Code Optimization by Integer Linear Programming

Daniel Kästner* and Marc Langenbach

Universität des Saarlandes, Fachbereich Informatik
Postfach 15 11 50, D-66041 Saarbrücken, Germany
Phone: +49 681 302 5589, Fax: +49 681 302 3065
{kaestner,mlangen}@cs.uni-sb.de
http://www.cs.uni-sb.de/~{kaestner,mlangen}

Abstract. The code quality of many high-level language compilers in the field of digital signal processing is not satisfactory. This is mostly due to the complexity of the code generation problem together with the irregularity of typical DSP architectures. Since digital signal processors mostly are traded on the high volume consumer market, they are subject to serious cost constraints. On the other hand, many embedded applications demand high performance capacities. Thus, it is very important that the features of the processor are exploited as efficiently as possible. By using integer linear programming (ILP), the deficiencies of the decoupling of different code generation phases can be removed, since it is possible to integrate instruction scheduling and register assignment in one homogeneous problem description. This way, optimal solutions can be found—albeit at the cost of high compilation times. Our experiments show, that approximations based on integer linear programming can provide a better solution quality than classical code generation algorithms in acceptable runtime for medium sized code sequences. The experiments were performed for a modern DSP, the Analog Devices ADSP-2106x.

1 Introduction

In the last decade, digital signal processors (DSPs) have established on the high-volume consumer market to be the processors of choice for embedded systems. The high-volume market imposes stringent cost constraints to the DSPs; on the other hand, many embedded applications demand high performance capacities. High-level language compilers often are unable to generate code meeting these requirements [25]. This is mostly due to the complexity of the code generation problem together with the irregularity of typical DSP architectures. Especially, the phase coupling problem between instruction scheduling and register allocation plays an important role.

Since instruction scheduling and register allocation are $\mathcal{NP}$-hard problems, they are mostly solved in separate phases by using heuristic methods. Classical

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

S. Jähnichen (Ed.): CC'99, LNCS 1575, pp. 122–137, 1999.
© Springer-Verlag Berlin Heidelberg 1999

heuristic methods are register allocation by heuristically guided graph coloring [5,6] or instruction scheduling by *list scheduling* [16], *trace scheduling* [8], *percolation scheduling* [21] or *region scheduling* [12]. These algorithms are very fast, but usually produce only suboptimal solutions without any information about the solution quality.

The task of instruction scheduling is to rearrange a code sequence in order to exploit instruction level parallelism. In register allocation, the values of variables and expressions of the intermediate representation are mapped to registers in order to minimize the number of memory references during program execution. As the goals of these two phases often conflict, that phase which is executed first imposes constraints on the other; this can lead to inefficient code. That problem is known as the *phase ordering problem*.

Formulations based on integer linear programming (ILP) offer the possibility of integrating instruction scheduling and aspects of register allocation in an homogeneous problem description and of solving them together. Moreover, it is possible to get an optimal solution of the considered problems—albeit at the cost of high calculation times. We have shown that by using ILP-based approximations, the computation time can be significantly reduced. The resulting code quality is better than that of conventional graph-based algorithms. Moreover, with integer linear programming, lower bounds on the optimal schedule length can be calculated. This way, the quality of an approximate solution can be estimated, if no optimal solution is available.

The paper is organized as follows: In Section 2, we will give a short overview on related work. After an introduction to integer linear programming, we will present an ILP-formulation for combined instruction scheduling and register assignment. In Section 5, some additional constraints are introduced which are required to adapt the formulation to a real-world target architecture, the ADSP-2106x. Then we will give an overview on some ILP-based approximations in Section 6. The article concludes with a short summary and an outline of future work.

2 Related Work

During the last years, the development of phase coupling code generation strategies has gained increasing attention. In [4], Bradlee has developed a code generation policy where instruction scheduling and register allocation communicate with each other. First, a pre-scheduler is invoked which computes schedule cost estimates which allow the subsequent register allocation phase to quantify the effect of its choices on the scheduler. After the allocation pase, the final schedule is produced.

The AVIV retargetable code generator [13] builds on the retargetable code generation framework SPAM for digital signal processors [26,27]. It uses a heuristic branch-and-bound algorithm that performs functional unit assignment, operation grouping, register bank allocation, and scheduling concurrently. Register allocation proper is carried out as a second step. Bashford and Bieker [3] are de-

veloping a framework for scheduling, compaction, and binding using constraint logic programming. Both approaches are still work in progress, so final results are not available yet.

There have been only few approaches to incorporate ILP-based methods into the code generation process of a compiler. An approach for ILP-based instruction scheduling for vector processors has been presented in [2]. Wilson et al. [28] use an ILP-formulation for simultaneously performing scheduling, allocation, binding, and selection among different code alternatives. However the complexity of the resulting formulations leads to very high computation times. Leupers has developed a retargetable compiler for digital signal processors [18] where local compaction is performed by integer linear programming. However the formulation captures only the problem of instruction scheduling and no approximations or partitioning techniques are considered. Other ILP-based approaches have been developed in the context of software pipelining [24,11].

3 Basics of Integer Linear Programming

In integer programming problems, an objective function is maximized or minimized subject to inequality and equality constraints and integrality restrictions on some or all of the variables. The calculation of an optimal solution of an integer linear program is $\mathcal{NP}$-hard; yet many large instances of such problems can be solved. This, however, requires the selection of a structured formulation and no ad-hoc approach [7].

In this paper, we will just sketch the basics of integer linear programming, which are essential for the understanding of the presented ILP-approaches. For further information see e.g. [20], [19], [22], or [7].
Let $P_F = \{x \mid Ax \geq b,\ x \in \mathbb{R}^n_+\},\ c \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$. Then Integer linear programming (ILP) is the following optimization problem:

$$\begin{aligned} \min \quad & z_{IP} = c^T x \\ & x \in P_F \cap \mathbb{Z}^n \end{aligned} \tag{1}$$

The set P_F is called *feasible region.* If some of the variables have to be integral while the others also can take real values, the problem is called *mixed integer linear problem (MILP).* The feasible area P_F is called integral, if it is equal to the convex hull P_I of the integer points ($P_I = conv(\{x \mid x \in P_F \cap \mathbb{Z}^n\})$; see Fig. 1). In this case, the optimal solution can be calculated in polynomial time by solving its LP-relaxation. This means, that linear programming algorithms can be used, since the solution of the (non-integer) linear program is guaranteed to be integral. 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 [10,23]. Unfortunately for most problems it is not known how to formulate these additional inequalities—and there could be an exponential number of them [19].

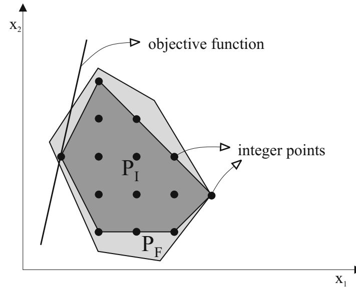

Fig. 1. Feasible Areas.

In general, $P_I \subsetneq P_F$, 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.

4 The ILP Model

In this section, the problem of instruction scheduling is formally introduced. An ILP formulation for instruction scheduling is presented and is extended to include the problem of register assignment. These formulations work on basic-block level; in [14] it is shown how they can be extended to programs with structured control flow.

4.1 Instruction Scheduling

Basic Definitions
Let a sequence of partially ordered microoperations be given. Then the task of instruction scheduling is to find a schedule which minimizes the execution time of the instruction sequence and respects its partial order. This partial order among the instructions is induced by the data dependences. If a microoperation i is data dependent of another microoperation j, then the ordering of i and j must not be changed; otherwise the semantics of the program would be changed. The data dependences are modelled by the data dependence graph $G_D = (V_D, E_D)$ whose nodes correspond to the microoperations of the input program and whose edges reflect dependences between the adjacent nodes. There are three different types of data dependences:

- true dependence: i defines a resource which is used by j $((i,j) \in E_D^{true})$
- output dependence: i defines a resource which is also defined by j $((i,j) \in E_D^{output})$
- anti dependence: i uses a resource which is defined by j $((i,j) \in E_D^{anti})$

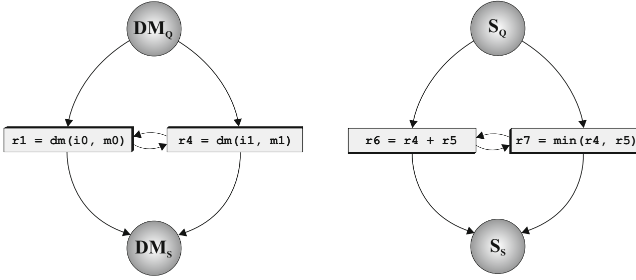

Fig. 2. Resource Flow Graph for two Instructions Executed on an ALU and the Data Memory.

Each operation of the input program can be executed by a certain resource type. In order to describe the mapping of instructions to hardware resources, the resource graph G_R is used [29]. G_R is a bipartite directed graph $G_R = (V_R, E_R)$, where $(j, k) \in E_R$ means that instruction j can be executed by the resources of type k.

An ILP-Formulation for Instruction Scheduling

In the area of architectural synthesis, several ILP-formulations have been developed for the problem of instruction scheduling and resource allocation. We have investigated two well-structured formulations in detail: OASIC [9,10], which is a time-based formulation[1] and SILP [29,14], which is an order-based formulation.

In the scope of this paper, we will concentrate on SILP (*Scheduling and Allocating with Integer Linear Programming*); an investigation of OASIC and a comparison of both models can be found in [14,15]. First we will give an overview of the SILP-terminology:

- The variable t_i indicates the relative position of a microoperation within the instructions of the optimized code sequence; the t_i-values have to be integral.
- w_j describes the execution time of instruction $j \in V_D$.
- z_j denotes the latency of the functional unit executing operation 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.
- τ_j describes the length of the life range of a variable created by operation j.

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

[1] In a *time-based* ILP-formulation the choice of the decision variables is based on the time the modelled event is assigned to. In an *order-based* formulation, the decision variables reflect the ordering of the modelled events.

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. Fig. 2 shows an example resource flow graph for two resource types of our target processor ADSP-2106x (see Sec. 5); on each resource type, two independent instructions are executed. The *resource flow graph* G_F is a directed graph $G_F = (V_F, E_F)$ with $V_F = \bigcup_{k \in V_K} V_F^k$ and $E_F = \bigcup_{k \in V_K} E_F^k$. The set V_F^k contains the resource nodes for resource type k and all operations of the input program which are executed by k. E_F^k is the set of edges connecting nodes in V_F^k. 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 $x_{ij}^k = 1$.

The goal of this ILP-formulation is to minimize the execution time of the code sequence to be scheduled. The execution time is measured in control steps (clock cycles). The ILP-formulation for the problem of instruction scheduling reads as follows:

$$\min \quad M_{steps} \tag{2}$$

$$t_j \leq M_{steps} \quad \forall\, j \in V_D \tag{3}$$

$$t_j - t_i \geq w_i \quad \forall\, (i,j) \in E_D^{output} \cup E_D^{true} \tag{4}$$

$$t_j - t_i \geq 0 \quad \forall\, (i,j) \in E_D^{anti} \tag{5}$$

$$\sum_{(i,j) \in E_F^k} x_{ij}^k - \sum_{(j,i) \in E_F^k} x_{ji}^k = 0 \quad \forall\, j \in V_D,\ \forall\, k \in V_k :\ (j,k) \in E_R \tag{6}$$

$$\sum_{\substack{k \in V_K : \\ (j,k) \in E_R}} \sum_{(i,j) \in E_F^k} x_{ij}^k = 1 \quad \forall\, j \in V_D \tag{7}$$

$$\sum_{(k,j) \in E_F^k} x_{kj}^k \leq R_k \quad \forall k \in V_K \tag{8}$$

$$t_j - t_i \geq z_i + \Big(\sum_{\substack{k \in V_K : \\ (i,j) \in E_F^k}} x_{ij}^k - 1 \Big) \cdot \alpha_{ij} \quad \forall (i,j) \in E_F^k \tag{9}$$

The time constraints (equation (3)) guarantee, that for no instruction the start time may exceed the maximal number of control steps M_{steps} (which is to be calculated). Equations (4) and (5) are precedence constraints which are used to model the data dependences. When instruction j depends on instruction i, then j may be executed only after the execution of i is finished. The flow conservation constraints (equation (6)) assert that the value of the flow entering a node equal the flow leaving that node. Moreover, each operation must be executed exactly once by one hardware component. This is guaranteed by equation (7). The Resource constraints (8) are necessary, since the number of available resources of all resource types must not be exceeded. The constraints (9) are called 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 $x_{ij}^k = 1$. 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. In [29], it is shown that the tightest polyhedron is described by using the value $\alpha_{ij} = z_i - asap(j) + alap(i)$.

4.2 Integration of Register Asignment

Up to now, the presented ILP-formulation covers only the problem of instruction scheduling. To take into account the problem of register assignment, this formulation has to be extended. Register assignment is a subtask of register allocation. The goal is to determine the physical register which is used to store a value that has been previously selected to reside in a register. The choice of these registers interacts with the reordering facilities of instruction scheduling.

Again following the concept of flow graphs, the register assignment problem is formulated as register distribution problem. The register flow graph $G_F^g = (V_F^g, E_F^g)$ is a directed graph. The set $V_F^g = V_g \cup G$ is composed of two subsets: $G = \{g\}$ represents a homogeneous register set and the nodes in V_g represent operations performing a write access to a register. Each node $j \in V_g$ is associated with the generated variable whose lifetime is denoted by τ_j. Each arc $(i, j) \in E_F^g$ represents a possible flow of a register from i to j and is mapped 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 associated with true dependences. If 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, additional variables $b_{ij} \geq 0$ are introduced which measure the distance between a defining instruction i and a corresponding use j. The formulation of the precedence relation is replaced by the following equation:

$$t_j - t_i - b_{ij} = w_i \tag{10}$$

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

$$\tau_i \geq b_{ij} + w_i \qquad \forall\ (i, j) \in E_D^{true} \tag{11}$$

An instruction j may only write to the same register as a preceding instruction i, if j is executed at a time when the lifetime of i, τ_i is already terminated. This is modelled by the *register serial constraints*:

$$t_j - t_i \geq w_i - w_j + \tau_i + (x_{ij}^g - 1) \cdot 2T \tag{12}$$

Here, T represents the number of machine operations of the input program, which is a safe 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:

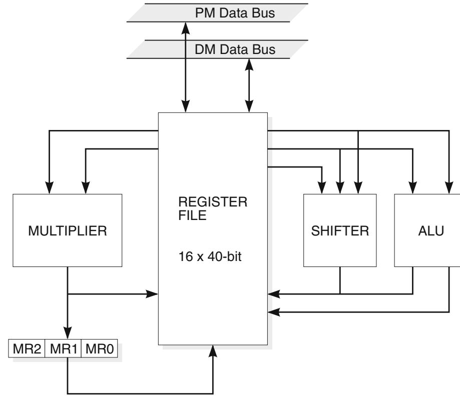

Fig. 3. Simplified Block Diagram.

$$\sum_{(g,j)\in E_F^g} x_{gj}^g \leq R_g \qquad \forall g \in G \tag{13}$$

$$\sum_{(i,j)\in E_F^g} x_{ij}^g = 1 \qquad \forall\, j \in V_g \tag{14}$$

$$\sum_{(i,j)\in E_F^g} x_{ij}^g - \sum_{(j,i)\in E_F^g} x_{j,i}^g = 0 \qquad \forall\, j \in V_F^g \quad \forall g \in G \tag{15}$$

$$t_j - t_i - \tau_i \geq w_i - w_j + (\sum_{g\in G} x_{ij}^g - 1)\cdot 2T \quad \forall (i,j) \in E_F^g \tag{16}$$

The total number of constraints is $\mathcal{O}(n^2)$, where n is the number of operations in the input program. The number of binary variables is bounded by $\mathcal{O}(n^2)$. The proofs are given in [29,14].

The ILP-formulation as presented here can model only sequential code. However, it is possible to integrate the control structure of the input program into an ILP, so that the movement of instructions across basic block boundaries can be handled internally by the ILPs. This is covered in detail in [15].

5 Adaptation to the ADSP-2106x

We have adapted the investigated formulations to a modern 32-bit digital signal processor with a load/store architecture, the ADSP-2106x SHARC (*Super Harvard Architecture Computer*) [1]. The processor contains three functional units: an ALU, a shifter, and a multiplier. The memory consists of a data memory DM and a program memory PM which can be used to store instructions and data (see Fig. 3). Most arithmetic operations can be executed in parallel with a data memory and a program memory access and in some cases, also the ALU and the multiplier can operate in parallel.

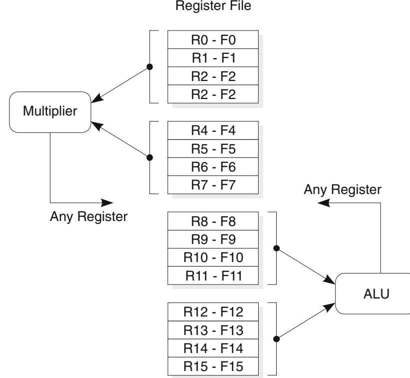

Fig. 4. Register Groups and Usage in Multifunctional Instructions.

The register file consists of two sets of sixteen 40-bit registers, which are used to store both fixed and floating point data. Furthermore, each set is divided into four groups of four consecutive registers. ALU and multiplier can only operate in parallel if the operands come from the appropriate register group (Fig. 4).

The execution of instructions is pipelined. In sequential program flow, when one instruction is being fetched, the instruction fetched in the previous cycle is being decoded, and the instruction fetched two cycles before is being executed. Thus, the throughput is one instruction per cycle.

5.1 Prevention of Incorrect Parallelism

In the presented ILP-formulation, parallel execution of instructions assigned to the same resource type is excluded by the serial constraints. Instructions assigned to different resource nodes can always be executed in parallel. As this parallelism is restricted in the considered architecture, additional constraints are required which explicitly prohibit the parallel execution of a certain pair of operations.

For two operations i and j, which must not be executed in parallel, i.e. for which $t_i \neq t_j$ must hold, constraints are formulated which represent the disjunction $(t_i > t_j) \vee (t_i < t_j)$. Let I denote the number of operations in the input program; then the following inequalities are required:

$$t_i - t_j > -v_{ij}T \tag{17}$$
$$t_i - t_j < (1 - v_{ij})T \tag{18}$$
$$v_{ij} \in \{0, 1\} \tag{19}$$
$$T = 2I + 1 \tag{20}$$

A correctness proof is provided in [14].

5.2 Irregular Register Sets

The operands of multifunction-instructions using ALU and multiplier are restricted to a set of four registers whithin the register file (see Fig. 3). Thus,

there are four different register groups to be considered and no homogeneous register set. For each such group, an own register node is inserted into the register flow graph ($G = \{g_1, g_2, g_3, g_4\}$).

When instructions i and j are combined to form a multifunction-instruction, so that for the reaching definition m, the target register set is restricted to exactly one $g \in G$, it must be guaranteed that m in fact uses a register of register set g. Then, a constraint of the form $\sum_{(i,m) \in E_F^g} x_{im}^g \geq 1$ must hold. Since $\sum_g \sum_{(i,m) \in E_F^g} x_{im}^g = 1$, this automatically excludes the use of other register sets. The formulation presented below uses two binary variables p_{ij} and q_{ij} which are defined by following constraints.

$$t_i - t_j \geq -p_{ij} T \quad (21)$$

$$t_i - t_j \leq q_{ij} T \quad (22)$$

$$p_{ij} + q_{ij} = 1 \quad (23)$$

where $T = 2I + 1$. Using these values, the register constraints can be formulated as follows:

$$\sum_{(i,m) \in E_F^g} x_{im}^g \geq 1 - (t_i - t_j) - p_{ij} T \quad (24)$$

$$\sum_{(i,m) \in E_F^g} x_{im}^g \geq 1 + (t_i - t_j) - q_{ij} T \quad (25)$$

The correctness proofs are omitted in this paper; they are explicitly given in [14].

6 Approximations

The computation time required to solve the generated ILPs is high. 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. In this paper, we give an overview of the investigated approximation algorithms; they are treated in detail in [14].

6.1 Approximation by Rounding

Approximation by rounding is a straightforward approach: the flow variables x_{ij}^k are relaxed[2] and the resulting relaxation is solved. Then the variable with a non-integral value closest to 0 or 1 is fixed to that integer and the new mixed integer linear program is solved. This is repeated until an integral solution has been found. However, the solution quality is not convincing enough for this method to be considered promising; moreover the calculation time can be high since usually backtracking steps are required.

[2] This means that the integrality constraint $x_{ij}^k \in \{0, 1\}$ is replaced by $0 \leq x_{ij}^k \leq 1$.

6.2 Stepwise Approximation

The algorithm starts by solving the MILP obtained by relaxing all flow variables. Then the following approach is repeated for all control steps. The algorithm checks whether any operations were scheduled to the actual control step in spite of a serial constraint formulated between them and the corresponding variables are redeclared binary. Let M_S^c be the set of these variables. After solving the resulting MILP, the variables $x \in M_S^c$ with $x = 1$ are fixed to their actual value and the next iteration starts. After considering each control step, the set of all flow variables which still have non-integral values is determined. These variables are redeclared binary and the MILP is solved again. This is repeated until a feasible solution has been found. Since in each step optimal solutions with respect to the already fixed variables are calculated, it can be expected that the approximation leads to a good global solution. This is confirmed by the test results.

6.3 Isolated Flow Analysis

In this approach, only the flow variables corresponding to a certain resource type $r \in R$ are declared as binary. The flow variables related to other resources are relaxed. Then, an optimal solution of this MILP is calculated and the flow variables x 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. 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.

6.4 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 flow variables x 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 in our experimental results, the solutions provided by this approximation are as good as the results of the two previously presented approximations. In the following, we denote this approximation by $\mathcal{SF}$.

7 Implementation and Experiments

In our experiments, we use ADSP-2106xassembler programs as input. These programs can be generated by the `gcc`-based compiler `g21k`, shipped with the

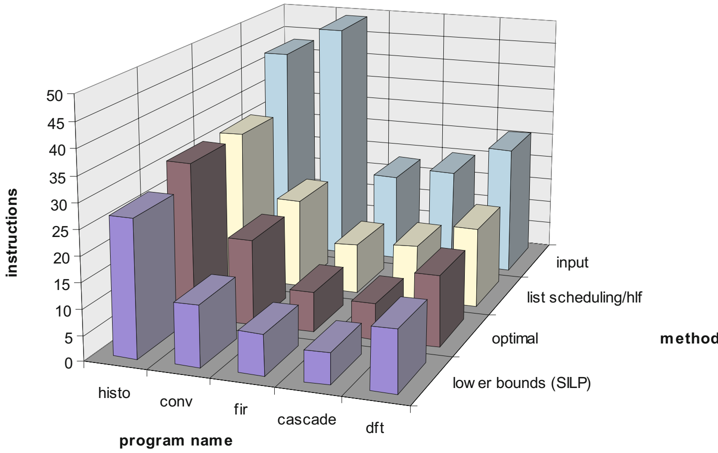

Fig. 5. Comparison of Solution Quality for Different Methods.

ADSP-2106x or can be written by hand. We have chosen as input programs typical applications of digital signal processing: a finite impulse response filter (`fir`), an infinite impulse response filter (`cascade`), a discrete fourier transformation (`dft`), one function of the whetstone-suite (`whetp3`), a histogramm (`histo`), and a convolution algorithm (`conv`). The input programs make no use of the available instruction-level parallelism of the processor. The run times have been measured on a SPARC ULTRA 2x200 with 1024 MB RAM under Solaris 2.5.1.

An overview on the experimental results is given in Fig. 5. The input programs contain between 18 and 49 instructions. In order to oppose the ILP-based techniques to classical algorithms, we have implemented several local scheduling algorithms and a graph-based register allocator. The graph-based algorithm which gave the best results was list scheduling with highest level first heuristic [14,17]. Even with an optimal register assignment given, the produced code sequences contained on average 13 % more instructions than the optimal code sequences; the solution time was less than one second for each of the input programs. The investigated SILP approximations however produced optimal code for each input program; the only exception was `whetp3`. ILP-based lower bounds could be calculated within several seconds by solving the LP-relaxation of the given problems. These bounds are on average 15 % below the optimal instruction number. The most important characteristics of the ILP-based algorithms are shown in Tables 1 and 2. For reasons of complexity, only instruction scheduling has been performed for the input programs `whetp3, histo` and `conv`; integrated instruction scheduling and register assignment has been performed for `fir, cascade` and `dft` (entry "is+ra" in column "mode"). The generated ILPs contain between 383 and 2143 constraints, between 93 and 946 binary variables and the textual representation of the largest one takes 294.59 KB. As can be seen in Table 2, even for programs whose exact solution (opt) took more than 24 hours, an ILP-based approximation could be calculated in less than 2 minutes

name	mode	constr	bin (expl)	size [KB]
fir	is+ra	422	675 (464)	58.94
cascade	is+ra	606	950 (707)	89.37
dft	is+ra	2143	1201 (946)	294.59
whetp3	is	383	214 (93)	15.26
histo	is	716	434 (168)	29.66
conv	is	963	452 (163)	40.09

Table 1. Characteristics of the ILPs generated by the SILP-based formulation. (all figures refer to the fastest approximation $\mathcal{SF}$). With exception of `whetp3`, all solutions provided by the approximation were in fact optimal.

name	mode	method	instr	CPU-time
fir	is+ra	def	8	1.69 sec
fir	is+ra	app	8	19.58 sec
cascade	is+ra	def	-	> 24 h
cascade	is+ra	app	7	86.72 sec
dft	is+ra	def	-	> 24 h
dft	is+ra	app	14	9 min 20 sec
whetp3	is	def	20	2h 4 min
whetp3	is	app	21	85.96 sec
histo	is	def	-	> 24 h
histo	is	app	31	1 h 2 min
conv	is	def	17	2 h 1 min
conv	is	app	17	53.66 sec

Table 2. Runtime characteristics of the SILP-based ILPs.

8 Ongoing and Future Work

Even by using ILP-based approximations, the computation time can grow high for large input programs. Thus, when compiling large programs, ILP-based solutions cannot replace the classical methods. However it is profitable to integrate ILP-based methods into conventional algorithms where an ILP-based optimization is only performed for code sequences offering a high degree of potential parallelism and whose efficiency is critical for the application. A typical example are inner loops. Moreover, partitioning techniques have to be developed which allow also for larger code sequences to be optimized by integer linear programming. Up to now, only one target architecture has been considered. However, in the field of digital signal processing, retargetability is an important issue. Currently, a retargetable framework for the presented optimization techniques, called PROPAN (Postpass Retargetable Optimizer and Analyzer) is developed. The goal is to make the described optimizations generic, so that they can be

adapted to a new target processor by a concise specification of the important hardware features.

9 Conclusions

We have shown that the the problem of instruction scheduling and register assignment can be modeled completely and correctly as an integer linear program for an irregular target architecture.

Based on a well-structured ILP-formulation, SILP, several approximations can be calculated while solving the ILPs, leading to good results in relatively low calculation times. The optimality of the result is not guaranteed by such heuristics; yet better results can be obtained than with the conventional, graph-based algorithms [17].

In conventional, graph-based algorithms, it is not possible to estimate the quality of a solution. By solving partial relaxations of the ILP, lower bounds to the optimal solution can be calculated. For the tested programs, the quality of these lower bounds corresponds to the quality of solutions which are calculated by conventional, graph-based algorithms. Thus, it is possible to give an interval which safely contains the optimal solution and to obtain an estimate for the quality of an approximate solution. This holds even when the optimal solution cannot be calculated for reasons of complexity.

References

1. Analog Devices. *ADSP-2106x SHARC User's Manual*, 1995. 129
2. Siamak Arya. An Optimal Instruction Scheduling Model for a Class of Vector Processors. *IEEE Transactions on Computers*, C-34, November 1985. 124
3. Ulrich Bieker and Steven Bashford. Scheduling, Compaction and Binding in a Retargetable Code Generator Using Constraint Logic Programming. *4. GI/ITG/GME Workshop "Methoden des Entwurfs und der Verifikation digitaler Systeme"*, March 1996. Kreischa, Germany. 123
4. D. G. Bradlee. Retargetable Instruction Scheduling for Pipelined Processors. Phd thesis, Technical Report 91-08-07, University of Washington, 1991. 123
5. P. Briggs, K. Cooper, and L. Torczon. Improvements to Graph Coloring Register Allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428 – 455, 1994. 123
6. D. Callahan and B. Koblenz. Register Allocation via Hierarchical Graph Coloring. *Proceedings of the ACM PLDI Conference*, pages 192 – 202, 1991. 123
7. S. Chaudhuri, R.A. Walker, and 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(4):456 – 471, December 1994. 124
8. J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478 – 490, July 1981. 123
9. C. H. Gebotys and M.I. Elmasry. *Optimal VLSI Architectural Synthesis*. Kluwer Academic, 1992. 126

10. C. H. Gebotys and M.I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-12(9):1266 – 1278, September 1993. 124, 126
11. R. Govindarajan, Erik R. Altman, and Guang R. Gao. A Framework for Resource Constrained Rate Optimal Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 7(11), November 1996. 124
12. Rajiv Gupta and Mary Lou Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, 1990. 123
13. Silvina Hanono and Srinivas Devadas. Instruction Scheduling, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the DAC 1998*, San Francisco, California, 1998. ACM. 123
14. Daniel Kästner. Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x. Master's thesis, University of the Saarland, 1997. 125, 126, 129, 130, 131, 133
15. Daniel Kästner and Marc Langenbach. Integer Linear Programming vs. Graph Based Methods in Code Generation. Technical Report A/01/98, University of the Saarland, Saarbrücken, Germany, January 1998. 126, 129
16. David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallet. Local Microcode Compaction Techniques. *ACM Computing Surveys*, 12(3):261–294, 1980. 123
17. Marc Langenbach. Instruktionsanordnung unter Verwendung graphbasierter Algorithmen für den digitalen Signalprozessor ADSP-2106x. Master's thesis, University of the Saarland, 1997. 133, 135
18. Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997. 124
19. G.L. Nemhauser, A.H.G. Rinnooy Kan, and M.J. Todd, editors. *Handbooks in Operations Research and Management Science*, volume 1 of *Handbooks in Operations Research and Management Science*. North-Holland, Amsterdam; New York; Oxford, 1989. 124
20. G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988. 124
21. Alexandru Nicolau. Uniform parallelism exploitation in ordinary programs. In *International Conference on Parallel Processing*, pages 614–618. IEEE Computer Society Press, August 1985. 123
22. C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, 1982. 124
23. C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*, chapter 13, pages 318 – 322. Prentice-Hall, Englewood Cliffs, 1982. 124
24. John Ruttenberg, G.R. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI 96)*, 31(5):1 – 11, May 1996. 124
25. M.A.R. Saghir, P. Chow, and C.G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *http://www.eecg.toronto.edu/~saghir/papers/asplos7.ps*, 1996. 122
26. SPAM Research Group, http://www.ee.princeton.edu/spam. *SPAM Compiler User's Manual*, September 1997. 123

27. Ashok Sudarsanam. *Code Optimization Libraries for Retargetable Compilation for Embedded Digital Signal Processors*. PhD thesis, University of Princeton, November 1998. 123
28. Tom Wilson, Gary Grewal, Shawn Henshall, and Dilip Banerji. An ILP-Based Approach to Code Generation. In Peter Marwedel and Gert Goossens, editors, *Code Generation for Embedded Processors*, chapter 6, pages 103–118. Kluwer, Boston; London; Dortrecht, 1995. 124
29. L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, University of the Saarland, Technical Faculty, 1996. 126, 128, 129