Integer Linear Programming vs. Graph-Based Methods in Code Generation

Daniel Kästner
kaestner@cs.uni-sb.de

Marc Langenbach
mlangen@cs.uni-sb.de

February 17, 1998

Abstract

A common characteristic of many embedded applications is that they are aimed at the high-volume consumer market, which is extremely cost-sensitive. However many of them impose stringent performance demands on the underlying system. Therefore, the code generation must take into account the restrictions and features given by the target architecture while satisfying these performance demands. High-level language compilers often are unable to generate code meeting these requirements. One reason is the phase coupling problem between instruction scheduling and register allocation. Many compilers perform these tasks separately with each phase ignorant of the requirements of the other. Commonly, each task is accomplished by using heuristic methods. As the goals of the two phases often conflict, whichever phase is performed first imposes constraints on the other, sometimes producing inefficient code. Integer linear programming (ILP) provides an integrated approach to the combined instruction scheduling and register allocation problem. This way, optimal solutions can be found—albeit at the cost of high compilation times. In our experiments, we considered as target processor the 32-bit DSP ADSP-2106x. We have examined two different ILP formulations and compared them with conventional approaches including list scheduling and the critical path method. Moreover, we have investigated approximations based on the ILP formulations; this way, compilation time can be reduced considerably while still producing near-optimal results. From the results of our implementation, we have concluded that integrating ILP formulations in conventional global algorithms is a promising method for generating high-quality code.

1 Introduction

In the last decade, digital signal processors (DSP's) have emerged as the processors of choice for implementing embedded systems for the high-volume consumer market. The placement on the high-volume market leads to a constraint of low prices; on the other hand, many embedded applications impose stringent performance demands on the underlying system. High-level language compilers often are unable to generate code meeting these requirements [SCL96]. Much of the research for optimizing compilers has concentrated on general purpose processors or machine independent optimizations so that special hardware features of typical DSP's as, e.g., irregular register sets and dual banked memory are not efficiently used. Another reason is the complexity of code generation itself; here, the phase coupling problem between instruction scheduling and register allocation plays an important role.

To provide an insight into the underlying problem, the basics of code generation are sketched in this section, following the notions in [WM97] and [Bas95]. A compiler takes an input-program and, performing syntactic and semantical analysis, transforms it into an intermediate representation. Subsequently, code generation is performed producing a semantically equivalent program in the target language, the target machine's instruction set. The task of code generation is composed of three subtasks: code selection, instruction scheduling and register allocation. Since all these subtasks represent complex problems—in fact instruction scheduling and register allocation are $\mathcal{NP}$-hard—many compilers perform code generation in three largely independent phases with each subproblem solved by using heuristic methods.

- The task of code selection is to generate a semantically equivalent target machine program for an intermediate-language program. In the worst case, code selection can be $\mathcal{NP}$-hard too, but at least for RISC-processors, the problem is easier because of the simpler instruction set architecture.
- The goal of register allocation is to map values of variables and registers of the intermediate representation to physical registers in order to minimize the number of memory references during program execution. Register allocation itself consists of two subtasks:
  - In general, the number of simultaneously live variables exceeds the number of physical registers. In order to minimize data transfer to and from memory, the allocator has to decide which values are to be held in registers. In the context of distributed register sets, it is a task of increasing importance to map values to certain register sets.
  - After the allocation, the physical registers have to be determined, in which the values are to reside. This subtask is called register assignment.
- Instruction scheduling: the instruction sequence selected by the code selector is to be reordered in order to efficiently exploit the parallel-processing capabilities and the instruction pipeline of the target machine.

Since code selection is easier to perform for RISC processors than for CISCs, the importance of interaction between register allocation and code selection decreases. Instead, the interaction between register allocation and instruction scheduling becomes increasingly important. In the context of instruction scheduling, registers are used to exploit instruction level parallelism; when register allocation is performed, they are required to reduce memory accesses. As the goals of these two phases often conflict, whichever phase is executed first imposes constraints on the other, sometimes resulting in inefficient code. This is called the **phase ordering problem**. When register allocation is performed before instruction scheduling, it can limit the reordering capabilities of the scheduler by assigning the same physical register to independent intermediate values. This prevents the corresponding operations from being overlapped by the scheduler. When instruction scheduling precedes register allocation, the number of simultaneously alive values can be increased so much that many of these values have to be spilled to main memory.

Over the years, several heuristic methods have been developed for the problem of instruction scheduling, e.g. *list scheduling* [LDSM80], *percolation scheduling* [Nic85] or *region scheduling* [GS90], which use a graph-based representation of the program. While being very fast, these classical heuristic methods have the disadvantage of only being able to find approximative, suboptimal solutions without any information about the quality of the solution.

Formulations based on integer linear programming (ILP) offer the possibility of integrating instruction scheduling and aspects of register allocation in a homogeneous problem description and of solving them together. Moreover, it is possible to get an optimal solution of the problem of instruction scheduling and register allocation—albeit at the cost of high calculation times. Another feature is the ability to provide lower bounds on the optimal schedule length. Since graph-based heuristic approaches always calculate an upper bound, the quality of an approximative solution can be estimated, even if no exact solution could be obtained due to the complexity of the problem. We have investigated two such ILP-formulations, OASIC [GE92, GE93] and SILP [Zha96], developed in the area of architectural synthesis.

The paper is organized as follows: In the second section we describe our target architecture, the digital signal processor ADSP-2106x. After a short presentation of the most important intermediate representations in Section 3, an overview of conventional graph-based algorithms for instruction scheduling and register allocation is given in Sections 4 and 5. The basics of integer linear programming are sketched in Section 6, followed by an overview of the investigated ILP-formulations SILP and OASIC. Then extensions to these models required to cope with global analyses and with the architectural features of the ADSP-2106x are outlined. The rest of Section 6 deals with some approximation algorithms based on ILP-formulations. These allow for nearly-optimal solutions to be obtained in considerably shorter computation times. Our implementation is described and experimental results are given in Section 7; Section 8 concludes and provides an outlook.

2 Architecture

In the scope of our paper, we are considering as target-architecture a 32 bit digital signal processor with load/store architecture, the ADSP-2106x (*super harvard architecture computer*) [Ana96, Ana95c, Ana91, Ana95a, Ana95b]. Its core processor consists of the register file, three functional units, a control unit, two address generators (DAG1 and DAG2), a timer and the instruction cache (see figure 1). Data can be transported via three buses: PM-, DM- and I/O-bus, which provide connection to the program memory,[1] data memory and the I/O-processor.

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.

The three functional units—an arithmetic-logical unit (ALU), a shifter and a multiplier—can operate in parallel (with some restrictions listed below), if they use certain register groups as source operands (see figure 1). Multifunctional instructions use the multiplier and the ALU concurrently or perform two simple instructions in the ALU, e.g. a combined addition/subtraction.

Providing two address generators, the ADSP-2106x is capable of fetching a 48-bit data from PM and a 40-bit data from DM simultanously. This can be done in parallel with the arithmetic operations.

Instructions are executed in three clock cycles:

- In the *fetch*-cycle, the ADSP-2106x reads the instruction from either the on-chip instruction cache or from program memory.
- In the *decode*-cycle, the instruction is decoded.
- In the *execute*-cycle, the instruction is executed, i.e. the operations to be executed are completed.

These cycles are 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.

[1]The program memory is used to store both data and instructions.

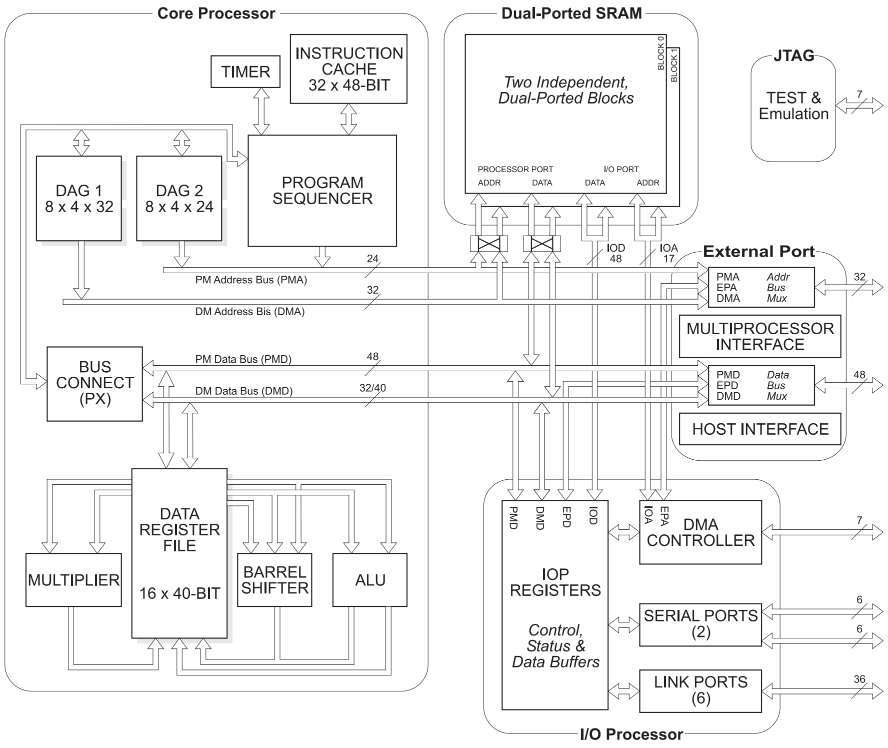

Figure 1: Overview of the architecture of the ADSP-2106x.

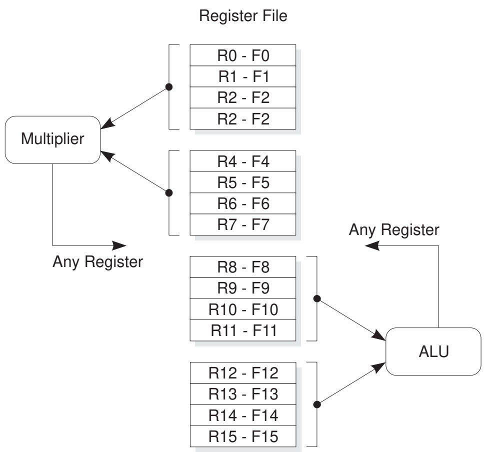

Figure 2: The use of the register file groups for a combined multiply/add-instruction.

3 Intermediate Representation

A compiler internally represents the source program in several ways. This section briefly describes the intermediate representations needed by the different schedule algorithms.

Definition 3.1 *A* basic block *in a given control flow graph is a path of maximal length, such that at most the first node of this path has more than one incoming edge and at most the last node has more than one outgoing edge.*

For every basic block holds: if the first instruction is executed, then all remaining instructions will be executed too (assuming no run-time errors, exception, etc.).

Definition 3.2 *The* basic block graph G_B *of a given control flow graph* G_{cf} *is derived from* G_{cf} *by replacing basic blocks by nodes. Edges in* G_{cf} *that lead to the first node of a basic block are connected to the node representing the basic block. Edges that leave the last node of a basic block in* G_{cf}*, become outgoing edges of the basic block node.*

The sequence of instructions within a basic block may be rearranged with respect to certain restrictions, which are determined by the data dependencies between instructions. These dependencies are pairs of reade or write accesses to the same register or other components that influence the overall state of the machine. Write accesses are calles *definitions*, read accesses *uses*. Data dependencies can be categorized as

- true dependencies (def-use)
- output dependencies (def-def)
- anti dependencies (use-def)

In neither case, the position of the defining and the using instructions may be interchanged. The data dependencies for a given basic block are given by its data dependence graph.

Definition 3.3 *Given a basic block* B*. Its* data dependence graph *is a labelled, acyclic, directed graph* $G_D = (V_D, E_D)$ *whose nodes are labelled with the instructions of* B*. An edge exists between nodes* x *and* y*,* $x, y \in V_D$*, if there is a sequence of instructions* $(x = n_1, n_2, \ldots, n_k = y)$ *such that*

- x *is a definition,* y *is a use of the same resource and there is no other use on a path from* x *to* y *(true dependence), or*
- x *uses a resource which will be written by* y *and there is no other write access to that resource on a path from* x *to* y *(anti dependence), or*
- x *and* y *are definitions of the same resource and there is neiter another definition nor a use on any path from* x *to* y *(output dependence).*

If E_D^{true} *denotes the true dependencies,* E_D^{anti} *the anti dependencies and* E_D^{output} *the output dependencies, then the set of edges can be rewritten as*

$$E_D = E_D^{true} \cup E_D^{anti} \cup E_D^{output}$$

4 Conventional Approaches

Current microprocessors usually provide several hardware resources, that can be used in parallel. These so called *horizontal* processors force the compiler to reorder the created microcode in order to improve its efficiency. This task is known as *compaction*, for two or more instructions that are to be executed in parallel are packed into the same instruction word.

Compaction methods can be classified as *local* or *global*. Local techniques only consider *basic blocks* as their compaction scope, whereas global techniques try to resolve the basic block boundaries by different means. A well-known example for global techniques is trace scheduling, which combines basic blocks, that are often executed successively, to form a single trace. Then, these traces are scheduled.

4.1 Critical Path Method

Ramamoorthy and Tsuchiya [LDSM80] introduced critical path (CP) algorithms for microcode compaction in 1974. Their work has some similarities to the critical path approach to processor scheduling.

The minimum amount of cycles needed to schedule a basic block corresponds to the maximum depth of its data dependence graph. This longest path ist called *critical path*. The ordering of instructions in the critical path is implied by their data dependencies. The remaining instructions are then placed into the critical path, what may lead to additional cycles.

The calculation of the critical path takes three steps. First, an early and a late partition are created. In the *early partition*, an instruction is scheduled into the first cycle if no data dependences are violated. Hereby, resource conflicts are not taken into account. The *late partition* is computed analogously, with the direction of the edges in the data dependence graph being reversed. Doing so, instructions are scheduled into the last possible cycle. The amount of cycles needed to schedule the instructions of the basic block in the early partition is equal to the one needed for the late partition.

The critical path consists of those instructions that are mapped to the same cycle in the early as well as in the late partition. These instructions form the *critical partition* and are a frame for adding the remaining operations. In a following step, resource conflicts in the critical partition are resolved; this may lengthen the schedule.

In the last step, the remaining instructions are inserted into the revised critical partition. This is done for each instruction by testing the cycles between the early and late partition with respect to data dependencies and resource conflicts. This may also lengthen the schedule. A schema of the implementation is given in algorithm 1.

The critical path method is not always able to create an optimal schedule. This is due to the fact that subsequent subframes[2] (these are newly created cycles) cannot be merged together although this would be permitted by the data dependencies and the use of resources.

[2]For each cycle in the revised critical partition a *frame* is created. Inserting new cycles due to resource conflicts would demand a recalculation of the early and the late partition. To avoid this, subframes are inserted instead of new cycles to avoid this computation.

```
critical_path(BASICBLOCK bb)
{
  LIST frames[ddg_depth];
  LIST non_critical;
  MICROOP m;
  int c;

  create_partitions(bb);

  forall(bb, m)
    if (m->ep == m->lp)
      insert_mop(frames[m->ep], m);
    else
      append(non_critical, m);

  forall(non_critical, m)
    for (c=m->ep; c<=m->lp; c++)
      if (insert_mop(frames[c], m))
        break;
}
```

Algorithm 1: Scheme of the implementation of critical path method.

4.2 List Scheduling

List scheduling is a local scheduling method. It evolved out of a heuristic branch-and-bound algorithm [LDSM80] and is also used by global methods such as *trace scheduling* [Fis81].

In addition to a list of instructions in the basic block the data dependence graph is needed. While executing, list scheduling maintains a set called *data ready*, in which all instructions reside that have no predecessor in the data dependence graph or whose predecessors have already been scheduled.

The algorithm can now easily be explained: starting with the first cycle of the basic block, an instruction is taken out of the data ready set and put into the current cycle as long as there are instructions available and no resource conflict is encountered. Then the cycle is incremented and the data ready set is updated. This is done until all instructions are scheduled. A pseudo code of list scheduling is presented in algorithm 2.

In order to select an instruction from the data ready set, a heuristic is used that assigns priority values to the instructions. Instructions with higher priority are preferred. Possible heuristics are:

first fit The first instruction found in the data ready set is taken.

longest remain After every update, a counter for each microinstruction is incremented. The instruction with the greatest counter value is selected.

max depend The priority of an instruction is determined by the number of successors in the data dependence graph.

highest level The priority equals the length of the longest path in the data dependence graph starting from that instruction.

As an extension to list scheduling, we have integrated register allocation into the task of scheduling.

```
list_scheduling(BASICBLOCK bb)
{
  LIST mops;
  int cycle=0;
  MICROOP m;

  update_data_ready();
  while (entries(data_ready) > 0) {
    cycle++;

    while ((m=choose_microop()) != NULL)
      if (are_parallel(m, mops))
        append(mops, m);

    forall(mops, m)
      m->cycle=cycle;

    remove_from_data_ready(mops);
    update_data_ready();
    empty(mops);
  }
}
```

Algorithm 2: Scheme of the implementation of list scheduling.

First a pre-run scheduler is invoked to gather information that can be used by the register allocator, which assigns machine registers to symbolic registers. Then the scheduler performs the final reordering of the instructions. To be more precise, the scheduler identifies operations that cannot be compacted due to restrictions on register usage.[3] For different instruction different register groups are required; therefore colliding instructions can be marked with the groups they need to belong to in order to resolve the conflict. The register allocator tries to satisfy these group assignments as long as they do not inhibit the graph coloring.

4.3 Global Methods

In our implementation, we didn't consider any global graph-based algorithm, since these are suited for large input programs. Our test programs only contain a few basic blocks and so the global methods would have found little more parallelism (if any at all). Since the aim of our work is a comparison between conventional and ILP-based methods, we had to choose programs suitable for both approaches. Larger programs would have prevented the ILP scheduler from computing complete solutions within a bearable amount of time. For the complete `conv` program, e.g., the optimal solution (using ILP-methods) could not be calculated whithin twenty-four hours, so we stopped the calculation process. The program `conv` examined in this paper is only the first basic block of the original convolution filter to make a useful comparison possible.

[3]Recall that for multifunctional instructions certain register groups are required (see figure 2).

5 Register Allocation

The input for the register allocator is an intermediate representation of the source program. Herein, a symbolic register is assigned to every operation and every modified variable. This unbounded number of symbolic registers is to be mapped onto the limited number of machine registers. A register cannot be assigned to two different symbolic registers if their life ranges overlap.

Definition 5.1 *A symbolic register r is* live *at a program point p, if r is defined on a program path from the entry node of the procedure to p and there exists a path from p to a use of r on which r is not defined. The* live range *of a symbolic register r is the set of program points at which r is live.*

Definition 5.2 *Two live ranges of symbolic registers* interfere with one another*, if one of them is defined during the life range of the other. The* register interference graph *is an undirected graph. Its nodes are life ranges of symbolic registers and there is an edge between every two interfering life ranges.*

5.1 Graph Coloring

Graph coloring is a common method to solve the problem of register assignment [WM97]. The register interference graph is the graph to be colored and the number of actual machine registers is the number of colors to be used. The graph coloring problem is to assign one of the k possible colors to every node such that every two nodes that share an edge have different colors. For $k > 2$ the problem is $\mathcal{NP}$ complete, but in the context of register allocation, there exist a number of heuristic methods which have been well tried and tested in practice.

One heuristic method is the following: if the graph contains a node n of degree less than k, then n can be assigned a color that differs from the color of its neighbours. n is removed from the graph which results in a new graph with one node and several edges fewer. The problem was reduced recursivly to a smaller one.

If a k-coloring cannot be obtained by this method—which does not mean that there doesn't exist such a coloring—some life ranges have to be spilled, i.e. stored to memory.

6 ILP-based Methods

6.1 General Introduction

The application area of integer linear programming covers a rich variety of problems. 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. ILP methods are used to solve scientific or economic problems [NW88, NKT89]. Typical applications are concerned with the management and efficient use of scarce resources to increase productivity. These applications include production scheduling, machine sequencing [Brü95], VLSI-design, portfolio analysis [DK96], as well as problems in molecular biology, high energy physics and x-ray crystallography. 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 [CWM94].

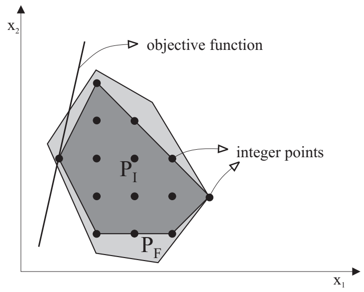

Figure 3: feasible areas.

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. [NW88], [NKT89], [PS82a] or [CWM94].

Integer linear programming (ILP) is the following optimization problem:

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

where

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

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)**. We will assume that $A \in \mathbb{Z}^{m \times n}$ and $b \in \mathbb{Z}^m$ holds. Then the optimal solution of an integer linear program can be calculated by solving the following problem [NW88]:

$$\begin{aligned} \min \quad z_{IP} &= c^T x \\ x &\in P_I \end{aligned} \tag{2}$$

where

$$P_I = conv(\{x \mid x \in P_F \ \cap \mathbb{Z}^n\}).$$

Here, *conv* denotes the convex hull. In the following, we need another definition:

Definition 6.1 (Relaxation) *Let Q be an optimization problem with a feasible region $X(Q)$. An optimization problem Q^R is called a* ***relaxation*** *of Q, if for the feasible region $X(Q^R)$ the following holds:*

$$X(Q) \subset X(Q^R).$$

For the two-dimensional case, a representation of P_F und P_I (equations 1 and 2) is given in figure 3.

The integral points within P_F denote the feasible solutions to the integer linear problem; depending on the objective function, at least one of them represents an optimal solution. The feasible region of (1) consists only of the integer points, whereas the feasible region of (2), P_I, consists of the convex hull of these points.

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 [NKT89]. Therefore one can try to solve a related problem, called the **LP-relaxation** of the integer linear problem, which reads as follows:

$$\begin{aligned} \min \quad z_R &= c^T x \\ x &\in P_F \end{aligned} \tag{3}$$

with

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

Since $P_I \subseteq P_F$, one can conclude from (2) and (3) 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 [GE93], [PS82b]. Unfortunately for most problems it is not known how to formulate these additional inequalities—and there could be an exponential number of them [NKT89].

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.

By using such techniques, the ILP-formulations examined in this paper try to get to an efficient solution of the integrated instruction scheduling and register allocation problem. They are based on formulations which were developed in the area of architectural synthesis. The goal of architectural synthesis consists of finding either the fastest architecture for a given code sequence or the cheapest architecture for a given performance requirement [GE92]. This problem formulation is closely related to the integrated instruction scheduling and register allocation problem. For a correct synthesis, the problem of instruction scheduling has to be taken into account; as part of a more general resource allocation, the register allocation is also part of the problem. The difference is mainly that in order to solve the problem of instruction scheduling in a compiler, the hardware is fixed. Thus, the input code has to be transformed to allow an efficient use of the given hardware resources maintaining the program's semantics. Resource allocation and minimization of hardware costs are less important in this scope.

When no resource constraints have to be considered, local instruction scheduling can be performed by applying the **critical path method** (CPM) to the data dependence graph (see section 4.1). This algorithm calculates for each node of the dependence graph, i.e. for each instruction of the input program the earliest possible (*asap*-control step; *as soon as possible*) and the latest possible execution time (*alap*-control step;*as late as possible*). Instructions scheduled to the same control step are executed in the same clock cycle of the target machine. The *asap*- and *alap*-values are important for both ILP-formulations considered in this paper. The reason is, that they allow for each instruction to be assigned to an interval of valid control steps, in which the execution may take place. Thus, the size of the feasible region of the created ILP is reduced.

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, a resource graph is used which is defined following [Zha96].

Definition 6.2 (resource graph) *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.*

6.2 SILP

The ILP-formulation described in this section was presented in [Zha96] under the name **SILP** (*Scheduling and Allocation with Integer Linear Programming*). First we will give an overview of the terminology used:

- 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. For linear program flow, the mapping of a microoperation to the c-th instruction of the machine program is equivalent to assigning the starting time for the execution of this microoperation to the c-th clock cycle (control step). For non-linear program flow, this correspondance need not be correct.
- 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 succesive data inputs to this functional unit).
- The number of available resources of type $k \in V_K$ is R_k.
- τ_j describes 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 program's instructions; 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 figure 4.

Definition 6.3 (resource flow 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

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

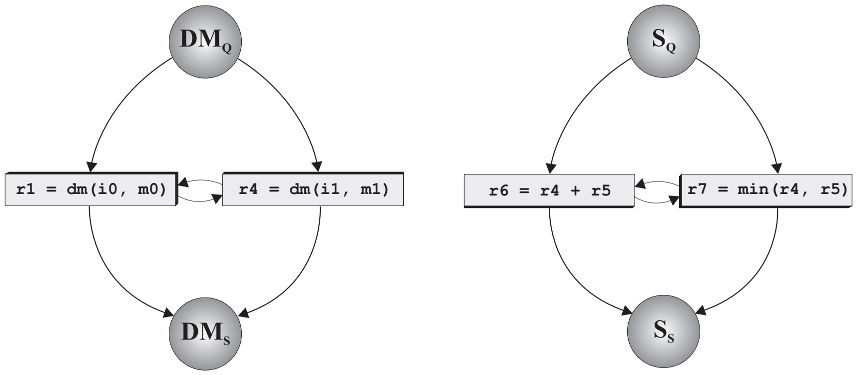

Figure 4: Resource flow graph for two instructions executed on an ALU and the data memory, resp.

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:

$$\Phi_j^k = \sum_{(i,j) \in E_F^k} x_{ij}^k; \qquad \Psi_j^k = \sum_{(j,i) \in E_F^k} x_{ji}^k \tag{4}$$

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

$$\min \quad M_{steps} \tag{5}$$

- 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)

$$t_j \leq M_{steps} \quad \forall\, j \in V_D \tag{6}$$

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

$$\begin{aligned} t_j - t_i &\geq w_i \quad \forall\, (i,j) \in E_D^{output} \cup E_D^{true} \\ t_j - t_i &\geq 0 \quad \forall\, (i,j) \in E_D^{anti} \end{aligned} \tag{7}$$

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

$$\Phi_j^k - \Psi_j^k \;=\; 0 \quad \forall\, j \in V_D,\ \forall\, k \in V_k :\ (j,k) \in E_R \tag{8}$$

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

$$\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \Phi_j^k = 1 \quad \forall\, j \in V_D \tag{9}$$

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

$$\sum_{(k,j) \in E_F^k} x_{kj}^k \leq R_k \qquad \forall k \in V_K \tag{10}$$

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 $x_{ij}^k = 1$.

$$t_j - t_i \geq z_i + \Big(\sum_{\substack{k \in V_K: \\ (i,j) \in E_F^k}} x_{ij}^k - 1\Big) \cdot \alpha_{ij} \ \forall (i,j) \in E_F^k \tag{11}$$

   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 [Zha96], it is shown, that the tightest polyhedron is described by using the value $\alpha_{ij} = z_i - asap(j) + alap(i)$.

6.2.1 Integration of Register Allocation

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 modified. Again following the concept of flow graphs, the register assignment problem is formulated as register distribution problem.

Definition 6.4 (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 a resource node g representing the available register set, $G = \{g\}$. 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:

$$t_j - t_i - b_{ij} = w_i \tag{12}$$

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

$$\tau_i \geq b_{ij} + w_i \qquad \forall\ (i,j) \in E_D^{true} \tag{13}$$

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*:

$$t_j - t_i \geq w_i - w_j + \tau_i + (x_{ij}^g - 1) \cdot 2T \tag{14}$$

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:

$$\begin{aligned}
\Psi_g^g &\leq R_g && && (15)\\
\Phi_j^g &= 1 && \forall\, j \in V_g && (16)\\
\Phi_j^g - \Psi_j^g &= 0 && \forall\, j \in V_F^g && (17)\\
t_j - t_i &\geq w_i - w_j + \tau_i + (x_{ij}^g - 1) \cdot 2T && \forall (i,j) \in E_F^g && (18)
\end{aligned}$$

Moreover, the ILP-formuation can be tightened by an identification of redundant serial constraints and the insertion of valid inequalities; for further information see [Zha96, Käs97].

Following [CWM94], 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 [Zha96, Käs97].

6.3 OASIC

In this section, a different modelling approach, called **OASIC** (*Optimal Architectural Synthesis with Interface Constraints*) [GE92, GE93] is presented. Again, results of polyhedral theory are used to formulate constraints which reduce the size of the feasible region thus increasing the solution efficiency.

First, we will give an overview of the used terminology, which differs in some points from the SILP-terminology:

- The main decision variables are called x_{jn}^k, where $x_{jn}^k = 1$ means, that microoperation j is scheduled in instruction $n \geq 1$ and is executed by an instance of resource type k. Again, in the case of sequential control flow, the relative position of an instruction can be considered as synonymous to the clock cycle of the execution of this instruction. (see section 6.2).
- t_j describes the relative position of a microoperation j within the instructions of the optimized code sequence. This variable is introduced just for the sake of clarity and is not used in the linear programs; instead the following equation is used:

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

- $N(j) = \{asap(j), asap(j)+1, \ldots, alap(j)\}$ is the set of possible control steps, in which an execution of j can take place. $asap(j)$ describes the earliest possible execution time, $alap(j)$ the latest possible one.

As already mentioned in Section 6.1, for every bounded system of linear inequalities, there exists an integral polytope that contains the same integer points. When this integral polytope is used as feasible region of an optimization problem, an integer optimal solution can be calculated by using linear programming algorithms [NW88, PS82a]. The constraints necessary to define the integer vertices of the polytope are called *integral facets*.

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. 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 gained. This is covered in detail in [GE92, GE93]; an overview is given in [Käs97].

In the following, the ILP-formulation given in [GE92, GE93] is presented in a slightly modified form.

- objective function

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

- constraints

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

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

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

$$\begin{aligned}
\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_j \leq n \\ n_j \in N(j)}} x^k_{jn_j} + \sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_i \geq n - Q^k_i + 1 \\ n_i \in N(i)}} x^k_{in_i} \;&\leq\; 1 \qquad \forall\; (i,j) \in E^{true}_D \cup E^{output}_D, \\
n \in (\{n + Q^k_i - 1 \mid n \;&\in\; N(i)\} \cap N(j)) \\
\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_j < n \\ n_j \in N(j)}} x^k_{jn_j} + \sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{\substack{n_i \geq n - Q^k_i + 1 \\ n_i \in N(i)}} x^k_{in_i} \;&\leq\; 1 \qquad \forall\; (i,j) \in E^{anti}_D, \\
n \in (\{n + Q^k_i - 1 \mid n \;&\in\; N(i)\} \cap N(j))
\end{aligned} \tag{21}$$

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

$$\sum_{\substack{k \in V_K: \\ (j,k) \in E_R}} \sum_{n \in N(j)} x^k_{jn} \;=\; 1 \qquad \forall\; j \in V_D \tag{22}$$

  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.

$$\sum_{j \in V_D:(j,k)\in E_R} \sum_{n \in N'(j,k,n')} x^k_{jn} \leq R_k \qquad \forall k \in V_K \wedge 0 \leq n' \leq M_{steps}$$
$$\text{with } N'(j,k,n') = \{n \in N(j) : n = n' + p,\ 0 \leq p \leq Q^k_j - 1\} \qquad (23)$$

In the ideal case, the resulting polytope is integral; then the relaxation provides an optimal integer solution. However, in most cases some variables will have non-integral values. Then, these variables are specified as binary and the resulting MILP is solved. This is repeated until all variables of the solution have integral values.

6.3.1 Integration of Register Allocation

In order to take into account the problem of register assignment with respect to a homogeneous register set, the above presented formulation just 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 produced the variable and the operation j, that last used the variable. However, each variable can be used within more than one operation. In consequence, lifetime-defining edges are possibly not unique, since when simultaneous instruction scheduling and register allocation is performed, the order of the operations is not fixed. 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 informations, see [GE92, GE93, Käs97].

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.

$$\begin{aligned} \sum_{j_a \prec j_b \in M'(n)} \Big(\sum_{k \in V_K} \sum_{\substack{n_1 \leq n \\ n_1 \in N(j_a)}} x^k_{j_a n_1} + \sum_{k \in V_K} \sum_{\substack{n_2 > n \\ n_2 \in N(j_b)}} x^k_{j_b n_2} \; - \\ \sum_{k \in V_K} \sum_{\substack{n_3 \leq n \\ n_3 \in N(j_b)}} x^k_{j_b n_3} - \sum_{k \in V_K} \sum_{\substack{n_4 > n \\ n_4 \in N(j_a)}} x^k_{j_a n_4} \Big) \;\; &\leq \;\; 2 \cdot R \qquad (24) \\ \forall\, n \;\wedge\; \forall\, M'(n) \in M(n) \end{aligned}$$

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 (24) 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 dependent of 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 [GE92, GE93, Käs97]).

6.4 Global Analysis

In the context of global analysis, the consideration of data dependences is not sufficient in order to maintain the program's semantics. It might be possible for some instructions to be moved from one basic block to another which is executed under different control conditions, without violating any data dependences. A common approach to this problem is the prevention of code motion across basic block boundaries, e.g. by inserting dummy nodes at the begin and end of each basic block [Ell86, GE92]. In this section, an entirely different approach is presented. Let a sequence of machine operations be given which is to be optimized. For each instruction, the semantics of the input program define the valid control conditions; provided there are no data dependences, it can be executed in every basic block, that is subject to exactly the same control conditions. Thus, for each instruction, the set of basic blocks is determined, in which it can be inserted and the resulting disjunction is integrated into the integer linear program.

Let a basic block B_k be given. This basic block is assigned a start-time t_k^A and an end-time t_k^E. An instruction is contained in basic block B_k if and only if $t_k^A \leq t_j \leq t_k^E$. Thus, the set X_k of all instructions that can belong to basic block B_k can be defined as follows:

$$X_k = \{j \in V_D \mid t_k^A - t_j \leq 0 \ \wedge \ t_j - t_k^E \leq 0\}.$$

Now we want to represent the fact that an instruction j may be scheduled in exactly one of l possible basic blocks $B_1, \ldots, B_l$ by a system of inequalies including binary variables $y_1, \ldots, y_l$. First, an upper bound T is needed, so that $t_j \leq T \quad \forall\, j \in V_D$. When a constraint $t_j - t_\alpha \leq 0$ is replaced by $t_j - t_\alpha \leq T$, it becomes redundant and doesn't restrict the feasible region any more. So we define $y_j = 0$ if and only if $j \in B_k$, and $y_j = 1$ if and only if $j \notin B_k$. The value of the upper bound is chosen to be $T = 2I + 1$, where $I = |V_D|$ indicates the number of operations of the input program.

The following constraints guarantee that an instruction j is contained in exactly one of the sets $X_1, \ldots, X_l$:

$$\begin{aligned} t_k^A - t_j - T y_j^k &\leq 0 \quad \forall\, k = 1, \ldots, l \\ t_j - t_k^E - T y_j^k &\leq 0 \quad \forall\, k = 1, \ldots, l \\ \sum_{k=1}^{l} y_j^k &= l - 1 \\ y_j^k &\in \{0, 1\} \quad \forall\, k = 1, \ldots, l \end{aligned} \tag{25}$$

Example 6.1 When an instruction j can be assigned to three basic blocks B_1, B_2 und B_3, the following constraints are generated:

$$
\begin{array}{ccccccc}
t_1^A & - & t_j & - & Ty_j^1 & \leq & 0 \\
t_j & - & t_1^E & - & Ty_j^1 & \leq & 0 \\
t_2^A & - & t_j & - & Ty_j^2 & \leq & 0 \\
t_j & - & t_2^E & - & Ty_j^2 & \leq & 0 \\
t_3^A & - & t_j & - & Ty_j^3 & \leq & 0 \\
t_j & - & t_3^E & - & Ty_j^3 & \leq & 0 \\
y_j^1 & + & y_j^2 & + & y_j^3 & = & 2
\end{array}
$$

When a feasible solution provides $y_j^2 = 0$, then instruction j must be scheduled in basic block B_2 as can easily be verified.

$\square$

The set of basic blocks, which an instruction can be assigned to, can be calculated using the control dependence graph [Bas95]. When instructions of different basic blocks have the same predecessor in the control dependence graph and the types of these edges to these predecessors are identical, all these instructions can be assigned to each of these basic blocks—provided this is not prevented by data dependences.

In order for the assignment of instructions to basic blocks to be well defined, neither branches nor loop-instructions may be removed from their actual basic blocks, since these instructions define the basic block structure. When a basic block B_k is introduced by a loop-instruction i_1 and is finished by a branch i_2, the assignment of an operation j to B_k means that j is scheduled between i_1 and i_2. Any reordering of basic blocks is excluded, i.e. the order of the basic-blocks in the output program must be the same as in the input program. This way, just the ordering of control instructions is fixed; any other instructions can be moved between the basic blocks which underly the same control conditions. The order of the basic blocks is fixed in the ILP by creating the following inequalities for each pair of subsequent basic blocks:

$$
\begin{aligned}
t_{k1}^A - t_{k1}^E &\leq 0 \qquad (26)\\
t_{k2}^A - t_{k2}^E &\leq 0 \\
t_{k1}^E &< t_{k2}^A
\end{aligned}
$$

Thus, not all reorderings of instructions are allowed. However the quality of the generated code depends of the degree of parallelism; this is not severely affected by the order of basic blocks. So in most cases, this restriction will be of no importance.

These constraints are sufficient, when only the problem of instruction scheduling is considered. When integrated instruction scheduling and register allocation is performed, additional constraints are necessary for a correct representation of lifetimes in the presence of branches and loop-instructions [Käs97]. Unfortunately, these constraints require additional binary variables, so that the complexity of the generated ILPs is increased. Therefore, in our implemenation, only the optimal register set assignment is calculated. After solving the ILPs a conventional register allocator has to be invoked. This allocator uses the previously calculated register set assignment to get a feasible register assignment. In the scope of this paper, we cannot give more details; however, they can be found in [Käs97].

6.5 Extensions Required by the Target Architecture

6.5.1 Extensions to SILP

As can be seen in the previous sections, the assignment of instructions to hardware resources plays a significant role for the ILP-modeling. In order to correctly model the processing of the ADSP-2106x, we need the following resource nodes:

- DM: models accesses to data memory
- PM: models accesses to program memory
- S (standard): models ALU- and Shifter-operations as well as several miscellaneous instructions. These instructions can never be executed in parallel. Since the motivation to introduce resource nodes is the capability of parallel execution, the assignment of these instructions to different nodes would not be useful.
- MU: models multiplier-instructions
- C: models control flow instructions

Prevention of Incorrect Parallelism

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. However, the considered architecture just implements limited parallelism: only certain ALU- and multiplier-accesses can be executed in parallel and parallel accesses to memory are not possible for all load-/store-operations. Moreover, the degree of potential parallelism also depends of the register assignment (see Section 2, figure 1). Therefore, 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 $(t_i > t_j) \vee (t_i < t_j)$. The following inequalities are required:

$$t_i - t_j > -v_{ij} T \tag{27}$$
$$t_i - t_j < (1 - v_{ij}) T \tag{28}$$
$$v_{ij} \in \{0, 1\} \tag{29}$$
$$T = 2I + 1 \tag{30}$$

A correctness proof is provided in [Käs97].

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 Section 2). 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, so that the set G of available register sets has to be extended to $G = \{g_1, g_2, g_3, g_4\}$. The definition of the register flow graph has to be modified and the constraints (15), (16) and (18) are be replaced by the following constraints:

$$
\begin{aligned}
\Psi_g^g &\leq R_g \qquad \forall\, g \in G & (31)\\
\sum_{g\in G} \Phi_j^g &= 1 \qquad \forall\, j \in V_g & (32)\\
\Phi_j^g - \Psi_j^g &= 0 \qquad \forall\, j \in V_F^g \quad \forall\, g \in G & (33)\\
t_j - t_i &\geq w_i - w_j + \tau_i + (\sum_{g\in G} x_{ij}^g - 1)\cdot \alpha_{ij} \qquad \forall\, (i,j) \in E_F^g & (34)
\end{aligned}
$$

Inequality (31) assures that at most four instances of each of the four register sets are used; the constraint (32) guarantees, that each variable is assigned to exactly one register. The flow conservation is maintained by (33) and the new formulation of the serial constraints is given in (34).

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 $\Phi_m^g \geq 1$ must hold. Since $\sum_g \Phi_j^g = 1$, 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.

$$
\begin{aligned}
t_i - t_j &\geq -p_{ij}T & (35)\\
t_i - t_j &\leq q_{ij}T & (36)\\
p_{ij} + q_{ij} &= 1 & (37)
\end{aligned}
$$

where $T = 2I + 1$.

Using these values, the register constraints can be formulated as follows:

$$
\begin{aligned}
\Phi_m^g &\geq 1 - (t_i - t_j) - p_{ij}T & (38)\\
\Phi_m^g &\geq 1 + (t_i - t_j) - q_{ij}T & (39)
\end{aligned}
$$

The correctness proofs are omitted in this paper; they are explicitly given in [Käs97].

6.5.2 Extensions to OASIC

Prevention of Incorrect Parallelism

Let two instructions i and j be given which cannot be executed in parallel, so that $t_i \neq t_j$ must hold. When i and j can never be assigned to the same basic block or when parallel execution is prevented by data dependences, no additional constraints are required. Otherwise, the following constraints have to be added to the formulation:

$$
\sum_{k\in V_K:(i,k)\in E_R} x_{in}^k + \sum_{k\in V_K:(i,k)\in E_R} x_{jn}^k \leq 1 \qquad \forall\, n \in N(i)\cap N(j) \qquad (40)
$$

Irregular Register Sets

The constraints given in Section 6.3 have to be modified, since the register file is divided into subsets to be considered separately. So, new variables have to be introduced. When instruction i

performs a write access to a register, the binary variable x^k_{in} has to be replaced conformingly to the following equation:

$$x^k_{in} = \sum_{m=1}^{4} x^{km}_{in}$$

When instruction i is executed in control step n by resource type k and a write access is performed to a register of register set m, then $x^{km}_{in} = 1$ must hold. The following additional constraints are required:

- In any register only one variable must be saved at one time, thus in each of the four register groups at most $R_m = 4$.

$$\sum_{(k,j)\in E_R} \sum_{j:n\in N(j)} x^{km}_{jn} \leq R_m \qquad \forall\, m \quad \forall\, n \tag{41}$$

- For each register group, the number of crossing edges of each control step must not exceeed the number of available registers.

$$\begin{aligned} \sum_{j_a \prec j_b} \Big(\sum_{k\in V_K} \sum_{\substack{n_1 \leq n \\ n_1\in N(j_a)}} x^{km}_{j_a n_1} + \sum_{k\in V_K} \sum_{\substack{n_2 > n \\ n_2\in N(j_b)}} x^{km}_{j_b n_2} & - \\ \sum_{k\in V_K} \sum_{\substack{n_3 \leq n \\ n_3\in N(j_b)}} x^{km}_{j_b n_3} - \sum_{k\in V_K} \sum_{\substack{n_4 > n \\ n_4\in N(j_a)}} x^{km}_{j_a n_4} \Big) & \leq 2 \cdot R_m \qquad \forall n \quad \forall m \end{aligned} \tag{42}$$

- When operation l is restricted to register set m_l because of the combination of operations i and j to a multifunction-instruction, the following constraints have to be added.

$$t_i - t_j \geq -p_{ij}T \tag{43}$$

$$t_i - t_j \leq q_{ij}T \tag{44}$$

$$p_{ij} + q_{ij} = 1 \tag{45}$$

$$\sum_{n\in N(l)} x^{km_l}_{ln} \geq 1 - (t_i - t_j) - p_{ij}T \tag{46}$$

$$\sum_{n\in N(l)} x^{km_l}_{ln} \geq 1 + (t_i - t_j) - q_{ij}T \tag{47}$$

  where $T = 2|V_D|+1$ and $p_{ij}, q_{ij} \in \{0,1\}$. This formulation is analogous to that one presented in Section (6.5.1).

Using this formulation, the optimal assignment of register sets can be calculated. Moreover, the structure of these constraints ascertains that enough registers are available to execute the resulting code without inserting spillcode. However, no concrete register assignment is performed by the ILP formulation. The concrete assignment has to be left to a conventional graph-based register assigner that takes into account the previously calculated register set assignment.

The reason for not integrating register assignment completely in the ILP-formulation is the complexity of the ILP to be generated. All decision variables must have integral values. Using a branch-and-bound algorithm to solve the ILP, 2^n nodes have to be generated in the worst case, where n is the number of variables specified as binary. While, for a homogeneous register set $N(j)$ binary variables are introduced per instruction, this number increases to $4N(j)$ when the four different register sets are taken into account. If an own binary variable was defined for each register, $16N(j)$ binary variables would be required. Since $\mathcal{O}(2^{N(j)}) \subset \mathcal{O}(2^{16\cdot N(j)}) = \mathcal{O}((2^{N(j)})^{16})$, the increase in complexity would be intolerably high.

6.6 Approximations

The computation time required to solve the generated ILP's 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 [Käs97].

6.6.1 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 [Zha96]). As for the other binary variables introduced to handle global analyses, interdictions of parallelism, register set assignment etc., a relaxation of the integrality constraint would affect the structure of the ILP too much (see also [PS82a]). Thus the presented rounding approach is applied exclusively to the variables $x \in M_S$.

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 $x \in M_S$ which 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 by 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.

6.6.2 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 is calculated, it can be expected that the resulting fixations also lead to a good global solution. This is confirmed by the test results.

6.6.3 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.

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

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.

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.

6.6.4 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 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 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 $\mathcal{SF}$.

6.6.5 Rounding Approximation for OASIC

For the OASIC-formulation, only one approximation has been found—the rounding approach. Similarly as described above for SILP, a non-integral variable with smallest distance to an integer is rounded to that value by adding an appropriate equation to the constraints. This is repeated until all variables have integral values. Again, false fixations are changed by a backtracking approach, respectively, when this is not sufficient to obtain a feasible solution, the variables have to be redeclared as integral. Unfortunately, this approximation suffers from high computation times and doesn't produce satisfying results.

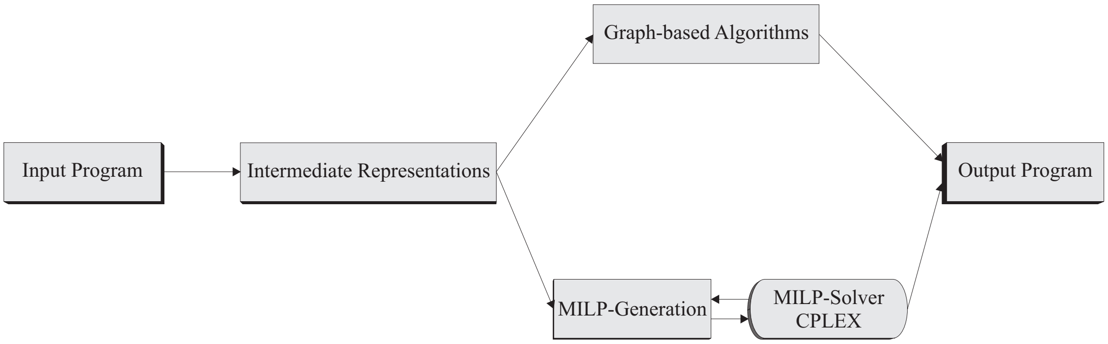

Figure 5: Overview of the Implementation

7 Implementation and Experiments

The source language is assembler based on the ADSP-2106x's instruction set. A source program can be generated by the `gcc`-based compiler `g21k`, shipped with the ADSP-2106x or can be written by hand. The implementation comprises several phases:

- The source program is analysed syntactically and sematically with `flex` and `bison`. These use a context-free grammar written for the ADSP-2106x's instruction set.
- The required intermediate representations, as e.g. control flow graph, data dependence graph, control dependence graph, the resource flow graph when using SILP, etc. are generated.
- Integrated instruction scheduling and register allocation is performed by two alternative approaches:
  - graph-based approaches. Each basic block is scheduled using the specified algorithm and the compacted instruction sequence is written to the result file.
  - ILP-based approaches. First, the ILP of the formulation type specified by the user (SILP or OASIC) is generated. This ILP is solved by using CPLEX, a callable library to optimize linear and integer linear problems [CPL95]. Finally, the optimization result is interpreted and the optimized instruction sequence is written to the result file.

This process can be visualized as depicted in figure 5. The input programs are typical applications of digital signal processing, e.g. a fourier transformation, a digital filter as well as imaging algorithms. All these programs are strictly sequential, i.e. they consist of a sequence of machine instructions, each containing only one microoperation. If only instruction scheduling is performed, an optimal register assignment is provided in order to allow a high amount of achievable parallelism.

Apart from the optimal, respective approximative schedules, lower bounds on the optimal value of the objective function are calculated. These bounds can be obtained simply by relaxing the integrality constraint of the variables $x \in M_S$ in the SILP-formulation. As for OASIC, the calculation is terminated after the first iteration (i.e. all decision variables x_{in}^k are relaxed) and the actual objective value is returned. The calculation time can be further reduced, when also the v- and p/q-variables are relaxed; however, this affects the quality of the bounds.

In the following, some of our experimental results are described. Table 1 shows the most important characteristics of our example programs: the number of instructions, basic blocks, loops and data

name	description	instructions	basis blocks	loops	dependences
fir	finite impulse response filter	18	3	1	59
cascade	infinite impulse response filter	20	1	0	63
dft	discrete Fourier-transformation	26	4	2	63
whetp3	function `p3` from Whetstone benchmark	26	1	0	103
histo	histogramm	43	4	2	186
conv	convolution filter	49	1	0	193

Table 1: Characteristics of the source programs.

name	mode	constr	bin (expl)	size [KB]
fir	is+ra	422	675 (464)	58.94
cascade	is	161	77 (11)	4.15
cascade	is+ra	606	950 (707)	89.37
dft	is	405	184 (102)	14.8
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 2: Characteristics of the ILP's generated by the SILP-based formulation.

name	mode	constr	bin	size [KB]
fir	is+ra	1233	799	2769.41
cascade	is	916	325	201.3
cascade	is+ra	1904	1208	6987.18
dft	is	1393	567	238.6
dft	is+ra	2143	946	45255.34
whetp3	is	6342	1605	158.58
histo	is	1190	359	1752.23
conv	is	4536	1344	3268.73

Table 3: Characteristics of the ILP's generated by the OASIC-based formulation.

Programm	*list scheduling* ff[a]	lr[b]	md[c]	hl[d]	*critical path*	optimal
fir	10	10	10	10	10	8
cascade	9	9	15	9	11	7
dft	16	16	16	16	16	14
whetp3	22	22	25	21	23	20
histo	31	31	31	31	31	31
conv	17	17	18	18	18	17

[a] *first fit*
[b] *longest remain*
[c] *max depend*
[d] *highest level first*

Table 4: Number of instructions in the result of the different scheduling algorithms.

dependences. The properties of the ILP's of the SILP- resp. OASIC-formulations are shown in tables 2 and 3.

Column "mode" indicates if instruction scheduling was considered in an isolated way or integrated with register allocation. The number of generated constraints is given in column "constr", the number of binary variables in column "bin". In the SILP approach, some flow variables need not be specified as binary, since they always take integral values due to the structure of the ILP. So, the numbers in brackets show how many variables are explicitly specified as binary. As for OASIC, the ILP's are solved iteratively by specifying the non-integral variables as binary and resolving the resulting MILP. Therefore, only the overall number of variables is given in the table. In the last column, the sizes of the generated ILP's are given in KBytes. It is obvious that in most cases, the size of the OASIC-generated ILP's is higher than for the SILP-formulation. This is especially true when considering integrated register allocation and instruction scheduling.

The calculation time of the graph-based algorithms is significantly lower than the time needed to solve the ILP's: all graph-based algorithms take less than one second to execute. The exact calculation times can be found in [Lan97]; in this paper, they are omitted. Results of list scheduling with integrated register allocation are not presented here, either. We want to oppose the best possible results of the conventional algorithms to the ILP-approaches. So, we assume that in the input programs, an optimal register assignment is given.

As we can see in table 5, using the first fit and longest remain heuristics we get the same results, because of two reasons. On the one hand, the data ready set is implemented as a list; instructions that reside in data ready for the longest time are at the beginning of the list. The list is checked for a suitable instruction from the beginning. On the other hand, the programs examined are—except for `whetp3`—hand-written. Therefore, the first-fitting operation equals the one with the longest remain-value. This also explains why the first fit and longest remain heuristics yield the same or a better result than max depend or highest level first (only considering the hand-written programs). A better result for `whetp3`, which was compiled via the `g21k` compiler, is gained from the max depend heuristic. Preferring such instructions that lie on longer paths in the data dependence graph enables parallel execution, which cannot be exploited by the other methods. `cascade` and `whetp3` demonstrate the heuristic nature of max depend. The deviation from the optimal result is 114.3% resp. 25%. The programs contain a large amount of data dependences between consecutive instructions. This imposes great restrictions on compaction which results in a worse schedule. An optimal schedule was found in only 6 of 30 cases. Table 5 shows the deviations from the optimal result. The highest level first heuristic, which is slightly more costly, yields better schedules than the others heuristics. First fit and longest remain still produce better results than the critical path method.

Programm	list scheduling ff[a]	lr[b]	md[c]	hl[d]	critical path	
fir	25%	25%	25%	25%	25%	25%
cascade	28.6%	28.6%	114.3%	28.6%	57.1%	53.6%
dft	14.3%	14.3%	14.3%	14.3%	14.3%	14.3%
whetp3	10%	10%	25%	5%	15%	13%
histo	0%	0%	0%	0%	0%	0%
conv	0%	0%	5.9%	5.9%	5.9%	3.5%
	13%	13%	30.75%	12.3%	19.6%	

[a] *first fit*
[b] *longest remain*
[c] *max depend*
[d] *highest level first*

Table 5: Deviations of the results obtained from list scheduling with several heuristics from the optimal solution written in percentage of the optimum.

name	mode	method	instr	CPU-time
fir	is+ra	def	8	1.69 sec
fir	is+ra	app	8	19.58 sec
cascade	is	def	7	0.19 sec
cascade	is	app	7	0.27 sec
cascade	is+ra	def	-	> 24 h
cascade	is+ra	app	7	86.72 sec
dft	is	def	14	47.42 sec
dft	is	app	14	42.58 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 6: Runtime characteristics of the SILP-based ILP's.

name	mode	method instr	CPU-time	
fir	is+ra	def	8	4 min 11 sec
fir	is+ra	app	11	80.35 sec
cascade	is	def	7	5.45 sec
cascade	is	app	7	1.04 sec
cascade	is+ra	def	7	11 min 7 sec
cascade	is+ra	app	7	65.97 sec
dft	is	def	14	3 h 30 min
dft	is	app	14	1 h 39 min
dft	is+ra	def	-	> 24 h
dft	is+ra	app	16	17 h 25 min
whetp3	is	def	20	18 min 30 sec
whetp3	is	app	21	23.8 sec
histo	is	def	-	> 24 h
histo	is	app	31	> 21 h 20 min
conv	is	def	17	1 h 45 min
conv	is	app	18	2 min 26 sec

Table 7: Runtime characteristics of the OASIC-based ILP's.

The runtime characteristics for the solution process of the ILP's are described in tables 6 and 7. Again, the column "mode" indicates if register allocation is considered together with instruction scheduling. The column "method" in table 6 shows, if an exact solution is computed (default; def) or an approximation is used (app). We describe only the approximation $\mathcal{SF}$ which takes the least computation time. The quality of the optimized code equals that of the code produced by $\mathcal{SF}$; only for `whetp3`, more instructions were needed. For details, see again [Käs97]. The columns "instr" in tables 6 and 7 give the number of instructions of the optimized program and the CPU-times needed to compute these results are shown in the last column. For the OASIC-approach, only the rounding approach was viable; in table 7, it is denoted by app.

As we can see in table 6, by the use of approximations the solution time of the SILP-based approach can be significantly reduced. While the calculation time of an optimal solution for the program `cascade` with integrated instruction scheduling and register allocation had to be broken off after more than twenty-four hours, the stepwise approximation of the isolated flow analysis $\mathcal{SF}$ could find a solution in 86.72 sec. Moreover, this solution was even optimal. In fact, only for the program `whetp3`, the approximation gave a suboptimal result, all other input programs were solved optimally. At first sight, it seems surprising that for some programs, the approximation take more time than the exact solution. However, the approximations require several mixed integer linear programs to be generated and solved. When the original problem is small, the creation and solution of the approximate MILPs consumes more time than is saved during the solution processes. The solutions found by the rounding approach for OASIC (see table 7) are less good than those found with the SILP-approximations. Moreover, the calculation time can still grow very high, so this approach is less satisfactory.

The results of the calculation of lower bounds are given in tables 8 and 9. We only present the results for the problem of integrated instruction scheduling and register allocation. Method $B1$ calculates the lower bounds by relaxing only the flow variables $x \in M_S$ of the SILP-based approach and for the OASIC-based approach the variables $x_{ij}k$. In method $B2$, the v- and p/q-variables are relaxed, too. Using the OASIC-based approach, no lower bounds could be calculated for the three largest programs, since the size of the ILP's grew too much. We can see, that for the SILP-approach, the deviation from the optimal solution is 14.98% for method $B1$ and 19.77% for method $B2$. However, with $B2$, a lower bound could be found within seconds for all input

program	method	lower bound	CPU-time	deviation [%]
fir	$B1$	8	1.2 sec	0
fir	$B2$	8	0.42 sec	0
cascade	$B1$	6	6.76 sec	14
cascade	$B2$	5	0.39 sec	28
dft	$B1$	12	27.142 sec	14
dft	$B2$	12	2.9 sec	14
whetp3	$B1$	16	24.9 sec	20
whetp3	$B2$	15	0.49 sec	25
histo	$B1$	27	156.46 sec	12.9
histo	$B2$	24	16.19 sec	22.6
conv	$B1$	12	30 min 22 sec	29
conv	$B2$	12	11.1 sec	29

Table 8: Calculation of lower bounds in the SILP-based formulation.

program	method	lower bound	CPU-time
fir	$B1$	8	18.6 sec
fir	$B2$	8	9.87 sec
cascade	$B1$	6	22.87 sec
cascade	$B2$	6	20.91 sec
dft	$B1$	12	50 min 31 sec
dft	$B2$	12	47 min 57 sec

Table 9: Calculation of lower bounds in the OASIC-based formulation.

programs. For the OASIC-approach, the quality of methods $B1$ and $B2$ didn't differ; the deviation is 7%—however, the calculation times were higher than using the SILP-based approach.

An overview of the solution quality of the different methods is given in figure 6. The results of the SILP approximation were in fact optimal for the programs shown. The output programs of the graph-based algorithms contain more instructions, i.e. they are not optimally compacted. This is although an optimal register assignment as already given in the input programs, whereas in the ILP-formulation the register assignment is treated together with instruction scheduling.

Comparing the results obtained with the SILP and the OASIC-formulation, it becomes obvious, that the SILP approach is better suited for the problem of integrated instruction scheduling and register allocation whithin a compiler. OASIC is more efficient, if an optimal solution of instruction scheduling for larger input programs is calculated. When register allocation is taken into account, the disk space devoured by the ILPs can explode—this is especially true for our target architecture because of the irregular register set. For the program `dft`, the size of the OASIC-based ILP with integrated instruction scheduling and register allocation was more than 42 MB. However, mainly it is the calculation of approximations that favours the SILP-formulation. The OASIC-approach is less suited for approximative calculations, while based on the SILP-formulation, very good approximative solutions (in fact optimal solutions, with one exception) can be obtained. This is also true for problems, which don't allow an exact solution due to their size. With the calculation of lower bounds, the SILP-modelling outperforms OASIC, too (see table 9). For all tested programs, a good lower bound could be calculated whithin seconds. With the OASIC-based formulation, in three cases lower bounds could just be calculated for the pure instruction scheduling problem. Considering the register allocation too, the calculation was broken off, since the size of the ILP's grew to much.

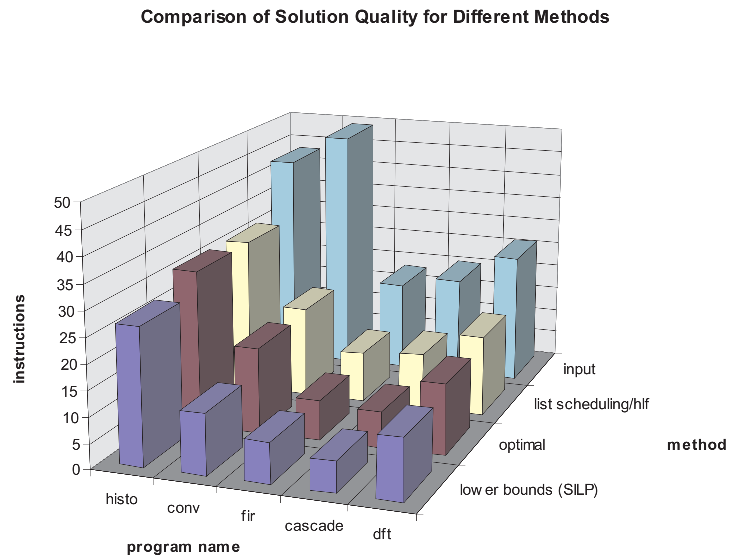

Figure 6: Comparison of Solution Quality for Different Methods

For the examined graph-based algorithms, the average distance from the optimal solution in our test-suite is 13%. Thus, the conventional algorithms exceed the optimal instruction number by at least 13% on average. On calculating lower bounds in the ILP-approach, an objective value is obtained which is on average 15% below the optimal instruction number. So, the quality of the lower bounds when using an ILP-approach is comparable to the quality of the solutions offered by convential, graph-based algorithms.

8 Conclusions

We have shown that the the problem of instruction scheduling for the underlying irregular target-architecture can be modeled completely and correctly as an integer linear program. This result was obtained by extending two structured ILP-formulations, SILP and OASIC. Since the ILPs are created for a fixed set of microoperations, it is not possible for the considered approaches to take into account the insertion or removal of instructions within a unique ILP-formulation. Live range splitting and insertion of spill code cannot be considered, so that a complete integration of instruction scheduling and register allocation is not possible. Two subtasks of register allocation, register set assignment and concrete register assigment however can be integrated. For reasons of complexity it is advisable to renounce of a complete solution to the register assignment problem whenever using the OASIC-approach and using the SILP-formulation, when considering input programs with non-linear program flow. Then only an optimal register set assignment is calculated.

The analysis of our experimental results has shown that for use in a compiler the SILP-modelling is superior to the OASIC-approach. Based on the SILP-formulation, several approximations can be calculated, 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 examined in [Lan97].

Another important application of the SILP-approach consists in calculating lower bounds on the optimal solution. 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.

The optimal schedule computed by the ILP methods is gained at a high price. The space and time complexity explodes with increasing program sizes and inhibits therefore the scheduling of complete applications with the ILP approach. It seems more promising to compact suitable code sequences, e.g. innermost loops. Graph-based methods are a real alternative to the ILP scheduling techniques. At the cost of losing optimal results a improved schedule can be found within a short time. This makes the heuristics attractive to be used within optimizing compilers.

At the moment, graph-based methods are the only way to schedule large programs within a bareable space of time. The quality of the schedule could be improved by integrating ILP methods into heuristics, that could identify certain code fragments and schedule them optimally using ILP.

References

[Ana91] Analog Devices. *ADSP-21020 User's Manual*, 1991.

[Ana95a] Analog Devices. *ADSP-21000 Family Assembler Tools and Simulator Manual*, 1995.

[Ana95b] Analog Devices. *ADSP-21000 Family C Tools Manual*, 1995.

[Ana95c] Analog Devices. *ADSP-2106x SHARC User's Manual*, 1995.

[Ana96] Analog Devices. *ADSP-2106x SHARC DSP Microcomputer Family*, 1996.

[Bas95] S. Bashford. Code Generation Techniques for Irregular Architectures. Technical Report 596, Universität Dortmund, November 1995.

[Brü95] W. Brüggemann. *Ausgewählte Probleme der Produktionsplanung*. Physica Verlag, Heidelberg, 1995.

[CPL95] CPLEX Optimization. *Using the CPLEX Callable Library*, 1995.

[CWM94] 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.

[DK96] W. Dinkelbach and A. Kleine. *Elemente einer betriebswirtschaftlichen Entscheidungslehre*. Springer, 1996.

[Ell86] J.R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.

[Fis81] J.A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30(7):478 – 490, July 1981.

[GE92] C. H. Gebotys and M.I. Elmasry. *Optimal VLSI Architectural Synthesis.* Kluwer Academic, 1992.

[GE93] 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.

[GS90] 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.

[Käs97] Daniel Kästner. Instruktionsanordnung und Registerallokation auf der Basis ganzzahliger linearer Programmierung für den digitalen Signalprozessor ADSP-2106x. Master's thesis, Universität des Saarlandes, 1997.

[Lan97] Marc Langenbach. Instruktionsanordnung unter Verwendung graphbasierter Algorithmen für den digitalen Signalprozessor ADSP-2106x. Master's thesis, Universität des Saarlandes, 1997.

[LDSM80] David Landskov, Scott Davidson, Bruce Shriver, and Patrick W. Mallet. Local Microcode Compaction Techniques. *ACM Computing Surveys*, 12(3):261–294, 1980.

[Nic85] Alexandru Nicolau. Uniform parallelism exploitation in ordinary programs. In *International Conference on Parallel Processing*, pages 614–618. IEEE Computer Society Press, August 1985.

[NKT89] 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.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley and Sons, New York, 1988.

[PS82a] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, 1982.

[PS82b] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity*, chapter 13, pages 318 – 322. Prentice-Hall, Englewood Cliffs, 1982.

[SCL96] 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.

[WM97] R. Wilhelm and D. Maurer. *Übersetzerbau. Theorie, Konstruktion, Generierung; zweite, überarbeitete und erweiterte Auflage.* Springer, Berlin; Heidelberg; New York, 1997.

[Zha96] L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming.* PhD thesis, Technische Fakultät der Universität des Saarlandes, 1996.