ILP-based Approximations for Retargetable Code Optimization

Daniel Kästner
Universität des Saarlandes &
AbsInt Angewandte Informatik GmbH
Saarbrücken, Germany
kaestner@absint.com

September 21, 2001

Embedded systems are characterized by high performance requirements and are subject to severe cost and power consumption restrictions. This has led to the development of specialized irregular hardware architectures for which traditional code generation and optimization techniques usually fail to generate machine code of satisfactory quality. The PROPAN system has been developed as a framework for generating machine-dependent postpass optimizations and analyzes on assembly level. From a concise hardware specification a machine-sensitive code optimizer is generated that is based on integer linear programming (ILP). The ILP modeling allows to integrate several code optimization phases in a homogeneous problem description which is an important precondition for producing high-quality code for irregular hardware architectures. Due to the complexity of phase-coupled code optimization the time for exactly solving the generated integer linear programs can grow high. In this article novel approximation techniques for ILP-based code optimization are presented. The presented approximation techniques mostly produce optimal results while reducing the computation time by orders of magnitude compared to the exact solution. The PROPAN framework has been retargeted to several representative standard digital signal processors. Practical experiments demonstrate the applicability of this approach.

1 Introduction

1.1 Embedded Processors

During the last years, the markets for telecommunication, embedded systems, and multimedia applications have been rapidly growing. The distinct cost sensitivity of these markets in connection with the stringent time constraints of real-time applications have led to the development of specialized, irregular hardware architectures designed to efficiently execute typical applications of digital signal processing. In the area of general-purpose processors, compiler technology has reached a high level of maturity. For irregular architectures however, the code quality achieved by traditional high-level language compilers is often not satisfactory [33, 26]. Generating efficient code for irregular architectures requires highly optimizing techniques that have to be aware of specific hardware features of the target processor. Since such techniques are usually not provided by standard compilers, many digital signal processing applications are developed in assembly language – a time-consuming and error-prone task suffering from bad portability and bad maintainability. The increasing complexity of embedded applications and the shrinking design cycles of embedded processors render this approach increasingly unacceptable. Therefore, there is an urgent need for *retargetable* code generation and optimization techniques that can be quickly adapted to different target architectures, e. g. by deriving machine-specific properties from a dedicated hardware description. At the same time these techniques must be able to produce high-quality code for the modeled architectures.

1.2 Code Generation

The code generation process can be subdivided into several subtasks most of which are $\mathcal{NP}$-complete problems: code selection, register allocation, instruction scheduling, register assignment and functional unit binding. In classical approaches they are addressed in separate phases by heuristic algorithms. Since these phases are interdependent, decisions made in one phase impose constraints to the subsequently addressed phases. This can lead to a suboptimal combination of suboptimal partial solutions resulting in a very poor code quality – especially for irregular architectures [33].

Search-based techniques allow to model the interactions of code generation phases in an exact way. One such technique – that additionally supports easy retargetability – is *integer linear programming* [24]. The use of integer programming models has increased significantly during the last years which is mostly due to the advances in algorithms for solving integer programs and the availability of reliable software packages [18]. Computing an optimal solution of an integer linear program is $\mathcal{NP}$-complete [10]. Recent research however has lead to an understanding of polyhedral properties that can be used to develop well-structured formulations permitting efficient computations [6, 7]. Moreover, as will be sketched in this article, ILP-based approximations allow to compute high-quality solutions in considerably reduced time compared to the exact solution.

1.3 The PROPAN System

The PROPAN system [22, 20, 24] has been developed as a retargetable framework for high-quality code optimizations and machine-dependent program analyzes at assembly level. From a concise hardware specification a machine-sensitive postpass optimizer is generated that is based on integer linear programming (ILP). It reads previously generated assembly programs and optimizes them. Experimental results show that large improvements can be achieved. In the past, research on retargetability has mainly focused on closed compilation systems [25, 26, 15, 14]. Using such a system in industry however mostly requires replacing the existing compiler infrastructure which causes high costs. Thus the use of retargetable compilers in industry is rare. Due to the postpass orientation, PROPAN can be integrated in existing tool chains with moderate effort and allows to improve the code quality of existing compilers. Thus the costs associated with changing the compiler infrastructure in a company can be avoided.

1.4 Overview

In this article ILP-based approximation techniques for two well-structured ILP-formulations for phase-coupled code optimization are presented. They have been implemented in the retargetable PROPAN framework [24]. Sec. 2 presents a short summary of related work. The basic ideas of the ILP modeling and the ILP-based approximations are outlined in Sec. 4. The PROPAN system and the experimental evaluation of the ILP-based optimizers for two contemporary digital signal processors, the Analog Devices ADSP-2106x SHARC and the Philips TriMedia TM1000, are summarized in Sec. 8; Sec. 9 concludes.

2 Related Work

During the last years the development of retargetable code generation systems has gained increasing attention. In most approaches the code generation subtasks and interactions between them are addressed by graph-based heuristic methods. The retargetable code generators MARION [4], CHESS [25], and CBC [8, 9] take phase interactions into account by heuristically estimating the effect of code generation decisions in one phase to other phases. The Express [14] and AVIV [15] compilers offer a heuristic coupling of code generation phases. In all these systems the quality of the generated code depends to a large degree on the quality of the chosen heuristics. The efficiency of a heuristics in turn depends on the specific target architecture so that a conflict between the goals of high-quality code and easy retargetability arises.

A higher code quality can be achieved by exact search-based methods – at the cost of higher compilation times. Bashford and Leupers [3] are developing a framework for phase-coupled code selection and register allocation using constraint logic programming. An early approach for ILP-based instruction scheduling for vector processors has been presented in [2]. In the area of architectural synthesis, the SILP [32] and the OASIC [11, 12] models have been developed for phase-coupled instruction scheduling and register assignment. Both have been extended in our work [20, 24] and will be summarized in more detail in Sec. 4. Leupers has developed a retargetable compiler for digital signal processors [26] where local instruction scheduling is performed by integer linear programming. Wilson et al. [27] use an ILP-formulation for simultaneously performing code selection, local instruction scheduling, register allocation and assignment. The complexity of the resulting formulations however leads to excessive computation times. An ILP model for local instruction scheduling and register allocation has been presented in [5]; experimental results are only reported for some small example programs and show excessive computation times. In [16] a well-structured ILP-formulation is presented that allows optimal instruction scheduling of large basic blocks for regular architectures in acceptable time.

For several classes of scheduling problems approximation algorithms have been developed that have polynomial time complexity and whose result is guaranteed to be at most ϵ times optimal (ϵ-approximations) [29, 13, 31, 30]. The basic idea of many of these algorithms is to use results of LP relaxations as criteria for making scheduling decisions. Mostly however they are restricted to "simple" classes of scheduling problems [29, 31, 30]. For some combinatorial problems it can be proved that there is no hope of finding an approximation algorithm of specified accuracy unless P=NP [10].

To our knowledge no ϵ-approximation for the problem of phase-coupled instruction scheduling, register assignment and resource allocation has been developed yet. Further complications are that additionally the control flow structure of the programs has to be taken into account. Moreover the code generation problem itself depends on the target architecture with respect to important parameters as the number of functional unit types, the number of instances of each functional unit type, and the execution times of operations. Due to the required retargetability, these machine properties cannot be assumed fixed. Since the ϵ-approximations are tailored to special problem classes more general approaches are required.

3 Code Generation

The task of a compiler is to translate input programs written in a high-level programming language into semantically equivalent machine programs of a specific target processor. Typically after syntactic and semantical analysis an intermediate representation of the program is computed which is used as the starting point of the code generation process. The task of code generation comprises several subtasks:

The goal of *code selection* is to map the intermediate representation to a semantically equivalent sequence of machine operations that is as efficient as possible.

The task of *register allocation* is to map the values of the intermediate representation to physical registers in order to minimize the number of memory references during program execution. It consists itself of two subtasks: *Register allocation proper* attempts to decide which variables and expressions are mapped to registers and which ones are kept in memory. The second subtask is called *register assignment*; its goal is to determine the physical registers that are used to store the values that have been previously selected to reside in registers.

Instruction scheduling is the task of reordering the produced instruction stream in order to minimize pipeline stalls and to exploit the available instruction-level parallelism thus minimizing execution time.

Resource allocation is concerned with binding operations to machine resources, e. g. functional units or buses; it is also called *functional unit binding*.

The task of instruction scheduling is to rearrange a code sequence in order to minimize the required execution time. During reordering, the data dependences of the operations have to be respected. Data dependences are caused by the order of read and write accesses to the components of the machine states, e. g. registers or memory cells. The data dependences induce a partial ordering $\prec$ among the operations; $i \prec j$ holds if operation j depends directly or indirectly on operation i.

Thus in its simplest form, instruction scheduling corresponds to the classical NP-complete problem of *precedence constrained scheduling*: Let a set V of tasks t with length $l(t) = 1$, m processors, a deadline D and a partial order $\prec$

on V be given. Is there an m-processor schedule for V that meets the deadline D and does not violate the precedence constraints? The tasks correspond to machine operations and the machines represent parallel functional units of the underlying processor, e.g. ALUs, multipliers, etc.

For real-world hardware architectures, the problem usually is more complex. In general each machine operation (task) has to be executed by one specific functional unit (dedicated machines) and the execution time (task length) can be any nonnegative integer. Another complication is that many architectures exhibiting instruction-level parallelism dispose of several identical functional units that represent instances of one functional unit type. In this case additional resource constraints have to be taken into account such that in the classification of [10] the instruction scheduling problem corresponds to precedence and resource constrained scheduling. The complexity further rises since often the assignment of operations to functional units is not uniquely determined and irregular restrictions of parallelism have to be taken into account.

Let V denote the set of machine operations from the input program. Before generating the integer linear programs, an interval is calculated for each operation i containing all clock cycles in which the execution of i can be started in any feasible schedule. This interval is defined as $N(i) = \{asap(i), \ldots alap(i)\}$. The $asap$ ($alap$) control step is the earliest (latest) control step in which i can be started without violating any data dependences. The interval $N(i)$ is refined using information about the availability of hardware resources [24].

4 ILP Models for Code Generation

ILP-formulations for code generation problems can be classified as time-indexed or order-indexed models [23]. In time-indexed formulations, the choice of the decision variables is based on the point of time the modeled event is assigned to. In order-indexed approaches, the decision variables reflect the ordering of the modeled events; the assignment to points of time is implicitly given.

Our work is based on two well-structured ILP formulations, the order-indexed SILP model [32, 20, 24] and the time-indexed OASIC-model [12, 20, 24]. In [24] we could establish a connection between the hardware design of the target processor and the appropriate ILP modeling style. The order-indexed SILP modeling allows an efficient integration of the register assignment problem and is well suited for irregular architecture where the resource competition is high. Time-indexed modeling presents itself for architectures with a high degree of instruction-level parallelism where a large number of functional unit types is available for the execution of each microoperation.

In [24] we have adapted the SILP and OASIC models to the postpass optimization problem and extended them to incorporate irregular hardware characteristics and exceed basic block boundaries. A novel superblock concept allows to extend the optimization scope across basic block and loop boundaries. For each superblock, an individual integer linear program is generated [24]. Since the approximative methods are also applicable for the extended formulations we will present only the basic versions of the SILP and OASIC formulations in the following.

4.1 SILP Formulation

In the order-indexed SILP model, the main decision variables describe the flow of resources through the instructions of the program: $x_{ij}^k \in \{0,1\}$ indicates whether operation i passes an instance of resource type k to operation j. The starting time for the execution of each operation is implicitly determined by the calculated resource flow.

Apart from the flow variables x_{ij}^k, the following types of decision variables are used: $t_i \in \mathbb{Z}$ denotes the starting time for the execution of operation i, $w_j \in \mathbb{Z}$ the number of control steps required to execute operation i. The latency of the functional unit executing operation j, i.e. the minimal time interval between successive data inputs to this functional unit is represented by $z_j \in \mathbb{Z}$. The set of all operations in the input program is denoted V_I. Finally $R_k \in \mathbb{Z}$ denotes the number of instances of resource type $k \in V_K$.

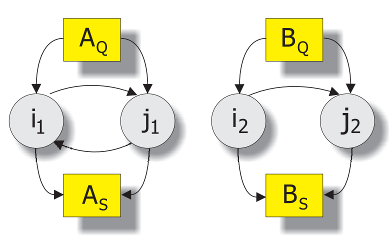


Figure 1: Example of a Resource Flow Graph

The ILP is generated from a *resource flow graph* $G_F = (V_F, E_F)$. This graph describes the execution of a program as a flow of the available hardware resources through the operations 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 $k_Q, k_S \in V_F$ where the nodes k_Q represent the sources and the nodes k_S the sinks. The first operation 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 operation using the same resource type. The last operation 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. Each edge (i, j) of the resource flow graph $(i, j) \in E_F^k$ 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 $x_{ij}^k = 1$.

Fig. 1 shows an example resource flow graph for two resource types A and B. The operations that have to be executed by A are independent, so there is no restriction of the operation ordering. However in the subgraph of resource type B, we can see that there is only one feasible ordering of the operations what can, e. g. , be caused by data dependences.

The basic idea of this modeling is that operations that are mapped to different resource types can be executed in parallel if this is not prevented by data dependences. The number of operations that can be simultaneously executed by the same resource type is given by the number of instances of this resource type. This way, the VLIW execution model is covered where an instruction can be composed of several micro-operations that are executed in parallel.

The goal of the 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). So the objective function and the time constraints required to define the total execution time M_{steps} can be given as follows:

$$\begin{aligned} \min \quad & M_{steps} && (1) \\ t_j \quad \leq \quad & M_{steps} \quad \forall\, j \in V_I && (2) \end{aligned}$$

The precedence constraints are used to represent the data dependences of the input program. When operation j depends on operation i, then j may be executed only after the execution of i is finished. In the following, only the constraint for true dependences is shown; other data dependence types are modeled analogously.

$$t_j - t_i \quad \geq \quad w_i \quad \forall\, i \prec j \qquad (3)$$

The flow conservation constraints (equation (4)) assert that the value of the flow entering a node is equal to the value of the flow leaving that node. Moreover, each operation must be executed exactly once by one hardware component. This is guaranteed by equation (5). The resource constraints (6) are necessary, since the number of available instances of all resource types must not be exceeded.

$$\sum_{(i,j)\in E_F^k} x_{ij}^k - \sum_{(j,i)\in E_F^k} x_{ji}^k \quad = \quad 0 \;\forall\, i,j \in V_I \quad (4)$$

$$\sum_{\substack{k\in V_K:\\ (j,k)\in E_R}} \sum_{(i,j)\in E_F^k} x_{ij}^k \quad = \quad 1 \;\forall\, j \in V_I \quad (5)$$

$$\sum_{(k,j)\in E_F^k} x_{kj}^k \quad \leq \quad R_k \;\forall k \in V_K \quad (6)$$

Finally, the serial constraints are required which guarantee a correct sequencing among independent operations assigned to the same resource instance (equation (7)).

$$t_j - t_i \geq z_i + \Big(\sum_{\substack{k\in V_K:\\ (i,j)\in E_F^k}} x_{ij}^k - 1\Big)\alpha_{ij} \;\forall (i,j) \in E_F^k \quad (7)$$

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 $x_{ij}^k = 1$. Here α_{ij} is a constant that represents an upper bound of the maximal distance between i and j.

The integer linear program presented above only covers the problem of instruction scheduling and resource allocation. The problem of register assignment is also modeled as a flow problem; the required constraints are similar to the instruction scheduling constraints. More details can be found in [24].

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 $\mathcal{O}(n^2)$.

In [32] it has been shown that the polytope of the precedence constraints is integral if the execution time of the machine operations does not depend on the functional unit binding. In [24] we have proven that the polytopes of the flow conservation constraints (equation (4)), execution constraints (5) and the resource constraints (6) are integral. The combined polytope of the flow conservation constraints and the execution constraints is not integral in the general case whereas the combined polytope of the flow conservation and the resource constraints as well as the combined polytope of the execution and the resource constraints are integral. The same integrality properties also hold for the polytopes of the corresponding register flow constraints. Thus, deviations of the feasible region of the SILP model from the integral polytope are only due to the serial constraints and to the combination of flow conservation and execution constraints.

4.2 OASIC Formulation

In [12] a time-based formulation is presented where the main decision variables describe the assignment of operations to control steps. The flow of the resources is determined implicitly by the operation sequencing. The main decision variables are called $x_{jn}^k \in \{0,1\}$, where $x_{jn}^k = 1$ means, that microoperation j is assigned to the nth control step ($n \geq 1$) in the generated schedule and is executed by an instance of resource type k. In this model, the t-variables are derived from the main decision variables x_{jn}^k:

$$t_i = \sum_{k:(i,k)\in E_R} \sum_{n\in N(i)} n \cdot x_{in}^k$$

In [21, 24] the ILP formulation of [12] is adapted to our problem setting; its basic structures is presented in the following. The objective function and the time constraints are identical to the SILP formulation:

$$\min \quad M_{steps} \tag{8}$$

$$t_j \leq M_{steps} \quad \forall\, j \in V_I \tag{9}$$

The data dependences of the input program are represented by the precedence constraints which ensure that an operation j must not be started before operation i if there is a data dependence from i to j. Again only the constraints for true dependences are shown:

$$\sum_k \sum_{\substack{n_j \leq n \\ n_j \in N(j)}} x_{jn_j}^k + \sum_k \sum_{\substack{n_i \geq n-w_i+1 \\ n_i \in N(i)}} x_{in_i}^k \leq 1 \;\forall i \prec j,$$
$$n \in \{n' + w_i - 1 | n' \in N(i)\} \cap N(j)) \tag{10}$$

Additionally, assignment constraints are required which ensure that the execution of an operation starts in exactly one control step and is performed by exactly one resource type.

$$\sum_k \sum_{n\in N(j)} x_{jn}^k = 1 \qquad \forall\, j \in V_I \tag{11}$$

Finally, the resource constraints guarantee that the number of available instances of each resource type must not be exceeded, so that in no control step, more than R_k operations may be executed by resource type k. Let $N_{jm} = \{n : m = n + p,\ 0 \leq p \leq w_j - 1\}$, then the constraints read as follows:

$$\sum_{j\in V_I} \sum_{n\in N_{jm}\cap N(j)} x_{jn}^k \leq R_k \qquad \forall m \leq M_{steps} \tag{12}$$

In [11, 12] it has been shown that the polytope of the precedence constraints is not integral. The formulation of the constraints (10) however respects integral facets derived from a transformation to the node packing polytope. Thus while the polytope cannot be guaranteed to be integral it represents a good approximation to the integral polytope [11, 12]. In [24] we have proven that the polytopes of the execution constraints, of the resource constraints and the combined polytope of the execution and the resource constraints are integral. Thus any deviations of the feasible region from the integral polytope are due to the precedence constraints.

5 ILP-Based Approximations

Since register assignment and instruction scheduling are NP-complete problems even when addressed in isolation there is an upper bound on the size of the code sequences for which a provably optimal solution can be computed in practicable time. In order to be able to address code sequences exceeding this bound we have developed approximative methods for the code optimization problem that cannot guarantee an optimal solution but lead to a significant reduction of computation time while still obtaining a high solution quality.

The basic idea of the approximations is to iteratively solve *partial relaxations* of the original problem. After the solution of one partial relaxation has been computed certain parts of the solution that are guaranteed to be integral are extracted. The corresponding variables are fixed to their current value by additional equality constraints. In subsequent iterations they are treated as constants and do not contribute to the problem complexity any more. Then the next partial relaxation is addressed. In the end an integral solution is obtained. There is no analytically determined performance guarantee that holds for all possible problem instances. Our experimental results however show that in most cases optimal solutions can be obtained.

Previous studies [21] have shown that a simple rounding mechanism cannot be considered promising since the resulting code quality is not satisfactory and due to the need for backtracking the computation can take longer than the calculation of an exact solution. Thus more elaborate approaches are required.

6 SILP-Approximations

The approximations developed for the SILP formulation start with an initial relaxation where the integrality restrictions of all flow variables are removed. The set X^{rel} denotes the set of all relaxed flow variables, i. e. of all flow variables that are explicitly specified as integers in the original ILP formulation.

6.1 Stepwise Approximation

The algorithm of the *stepwise approximation* is based on an approximative method sketched in [32]. It consists of two phases; first the flow variables of the execution units are addressed and then the register flow variables. Each phase starts with computing an optimal solution of the current relaxed mixed integer linear program. The core of each phase is a loop that iterates over all control steps c until an upper bound for the execution time of the input program is reached.

Let X_R be the set of all flow variables $x_{ij}^k \in X^{rel}$ such that k is a functional unit type $k \in N_R^F$, $t_i + z_i - 1 \geq c$, $t_i \leq c$, $t_j + z_j - 1 \geq c$, and $t_j \leq c$ in the solution of M_1. Then in the current schedule both operations i and j use functional units of type k such that those are not ready to accept new data inputs in control step c. All variables contained in X_R are specified as binary and the resulting mixed integer linear program is solved. Thus flow variables are specified only as binary when this is inevitable in to prevent violations of the serial constraints with respect to the current control step c.

Then variables $x_{ij}^k \in X_R$ that are associated with an operation scheduled to control step c

($t_i = c \vee t_j = c$) and that have value 1 in the current solution are fixed to 1. In subsequent optimization steps they can be considered as constants and do not contribute to the problem complexity any more. Fixing variables $x_{ij}^k \in X_R$ where $x_{ij}^k = 0$ can make the problem infeasible since non-integral values of other variables might shadow violations of the flow constraints. Due to the structure of the flow constraints this cannot occur when only fixing the variables with value 1. Since the definition of X_R implies that the algorithm can only fix flow variables between independent operations the fixations do not lead to violations of the precedence constraints. Thus the feasibility of the modified MILP is ensured.

In the second phase of the algorithm the same proceeding is repeated for the register flow variables. After all control steps have been traversed, the flow variables of X^{rel} that have not been fixed to an integer up to now are specified as binary and the resulting MILP is solved. This solution is guaranteed to be integral since all previously relaxed variables have been fixed to an integral value, or have been respecified as binary.

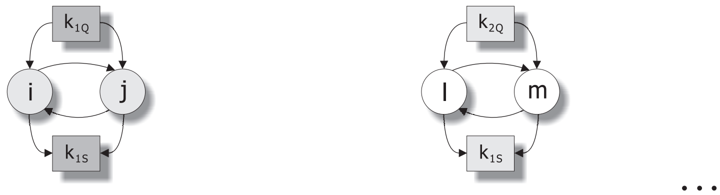
In the main loop of the algorithm only operations whose current scheduling violates the serial constraints are specified as binary. This way the number of variables simultaneously specified as integers is reduced. Then an optimal solution of the resulting mixed integer linear program is computed and the ordering among the colliding operations is fixed. In addition to further reducing the number of variables simultaneously specified as binary the fixations have the effect that serial constraints are transformed into precedence constraints. Deviations of the SILP polytope from the integral polytope are mostly caused by the serial constraints. Thus with each fixation one serial constraint is effectively eliminated such that the resulting polytope approximates the integral polytope more closely. In consequence a significant reduction of computation time can be achieved.

6.2 Isolated Flow Analysis

In the SILP formulation a separated flow network is generated for each resource type. The concept of resource flows offers a natural method of problem decomposition that is at the basis of the *approximation of isolated flows*: The resource types of the target processor are addressed iteratively; only the flow variables associated with the current resource type are specified as binary. This way the number of variables that are simultaneously specified as binary can be significantly reduced.

In the initial relaxation, all flow variables $x_{ij}^k \in X^{rel}$ are relaxed. In the main loop of the algorithm the hardware resources of the target processor are traversed. In each iteration, the flow variables associated with the current resource type k are specified as binary and the modified MILP is solved. Then the flow variables x_{ij}^k of the current resource type k are fixed to their current value by additional equality constraints and the next resource type is addressed. This is repeated for all resource types, so a feasible integral solution is obtained in the end.

In each iteration only the flow variables of one resource type k are specified as binary. The variables of the other resource types either have been addressed in a previous iteration and can be considered as constants or they are still relaxed



$x_{ij}^{k_1}, x_{ji}^{k_1}, \ldots \in \{0,1\}$ $\qquad$ $0 \leq x_{lm}^{k_2}, x_{ml}^{k_2}, \ldots \leq 1$

Figure 2: Isolated flow analysis.

(see Fig. 2).

The order in which the resource types of the target processor are traversed is based on the number of flow variables x_{ij}^k associated with each resource type k. First the resource types with the lowest number of flow variables are addressed. This way the search space for the largest subproblems (in terms of associated flow variables) is reduced due to the fixations of the preceding iterations.

In each iteration of the algorithm an optimal solution with respect to the current resource type is computed. Since no constraints are omitted the complete problem information is used such that the solution is also influenced by the effects on other resource flows. While no optimal solution can be guaranteed a high solution quality can be expected since the overall solution is composed of the individually optimal solutions of all subproblems. The experimental results indicate that in fact an optimal solution is obtained in most cases. The computation time is reduced significantly since only the flow variables associated to one resource type are considered as binary at a time and the search space is tightened after each approximation step.

6.3 Stepwise Approximation of Isolated Flows

The *stepwise approximation of isolated flows* is a combination of the two previously presented algorithms. Again all resource types of the target processor are traversed, but instead of exactly solving the decomposed problem as in the approximation of isolated flows, in each iteration the stepwise approximation is used for the variables of the current resource type. Thus for each resource type, all control steps are traversed in an inner loop until an upper bound for the execution time of the input program is reached. In each iteration of the inner loop the variables of the current resource type are specified as binary if they cause violations of serial constraints with respect to the current control step c. Fixations of variables induce a partial ordering among the operations and reduce the number of active serial constraints. After all control steps have been traversed, feasible resource flows have been computed for the current resource type. After all resource types have been traversed, a feasible integer solution results.

The setup overhead of this approximation is larger than that of the previously presented approximations so that the computation times for small input programs are higher. For larger input programs however experimental results show that in most cases the stepwise approximation of isolated flows leads to the shortest computation time while producing solutions of very high quality.

7 OASIC-Approximations

Problem decomposition is more difficult in the OASIC formulation than in the SILP formulation. In the OASIC model each decision variable x_{in}^k describes the starting time of an operation and the functional unit it is assigned to. After fixing a variable x_{in}^k, the starting time of the operation, its completion time and the functional unit it is assigned to become invariant.

In contrast to this, if a flow variable x_{ij}^k of the SILP formulation is fixed, the assignment to functional units and the relative ordering of the operations i and j are fixed; the starting time however may vary. If the resource flows are addressed iteratively, after each iteration the operations of the current resource type are totally ordered but still the distance between them can be changed, if this is necessary due to the sequentialization of operations of other resource types.

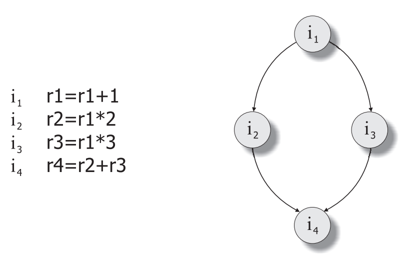


Figure 3: Fixation problem for time-indexed models.

If a variable x_{in}^k of the time-based OASIC formulation is fixed, its starting time cannot be changed later on. The immediate consequence is that a problem decomposition based on resource types is not possible. This can be illustrated by a simple example. Consider the situation of Fig. 3. Assume that the target architecture disposes of one ALU and one multiplier. If only variables associated with the ALU are specified as binary, it would be possible to set $t_{i_1} = c$ and $t_{i_2} = c + 2$. The necessary sequentialization of i_2 and i_3 will only become apparent in the next iteration when the multiplier variables are specified as binary. But then there would be no feasible solution any more while in the SILP formulation the modification of the starting time would still be possible.

Thus among the approximations presented in Sec. 6.1 – Sec. 6.3 only the stepwise approximation can be applied to the OASIC formulation. A difference to the stepwise approximation of the SILP formulation is the usage of the lookahead set $L^A(l, c)$ where l is a lookahead value and c denotes the current control step. The lookahead can be specified as a command line parameter in the PROPAN-system. For a lookahead of l cycles the set $L^A(l, c)$ is defined as follows:

$$L^A(l, c) = \{x_{in}^k \mid x_{in}^k \in X^{rel} : alap(i) \leq c + l\}$$

It contains the relaxed decision variables associated with all operations whose latest possible starting time is reached before l cycles from the current control step. Those variables are specified as binary.

In the time-indexed OASIC formulation for each control step n where the execution of an operation i may be started a dedicated variable

x_{in}^k is introduced. The set of feasible control steps is determined by *asap*/*alap* analysis with the help of an upper bound of the execution time of the input program. No variable will be generated that allows an operation to be scheduled after its *alap* control step. However it is a reasonable approach to use the execution time of the schedule of the input program as an upper bound and enforce this upper bound by an additional constraint in the generated integer linear programs. But then suboptimal decisions during the approximation could force an operation to be scheduled after its *alap* control step which leads to infeasibility. The lookahead can be used to prevent that situation. In the experimental results a lookahead of two cycles has been sufficient.

First the initial relaxation is solved where all variables x_{in}^k associated with functional units of type $k \in N_R^F$ are relaxed. Then all control steps are traversed until an upper bound for the execution time of the input program has been reached. In each iteration a set X is computed that contains all relaxed variables associated with the current control step c and the variables of the lookahead set $L^A(l, c)$. The variables $x_{in}^k \in X$ are specified as binary and the modified mixed integer linear program M_2 is solved. Then for all variables $x_{in}^k \in X$ where $x_{in}^k = 1$ in the solution of M_2 a constraint is added to the MILP fixing that variable to 1. Additionally all other variables from the set X that are associated with operation i can be set to 0. Finally all variables $x_{in}^k \in X$ are relaxed again and the next control step is addressed. The fixed variables are treated as constants in the next iteration and do not contribute to the problem complexity any more.

Again in each iteration variables are only specified as binary where this is inevitable in order to obtain a feasible solution. The number of variables that are simultaneously specified as binary is reduced such that the computation is significantly accelerated. As the experimental results show in most cases an optimal result is obtained.

8 Experimental Results

The PROPAN system (*Postpass-oriented Retargetable Optimizer and Analyzer*) has been designed as a retargetable framework for postpass optimizations and analyzes. PROPAN combines the issues of machine description driven retargetability and of search-based postpass optimizations to improve the quality of machine code especially for irregular hardware architectures.

The retargetability concept of PROPAN is based on the combination of generic and generative techniques. The core system is composed from generic and generated program parts. Generic program parts are independent from the target architecture and can be used for different processors without any modification. If hardware-specific knowledge is required all information is retrieved in a standardized way from an architecture 'database' that is generated from a central machine description. The program parts that change with the specific target architecture are automatically generated from the machine description. Thus retargeting the system to a different architecture only requires an appropriate specification of the target machine. Additionally the integration of user-supplied functionality is supported.

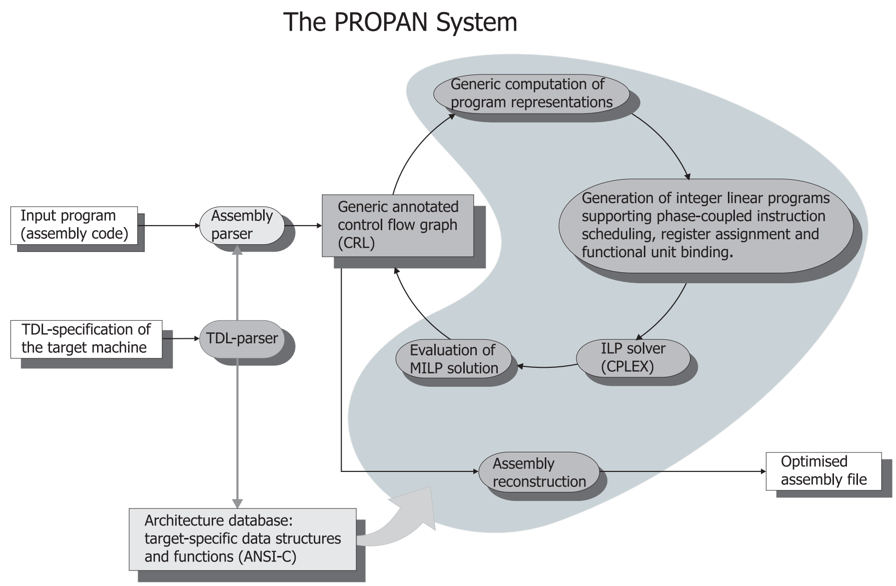


Figure 4: The PROPANsystem.

This may include dedicated program analyzes or hardware-specific program transformations. Those external program parts can communicate with the core system via a well-defined interface and can access the architecture database generated from the machine description.

A detailed overview of the PROPAN framework is shown in Fig. 4. The input of PROPAN consists of a TDL-description of the target machine and of the assembly programs that are to be analyzed or optimized. TDL (*Target Description Language*) [19, 24] is a dedicated machine description language that allows to specify the hardware resources of the target processor, its instruction set, the assembly language and irregular hardware constraints [24]. The TDL specification is processed once for each target architecture; from the TDL description a parser for the specified assembly language and an architecture database are generated. The architecture database consists of a set of ANSI-C files where data structures representing all specified information about the target architecture and functions to initialize, access and manipulate them are defined. The core system of PROPAN is generic; if hardware-specific knowledge is required the architecture database is referenced. For each target architecture, the generic core system is linked with the generated files yielding a dedicated hardware-sensitive postpass optimizer.

The generated assembly parser reads the input programs and computes their control flow graphs from which the necessary program representations as, e. g. , the data dependence and the control dependence graphs are calculated by generic algorithms. The central part of the PROPAN system is the modeling of phase-coupled code optimization by integer linear programming. In the last step of the optimization process the solutions of the integer linear programs are evaluated and the optimized assembly file is generated.

The PROPAN framework has been retargeted to several different target architectures. It has been used to generate ILP-based postpass optimizers for two widely used contemporary digital signal processors with considerably different hardware characteristics, the Analog Devices ADSP-2106X [1] and the Philips TriMedia TM1000 [28]. The experiments have been performed under SunOS 5.7 on a SPARC Ultra-Enterprise 10000; the mixed integer linear programs are solved by the CPLEX 6.5 library [17].

8.1 The ADSP-2106x SHARC

The ADSP-2106X is a digital signal processor with an irregular hardware architecture and restricted instruction-level parallelism. The instruction scheduling interacts with the register assignment problem: some operations can only be executed in parallel if all operands are assigned to specific register groups within the heterogeneous register file.

The optimizer generated for the ADSP-2106X is based on the SILP modeling and performs integrated instruction scheduling and register reassignment. All target-specific information is provided by the TDL specification. The investigated input programs are typical applications of digital signal processing. They comprise the computation routines of the DspStone benchmark [33] that have been compiled to assembly code by the `gcc`-based `g21k` compiler. The input programs contain between 14 and 55 microoperations. An overview of the results for some representative input programs is given in Tab. 1. The number of compacted instructions in the exact, i. e., provably optimal ILP-based solution (e) is compared to the approximation by isolated flows (a_1), the stepwise approximation by isolated flows (a_2), the stepwise approximation (a_3) and a generic list scheduling algorithm with the common highest-level-first heuristic (`l`). For each input program the number of microoperations is given in braces; the column I denotes the number of compacted instructions in the optimized code. In order to ensure a conservative comparison, an optimal register assignment is given as an input to the list scheduling algorithm while it is explicitly computed by the ILP-based methods. A time limit of 8 hours has been specified for the exact ILP-based solution; if it is exceeded, the best feasible solution found so far is returned; the corresponding programs are marked with `(p)`.

The approximation by isolated flows and the stepwise approximation produce an optimal result for each input program, the stepwise approximation by isolated flows exceeds the optimal number of instructions by one for one input program (`waveleti`). The result of the established list scheduling algorithm exceeds the optimal number of instructions for four input programs. The maximum overhead has been measured for `whetp3` where it exceeds the optimal number of instructions by 21.05%. The `g21k`-generated input code could be improved in all cases; on average, it exceeds the optimal number of instructions by 8.2%.

For the smaller input programs the exact ILP-based solution takes less time than the ILP-based approximations which is due to the setup overhead of the approximations. For the larger input programs however the computation time

Name	m	t	I
`dconvolu`	a_1	4.2”	14
(17)	a_2	8.7”	14
	a_3	3.29”	14
	e	5.01”	14
	l	0.19”	14
`ddotprod`	a_1	3’ 21.03”	20
(22)	a_2	3’ 8.09”	20
	a_3	2’ 10.05”	20
	e	1h 24’ 18.21”	20
	l	0.39”	20
`dft`	a_1	10.4”	14
(26)	a_2	30.63”	14
	a_3	58.56”	14
	e	7.67”	14
	l	0.46”	14
`fir`	a_1	2.65”	8
(18)	a_2	8.14”	8
	a_3	4.43”	8
	e	1.3”	8
	l	0.17	9
`dmat1x3`	a_1	50.23”	22
(24)	a_2	49.78”	22
	a_3	3’ 34.87”	22
	e	7’ 42.67”	22
	l	0.48	22
`waveleti`	a_1	4h 16’ 44.84”	35
(49)	a_2	2’ 51.13”	36
	a_3	1h 45’ 2.53”	35
	e	8h 1’ 57.46”(p)	39
	l	2.76	37
`waveletk`	a_1	9’ 25.9”	29
(39)	a_2	40.22”	29
	a_3	2’ 58.6”	29
	e	8h 1’ 8.33”(p)	30
	l	1.41”	32
`whetp3`	a_1	15’ 2.88”	19
(26)	a_2	22’ 22.71”	19
	a_3	12h 37’ 58.47”	19
	e	8h 14’ 1.43”(p)	21
	l	0.86”	23

Table 1: Performance of the SILP-based optimizations for the ADSP-2106X.

can be reduced by orders of magnitude. The time required for the stepwise approximation strongly varies with the input programs; for some programs it is the fastest approximative method, for others it is the slowest. For the large input programs the stepwise approximation of isolated flows is faster than the approximation of isolated flows. For all input programs an ILP-based solution could be obtained within minutes.

Not surprisingly, the list scheduling algorithm is faster than the ILP-based methods; it produces its results in a few seconds. However the list scheduling algorithm addresses each basic block separately. The computation times for the ILP-based methods shown in Tab. 1 are those that have been measured when the complete input program is represented by a single integer linear program. If the ILP-based methods are restricted to basic-block level, the required computation times are reduced.

8.2 The Philips TriMedia TM1000

The TriMedia TM1000 is a multimedia processor with a VLIW architecture. Here the scheduling interacts with the slot allocation problem: all operations have to be assigned to issue slots in the instruction word and the assignment of operations to issue slots is restricted. Moreover, the operations have to be synchronized with respect to the write-back bus; no more than five operations may write their result simultaneously on the bus. Again, the corresponding optimizer is generated automatically from the TDL description. for the TM1000 with its VLIW architecture the time-indexed OASIC model is better suited than the order-indexed SILP formulation. The input programs have been generated with the highly optimizing Philips `tmcc` compiler and contain between 19 and 95 microoperations. Again they are typical applications of digital signal processing, mostly from the DSP-stone benchmark.

In Tab. 2 computation time and result are compared for the stepwise approximation (a), the exact solution when using the schedule of the input programs as start solution (e), the exact solution calculated from scratch (n) and a generic list scheduling algorithm with highest level first heuristic that performs resource allocation on the fly.

In most cases the ILP-based approximation produces optimal results, only for three of the input programs the number of compacted instructions exceeds the optimal number of instructions by one. The schedule produced by the generic list scheduling algorithm exceeds the optimal number of instructions on average by 7.29% (for all evaluated programs). The programs where it gives an optimal result offer not enough instruction-level parallelism for the allocation decisions to negatively affect the code quality. The assembly programs *dctk* and *fidctk* offer a high level of parallelism and make use of special DSP operations that can only be assigned to few issue slots. For those programs, the interaction between instruction scheduling and resource allocation has a significant effect which becomes apparent in the result of the list scheduling algorithm. The optimal number of instruction is exceeded by more than 25% respectively more than 42%.

Due to the approximation setup time, for the smaller input programs the exact ILP-based solution takes less time than the ILP-based ap-

Name	m	t	I
`dbiquadn`	a	3’ 31.73”	18
(41)	e	5.23”	18
	n	4’ 33.11”	18
	l	0.33”	20
`dconvolu`	a	43’ 37.78”	21
(47)	e	11h 2’ 25.42”	21
	n	19h 47’ 20.76”	21
	l	0.48”	21
`dmat1x3`	a	9’ 22.64”	25
(31)	e	4’ 12.53”	25
	n	6’ 6.25”	25
	l	0.17	25
`dncompup`	a	4’ 0.16”	14
(33)	e	11.92”	14
	n	5’ 9.81”	14
	l	0.20”	14
`dnrealup`	a	9’ 52.27”	16
(41)	e	4’ 47.33”	15
	n	12’ 11.56”	15
	l	0.31	16
`dcompmul`	a	4.34”	11
(19)	e	0.48”	11
	n	2.27”	11
	l	0.05”	11
`dctk`	a	9’ 30.25”	22
(95)	e	30.21”	21
	n	1h 20’ 7.75”	21
	l	3.89”	30
`fidctk`	a	3’ 0.22”	16
(74)	e	6.93”	15
	n	18’9.06”	15
	l	1.89”	19

Table 2: Performance of the OASIC-based optimizations for the TriMedia TM1000.

proximation. With increasing size of the input programs the computation time is significantly reduced compared to the exact solution (method n). For all investigated programs, an ILP-based solution could be obtained within some minutes. Another observation is that using the schedule of the input code as a start solution can speed up the calculation of the exact solution significantly (method e). Thus proving a given solution to be in fact optimal can be done comparatively fast. The list scheduling algorithm again produces its results within some seconds.

9 Conclusion and Outlook

The PROPAN system is a framework for generating machine-dependent postpass optimizations and analyzes that has been especially designed to perform high-quality optimizations for irregular architectures. The novel machine description language called TDL allows to concisely specify all relevant information about the target architecture. Irregular hardware constraints are modeled in a generic way that allows them to be exploited in generic search-based optimization algorithms. The core system of PROPAN has been implemented in a generic way; if target-specific information is required, the architecture database generated from the TDL description is referenced. For each target architecture the architecture database is linked with the generic core system yielding a dedicated hardware-sensitive postpass optimizer.

The optimizations are based on integer linear programming and allow a phase-coupled modeling of instruction scheduling, register assignment and resource allocation taking precisely into account the hardware characteristics of the target architecture. By using ILP-based approximations, the calculation time can be drastically reduced while obtaining a solution quality that is superior to conventional graph-based approaches. In contrast to most other phase-coupled approaches allowing an exact solution, the optimization scope is not restricted to basic blocks. A novel superblock concept allows to extend the optimization scope across basic block and loop boundaries. The applicability of our approach has been demonstrated by modeling two widely used processors with considerably different hardware characteristics.

Future research aims at different directions. There is ongoing work to model additional hardware architectures and to extend the TDL language to cover superscalar architectures with complex pipelines. By investigating the polytope structure of both ILP formulations, the constraints could be identified that contribute most to the required computation times. Future research aims at tightening the formulations in order to further improve the solution efficiency. Currently a separation algorithm is developed that allows to efficiently integrate the register assignment task also in the OASIC model.

References

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

[2] S. Arya. An Optimal Instruction Scheduling Model for a Class of Vector Processors. *IEEE Transactions on Computers*, C-34, November 1985.

[3] S. Bashford and R. Leupers. Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths. *Design Automation for Embedded Systems*, pages 1–50, 1999.

[4] D.G. Bradlee, R.R. Henry, and S.J. Eggers. The Marion System for Retargetable Instruction Scheduling. *Proceedings of the PLDI*, pages 229–240, 1991.

[5] C-M. Chang, C-M. Chen, and C-T. King. Using Integer Linear Programming for Instruction Scheduling and Register Allocation in Multi-Issue Processors. *Computers and Mathematics with Applications*, 34(9):1–14, November 1997.

[6] 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) Systems*, 2(4):456–471, December 1994.

[7] F. Eisenbrand. *Gomory-Chvatal Cutting Planes and the Elementary Closure of Polyhedra*. PhD thesis, Saarland University, 2000.

[8] A. Fauth. Beyond Tool-Specific Machine Descriptions. In *[27]*, chapter 8, pages 138–152. Kluwer, 1995.

[9] A. Fauth and A. Knoll. Translating Signal Flowcharts into Microcode for Custom Digital Signal Processors. *Proceedings of the IEEE International Conference on Signal Processing*, pages 65–68, October 1993.

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

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

[12] C.H. Gebotys and M.I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1266–1278, 1993.

[13] M.X. Goemans. Improved Approximation Algorithms for Scheduling with Release Dates. *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 591–598, 1997.

[14] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *Proceedings of the DATE99*, 1999.

[15] S. Hanono and S. Devadas. Instruction Scheduling, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the Design Automation Conference 1998*, San Francisco, California, 1998. ACM.

[16] M. Heffernan, J. Liu, and K. Wilken. Optimal Instruction Scheduling Using Integer Programming. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 121–133, June 2000.

[17] ILOG S.A. *ILOG CPLEX 6.5. User's Manual*, 1999.

[18] E.L. Johnson, G.L. Nemhauser, and M.W.P. Savelsbergh. Progress in Integer Programming: An Exposition. Technical Report LEC-97-02, Georgia Institute of Technology,

School of Industrial and Systems Engineering, Atlanta, CA 30332-0205, January 1997. http://tli.isye.gatech.edu/reports.html.

[19] D. Kästner. TDL: A Hardware and Assembly Description Language. Technical Report TDL1.3, Transferbereich 14, Saarland University, 1999.

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

[21] D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph Based Methods in Code Generation. Technical Report A/01/98, Saarland University, Saarbrücken, Germany, January 1998.

[22] D. Kästner and M. Langenbach. Code Optimization by Integer Linear Programming. In Stefan Jähnichen, editor, *Proceedings of the 8th International Conference on Compiler Construction CC99*, pages 122–136. Springer LNCS 1575, March 1999.

[23] D. Kästner and R. Wilhelm. Operations Research Methods in Compiler Backends. *Journal of Mathematical Communications*, 1999.

[24] Daniel Kästner. *Retargetable Code Optimisation by Integer Linear Programming*. PhD thesis, Saarland University, 2000.

[25] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation For Embedded DSP Processors. In *[27]*, pages 85–102. Kluwer, 1995.

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

[27] P. Marwedel and G. Goossens. *Code Generation for Embedded Processors*. Kluwer, Boston; London; Dortrecht, 1995.

[28] Philips Electronics North America Corporation. *TriMedia TM1000 Preliminary Data Book*, 1997.

[29] Martin W.P. Savelsbergh, R.N. Uma, and Joel Wein. An Experimental Study of LP-Based Approximation Algorithms for Scheduling Problems. Technical Report LEC-97-11, Georgia Institute of Technology, Atlanta, GA 30332-0205, 1997. http://tli.isye.gatech.edu/reports.html.

[30] A.S. Schulz. *Polytopes and Scheduling*. PhD thesis, Technische Universität Berlin, February 1996.

[31] A.S. Schulz. Scheduling to Minimize Total Weighted Completion Time: Performance Guarantees of LP-Based Heuristics and Lower Bounds. *Proceedings of the 5th International Conference on Integer Programming and Combinatorial Optimization*, pages 301–315, June 1996.

[32] L. Zhang. *SILP. Scheduling and Allocating with Integer Linear Programming*. PhD thesis, Saarland University, 1996.

[33] V. Zivojnovic, J.M. Velarde, C. Schläger, and H. Meyr. DSPSTONE: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, 1994.