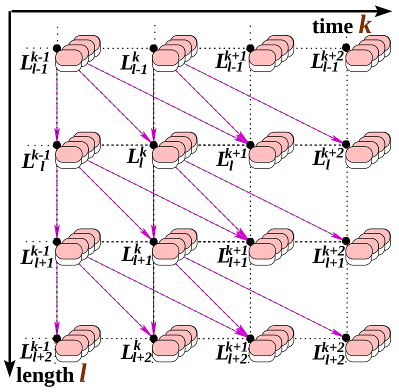


Figure 6: Structuring the space of partial solutions as a two-dimensional grid.

The final algorithm is given in Figure 7. The correctness follows from Theorem 1 by induction. Note that partial schedules that exceed a given, maximum register need $m_{\max}$ are discarded immediately.

```
function timeopt ( DAG $G$ with $n$ nodes and set $z_0$ of leaves)
 List<ESnode> $L_l^k \leftarrow$ empty list $\forall l\ \ \forall k$;
 $L_0^0 \leftarrow$ new List<ESnode> ($\eta_0$ = new ESnode$(z_0, P_0, t_0)$);
 $\eta_0$.schedule $\leftarrow \emptyset$;
 for $k = 0, ..., n \cdot D$ do    // outer loop: over time axis
     for level $l$ from 0 to $n-1$ do
         for all $\eta = (z, t, P) \in L_l^k$ do
             for all $v \in z$ do
                 $(z', INDEG') \leftarrow selection(v, z, INDEG)$;
                 for all $y \in \Psi(v)$ do
                     $s' \leftarrow \eta.schedule \bowtie y$;         // append $y$
                     if $m(s') > m_{\max}$ then continue fi;
                     $k' \leftarrow \tau(s')$;
                     $P' \leftarrow profile(s')$;
                     $t'$ is the time reference point of $P'$ in $s'$;
                     $\eta' \leftarrow$ new ESnode$(z', P', t')$;
                     $\eta'.schedule \leftarrow s'$;
                     for all $L_{l+1}^j$ with $k \le j \le k + D$ do
                         if $(\eta'' \leftarrow L_{l+1}^j.lookup(z', P'))$ exists
                             then break fi;
                     od
                     if $\eta'' = (z', P', t'')$ exists then
                         if $k' < j$ then
                             $L_{l+1}^j.remove(\eta'')$;  $L_{l+1}^{k'}.insert(\eta')$;
                         else forget $\eta'$  fi
                     else $L_{l+1}^{k'}.insert(\eta')$;  fi
                 od
             od
         od
     od
     if $L_n^k.nonempty()$
         then return $\eta.schedule$ for some $\eta \in L_n^k$ fi
 od
end timeopt
```

Figure 7: The algorithm for determining a time-optimal schedule, taking instruction selection into account.

5. IMPLEMENTATION, FIRST RESULTS

We use the LEDA library [33] for the implementation of the most important data structures, such as DAGs, selection node lists, and zero-indegree sets. Furthermore, we have connected a C front end to our code generator prototype. The new prototype is intended to become the technical basis for a future system for integrated code generation, called OPTIMIST.

First, we use randomly generated basic blocks as input to test our algorithm with problem instances of a reasonable size. Our method for generating random DAGs is controlled by several parameters such as the number of nodes or the number of leaf nodes. We considered different parameter sets for f, ω, ℓ, and Ψ to model various target architectures.

Figure 8 shows the optimization times depending on the DAG size for a large test series with $f = 2$, $\omega = 1$, $\ell = (1, 2)$. All arithmetic instructions go to U_1, the `Loads` to U_2. Figure 9 shows the case $f = 3$, $\omega = 1$, $\ell = (1, 2, 3)$ where additions go to U_1, multiplications to U_2, and `Loads` to U_3. Our algorithm turns out to be practical for DAGs with up to 50 instructions. It ran out of space (256 MB main memory plus 200 MB swap space) only for a few cases in the range between 38 and 45 instructions, for about 50% of the DAGs with 46 to 50 nodes, and for most DAGs with more than 50 instructions. Note that basic blocks that can still be optimized

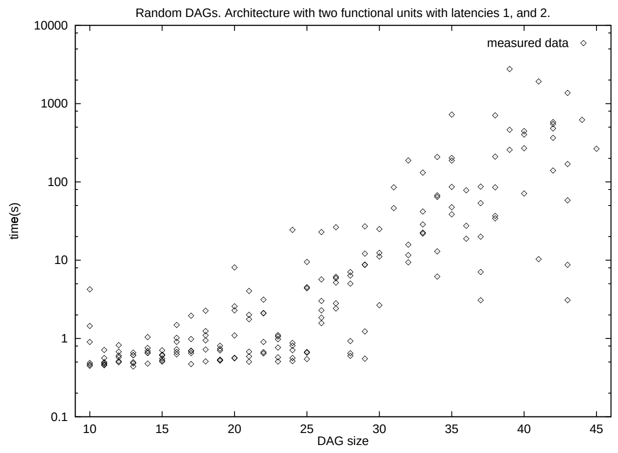


Figure 8: Optimization times for 200 random DAGs with $f = 2$, $\omega = 1$, $\ell = (1, 2)$.

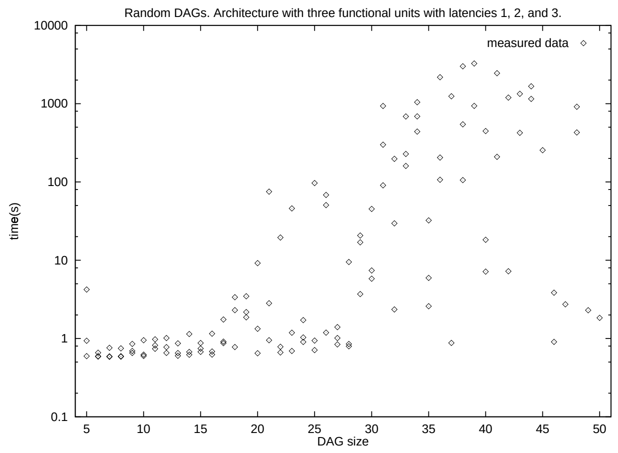


Figure 9: Optimization times for 120 random DAGs with $f = 3$, $\omega = 1$, $\ell = (1, 2, 3)$.

Source program, basic block	# IR nodes n	optimization time $f = 2$, $\omega = 1$, $\ell = (1, 2)$	optimization time $f = 3$, $\omega = 1$, $\ell = (1, 2, 3)$
FIR filter `fir.c`			
BB1 (first loop body)	16	3.5 s	4.0 s
BB2	16	8.0 s	9.5 s
BB3 (main loop body)	30	3:21:50.2 s	4:40:44.9 s
Matrix multiplication			
BB2 (main loop body)	30	1:05.0 s	1:41.8 s
same, unrolled once	40	6:08.5 s	9:47.2 s
Jacobi-style grid relaxation			
loop body, 5-point-stencil	40	1:15.8 s	1:31.8 s
loop body, 9-point-stencil	53	1:36:13.2 s	2:00:51.5 s

Table 1: Optimization times for basic blocks taken from compiled C programs. All measurements were taken on an Ultra Sparc 10 with 256 MB RAM and 200 MB swap space.

with a naive enumeration algorithm may have up to 15 instructions only. It can be well observed how the combinatorial explosion is deferred towards larger problem sizes by our algorithm.

Second, we have tested our algorithm with real-world example programs written in C. Table 1 shows the results. We observed for the examples with compiled code similar time and space requirements of the optimization as for the randomly generated basic blocks. Note that the optimization times depend considerably on the individual structure of the DAG. For instance, optimization of the FIR filter main loop body with 30 IR nodes takes about 200 times longer than for the matrix multiplication loop body with the same number of IR nodes. The source codes and the visualizations of their basic block IRs can be inspected at the project homepage[2].

6. POSSIBLE EXTENSIONS

Currently we still target architectures with a homogeneous register set but aim at generalizing this in the near future. For that purpose, we extend the time-profiles to *time-space-profiles*, but the implementation of this extension is not yet finished. A time-space profile for a zero-indegree set z additionally contains a *space profile*, that is a description which values in *alive*(z) reside in which class of registers. As the memory is also modeled as a register class, this covers even spilling and (optimal) scheduling of spill code. The space profile is used e.g. to restrict the alternatives for instruction selection for the next node to be scheduled. A more complete discussion of space profiles requires experimental evaluation and is beyond the scope of this paper.

Furthermore, to be able to retarget our system to different architectures, we will develop or adopt a suitable hardware description language.

The algorithm exhibits potential for exploiting massive loop-level parallelism. In principle, all but the outermost two loops may run in parallel, provided that a sequentially consistent data structure is used for the extended selection DAG that applies a locking mechanism to avoid multiple insertion of ESnodes. The parallelization of the two outermost loops must take the abovementioned dependencies among the ESnodes' construction into account, such that multiple processors may work on expanding different ESnode lists located on the same diagonal.

A generalization to more advanced instruction selection that allows to cover multiple IR nodes by a single target instruction, as applied by state-of-the-art tree pattern matchers, is straightforward but not yet implemented. The dynamic programming algorithm processes the IR nodes one after another. At some IR nodes v there is a choice between using a simple target instruction that matches v, or a more powerful instruction that may cover v and several subsequently scheduled nodes. For instance, an ADD operation may be translated directly to an `add` instruction, but it may also be executed as part of a combined `addmul` operation that requires a matching MUL operation. Or, a 32-bit adder may be used in some processors as two 16-bit adders operating in SIMD-parallel; where a 16-bit adder is enough for an ADD operation v, another 16-bit ADD operation can be sought to complement a double-16-bit addition. Unfortunately, it is not yet known when selecting an instruction y for v whether other DAG nodes may be scheduled subsequently that, together with v, would match such a more powerful instruction y'. This problem is solved by applying *speculative instruction selection.* Those instructions y' that require certain other IR operations to follow are considered but marked as speculative. As soon as the speculation turns out to be wrong, this variant is discarded. This method preserves the leveling property of the leveled extended selection DAG. The speculation may be complemented by a preceding static analysis of the DAG structure, such that speculation needs only be applied where such a cover is possible.

The time profiles may also be used as a mechanism to describe

[2]`http://www.ida.liu.se/~chrke/optimist`

the boundaries of schedules for basic blocks. Additionally, the computation of a target-schedule from a given IR-schedule may take, as an optional parameter, a time profile as an offset. This allows to propagate boundary descriptions of basic blocks along the program's control flow graph and switching back and forth between a local and global scope of scheduling. In this way, loop scheduling could be solved by an iterative process. This is an issue of future research.

7. RELATED WORK

We classify related work into heuristic methods and optimal methods and distinguish between isolated and integrated approaches.

7.1 Heuristics

The critical path list scheduling algorithm is the most popular heuristic for local instruction scheduling [35]. List scheduling determines a schedule by topological sorting, as described above. The edges of the DAG are annotated by weights which corresponds to latencies. In the critical path list scheduling, the priority for selecting an node from the zero-indegree set is based on the maximum-length path from that node to any leaf node. There exist extensions to list scheduling which take more parameters into account, data locality for example.

Global instruction scheduling methods, such as trace scheduling [14] and region scheduling [20] can move instructions across basic blocks to produce more efficient code.

Goodman and Hsu [19] try to break the phase-ordering dependence cycle and solve the register-constrained time optimization problem for large basic blocks by a mix of several heuristics and switching back and forth between space and time optimization depending on the current register pressure. Freudenberger and Ruttenberg [16] extend this approach to trace scheduling for pipelined VLIW processors. Further heuristic approaches [27, 34] have addressed the problem of register-constrained scheduling problem for superscalar and VLIW processors with arbitrary latencies. Kiyohara and Gyllenhaal [27] consider the special case of DAGs in unrolled loops; spill code is generated if necessary. Mutation scheduling [36] is an integrated, heuristic based method that integrates code selection, register allocation and instruction scheduling into a unified framework.

In general, heuristic methods can produce effective results within time and space limitation, in particular for standard processor architectures. But they do not guarantee optimality.

7.2 Optimal approaches

Integer linear programming. ILP-based methods are widely used today in code generation. The difficult part is to specify the linear program which is solved usually by third party linear solvers. Once the problem is specified as an ILP instance, the connection to the DAG is lost — the solver does not access extra information about the original program to solve the problem.

Wilken et al. [45] present a set of extensive transformations on the DAG, that help to derive an advanced ILP formulation for the instruction scheduling phase. First, they show that the basic formulation of the ILP of a given problem leads to unacceptable compile time. Then, providing transformations on the DAG, they decrease the complexity of the ILP, such that it can cope with basic blocks up to 1000 instructions in acceptable time. The resulting schedule is optimal. However, the register allocation problem is ignored during the optimization.

For instance, Wilken et al. show that an hourglass-shaped DAG (see Figure 10) can be scheduled using a divide-and-conquer step: An optimal schedule for such hourglass DAGs can be determined as a concatenation of optimal schedules for the two subDAGs below and above the articulation node. Note that explicit application of this simplification is not necessary in our approach, as such structural properties are automatically exploited by our algorithm, which creates for an hourglass-shaped DAG also an hourglass-shaped selection DAG, that is, the optimization is implicitly decomposed into two separate parts; preprocessing the DAG is not necessary.

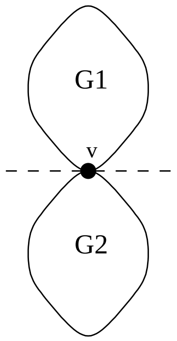


Figure 10: An hourglass-shaped DAG. Each node in G_2 is a predecessor of the articulation node v, and each node in G_1 is a successor of v. There is no path from a node in G_2 to a node in G_1 that does not contain v.

An interesting approach, from a technical and economical point of view, consists in performing post-pass optimization. In the PROPAN framework [25], Kästner implemented a phase coupled optimizer generator. The generator reads in a processor specification described in a *Target Description Language* (TDL) and generates a phase coupled optimizer specified as an ILP instance that takes restrictions and features of the target processor into account. An exact and optimal solution is produced, or a heuristic based, if the time limit is exceeded. In this framework, the full phase integration is not possible for larger basic blocks, as the time complexity is too high.

Dynamic programming. Aho and Johnson [2] use a linear-time dynamic programming algorithm to determine an optimal schedule of expression *trees* for a single-issue, unit-latency processor with homogeneous register set and multiple addressing modes, fetching operands either from registers or directly from memory.

Vegdahl [44] proposes an exponential-time dynamic programming algorithm for time-optimal scheduling that uses a similar compression strategy as described in Section 3 for combining all partial schedules of the same subset of nodes. In contrast to our algorithm, he first constructs the entire selection DAG, which is not leveled in his approach, and then applies a shortest path algorithm. Obviously this method is, in practice, applicable only to small DAGs. In contrast, we take the time and space requirements of the partial schedules into account immediately when constructing the corresponding selection node. Hence, we need to construct only those parts of the selection DAG that could still lead to an optimal schedule. Vegdahl's method directly generalizes to a restricted form of software pipelining of loops containing a single basic block, where the execution of independent instructions of subsequent loop iterations may overlap in time, by computing a shortest cycle. Note that our method could be generalized accordingly. In [44], register requirements are not considered at all.

Space-optimal schedules for DAGs are generated by a predecessor of our algorithm [26]. Although the worst case complexity for this algorithm is also exponential, this algorithm is, as shown empirically, practical for medium size basic blocks with up to 50 instructions.

Branch-and-bound. Yang et al. [46] present an optimal scheduling algorithm for a special architecture where all instructions take one time unit, and all functional units are identical. Note that optimal scheduling is even for such an architecture still NP-complete.

Enumeration. Chou and Chung [11] enumerate all possible target-schedules to find an optimal one. They propose methods to prune the enumeration tree based on structural properties of the DAG. Their algorithm is suitable for basic blocks with up to 30 instructions.

Constraint logic programming. Ertl and Krall [13] specify the instruction scheduling problem as a constraint logic program. A time-optimal schedule is achieved for small and medium size basic blocks.

7.3 Special cases

Where the DAG happens to be a *tree* or the target processor architecture is very simple, a time-optimal schedule can be found in polynomial time [4, 5, 24, 28, 37, 39, 41].

8. SUMMARY AND CONCLUSIONS

Several integrated methods based on heuristics, especially integrating register allocation and instruction scheduling, can be found in the literature. The heuristic methods may improve generated code, but do not indicate how far the produced code is from being optimal. Optimal solutions can be derived, if at all, only for very special cases or for isolated phases, but cannot be regarded optimal from a more general point of view. There are only a few approaches that guarantee to find an integrated optimal solution, and their high complexity causes them to be applicable only to quite small problem instances.

In this paper, we have made the first step towards the highly ambitious goal of fully integrated, optimal code generation. We have presented the theory and implementation of an integrated code generation method that is based on dynamic programming. We have exploited problem-specific properties for the compression and the efficient traversal of the solution space, which makes, for the first time, an integrated optimal solution of instruction selection and instruction scheduling tractable also for medium-sized basic blocks. We are now working on extending our method, by taking also non-homogeneous register sets into account, generating and scheduling spill code optimally, and allowing more aggressive forms of instruction selection. Future work in this project will also aim at going beyond the basic block scope of optimization.

9. REFERENCES

[1] A. Aho, S. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1), Jan. 1977.

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

[3] S. Bashford and R. Leupers. Phase-coupled mapping of data flow graphs to irregular data paths. In *DAES*, pages 1–50, 1999.

[4] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay on one cycle. *ACM Trans. Program. Lang. Syst.*, 11(1):57–67, Jan. 1989.

[5] D. Bernstein, J. M. Jaffe, R. Y. Pinter, and M. Rodeh. Optimal scheduling of arithmetic operations in parallel with memory access. Technical Report 88.136, IBM Israel Scientific Center, Technion City, Haifa (Israel), 1985.

[6] D. Bernstein, M. Rodeh, and I. Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.*, 38(9):1308–1314, Sept. 1989.

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

[8] T. S. Brasier, P. H. Sweany, S. J. Beaty, and S. Carr. Craig: A practical framework for combining instruction scheduling and register assignment. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, 1995.

[9] J. Bruno and R. Sethi. Code generation for a one–register machine. *J. ACM*, 23(3):502–510, July 1976.

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

[11] H.-C. Chou and C.-P. Chung. An Optimal Instruction Scheduler for Superscalar Processors. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):303–313, 1995.

[12] A. P. Ershov. *The Alpha Programming System*. Academic Press, London, 1971.

[13] M. A. Ertl and A. Krall. Optimal Instruction Scheduling using Constraint Logic Programming. In *Proc. 3rd Int. Symp. Programming Language Implementation and Logic Programming (PLILP)*, pages 75–86. Springer LNCS 528, Aug. 1991.

[14] J. Fisher. Trace Scheduling: A General Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[15] R. Freiburghouse. Register Allocation via Usage Counts. *Comm. ACM*, 17(11), 1974.

[16] S. M. Freudenberger and J. C. Ruttenberg. Phase Ordering of Register Allocation and Instruction Scheduling. In *Code Generation: Concepts, Tools, Techniques [18]*, pages 146–170, 1992.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979. With an addendum, 1991.

[18] R. Giegerich and S. L. Graham, editors. *Code Generation - Concepts, Tools, Techniques.* Springer Workshops in Computing, 1991.

[19] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Proc. ACM Int. Conf. Supercomputing*, pages 442–452. ACM Press, July 1988.

[20] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Trans. on Software Engineering*, 16(4):421–431, Apr. 1990.

[21] R. Güttler. Erzeugung optimalen Codes für series–parallel graphs. In *Springer LNCS 104*, pages 109–122, 1981.

[22] S. Hanono and S. Devadas. Instruction scheduling, resource allocation, and scheduling in the AVIV Retargetable Code Generator. In *Proc. Design Automation Conf.* ACM Press, 1998.

[23] J. Hennessy and T. Gross. Postpass Code Optimization of Pipeline Constraints. *ACM Trans. Program. Lang. Syst.*, 5(3):422–448, July 1983.

[24] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(11), 1961.

[25] D. Kästner. PROPAN: A Retargetable System for Postpass Optimisations and Analyses. In *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*, June 2000.

[26] C. W. Keßler. Scheduling Expression DAGs for Minimal Register Need. *Computer Languages*, 24(1):33–53, Sept. 1998.

[27] T. Kiyohara and J. C. Gyllenhaal. Code Scheduling for VLIW/Superscalar Processors with Limited Register Files. In *Proc. 25th Annual IEEE/ACM Int. Symp. Microarchitecture*. IEEE Computer Society Press, 1992.

[28] S. M. Kurlander, T. A. Proebsting, and C. N. Fisher. Efficient Instruction Scheduling for Delayed-Load Architectures. *ACM Trans. Program. Lang. Syst.*, 17(5):740–776, Sept. 1995.

[29] E. Lawler, J. K. Lenstra, C. Martel, B. Simons, and L. Stockmeyer. Pipeline Scheduling: A Survey. Technical Report Computer Science Research Report, IBM Research Division, San Jose, CA, 1987.

[30] R. Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer, 1997.

[31] H. Li. Scheduling trees in parallel / pipelined processing environments. *IEEE Trans. Comput.*, C-26(11):1101–1112, Nov. 1977.

[32] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer, 1995.

[33] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. The LEDA User Manual, Version 3.6. Technical report, Max-Planck-Institut für Informatik, Saarbrücken, Germany, Im Stadtwald, 66125 Saarbrücken, Germany, 1998.

[34] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining Register Allocation and Instruction Scheduling (Technical Summary). Technical Report TR 698, Courant Institute of Mathematical Sciences, New York, July 1995.

[35] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[36] S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grained parallelism. In *Proc. Annual Workshop on Languages and Compilers for Parallel Computing(LCPC'94)*, pages 16–30. Springer LNCS 892, 1994.

[37] K. V. Palem and B. B. Simons. Scheduling time–critical instructions on RISC machines. *ACM Trans. Program. Lang. Syst.*, 15(4), Sept. 1993.

[38] T. Proebsting. Least-cost Instruction Selection for DAGs is NP-Complete. unpublished, http://www.research.microsoft.com/ ˜toddpro/papers/proof.htm, 1998.

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

[40] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *ACM SIGPLAN Notices*, 31(9):234–243, Sept. 1996.

[41] V. Sarkar and B. Simons. Anticipatory Instruction Scheduling. In *Proc. 8th Annual ACM Symp. Parallel Algorithms and Architectures*, pages 119–130. ACM Press, June 24–26 1996.

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

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

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

[45] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pages 121–133, 2000.

[46] C.-I. Yang, J.-S. Wang, and R. C. T. Lee. A Branch-and-Bound Algorithm to Solve the Equal-Execution Time Job Scheduling Problem with Precedence Constraints and Profile. *Computers Operations Research*, 16(3):257–269, 1989.