Operations research methods in compiler backends

Daniel Kästner*†and Reinhard Wilhelm‡

Abstract. *Operations research can be defined as the theory of numerically solving decision problems. In this context, dealing with optimization problems is a central issue. Code generation is performed by the backend phase of a compiler, a program which transforms the source code of an application into optimized machine code. Basically, code generation is an optimization problem, which can be modelled in a way similar to typical problems in the area of operations research. In this article, that similarity is demonstrated by opposing integer linear programming models for problems of the operations research and of code generation. The time frame for solving the generated integer linear programs (ILPs) as a part of the compilation process is small. As a consequence, using well-structured ILP-formulations and ILP-based approximations is necessary. The second part of the paper will give a brief survey on guidelines and techniques for both issues.*

Sažetak. *Operacijska istraživanja mogu se definirati kao teorija o numeričkom rješavanju problema odlučivanja. U tom kontekstu, najveća pažnja posvećuje se problemima optimizacije. Prevodilac (compiler) je program koji pretvara izvorni kod neke aplikacije u optimizirani strojni kod. Pritom se samo generiranje koda obavlja u drugoj (backend) fazi prevođenja. Generiranje koda zapravo je jedan problem optimizacije, koji se može modelirati slično kao i tipični problemi iz područja operacijskih istraživanja. U ovom članku, ta sličnost se prikazuje tako što se uspoređuju modeli cjelobrojnog linearnog programiranja za probleme iz operacijskih istraživanja s modelima za problem generiranja koda. Raspoloživo vrijeme za rješavanje generiranih cjelobrojnih linearnih programa (CLP-a) unutar procesa prevođenja je vrlo kratko. Zato se nužno moramo prikloniti jednom od sljedeća dva pristupa: korištenje dobro strukturiranih formulacija CLP-a, odnosno prelazak na aproksimacije zasnovane na CLP-u. Drugi dio članka daje kratak pregled smjernica i tehnika za oba pristupa.*

Received September 20, 1998 Accepted October 2, 1998

*Member of the Graduiertenkolleg "Effizienz und Komplexität von Algorithmen und Rechenanlagen" (supported by the DFG).

†Universität des Saarlandes, Fachbereich Informatik, Postfach 151150, D-66 041 Saarbrücken, Germany, e-mail: kaestner@cs.uni-sb.de

‡Universität des Saarlandes, Fachbereich Informatik, Postfach 151150, D-66 041 Saarbrücken, Germany, e-mail: wilhelm@cs.uni-sb.de

1. Introduction

In the field of Operations Research, optimization problems play an important role. Since the problem of code generation basically is an optimization problem, it is not surprising that there are similarities between the models and solution techniques which are used. An important topic of Operations Research is *production planning*. Two different directions can be distinguished: *aggregate production planning for product types* and *production scheduling*. In this paper, we will focus on *production scheduling* which deals with operational decisions [16]. The analogy between these problems and the problem of code generation is pointed out by opposing the job shop problem to the problems of instruction scheduling and register assignment. Traditionally, instruction scheduling and register allocation are performed in two largely independent phases with each subproblem being solved by using graph-based heuristics. However, since the two problems are in fact interdependent, this phase decoupling can lead to inefficient code (phase ordering problem). Formulations based on integer linear programming (ILP) offer the possibility of integrating instruction scheduling and register assignment in a homogeneous problem description and of solving them together. Thus the technique of ILP is a universal approach suited for optimization problems in operations research as well as in the time sensitive field of compiler construction.

This article is structured as follows: in *Section 2.*, the job shop and flow shop problems are defined. The task of compiling is detailed in *Section 3.*; here, the focus is on the code generation part of the compilation process. The different subproblems of code generation are presented and the phase ordering problem is explained. The next two sections deal with different approaches for formulations of integer linear programs. *Section 4.* concentrates on time-based approaches, where the choice of the decision variables is based on the time the modelled event is assigned to. Time-based formulations for the flow shop and the instruction scheduling problems are opposed. In *Section 5.*, order-based formulations for the job shop and the instruction scheduling problem are presented. In order-based approaches, the decision variables reflect the ordering of the instructions or tasks which have to be calculated. *Section 6.* explains how the ILP-formulations for instruction scheduling presented in *Sec. 4.* and *Sec. 5.* can be extended to integrate the problem of register assignment. Approximation algorithms for the order-based ILP formulation for instruction scheduling and register assignment are presented in *Sec. 7.* and *Sec. 8.* concludes.

2. Job shop scheduling

2.1. Definition

For the definition of the *job shop* problem, we follow the definitions given in [4]. A job shop consists of a set of different machines that perform tasks making up a set of jobs. For each job, it is known how long its tasks take and in which order they have to be processed on each machine[1]. So, a job is composed of an ordered list of tasks; each task is characterized by the machine on which it has to be executed

[1]This ordering is called the *machine sequence* of the job.

and by the required processing time. Three constraints on jobs and machines have to be satisfied:

- there are no precedence constraints among tasks of different jobs;
- tasks cannot be interrupted, and each machine can handle only one job at a time;
- each job can be performed only on one machine at a time.

A frequently considered optimization goal for this problem is to find the *job sequences* on the machines which minimize the maximum of the completion times of all tasks, the so-called *makespan*. The *flow shop problem* is a special case of the job shop problem. In a job shop, each job has its own machine sequence and different jobs can have different machine sequences. In a flow shop, the machine sequences of all jobs have to be identical.

2.2. Graphical representation

The machine sequences can be visualized in a graph in which the nodes represent the tasks and directed edges denote the ordering of the tasks in a job [4, 5]. Additionally, two virtual tasks t_0 and t_{n+1} are introduced for the beginning and the end of the processing.

The decision problem of the job shop scheduling problem can be described by extending this *machine sequence graph* to the *disjunctive graph* [5]. A *disjunctive edge* is an originally undirected edge between two tasks which have to be processed on the same machine. Determining the processing order between these two tasks on the appropriate machine corresponds to selecting one of two possible directions for their disjunctive edge. Thus, the directed edges between tasks represent the precedence constraints on the tasks of the same job, and the disjunctive edges are added to ensure that each machine can handle at most one task at a time. The directions of the disjunctive edges are not determined by the input data but are subject to the scheduling process. A solution of the job shop scheduling problem is calculated by fixing the directions for all disjunctive edges.

In order to integrate the processing times of the tasks into the disjunctive graph, each node is marked with the processing time of the corresponding task. A feasible solution is calculated by fixing the direction of all disjunctive edges so that the resulting graph G is acyclic and the length of a longest weighted path between T_0 and T_{n+1} is minimal. Such a path is called a *critical path*. The length of a longest path in G connecting nodes T_0 and T_i is equal to the earliest possible starttime t_i of task T_i, $t_i = asap(T_i)$ (*as soon as possible*). Given an upper bound T of the makespan and given a feasible (not necessarily optimal) schedule, an upper bound of the latest possible starttime $t'_i = alap(T_i)$ (*as late as possible*) can be calculated. This is done by inverting the directions of all edges and subtracting the length of the longest path from t_{n+1} to t_i from the upper bound of the makespan. Methods to calculate these times in node- as well as edge-weighted graphs are the CPM- and MPM-method [8]. The resulting time window for the start of each task is exploited by many algorithms in order to increase efficiency [27, 13, 14].

Consider the following example. Let three jobs J_1, J_2, J_3 be given and let each task be specified by a triple (p, m, j) where p denotes the processing time of the task, m the number of the machine where it is executed and j the number of the job which the task belongs to. The task characteristics are given in *Table 1*; the corresponding disjunctive graph is shown in *Fig. 1*. There the dotted lines represent the precedence constraints between the tasks of each job and the solid lines correspond to the disjunctive edges. In *Fig. 1*, the tasks are numbered in increasing order of the jobs so that each task has its own number, e.g. task 5 denotes the first task of job J_2. Task 0 is the virtual start task, task 10 represents the virtual end task.

Jobs/Tasks	1	2	3	4
J_1	(5,1,1)	(5,2,1)	(6,3,1)	(6,1,1)
J_2	(15,1,2)	(6,3,2)		
J_3	(9,3,3)	(1,1,3)	(7,2,3)	

Table 1. *Example job shop*

Figure 1. *Disjunctive graph for the job shop of Tab. 1*

3. The process of compiling

Compilers for high-level programming languages are large, complex software systems. The task of a compiler is to transform an input program written in a certain source language into a semantically equivalent program in another language, the target language. This structure is illustrated in *Fig. 2*; there the source language is C and the target language is assembler code of a modern standard-DSP, the Analog

Devices ADSP-2106x ([1]).

Figure 2. *The process of compiling*

Assembler code is a human readable representation of executable machine code and can easily be transformed into the latter form.

Conceptually, the process of compiling can be subdivided into an analysis phase, the compiler frontend, and a synthesis phase, the compiler backend. In the analysis phase, the syntactic structure and some of the semantic properties of the source program, the so-called static semantics, are computed. The results of the analysis phase comprise either messages about syntactic or semantic errors in the program or an appropriate intermediate representation of the program. In this intermediate representation, the syntactic structure and the static semantic properties are exposed. The synthesis phase takes the intermediate representation as input and converts it into semantically equivalent target machine code.

The most important part of the compiler backend is the code generation phase which consists itself of several subtasks:

Code selection: the selection of instruction sequences for the program's constructs,

Register allocation: the determination of fast processor registers which should hold values of program variables and intermediate results,

Instruction scheduling: the reordering of the produced instruction stream for better exploitation of intraprocessor parallelism.

All tasks have high worst case complexity; register allocation and instruction scheduling are NP-hard. Therefore, heuristics for these problems are generally used.

For reasons of software complexity, a modular decomposition of code generation into these tasks is advisable. However, this will often lead to the combination of (locally) optimal or already nonoptimal solutions to even worse solutions as will be described shortly.

3.1. Code selection

Code selection is of different complexity for CISC and for RISC machines. A CISC machine usually offers many different meaningful, equivalent instruction sequences for a program, while this number is smaller for RISC machines. Code selection can be executed by a tree parser [9, 26] or a bottom up rewrite system [11] generated from an appropriate description of the target machine and its correspondance to the IR. In the tree parsing case, the different possible translations correspond to the different derivations of the IR according to the regular tree grammar describing the code selector.

Instructions are associated with costs which in turn combine to costs for the potential translations. A locally cheapest translation is usually selected.

3.2. Register allocation and assignment

The execution time of a program would profit if all values of live program variables and live intermediate results, often called *symbolic registers*, could be kept in fast processor registers. Liveness of a variable or result means that its current value will potentially be needed later on. The number of simultaneously live symbolic registers usually exceeds the number of available general purpose registers. Hence, a resource optimization problem results. Those values should be kept in processor registers which produce the highest benefits for the execution time.

Register allocation proper attempts to determine this set of symbolic registers. *Register assignment* then associates them with particular processor registers.

Variations of graph colouring algorithms are in use for register allocation. A *conflict graph* is built over the set of symbolic registers. Two such registers are connected by an edge if they are simultaneously alive. In this case, they can't be assigned to the same processor register. This conflict graph is then coloured with colours corresponding to processor registers. A heuristics is used to use as few registers as possible[6]. If the register allocator cannot find enough colours, i.e. registers, it introduces a temporary memory location for the corresponding value and the necessary spill code to access that memory location.

3.3. Instruction scheduling

Modern high speed processors offer some intraprocessor parallelism, e.g., parallel functional units and/or pipelines. Instruction scheduling attempts to reorder the (sequential) instruction sequences produced by previous phases in order to exploit these parallel capabilities. A program dependence analysis determines data and control dependences in the program. These limit the ways the instructions of the program can be reordered.

Control dependences refer to instructions whose execution depends on certain control conditions; they arise due to branch- or loop-instructions. Data dependences are pairs of read or write accesses to the same register or to other hardware components. Write accesses are usually called *definitions*, read accesses *uses*. Data dependences can be categorized as *true-dependences*(def-use), *output dependences* (def-def) and *anti-dependences*(use-def). If there is a true dependence between two instructions i and j, then i defines a value that is used by j. Swapping the two

instructions would lead to a different semantics of the program. The other types of dependences are defined analogously; only positions of independent instructions may be rearranged. The data dependences are modeled by the data dependence graph $G_D = (V_D, E_D)$ where $E_D = E_D^{true} \cup E_D^{anti} \cup E_D^{output}$. The set of nodes V_D represents the instructions of the input program, the set E_D models the different kinds of data dependences.

The data dependence graph defines a partial order among the instructions of the input program. We define a precedence relation $\prec$ on V_D where

$$i \prec j \;\Leftrightarrow i \xrightarrow[G_D]{+} j$$

Thus, $i \prec j$ holds if operation j depends directly or indirectly on operation i. The resulting problem is to rearrange the instructions of the input program so that the execution time is minimized, but no precedence constraints are violated. In its simplest form, instruction scheduling corresponds to the classical problem of *precedence constrained scheduling*. Let a set $\mathcal{T}$ of tasks of length 1 be given, a partial order $\prec$ on $\mathcal{T}$, m machines and an upper bound T for the schedule length. The goal is to find a schedule $\sigma : \mathcal{T} \rightarrow \{1, \ldots, T\}$, so that for all $t \in \{1, \ldots, T\}$ where $|\{i \in \mathcal{T} : \sigma(i) = t\}| \leq m$ holds:

$$i \prec j \qquad \Rightarrow \qquad \sigma(i) < \sigma(j)$$

This optimization problem is $\mathcal{NP}$-complete, except for the special case that there are only two processors [12] or each task i has at most one immediate predecessor with respect to the relation $\prec$. So, the tasks correspond to instructions and the machines represent parallel functional units of the underlying processor, e.g. ALUs, multipliers, etc.

In the scope of this paper, we will concentrate on the problem of instruction scheduling for VLIW-architectures (see *Fig. 3*).

Figure 3. *Example of the VLIW-architecture with 5 functional units*

In these architectures, most machine instructions can perform several independent tasks at a time. Thus, they can be considered as composed from several

RISC-like microoperations. These microoperations are subject to the scheduling process; the goal of the scheduling process is to pack the microoperations of the input program in as few instructions as possible. Each microoperation can be assigned to a control step, i.e. a clock cycle, in which the execution of the instruction containing that microoperation is started.

Several heuristic scheduling methods are in use to solve this complex task, e.g. *list scheduling* [10], *region scheduling* [15], and *percolation scheduling* [22]. Results bounding the distance from the optimum only exist for very regular architectures.

3.4. The phase ordering problem

The above listed phases of code generation are not independent. A decision improving one's result may make another's result worse.

Code selection naturally attempts to keep the costs low by using as many registers as possible. It is usually performed before register allocation assuming an infinite number of registers. It thus is in conflict with register allocation which has to cope with a limited number of registers. The code selector may produce an instruction sequence which it considers the least expensive using more registers than are available. Another instruction sequence using not more registers than are available may turn out to be cheaper, once the register allocator has inserted spill code.

A similar conflict exists between register allocation and instruction scheduling. Instruction scheduling uses registers to increase the instruction level parallelism. In the context of register allocation they are required to reduce the number of memory accesses. If register allocation is performed first, it can limit the amount of achievable instruction-level parallelism by assigning the same physical register to independent intermediate results. This prevents an overlapping of the corresponding operations by the scheduler. If instruction scheduling precedes register allocation, the number of simultaneously live values can be increased so much that many of these values have to be stored in main memory.

3.5. Phase integration by integer linear programming

Formulations based on integer linear programming (ILP) offer the possibility of integrating instruction scheduling and register assignment in a homogeneous problem description and of solving them toghether. However, this is not the only advantage over the traditional approaches treating these code generation subtasks in separate phases. By using integer linear programming, it is possible to get an optimal solution of both problems – albeit at the cost of high calculation times.

In order to speed up the solution process, approximations based on integer linear programs can be used. There, an approximative solution is calculated, e.g. by partial relaxations or problem decompositions, whose optimality cannot be guaranteed any more. However, in [17] approximative algorithms have been presented which produce optimal solutions in most cases.

Another feature is the ability to provide lower bounds on the optimal schedule length. The solution of graph-based heuristics is always an upper bound and it is not known, how much the optimal schedule length is exceeded. By solving relaxations of the corresponding ILP, the result is no feasible solution, but a lower bound on

the schedule length. This lower bound can be used to estimate the quality of the given heuristic solution. This is useful for input programs which could not be solved exactly by integer linear programming for reasons of complexity.

4. Time-based models

One well-known ILP-formulation which can be used to model the problem of instruction scheduling has been presented in the OASIC-approach [13, 14, 17, 18]. Compared to a standard formulation for the flow shop problem, the Wagner model, there are apparent similarities. In both formulations, the choice of the decision variables is based on the time the modelled event is assigned to. In contrast to order-based approaches, the variable name says nothing about the predecessors or successors of the associated task or instruction.

4.1. The Wagner modell

Wagner's model describes the *permutation flow shop*, in which all machines process the jobs in the same order [5, 25]. Once the job sequence on the first machine is fixed, it will be kept on all remaining machines. Any flow shop scheduling problem consisting of at most three machines has an optimal schedule which is a permutation schedule. However, in the general case the objective value of the optimal permutation schedule can be worse than that of the optimal flow shop schedule [24].

In the Wagner model, the following decision variables are used:

- $z_{ij} = \begin{cases} 1, & \text{if job } J_i \text{ is assigned to the } j\text{th position in the permutation} \\ 0, & \text{otherwise} \end{cases}$
- x_{jk} = idle time on machine M_k before the start of the job in position j in the permutation of jobs
- y_{jk} = idle time of the job in the jth position in the permutation after finishing processing on machine M_k, while waiting for machine M_{k+1} to become free.
- C_{max} = maximimum flow time of any job in the job set (makespan)

In the following, we assume that m machines $\{M_1, \ldots, M_m\}$ and n jobs $\{J_1, \ldots, J_n\}$ are given; p_{mj} denotes the processing time of the task of job J_j executed on machine M_m. Then the formulation reads as follows:

$$
\begin{aligned}
&\min && C_{max} \\
&\text{subject to} \\
&\sum_{j=1}^{n} z_{ij} &&= 1, \qquad i = 1, \ldots, n && (1)\\
&\sum_{i=1}^{n} z_{ij} &&= 1, \qquad j = 1, \ldots, n && (2)\\
&\sum_{i=1}^{n} p_{ri} z_{i,j+1} + y_{j+1,r} + x_{j+1,r} &&= y_{jr} + \sum_{i=1}^{n} p_{r+1,i} z_{ij} + x_{j+1,r+1}, \\
& && \qquad j = 1, \ldots, n-1; r = 1, \ldots, m-1 && (3)\\
&\sum_{j=1}^{n}\sum_{i=1}^{n} p_{mi} z_{ij} + \sum_{j=1}^{n} x_{jm} &&= C_{max} && (4)\\
&\sum_{r=1}^{k-1}\sum_{i=1}^{n} p_{ri} z_{i1} &&= x_{1k}, \qquad k = 2, \ldots, m && (5)\\
&y_{1k} &&= 0, \qquad k = 1, \ldots, m-1 && (6)
\end{aligned}
$$

Equations (1) and (2) assure that each job is processed exactly once by exactly one machine. Equation (3) guarantees that the execution and waiting times between all adjacent machines in the flow shop are consistent with each other. Equation (4) determines the makespan by summing the waiting times and the execution times of all tasks on the last machine in the schedule. Equation (5) adjusts the idle time of machine k: the processing of the first job on this machine must begin as early as the precedence relations allow. Equation (6) assures that the first job in the permutation passes immediately to each successive machine.

4.2. The OASIC model

The *OASIC*-formulation (*Optimal Architectural Synthesis with Interface Constraints*) has originally been developed for use in architectural synthesis [13, 14]. In [17], it has been adapted to the special needs of performing combined instruction scheduling and register assignment for a standard digital signal processor (Analog Devices ADSP 2106x) [2, 1].

First, we will give an overview of the terminology used:

- The main decision variables are called $x^k_{jn} \in \{0, 1\}$, where $x^k_{jn} = 1$ means, that microoperation j is assigned to the nth position ($n \geq 1$) in the generated schedule and is executed by an instance of resource type k.

- t_j describes the starting time of a microoperation j.

$$t_j = \sum_{k:(j,k)\in E_R} \sum_{n\in N(j)} n \cdot x^k_{jn}$$

- $N(j) = \{asap(j), asap(j)+1, \ldots, alap(j)\}$ is the set of possible control steps, in which an execution of j can be started.
- Q_i^k denotes the execution time of operation i on resource type k.

Each instruction of the input program can be executed by a certain resource type on the processor, e.g. by the arithmetic logic unit (ALU) or by the multiplier. In order to describe the mapping of instructions to hardware resources, a resource graph is used which is defined following [27].

Definition 1 *The* resource graph $G_R = (V_R, E_R)$ *is a bipartite directed graph. The set of nodes* $V_R = V_D \cup V_K$ *is composed of the nodes of the data dependence graph* V_D *and the available resource types represented by* V_K*. Its edge set* $E_R \subset V_D \times V_K$ *describes a possible assignment where* $(j,k) \in E_R$ *means that instruction* $j \in V_D$ *can be executed by the resources of type* k*.*

The goal of the ILP-formulation is to minimize the execution time of the code sequence to be optimized, so the objective function reads simply

$$\min \; M_{steps} \tag{7}$$

The correctness of the resulting schedule is guaranteed by several constraints:

1. time constraints

   No instruction may exceed the maximum number of control steps M_{steps}, which is to be calculated.

$$t_j \leq M_{steps} \qquad \forall\, j \in V_D \tag{8}$$

2. precedence constraints

   When instruction j depends on instruction i, then j has to be executed after completion of i.

$$\begin{aligned}
\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_j \leq n \\ n_j \in N(j)}} x_{jn_j}^k + \sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_i \geq n - Q_i^k + 1 \\ n_i \in N(i)}} x_{in_i}^k &\leq 1 \\
\forall\, (i,j) \in E_D^{true} \cup E_D^{output}&, \\
n \in (\{n + Q_i^k - 1 \mid n &\in N(i)\} \cap N(j)) \\
\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_j < n \\ n_j \in N(j)}} x_{jn_j}^k + \sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_i \geq n - Q_i^k + 1 \\ n_i \in N(i)}} x_{in_i}^k &\leq 1 \qquad \forall\, (i,j) \in E_D^{anti}, \\
n \in (\{n + Q_i^k - 1 \mid n &\in N(i)\} \cap N(j))
\end{aligned} \tag{9}$$

3. assignment constraints

   The execution of an operation must start in exactly one control step and is performed by exactly one resource type.

$$\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{n \in N(j)} x_{jn}^k = 1 \qquad \forall\, j \in V_D \tag{10}$$

4. resource constraints

   The number of available instances of a resource type must not be exceeded, so that in no control step, more than R_k operations may be executed by resource type k.

$$\sum_{j \in V_D : (j,k) \in E_R} \sum_{n \in N'(j,k,n')} x_{jn}^k \leq R_k \qquad \forall k \in V_K \wedge 0 \leq n' \leq M_{steps}$$
$$\text{with } N'(j,k,n') = \{n \in N(j) : n' = n + p,\ 0 \leq p \leq Q_j^k - 1\} \tag{11}$$

4.3. Comparison

In both formulations, the decision variables have a time-based semantics. Following the terminology of the flow shop problem, job J_i is assigned to the jth position in the permutation, if $z_{ij} = 1$. Each position can be considered as a time unit; the permutation, i.e. the ordering of the jobs is not reflected in the variables. In the OASIC-formulation, $x_{jn}^k = 1$ means that operation j is executed at control step n by a resource of type k. If all operations take one clock cycle to execute, this is equivalent to assigning operation j to the nth position in the generated schedule. So the ordering of the instructions corresponds to the job-permutations in the flow shop modelling.

Determining the value of the makespan (the optimal schedule length) has been shifted into the constraints in both formulations (see constraints (4) and (8)). The value of the makespan is equal to the sum of execution and waiting times of that machine which executes last. For the problem of instruction scheduling, the ordering of the instructions has to be determined by the optimization process, so the starting time of the last instruction is not yet known. Instead, the maximum of all starting times is considered.

From a theoretical point of view, the underlying scheduling models are not identical; however it is noticeable that important problems of different application areas share a great deal of formulation similarities. Both models use time-based variables; the machines of the flow shop correspond to the functional units of the instruction scheduling problem and the instructions correspond to the tasks of one job. The differences are that for instruction scheduling only one job has to be considered, whereas several instances of the machines have to be taken into account.

5. Order-based approaches

Another way of modelling the problem of instruction scheduling is used in the SILP-approach [27, 17, 18]. The generated ILP-formulation has similarities to the Manne-coding for the job shop scheduling problem. Both models can be called order-based, since the semantics of the decision variables reflect the ordering of the instructions or tasks which has to be calculated.

5.1. The Manne formulation

In order to solve the job shop problem, one classical formulation has been presented by Manne in [21]. Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be the set of tasks where T_0 is the virtual start task and T_{n+1} is the virtual end task. The set of machines is denoted by $\mathcal{P} = \{M_1, \ldots, M_m\}$. Each job J is composed of an ordered list of tasks from $\mathcal{T}$, so there is a precedence relation among the tasks of each job ($T_{i_1} \prec \cdots \prec T_{i_r}$, if $|J_i| = r$). The set $\mathcal{E}_k$ contains all pairs of tasks which can be executed on machine M_k. Binary variables w_{ij} are introduced for each such pair $(T_i, T_j) \in \mathcal{E}_k$). If $w_{ij} = 0$, task i has to be executed before task j on machine M_k; otherwise, if $w_{ij} = 1$, j has to be executed before i. The variables t_i denote the starting time for the processing of an task T_i and T is an upper bound of the makespan, e.g. the sum of all processing times p_j.

Then, the formulation can be given as follows:

$$
\begin{array}{rcll}
\min & & C_{max} & \\
\text{subject to} & & & \\
t_i + p_i & \leq & C_{max} & \forall\ i \text{ where } T_i \text{ is the last task of some job } J \quad (12)\\
t_i + p_i & \leq & t_j & \forall\ T_i \prec T_j \quad (13)\\
t_i + p_i & \leq & t_j + T w_{ij} & \forall k\ \forall (T_i, T_j) \in \mathcal{E}_k \quad (14)\\
t_j + p_j & \leq & t_i + T(1 - w_{ij}) & \forall k\ \forall (T_i, T_j) \in \mathcal{E}_k \quad (15)\\
w_{ij} & \in & \{0, 1\} & \quad (16)\\
t_i & \geq & 0 & \forall\ T_i \in \mathcal{T} \quad (17)
\end{array}
$$

The makespan is equal to the finish time for the execution of the last processed task (12). The constraints in (13) guarantee that the starting times of the tasks correspond to the machine sequences of each job. For a fixed job, a task may only begin after the processing of the preceding task of the same job has been finished. Constraints (14) and (15) guarantee that the starting times are not in conflict with the job sequences. If a task i has to be processed on a certain machine M_k before a task j belonging to another job ($w_{ij} = 0$), then j must be started after the processing of i has been finished.

5.2. The SILP formulation

The ILP-formulation described in this section was presented in [27] under the name *SILP* (*Scheduling and Allocation with Integer Linear Programming*). Again we will give an overview over the terminology first.

- The variable t_i indicates the starting time of a microoperation i (see *Sec. 3.3.*); the t_i values have to be integral.
- w_j describes the execution time of instruction $j \in V_D$.
- The busy time of the hardware component executing operation j is denoted by z_j (i.e. the minimal time interval between successive data inputs to this functional unit).

- The number of available resources of type $k \in V_K$ is R_k.

The ILP is generated from a resource flow graph G_F. This graph describes the execution of a program as a flow of the available hardware resources through the instructions of the program. For each resource type, this leads to a separated flow network. Each resource type $k \in V_K$ is represented by two nodes $k_Q, k_S \in V_F$; the nodes k_Q are the sources, the nodes k_S are the sinks in the flow network to be defined. The first instruction to be executed on resource type k gets an instance k_r of this type from the source node k_Q; after completed execution, it passes k_r to the next instruction using the same resource type. The last instruction using a certain instance of a resource type returns it to k_S. The number of simultaneously used instances of a certain resource type must never exceed the number of available instances of this type. An example resource flow graph for two different resource types and two instructions is given in *Fig. 4*. The instructions shown there belong to the instruction set of the digital signal processor ADSP-2106x SHARC. `DM` denotes the data memory and `S` the arithmetic and logic unit (ALU). The instruction `r1=dm(i0,m0)` is a load operation where the contents of a memory cell are stored in register `r1`. The address of the correct memory cell is obtained by adding the two registers `i0` and `m0`. The semantics of `r4 = dm(i1,m1)` is analogous. The instruction `r6 = t4 + r5` adds the contents of registers `r4` and `r5` and stores the result in register `r6`. Finally, `r7=min(r4,r5)` stores the minimum of the operand registers in `r7`.

Figure 4. *Resource flow graph for two instructions executed on an ALU and the data memory, resp.*

Definition 2 *Let $G_D = (V_D, E_D)$ be the data dependence graph for an input program and $G_R = (V_R, E_R)$ its resource graph. The* resource flow graph G_F *is a directed graph $G_F = (V_F, E_F)$ with*

$$V_F = \bigcup_{k \in V_K} V_F^k \quad \textit{und} \quad E_F = \bigcup_{k \in V_K} E_F^k$$

where

$$V_F^k = V_D^k \cup \{k_Q, k_S\} = \{u \in V_D \mid (u,k) \in E_R\} \cup \{k_Q, k_S\}$$

and

$$\begin{aligned} E_F^k &= \{(i,j)\ |i,j \in V_D^k\ \wedge\ j \textit{ not dependent on } i\ \wedge\ i \neq j\} \\ &\cup\ \{(k_Q,j)\ |(j,k) \in E_R\ \} \\ &\cup\ \{(j,k_S)\ |(j,k) \in E_R\ \} \end{aligned}$$

Each edge $(i,j) \in E_F^k$ *is mapped to a flow variable* $x_{ij}^k \in \{0,1\}$ *. A hardware resource of type* k *is moved through the edge* (i,j) *from node* i *to node* j*, if and only if* $x_{ij}^k = 1$*.*

V_D^k is the set of all nodes of the data dependence graph belonging to instructions which can be executed by resource type k. Each edge $(\mu,\nu) \in E_F^k$ describes a possible flow of resources of type $k \in V_K$ from μ to ν. The flow entering a node $j \in V_D$ is represented by the variable Φ_j^k and the flow leaving node j is denoted by Ψ_j^k. The exact definitions are given below:

$$\Phi_j^k = \sum_{(i,j)\in E_F^k} x_{ij}^k; \qquad \Psi_j^k = \sum_{(j,i)\in E_F^k} x_{ji}^k \tag{18}$$

The goal of this ILP-formulation is to transform a given set of machine instructions in order to minimize the number of clock cycles required for execution. The basic ILP-formulation for the problem of instruction scheduling with respect to resource constraints can then be given as follows:

- objective function

$$\min \quad M_{steps} \tag{19}$$

- constraints

  1. time constraints

     For no instruction the start time may exceed the maximal number of control steps M_{steps} (which is to be calculated)

$$t_j \leq M_{steps} \quad \forall\ j \in V_D \tag{20}$$

  2. precedence constraints

     When instruction j depends on instruction i, then j may be executed only after the execution of i is finished.

$$\begin{aligned} t_j - t_i &\geq w_i \quad \forall\ (i,j) \in E_D^{output}\ \cup\ E_D^{true} \\ t_j - t_i &\geq 0 \quad \forall\ (i,j) \in E_D^{anti} \end{aligned}$$

  3. flow conservation

     The value of the flow entering a node must equal the flow leaving that node.

$$\Phi_j^k - \Psi_j^k = 0 \quad \forall\ j \in V_D,\ \forall\ k \in V_k :\ (j,k) \in E_R \tag{21}$$

4. assignment constraints
   Each operation must be executed exactly once by one hardware component.

$$\sum_{\substack{k \in V_K : \\ (j,k) \in E_R}} \Phi_j^k \quad = \quad 1 \quad \forall\, j \in V_D \tag{22}$$

5. resource constraints
   The number of available resources of all resource types must not be exceeded.

$$\sum_{(k,j) \in E_F^k} x_{kj}^k \leq R_k \qquad \forall k \in V_K \tag{23}$$

6. serial constraints
   When operations i and j are both assigned to the same resource type k, then j must await the execution of i, when a component of resource type k is actually moved along the edge $(i,j) \in E_F^k$, i.e., if $x_{ij}^k = 1$.

$$t_j - t_i \geq z_i + (\sum_{\substack{k \in V_K : \\ (i,j) \in E_F^k}} x_{ij}^k - 1) \cdot \alpha_{ij} \; \forall (i,j) \in E_F^k \tag{24}$$

The better the feasible region of the relaxation P_F approximates the feasible region of the integral problem P_I, the more efficiently can the integer linear program be solved (see *Sec. 5.4.*). In [27], it is shown, that the tightest polyhedron is described by using the value $\alpha_{ij} = z_i - asap(j) + alap(i)$.

5.3. Comparison

The Manne-model is based on the disjunctive graph, the SILP-formulation on the resource flow graph. If we compare the two models, we can see that both graphs are basically equivalent. Each disjunctive edge corresponds to a pair of inversely directed edges in the resource flow graph. If there is a precedence relation between two instructions, only one edge is drawn between the corresponding nodes. So, the disjunctive graph corresponds to the resource flow graph augmented with the edges of the data dependence graph. Since the data dependence edges are only needed for the precedence constraints, they are not included in the resource flow graph.

Constraints (20) and (12) are basically also equivalent; the difference is that the SILP-formulation considers the starting time of each microoperation whereas Manne models the end time for the processing of the tasks. In both formulations, precedence constraints have to be respected. Inequalities (14) and (15) correspond to the flow conservation constraints (21) in connection with the serial constraints (24). So the tasks correspond to microoperations, the disjunctive edges to a pair of flow edges, the machines to the resource types and the makespan to the minimal execution time. As already mentioned in the previous section, the difference lies in the fact, that for the problem of instruction scheduling, only one "job" has to be considered; on the other hand, there can be several instances of each resource type.

5.4. Optimization of the model structure

Evidently, there are many alternatives for creating an ILP-formulation; yet it is very important to choose a formulation which is well-structured in order to be able to solve large instances of integer linear programming problems. The goal of integer linear programming is to find the integral point in a given feasible area, which minimizes or maximizes the considered objective function (see *Fig. 5*).

Figure 5. *Feasible areas*

More formally, *integer linear programming (ILP)* is the following optimization problem:

$$\begin{array}{rcl} \min \quad z_{IP} & = & c^T x \\ x & \in & P_F \cap \mathbb{Z}^n \end{array} \tag{25}$$

where

$$P_F = \{x \mid Ax \geq b,\ x \in \mathbb{R}^n_+\},\ c \in \mathbb{R}^n,\ b \in \mathbb{Z}^m,\ A \in \mathbb{Z}^{m \times n}$$

Since P_F is described only by equality and inequality constraints (no integrality constraints are required), any linear objective function over P_F can be optimized in polynomial time using linear programming algorithms. Unfortunately, in most cases, no representation of P_I as a system of linear equations is known; furthermore the number of inequality constraints required to describe the convex hull is usually extremely large [19]. Therefore, one can try to solve a related problem, called the *LP-relaxation* of the integer linear problem, which reads as follows:

$$\begin{array}{rcl} \min \quad z_R & = & c^T x \\ x & \in & P_F \end{array} \tag{26}$$

Since $P_I \subseteq P_F$, one can conclude from (25) and (26) that $z_R \leq z_{IP}$. If $P_F = P_I$, the polyhedron P_F is called integral and in this case, the equation $z_R = z_{IP}$ holds.

Thus, the optimal solution can be calculated in polynomial time by solving its LP-relaxation. Therefore, while formulating an integer linear program, one should attempt to find equality and inequality constraints such that P_F will be integral. It has been shown, that for every bounded system of rational inequalities there is an integer polyhedron [14], [23]. Unfortunately, for most problems it is not known how to formulate these additional inequalities—and there could be an exponential number of them [19].

In general, $P_I \subsetneqq P_F$, and the LP-relaxation provides a lower bound on the objective function. The efficiency of many integer programming algorithms depends on the tightness of this bound. The better P_F approximates the feasible region P_I, the sharper is the bound, so that for an efficient solution of an ILP-formulation, it is extremely important, that P_F is close to P_I. This can be achieved by developing tight descriptions of P_F that closely approximate P_I. Moreover, formal analysis can be used to determine new valid inequalities (the inequalities that arise due to the integrality of the variables), so that the formulations can be further tightened [7].

Both the SILP- and the OASIC-models try to explore these results in order to arrive at a well-structured formulation. The goal of the OASIC-approach is to formulate the integer linear program in a way that permits its transformation to a *node packing* graph, which has been partially characterized by its facets. Starting from a more intuitive formulation of the precedence constraints, additional valid inequalities could be developed and are actually subsumed in the given formulation of the precedence constraints. The resulting polytope is in general not identical with the integral polytope, but by taking into account the additional facets, a better approximation to the integral polytope is obtained. This is covered in detail in [13, 14]; an overview is given in [17, 18]. In the SILP-approach, additional valid linear inequalities have been added to the previously presented formulation in order to get a tighter approximation to the integral polytope. Moreover, in [27] it has been proven that the following value of the constant α in the serial constraints leads to the tightest possible polytope: $\alpha_{ij} = z_i - asap(j) + alap(i)$.

Similar optimizations have been performed by Liao and You in [20], where a slightly modified variant of the Manne-formulation is presented, which leads to a tigther approximation to the integral polytope. This is achieved by replacing constraints (14) and (15) by

$$z_{ij} = (t_j - t_i) + Tw_{ij} - p_j \qquad \forall\ k\ \forall\ (T_i, T_j) \in \mathcal{E}_k \tag{27}$$

$$z_{ij} \leq T - p_i - p_j \qquad \forall\ k\ \forall\ (T_i, T_j) \in \mathcal{E}_k \tag{28}$$

where $z_{ij} \geq 0$ is a non-negative slack variable.

In [3], the Manne model is extended by adding valid inequalities, e.g. based on cutting planes. Again the goal is to obtain better lower bounds in order to improve the performance of the used branch and bound algorithm.

6. Modelling of the register assignment

6.1. Extension of the OASIC model

Up to now, the presented ILP-formulation covers only the problem of instruction scheduling. In order to take into account the problem of register assignment with

respect to a homogeneous register set, the above presented formulation has to be extended by some additional constraints. It must be assured that in no control step more than R registers are used, so that there are at most R overlapping lifetimes.

A variable i is *defined* at a program point, when it is assigned a value; a variable is *used*, when it is referenced at a program point. For a given instruction sequence, the lifetime of a variable can be represented by a lifetime-defining edge $i \rightarrow j$ between the operation i, that defined the variable, and the operation j, that last used the variable. However, each variable can be used within more than one operation. Hence, lifetime-defining edges are possibly not unique, since the order of the operations is not fixed, when simultaneous instruction scheduling and register allocation is performed. Thus, in a naive approach a lifetime-defining edge will be inserted between a definition and each use. By means of transitivity analysis and of *asap-/alap*-analysis, the number of edges can be reduced. For more information, see [13, 14, 17].

In the constraints generated to take into account the problem of register allocation, the following terminology is used: An edge $i \prec j$ crosses control step n, if and only if $N(i) \cap \{0, 1, \ldots, n-(w_i-1)\} \neq \emptyset$ and $N(j) \cap \{n+1, n+2, \ldots, T\} \neq \emptyset$. The value $e_n(i)$ indicates the number of edges with head i crossing control step n; the set $M(n)$ represents the set of all maximal sets of edges $M'(n)$, which cross control step n and have unique heads.

$$\sum_{j_a \prec j_b \in M'(n)} \Big(\sum_{k \in V_K} \sum_{\substack{n_1 \leq n \\ n_1 \in N(j_a)}} x^k_{j_a n_1} + \sum_{k \in V_K} \sum_{\substack{n_2 > n \\ n_2 \in N(j_b)}} x^k_{j_b n_2} \quad - \quad \sum_{k \in V_K} \sum_{\substack{n_3 \leq n \\ n_3 \in N(j_b)}} x^k_{j_b n_3} - \sum_{k \in V_K} \sum_{\substack{n_4 > n \\ n_4 \in N(j_a)}} x^k_{j_a n_4} \Big) \quad \leq \quad 2 \cdot R \qquad (29)$$

$$\forall\, n \;\wedge\; \forall\, M'(n) \in M(n)$$

When e edges are crossing control step n and among these $e_n(i)$ have head i, while the rest of the edges has unique heads, inequality (29) has to be generated exactly $e_n(i)$ times for control step n. In the general case, the number of constraints to be generated for control step n is given by $\prod_i e_n(i)$. The register allocation constraint calculates two times the number of crossing edges for each control step. The relevant variables are partitioned into four groups and depending on their group they are used in the inequality either with positive or with negative sign.

As for the complexity of this ILP-formulation, the number of constraints is bound by $\mathcal{O}(n^3)$ when no register allocation is considered. The number of variables is $\mathcal{O}(n^2)$. Considering the problem of integrated instruction scheduling and register allocation, the worst case number of binary variables doesn't change, however, the number of constraints can grow exponentially due to the register crossing constraints (see [13, 14, 17]).

6.2. Extension of the SILP model

To take into account the problem of register assignment, the SILP formulation has to be modified, too. Again following the concept of flow graphs, the register assignment problem is formulated as register distribution problem.

Definition 3 (register flow graph) *The register flow graph $G_F^g = (V_F^g, E_F^g)$ is a directed graph with a set of nodes $V_F^g = V_g \cup G$ and a set of directed arcs E_F^g. The set G contains one resource node g for each available register set. A node $j \in V_g$ represents an operation performing a write access to a register, this way creating a variable with lifetime τ_j. Each arc $(i,j) \in E_F^g$ provides a possible flow of a register from i to j and is assigned a flow variable $x_{ij}^g \in \{0,1\}$. Then the same register is used to save the variables created by nodes i and j, if $x_{ij}^g = 1$.*

Lifetimes of variables are reflected by true dependences. When an instruction i writes to a register, then the life span of the value created by i has to reach all uses of that value. To model this, variables $b_{ij} \geq 0$ are introduced measuring the distance between a defining instruction i and a corresponding use j. The formulation of the precedence relation is replaced by the following equation:

$$t_j - t_i - b_{ij} = w_i \tag{30}$$

Then, for the lifetime of the register defined by instruction i:

$$\tau_i \geq b_{ij} + w_i \qquad \forall\ (i,j) \in E_D^{true} \tag{31}$$

An instruction j may only write to the same register as a preceding instruction i, if j is executed at a time when the life span of i, τ_i is already finished. In other words: If the variable produced by instruction i has lifetime τ_i and the output of instruction j is to be written into the same register (i.e. when $x_{ij}^g = 1$ holds), then $t_j - t_i \geq \tau_i$ must hold. This fact is caught by the following *register serial constraint*:

$$t_j - t_i \geq w_i - w_j + \tau_i + (x_{ij}^g - 1) \cdot 2T \tag{32}$$

Here, T represents the number of machine operations of the input program, which surely provides an upper bound for the maximal possible lifetime.

In order to correctly model the register flow graph, flow conservation constraints, as well as resource constraints and assignment constraints have to be added to the integer linear program. This leads to the following equalities and inequalities:

$$\Psi_g^g \leq R_g \tag{33}$$

$$\Phi_j^g = 1 \qquad \forall\ j \in V_g \tag{34}$$

$$\Phi_j^g - \Psi_j^g = 0 \qquad \forall\ j \in V_F^g \tag{35}$$

$$t_j - t_i \geq w_i - w_j + \tau_i + (x_{ij}^g - 1) \cdot 2T \qquad \forall (i,j) \in E_F^g \tag{36}$$

Moreover, the ILP-formuation can be tightened by an identification of redundant serial constraints and the insertion of valid inequalities; for further information see [27, 17].

Following [7], we will measure the complexity of an ILP-formulations in terms of the number of constraints and binary variables. The number of constraints is $\mathcal{O}(n^2)$, where n is the number of operations in the input program. The number of binary variables can be bound by $\mathcal{O}(n^2)$, however it's only the flow variables used in the serial constraints, that have to be specified as integers [27, 17].

7. ILP-based approximations

Experimental results have shown, that in spite of using well-structured formulations, the computation time for solving the ILP's for instruction scheduling and register assignment is high [17, 18]. Therefore, it is an interesting question to know, whether heuristics can be applied which cannot guarantee an optimal solution but can also deal with larger input programs. So, in the following we will give a short overview of some approximation algorithms for the SILP-formulation which have been developed and tested in [27, 17]. Note that no analytically evaluated accuracy can be guaranteed, so the use of the term "approximation" differs from the definition of "approximation algorithms" in [4].

7.1. Approximation by rounding

The basic idea of this approach is to solve only (partially) relaxed problems. Relaxed binary variables are fixed one by one to that value $\in \{0, 1\}$, which they would take presumably in an optimal solution. In the basic formulation, the SILP-approach requires only the flow variables appearing in the serial constraints to be specified as binary; these are forming the set M_S. Since these variables are multiplied by a large constant, one can assume, that a relaxed value close to 1 (0) indicates that the optimal value of that variable is also 1 (0) (see [27]).

First, the approximation algorithm replaces the integrality constraint $x \in \{0, 1\}$ for all $x \in M_S$ by the inequality $0 \leq x \leq 1$ and solves the resulting mixed integer linear program. After that, a non-integral variable $x \in M_S$ with smallest distance to an integer value is rounded to that value by adding an appropriate equation to the ILP-formulation. Then, the mixed integer linear program is solved again and the rounding step is repeated. It is possible that the rounding leads to an infeasible ILP—then the latest fixed variable is fixed to its complement. When the MILP is still unsolvable, an earlier decision was wrong. Then, in order to prevent the exponential cost of complete backtracking, integrality constraints are reintroduced. This is done by grouping the fixed binary variables by the distance they had to the next integral value before rounding and redeclaring them as binary beginning with those with the largest distance. It is clear, that in the worst case, the original problem has to be solved again.

Since only the variables $x \in M_S$ are relaxed, the calculation of the relaxations can take a long time; moreover due to backtracking and false rounding decisions, the computation time can be higher than with the original problem. The quality of the solution is worse than for the other approximations, so this approach cannot be considered promising.

7.2. Stepwise approximation

Again, the variables $x \in M_S$ are relaxed and the resulting MILP is solved. Then the following approach is repeated for all control steps, beginning with the first one. The algorithm checks whether any operations were scheduled to the actual control step in spite of a serial constraint formulated between them. Let M_S^c be the set of all variables corresponding to flow edges between such colliding operations with respect to the actual control step c. All $x \in M_S^c$ are declared binary and the resulting MILP

is solved. This enforces a sequentialisation of all microoperations which cannot be simultaneously executed in control step c and so cannot be combined to a valid instruction. Then, for all $x \in M_S^c$ which have solution value $x = 1$ this equation is added to the constraints of the MILP, so that these values are fixed. The integrality constraints for the $x \in M_S^c$ with value $x = 0$ are not needed any more and are removed. Then, the algorithm considers the next control step.

After considering each control step, it is still possible for some variables $x \in M_S$ to have non-integral values. Then the set of all $x \in M_S$ with non-integral value is determined iteratively, these variables are redeclared binary and the MILP is solved again. This is repeated until all variables have integral values.

This way, a feasible solution can always be obtained. Since for each control step optimal solutions with respect to arosen collisions are calculated, it can be expected that the resulting fixations also lead to a good global solution.

7.3. Isolated flow analysis

In this approach, only the flow variables $x \in M_S$ corresponding to a certain resource type $r \in R$ are declared as binary. The flow variables related to other resources are relaxed, i.e.

$$\begin{aligned} 0 \leq x \leq 1 \qquad & \forall\, x \ \in M_S \ \text{ mit } \ res(x) \neq r \\ x \in \{0,1\} \qquad & \forall\, x \ \in M_S \ \text{ mit } \ res(x) = r \end{aligned}$$

Then, an optimal solution of this MILP is calculated and the $x \in M_S$ executed by r are fixed to their actual solution value by additional equality constraints. This approach is repeated for all resource types, so a feasible solution is obtained in the end (see Fig. 6).

Figure 6. *Isolated flow analysis*

This way, in each step, an optimal solution with respect to each individual resource flow is calculated. Since the overall solution consists of individually optimal solutions of the different resource types, in most cases it will be equal to an optimal solution of the entire problem. This optimality, however, cannot be guaranteed,

as when analysing an individual resource flow, the others are only considered in their relaxed form. However, the computation time is reduced since only the binary variables associated to one resource type are considered at a time.

7.4. Stepwise approximation of isolated flow analysis

The last approximation developed for the SILP-Formulation is a mixture of the two previously presented approaches. At each step, the flow variables of all resources except the actually considered resource type r are relaxed; for the variables $x \in M_S$ with $res(x) = r$, the stepwise approximation is performed until all these variables are fixed to an integral value. Then the next resource type is considered. Clearly, this approximation is the fastest one, and experimental results have shown that the solutions provided by this approximation are as good as the results of the two previously presented approximations [17, 18].

7.5. Applicability and performance

The applicability of ILP-based approximations depends strongly on the chosen formulation. While for the SILP-formulation several efficient approximations could be developed, the OASIC-approach turned out not to be suited for approximations. The only approximation for this formulation is the rounding method which is not practicable due to the bad solution quality and the high calculation time. So, another criterion for chosing an ILP-formulation should be the applicability of approximative solution techniques. By using ILP-based approximations, the calculation time could be reduced considerably. Some problems whose exact solution took more than 24 hours could be calculated in several minutes; for further information, see again [17, 18]. In most cases, the solutions were optimal; however, in the general case, this optimality cannot be guaranteed.

8. Conclusion

In this article we have shown that there are apparent similarities between typical optimization problems of operations research and of compiler construction. These analogies were pointed out by opposing integer linear programming formulations for the job shop scheduling problem and the problem of instruction scheduling and register assignment.

The complexity of integer linear programming is high. Thus, it is important to choose a well-structured formulation. We have shortly explained the reason for this requirement and presented some important guidelines for optimizing an ILP-formulation.

Algorithms used in a compiler typically have to be fast, so that the calculation time for optimally solving large ILPs is not acceptable. We have presented several approximation algorithms which produce mostly optimal solutions in acceptable calculation time. While traditional graph-based heuristics suffer from the phase ordering problem, in this way a partial phase integration during code generation can be achieved.

References

[1] *Analog Devices. ADSP-2106x SHARC User's Manual*, 1995.

[2] *Analog Devices. ADSP-21000 Family C Tools Manual*, 1995.

[3] D. Applegate, W. Cook, *A computational study of the job sop scheduling problem*, ORSA Journal on Computing **3**(1991), 149–156.

[4] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, J. Weglarz, *Scheduling Computer and Manufacturing Processes*, Springer-Verlag, 1996.

[5] W. Brüggemann, *Ausgewählte Probleme der Produktionsplanung*, Physica Verlag, Heidelberg, 1995.

[6] G. J. Chaitin, *Register allocation and spilling via graph coloring*, in: Proc. SIGPLAN'82 Symp. on Compiler Construction, SIGPLAN Notices", vol. 17(6), 1982, pp. 201–207.

[7] S. Chaudhuri, R. A. Walker, J. E. Mitchell, *Analyzing and Exploiting the Structure of the Constraints in the ILP-Approach to the Scheduling Problem*, IEEE Transactions on Very Large Scale Integration (VLSI) System **2**(1994), 456–471.

[8] W. Dinkelbach, *Operations Research*, Springer, 1992.

[9] C. Ferdinand, C. H. Seidl, R. Wilhelm, *Tree automata for code selection*, Acta Informatica **31**(1994), 741–760.

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

[11] C. W. Fraser, D. R. Hanson, T. A. Proebsting, *Engineering a simple, efficient code-generator generator*, ACM Letters on Programming Languages and Systems **1**(1992), 213–226.

[12] M. Garey, D. S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, Freemann and Company, 1979.

[13] C. H. Gebotys, M. I. Elmasry, *Optimal VLSI Architectural Synthesis*, Kluwer Academic, 1992.

[14] C. H. Gebotys, M. I. Elmasry, *Global optimization approach for architectural synthesis*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **9**(1993), 1266–1278.

[15] R. Gupta, M. L. Soffa, *Region Scheduling: An Approach for Detecting and Redistributing Parallelism*, IEEE Transactions on Software Engineering **16**(1990), 421–431.

[16] C. A. Hax, *Hierarchical production planning*, in: Encyclopedia of Operations Research and Management Science, (S. I.Gass, and C. M. Harris, Eds.), Massachusetts Institute of Technology, Cambridge, editors = Gass, Saul I. and Harris, Carl M., Kluwer Academic Publishers, Dordrecht, 1996.

[17] D. KÄSTNER, *Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x*, Master's thesis, Universität des Saarlandes, 1997.

[18] D. KÄSTNER, M. LANGENBACH, *Integer Linear Programming vs. Graph-Based Methods in Code Generation*, Technical Report, Universität des Saarlandes, 1998.

[19] G. L. NEMHAUSER,, A. H. G. RINNOOY KAN, M. J. TODD, Eds., *Handbooks in Operations Research and Management Science*, North-Holland, Amsterdam; New York; Oxford, 1989.

[20] C. J. LIAO, C. T. YOU, *An improved formulation for the job-shop scheduling problem*, Journal of the Operational Research Society **43**(1992), 1047–1054.

[21] A. S. MANNE, *On the Job-Shop Scheduling Problem*, Operations Research **8**(1960), 210–223.

[22] A. NICOLAU, *Uniform Parallelism Exploitation in Ordinary Programs*, in: International Conference on Parallel Processing, IEEE Computer Society Press, 1985, pp. 614–618.

[23] C. H. PAPADIMITRIOU, K. STEIGLITZ, *Combinatorial Optimization, Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, 1982.

[24] C. N.POTTS, D. B. SHMOYS, D. P. WILLIAMSON, *Permutation vs. non-permutation flow shop schedules*, Operations Research Letters **10**(1991), 281–284.

[25] H. M. WAGNER, *An integer linear-programming model for machine scheduling*, Naval Research Logistics Quarterly **6**(1959), 131–140.

[26] R. WILHELM, D. MAURER, *Compiler Design*, Addison-Wesley, 1995.

[27] L. ZHANG, L., *SILP. Scheduling and Allocating with Integer Linear Programming*, PhD thesis, Technische Fakultät der Universität des Saarlandes, 1996.