PROPAN: A Retargetable System for Postpass Optimisations and Analyses

Daniel Kästner*
Saarland University, Saarbrücken, Germany
kaestner@cs.uni-sb.de

May 10, 2000

Abstract

PROPAN is a system that allows for the generation of machine-dependent postpass optimisations and analyses on assembly level. It has been especially designed to perform high-quality optimisations for irregular architectures. All information about the target architecture is specified in the machine description language TDL. For each target architecture a phase-coupled code optimiser is generated which can perform integrated global instruction scheduling, register reassignment, and resource allocation by integer linear programming (ILP). All relevant hardware characteristics of the target processor are precisely incorporated in the generated integer linear programs. Two different ILP models are available so that the most appropriate modelling can be selected individually for each target architecture. The integer linear programs can be solved either exactly or by the use of ILP-based approximations. This allows for high quality solutions to be calculated in acceptable time. A set of practical experiments shows the feasibility of this approach.

1 Introduction

During the last years, the markets for telecommunication, embedded systems, and multimedia applications have been rapidly growing. The stringent time constraints imposed by many of those applications in connection with severe cost constraints have led to the development of specialised, irregular architectures designed to efficiently execute typical applications of digital signal processing. Such hardware features include heterogeneous register files, simultaneous execution of multiple operations, support for low-overhead looping, and restricted connectivity of functional units to register files or buses.

In the area of general-purpose processors, compiler technology has reached a high level of maturity. However, the code quality achieved by traditional high-level language compilers for irregular architectures often cannot satisfy the requirements of the target applications [27, 33]. Generating efficient code for irregular architectures requires highly optimising techniques that must be aware of specific hardware features of the target processor. Since such techniques are usually not employed in standard compilers, many DSP applications are developed in assembly language. Due to the increasing complexity of typical applications and the shrinking design cycles of embedded processors, this approach becomes increasingly unacceptable. As a solution this article proposes retargetable postpass optimisation techniques that can be quickly adapted to different target architectures and yet produce high quality code. The goal of our work is the development of a fully retargetable, phase-coupled postpass optimisation framework that allows for an easy integration of program analyses and user-supplied optimisation routines.

Although several retargetable research compilers

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

have been developed during the last years (see Sec. 2), the use of such systems in industry is still rare. One reason is that changing the compiler leads to very high costs. So the question can be asked whether it is possible to improve the quality of existing compilers that fail to produce satisfying code for irregular architectures. A promising approach is using a postpass framework that can process inputs resulting from previous code generation passes. Previous studies [18] have shown that the integration of a postpass approach into existing tool chains can be done with moderate effort.

1.1 The Phase Coupling Problem

The process of generating code for high level language programs can be divided into several phases: code selection, register allocation, instruction scheduling, register assignment, and resource allocation. The goal of instruction scheduling is to rearrange a code sequence in order to exploit instruction level parallelism. The task of register allocation is to decide which variables and expressions of the intermediate representation are mapped to registers and which ones are kept in memory. The goal is to minimise the number of memory references during program execution. Register assignment is the task of determining the physical register that is used to store a value that has been previously selected to reside in a register[1]. The task of resource allocation is to allocate resources, e. g. functional units or buses to operations (see Sec. 5).

Most of these tasks are $\mathcal{NP}$-hard problems. In classical approaches they are solved separately by heuristic methods. However all these phases are interdependent, i. e., decisions made by one phase impose constraints on the following phases possibly leading to the generation of inefficient code. That problem is known as the phase coupling problem. In order to generate high-quality code for irregular architectures, it is essential that the phase coupling is addressed. However, due to complexity reasons a full phase integration is usually not possible. Instead, it is a promising approach to identify several phase groupings such that each of those groups is solved integratedly. One reasonable partition is a coupling of instruction selection and register allocation on the one hand, and of instruction scheduling, register assignment, and resource allocation on the other hand.

[1] Often the term register allocation is used in a more general sense to denote both the phases of register allocation proper and register assignment.

2 Related Work

In this overview, we will concentrate on code generation systems for irregular architectures. There are several approaches where phase coupling is done by heuristic methods. CHESS [22] has been designed as a retargetable code generation framework for fixed point DSPs. The target processor is modelled by instruction set graphs that are automatically extracted from an nML-description of the processor [6]. It is assumed that each instruction is executed in one machine cycle. Code selection, register allocation, and instruction scheduling are implemented as separated modules. The register allocation decisions are guided by estimates about the effects on instruction scheduling. This way, information is exchanged between the phases, but there is no true phase integration.

The retargetable CBC compiler [22] also specifies the target architecture using the nML language. An extended list scheduling algorithm is used that addresses the problems of register allocation and instruction scheduling in an integrated way. The scheduling and allocation decisions are guided by heuristics.

Express [12] integrates code selection and register allocation into instruction scheduling in a heuristic way (mutation scheduling) [23]. The information about the hardware is extracted from the machine description language Expression [12].

Apart from the heuristic approaches, there are also phase-integrated methods that allow to calculate exact, optimal solutions — usually at the cost of higher calculation times. Since in digital signal or real-time applications code quality is extremely important, higher calculation times are acceptable within reasonable limits. The AVIV retargetable code generator [13] builds on the SPAM library which is a retargetable code generation framework for digital signal processors [28]. AVIV uses a branch-and-bound algorithm that performs functional unit assignment, operation grouping, register bank allocation, and scheduling. Detailed register allocation is carried out as a second step. However this is work in progress and no final results are available yet.

Bashford and Leupers [3] are developing a framework for phase-coupled code selection and register allocation using constraint logic programming. Alternative solutions are kept as long as possible, so that the decision among them can be delayed until the scheduling phase.

There have been only few approaches to incorporate ILP-based methods into the code generation process of a compiler. An approach for ILP-based instruction scheduling for vector processors has been presented in [2]. Wilson et al. [22] use an ILP-formulation for simultaneously performing code selection, scheduling, register allocation and assignment; movements of operations across basic block boundaries are not addressed. The complexity of the resulting formulations however leads to prohibitive computation times. Leupers has developed a retargetable compiler for digital signal processors [21] where local instruction scheduling is performed by integer linear programming. Other ILP-based approaches have been developed in the context of software pipelining, e. g. [26, 10]; however those approaches mostly deal with homogeneous VLIW-like architectures.

3 The PROPAN Framework

In the following, the PROPAN framework is presented in more detail. PROPAN (Postpass Retargetable Optimiser and Analyser) has been designed as a retargetable framework for postpass optimisations and analyses. In Sec. 3.1, the architecture specification language TDL is presented; Sec. 3.2 gives a detailed overview of the design of PROPAN.

3.1 Specifying the Architecture

In order to allow for easy retargetability, a specification mechanism for the target architecture and its instruction set is required. That specification has to cover all information needed by optimisations and analyses and must allow a representation where the information can easily be accessed by target applications. Possible input formats are assembly code or compiler-specific intermediate code. PROPAN automatically generates a parser for the input language from the machine specification. Different analyses may require different views of the architecture. Therefore, the specification language should support different levels of abstraction and represent the information in a way that they can be easily accessed by target applications. In order to achieve high code quality, the generated optimisers must take the hardware characteristics of the target architecture precisely into account. Therefore irregular hardware properties have to be specified in a way that allows to automatically generate appropriate hardware-sensitive ILP constraints. Finally, the specification language has to be extendable and flexible in use, so that a wide range of architectures can be modelled.

There is a large number of existing architecture description formalisms, e. g. , [24], [6], [11], [12], [4]. Yet satisfying the demands mentioned above required the development of a dedicated specification language, called TDL (*Target Description Language*). Details about the design of TDL and a comparison to other architecture description languages can be found in [16]; in the following, only a short summary is given.

A TDL description has a modular structure. It is composed of a specification of the hardware resources, a description of the instruction set, a constraint section, an assembly section, and a pipeline section. In the resource section, all hardware resources that can influence the analyses and optimisations are introduced and their properties are specified by an extendable attribute mechanism. The description of the instruction set is given in the form of an attribute grammar [30]. The attribute mechanism allows to identify important properties of operations. In the constraint section, a set of constraints can be specified that have to be respected during code transformations in order to preserve the program semantics. This includes, e. g. , restrictions of the available parallelism of the target machine or dependences between instruction scheduling and register assignment (see Sec. 4.4). The assembly section deals with syntactic details of the assembly language. Typically

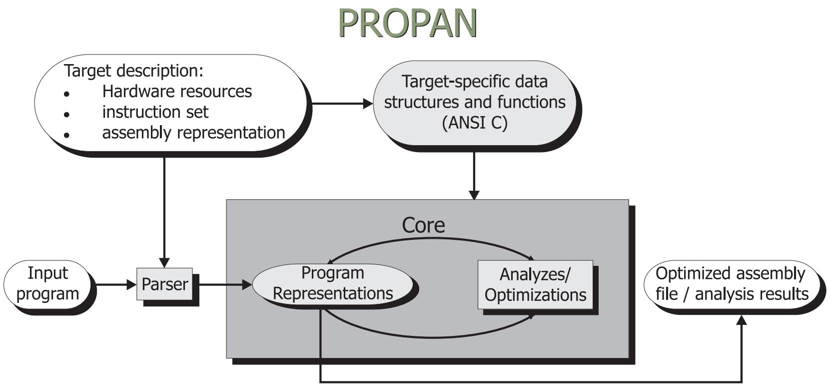

Figure 1: The PROPAN System

the assembly language is not a mere representation of the instruction set of the processor. Assembly expressions, directives as, e. g. , data declarations or segment directives, macros, comments and different forms of operation or instruction delimiters must be known.

Each TDL description is checked for semantical consistency, so input errors and inconsistencies in the hardware description can be detected early.

3.2 The Design of PROPAN

The input of the PROPAN framework (see Fig. 1) consists of a TDL-description of the target machine and of the assembly programs that are to be analysed or optimised. The TDL file has to be processed once for each target architecture and is used in two ways: first a parser for the specified input language is automatically generated. The parser reads the input programs and calculates the control flow graph by jump target and loop detection algorithms. The control flow graph is represented in a generic format, called *CRL* (Control Flow Representation Language) [20]. This representation serves as an interface that allows additional program analyses and optimisations to be easily incorporated into the system. Second, a set of ANSI C files containing fundamental data structures and functions is generated. All relevant information about the hardware resources and the instruction set is represented by the generated data structures; the functions are provided as an interface to initialise, access and manipulate the contained information. The corresponding files can be included in any target application and allow for generic accesses to architecture-specific properties. From the control flow graph, the required program representations as, e. g. , the data dependence and the control dependence graphs are calculated. If register reassignment is to be performed, a register renaming step is executed in order to remove spurious data dependences limiting the available parallelism.

The core of the optimisations is a generic algorithm for integrated instruction scheduling, register reassignment and functional unit allocation based on integer linear programming [18]. For each input program, integer linear programs (ILP) are generated that model the execution of the program for the specified target architecture. Two alternative ILP models are available (see Sec.4), so that the most appropriate modelling can be selected individually for each target architecture. Since the calculation time for the integer linear programs may be high, they can be either solved exactly or approximatively. The basic idea of the ILP-based approximations is the iterative solution of heuristically determined relaxations of the original problem. By the use of such approximations, the calculation time is usually significantly reduced and still high-quality solutions are obtained. More details about the approximation techniques and their use in practise can be found in [18, 15].

In contrast to most previous phase-integrated approaches, the optimisation scope is not restricted to a single basic block. Several basic blocks can be grouped to form a superblock, that can be extended across loop boundaries as long as the control flow paths are not disjunct. The control flow graph of the input program to be optimised is covered by superblocks; the maximal size of the superblocks can be defined by the user in order to influence the computation time. For each superblock, an individual integer linear program is generated and optimised; the optimisation starts with the most frequently executed superblocks, similar to the trace scheduling mechanism [8]. In order to reduce calculation time, the ILP-based optimisations can be restricted to important code sequences that are often executed, e. g. to nested loops. Details about the algorithms that are used can be found in [15].

4 The ILP Models

ILP-formulations can be classified as order-based or time-based models [19]. In time-based formulations, the choice of the decision variables is based on the point of time the modelled event is assigned to. In order-based approaches, the decision variables reflect the ordering of the modelled events; the assignment to points of time is implicitly given.

In the area of architectural synthesis, several ILP-formulations have been developed for the problem of instruction scheduling and resource allocation [9, 32]. In [17, 18] we have investigated the applicability of those approaches to the code generation problem. Based on the results of those studies, we have developed extensions of the order-based ILP model SILP [32] and of the time-based formulation OASIC [9]. The SILP model can be used to perform integrated instruction scheduling, resource allocation and register reassignment. The OASIC formulation is an alternative for processors where the register reassignment can be neglected; here the tasks of instruction scheduling and resource allocation can be integrated. As shown in [17], the integration of the register assignment phase in this model, although possible, is not practicable for complexity reasons [17].

In the following, some basic definitions are presented and then the structure of the ILP formulations is summarised and evaluated.

4.1 Basic Definitions

The task of instruction scheduling is to rearrange a code sequence in order to minimise the required execution time. During reordering, the data dependences of the operations have to be respected. The data dependences are modelled by the data dependence graph $G_D = (V_D, E_D)$ whose nodes correspond to the operations of the input program and whose edges reflect the dependences between the adjacent nodes. There are three different types of data dependences: *true dependences* (read after write, E_D^t), *output dependences* (write after write, E_D^o), *anti dependences* (write after read, E_D^a).

In order to describe the mapping of operations to resource types, the *resource graph* G_R is used that is derived from the machine description. G_R is a bipartite directed graph $G_R = (V_R, E_R)$, where $(j, k) \in E_R$ means that instruction j can be executed by the resources of type k.

4.2 An Order-Based Formulation

The SILP model is an order-based formulation. In this 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 $z_j \in \mathbb{Z}$. Finally $R_k \in \mathbb{Z}$ denotes the number of instances of resource type $k \in V_K$.

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 repre-

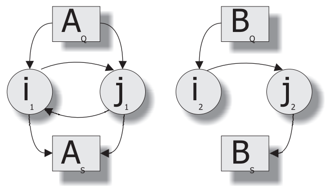

Figure 2: Example of a Resource Flow Graph

sent 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. 2 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 modelling 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 microoperations that are executed in parallel. If, however, the target processor has no VLIW-like architecture, this can simply be modelled by declaring only one (possible virtual) execution resource [16].

The goal of the ILP-formulation is to minimise 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:

$$\min \quad M_{steps} \tag{1}$$

$$t_j \leq M_{steps} \quad \forall\, j \in V_D \tag{2}$$

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.

$$t_j - t_i \geq w_i \quad \forall\, (i, j) \in E_D^t \tag{3}$$

$$t_j - t_i \geq w_i - w_j + 1 \quad \forall\, (i, j) \in E_D^o \tag{4}$$

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

The flow conservation constraints (equation (6)) 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 (7). The resource constraints (8) are necessary, since the number of available instances of all resource types must not be exceeded.

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

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

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

Finally, the serial constraints are required which guarantee a correct sequencing among independent operations assigned to the same resource instance (equation (9)).

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

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 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 $\mathcal{O}(n^2)$. More detailed descriptions including proofs are given in [32, 17].

4.2.1 The Register Assignment Problem

The integer linear program presented above only covers the problem of instruction scheduling and resource allocation. To take into account the problem of register assignment, the formulation has to be extended.

The central data structure is the register flow graph $G_F^R = (V_F^R, E_F^R)$ which is composed of the individual flow graphs for each register file of the target architecture. For each register file, a source and a sink node are introduced; the remaining nodes represent the operations of the input program that perform a write access to a register. Each marked edge $(i,j)_g \in E_F^R$ represents a possible flow of a register of register file $g \in G$ from i to j and is mapped to a binary flow variable $x_{ij}^g \in \{0,1\}$. If $x_{ij}^g = 1$, the same register is used to store the variables generated by operations i and j. Similar to Sec. 4.2, a set of constraints is required to ensure the well-definedness of the flow modelling. Additional constraints are used to model the lifetimes of the variables in the program: a register flow variable x_{ij}^g can only be set to 1, if j is scheduled at a control step where the life range of the variable defined by i has expired.

The number of additional constraints is $\mathcal{O}(m^2)$ where m is the number of operations in the input program performing write accesses to a register. The number of additional binary variables is bounded by $\mathcal{O}(m^2)$. More details including the complexity proofs can be found in [17].

4.3 A Time-Based Formulation

In [9] 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. For each operation i $N(i)$ denotes the set of feasible control steps for the execution of operation i calculated by ASAP/ALAP-analysis [17]. Similarly to Sec.4.2, w_i denotes the execution time of operation i and t_i the starting time for the execution of operation i. However 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 [17, 15] the ILP formulation of [9] 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:

$$\min \quad M_{steps} \tag{10}$$

$$t_j \leq M_{steps} \quad \forall\, j \in V_D \tag{11}$$

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.

$$\sum_k \sum_{\substack{n_j \leq n \\ n_j \in N(j)}} x_{jn_j}^k + \sum_k \sum_{\substack{n_i \geq n-w_i+1 \\ n_i \in N(i)}} x_{in_i}^k \leq 1 \ \forall (i,j) \in E_D^t,$$
$$n \in \{n' + w_i - 1 | n' \in N(i)\} \cap N(j)) \tag{12}$$

$$\sum_k \sum_{n_j \leq n} x_{jn_j}^k + \sum_k \sum_{\substack{n_i \geq n-w_i+w_j \\ n_i \in N(i)}} x_{in_i}^k \leq 1 \ \forall (i,j) \in E_D^o,$$
$$n \in \{n' + w_i - w_j | n' \in N(i)\} \cap N(j) \tag{13}$$

$$\sum_k \sum_{\substack{n_j <= n-w_j+1 \\ n_j \in N(j)}} x_{jn_j}^k + \sum_k \sum_{\substack{n_i > n \\ n_i \in N(i)}} x_{in_i}^k \leq 1 \ \forall (i,j) \in E_D^a,$$
$$n \in \{n' + w_j - 1 | n' \in N(j)\} \cap N(i)) \tag{14}$$

Additionally, the 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.

$$\sum_k \sum_{n\in N(j)} x_{jn}^k = 1 \quad \forall\, j \in V_D \tag{15}$$

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:

$$\sum_{j\in V_D} \sum_{n\in N_{jm}\cap N(j)} x_{jn}^k \leq R_k \qquad \forall m \leq M_{steps} \tag{16}$$

4.4 Modelling Extensions

Most architectures do not have a fully orthogonal instruction set. In the digital signal processor ADSP-2106X[1], there is a restricted parallelism between ALU and multiplier. Some operations can only be executed in parallel if all operands reside in uniquely defined register groups within the (heterogeneous) register file. If the operands are located in other registers, the operations cannot be grouped into one instruction [18]. Such restrictions can explicitly be specified in the constraint section of the TDL description. The logical formulas are built from the boolean operators conjunction, disjunction, and negation. Atomic expressions are available for specifying parallel execution of operations, operation sequencing and resource restrictions for operands [16].

```
(op1 in {AluOps} & op2 in {MulOps}):
 (op1 && op2) -> ((op1.src1 in {GroupA})
  & (op1.src2 in {GroupB})
  & (op2.src1 in {GroupC})
  & (op2.src2 in {GroupD}));
```

The logical constraints are transformed into integer linear constraints and added to the generated integer linear program. Each logical constraint consists of a premise and a rule. The premise denotes a precondition for the generation of the corresponding ILP constraint that can be statically evaluated. In the example, the ILP constraint will be generated for each pair of operations where the first one belongs to the operation group `AluOps` and the second one to `MulOps`. Those groups have been previously declared in the TDL description. The rule part constitutes a logical implication: if both operations are scheduled on the same control step, the operands must reside in the correct register groups. So the generated ILP constraint in the order-based formulation for two operations i and j is equivalent to the following implication:

$t_i = t_j \to \Phi^A_{d_1} = 1 \land \Phi^B_{d_2} = 1 \land \Phi^C_{d_3} = 1 \land \Phi^D_{d_4} = 1$

The expression $\Phi^g_k = \sum_{(i,k)\in E^{R,g}_F} x^g_{ik}$ represents the register flow of register group g entering an operation k. By forcing the appropriate sum to be equal to one for each operation d_k defining an operand of i and j, it is assured that the operands are located in the correct register groups g. The logical implication is translated into an ILP constraint with the help of binary variables [15, 31].

4.5 ILP Models and Hardware Architectures

In our experimental evaluation we could establish a relationship between the architectural design of the target processor and the appropriate ILP formulation style. The flow modelling allows an efficient representation of irregular architectures where the resource competition is high. It also allows an efficient integration of register assignment and instruction scheduling. However if there is a large number of available functional units, the increasing number of alternative resource flows leads to a decrease in solution efficiency. In that case, a time-based formulation is more efficient where the starting time for the operation execution is explicitly bound to control steps. Most ILP formulations for code generation known from literature are time-based formulations. Our results indicate that for irregular architectures, flow-based formulations are more appropriate.

5 Experimental Results

5.1 Modelled Architectures

Up to now, TDL descriptions have been written for the Analog Devices ADSP-2106X [1], for the Philips TriMedia TM1000 [25], the Infineon TriCore μC/DSP [5] and the Infineon C166 processor. The TDL description of the TriCore processor is used within a framework for calculating WCET guarantees for real-time systems [7]. The C166 description has been used to implement dataflow-based postpass optimisations and is part of a a commercial postpass optimisation tool. TDL descriptions for the TI320C6x [29], and for the Intel Embedded Pentium processor are under construction.

5.2 Optimiser Performance

In order to demonstrate the wide range of supported target architectures, we have evaluated the performance of the optimisers generated for the Analog Devices ADSP-2106X and for the Philips TriMedia

TM1000. The ADSP-2106X is a digital signal processor with an irregular architecture and restricted instruction-level parallelism. The TriMedia TM1000 is a VLIW architecture where restrictions on issue slot assignment and result bus synchronisation have to be considered. The integer linear programs have been solved by the CPLEX-library [14]; the calculation times have been measured under SunOS 5.7 on a SPARC Ultra-Enterprise 10000.

5.2.1 The ADSP-2106x SHARC

The optimiser generated for the ADSP-2106X processor performs integrated instruction scheduling and register assignment. The optimiser is generated automatically from the TDL-description and allows an optimal solution of those problems with respect to the given code selection; it uses the SILP-based formulation.

In the following we will show experimental results for a set of input programs that are typical applications of digital signal processing: a finite impulse response filter (*fir.s*), a discrete Fourier transformation (*dft.s*), a cascade filter (*casc.s*), and a wavelet transformation (*wave.s*); *wp3.s* is a routine from the whetstone benchmark. For each program, we compare the result of our optimisations to the optimal code sequence (verified by hand) and to the result of list scheduling algorithms with highest level first heuristics [17]. In order to provide for a fair comparison, an optimal register assignment has been used as an input to the graph-based algorithms (but not to the ILP-based optimisations).

In Table 1, the experimental results are summarised. The integer linear programs are solved either exactly or by the use of ILP-based approximations. For reasons of space, only the results of the fastest ILP-based approximation are shown. Details about the ILP-based approximations can be found in [15, 18]. Some of the input programs contain nested loops, yet they can all be represented by a single superblock. For each input program, the column *ops* gives the number of operations in the input programs, *opt* the optimal number of instructions. The solution method is shown in column *mode*; *I1* denotes the exact ILP-based solution, *I2* the ILP-based approximation and *LS* the result of list scheduling with optimal register assignment given. The result of each optimisation method is shown in column *res* as the number of compacted instructions in the output program. The last column indicates the required CPU-time.

input	ops	opt	mode	res	CPU-time
fir.s	18	8	I1	8	1.07 s
			I2	8	2.74 s
			LS	9	< 1 s
dft.s	26	14	I1	14	7.73 s
			I2	14	9.41 s
			LS	14	< 1 s
casc.s	23	8	I1	8	9.6 s
			I2	8	29.22 s
			LS	12	< 1 s
wp3.s	26	19	I1	–	> 24 h
			I2	19	10 min 29 s
			LS	24	1.9 s
wave.s	39	29	I1	29	1 min 55 s
			I2	29	1 min 43 s
			LS	32	< 1 s

Table 1: Optimisation Results (ADSP-2106X).

For the small input programs, the exact ILP-based solution takes less time than the approximation. This is due to the overhead caused by the approximation setup. However, for larger input programs, the computation time can be drastically reduced compared to the exact solution. In nearly all cases the ILP-based approximations provide an optimal solution. Even with an optimal register assignment given in the input programs, the code produced by the graph-based methods contained on average 19.82% more instructions than the optimal code sequences.

5.2.2 The Philips TriMedia TM1000

The TriMedia TM1000 has 128 general-purpose registers so that the detailed register assignment is less important. However, the instruction scheduling interacts with the slot allocation problem: all operations have to be assigned to issue slots and the assignment of operations to issue slots is restricted. Moreover, the operations have to be synchronised with respect to the write-back bus; no more than five operations may write their result simultaneously on the bus. Again, the corresponding optimiser is generated automatically from the TDL-description and

input	ops	opt	mode	res	CPU-time
ncu.s	33	14	I1	14	7 min 1 s
			I2	15	4 min 49.6 s
			tms	14	–
wp3.s	12	34	I1	34	3.82 s
			I2	34	44.9 s
			tms	34	–
conv.s	42	17	I1	17	5 min 57 s
			I2	17	1 min 25.2 s
			tms	17	–
m1x3.s	31	25	I1	25	4 min 22 s
			I2	26	9 min 17.2 s
			tms	29	–

Table 2: Optimisation Results (TriMedia TM1000).

allows an optimal solution of this problem with respect to the given code selection. Since the integration of register assignment is not needed, both the SILP- and the OASIC-based ILP-formulations can be applied. Our experimental results show that for this processor, the time-based model performs better than the SILP formulation. For reasons of space, only the results of the optimiser using the time-based ILP-formulation are shown. The input programs of the TriMedia TM1000 have been generated with the highly optimising Philips tmcc compiler. The information about the input schedule is not used during ILP solving, so that the scheduling of the input program is undone, and a comparison of the ILP-based methods and the genuine algorithm is possible. The input program *nru.s* is a filtering benchmark, *wp3* is a routine from the whetstone benchmark, *m1x3* is a matrix multiplication and *conv.s* the inner loop of a convolution algorithm.

The results are summarised in Table 2. The column *ops* shows the number of operations (nops are not counted), *opt* gives the optimal number of compacted instructions. The number of instructions in the result of each method is shown in column *res* and the required CPU-time is presented in the last column.

Again we can see that for the smaller programs, the exact ILP-based solution can be obtained faster than the approximative result which is caused by the approximation setup time. With increasing program size the calculation time becomes increasingly smaller compared to the exact solution. All ILP-based results could be calculated within some minutes; again the solution quality of the ILP-based approximation is very high.

6 Conclusion and Outlook

The PROPAN system has been presented as a framework that allows for the generation of machine-dependent postpass optimisations and analyses. PROPAN has been especially designed to perform high-quality optimisations for irregular architectures. For each target architecture, a phase-coupled code optimiser is generated from the TDL-specification that can perform integrated global instruction scheduling, register reassignment and resource allocation by integer linear programming (ILP). In the generated integer linear programs, the restrictions and features of the target architecture are precisely taken into account. We have demonstrated the applicability of our approach by modelling two widely used processors with considerably different hardware characteristics. Furthermore we have shown that 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 optimisation scope is not restricted to basic blocks. The measured calculation times are acceptable for practical use.

There is ongoing work to support complex, superscalar pipelines so that the range of target architectures can be further extended.

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*, 1985.

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

[4] F. Bodin, Z. Chamski, E. Rohou, and A. Seznec. *Functional Specification of SALTO: A Retargetable System for Assembly Language Transformation and Optimization. rev. 1.00 beta.* INRIA, 1997.

[5] E. Farquhar and E. Hadad. *TriCore Architecture Manual.* Siemens AG, 1997.

[6] A. Fauth, J. Van Praet, and M. Freericks. Describing Instruction Set Processors Using nML. In *Proceedings of the EDAC*, pages 503 – 507. IEEE, 1995.

[7] C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, J. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems - The USES Approach. *Proceedings of the ATPS*, 1999.

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

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

[10] R. Govindarajan, E. R. Altman, and G. R. Gao. A Framework for Resource Constrained Rate Optimal Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, (11), 1996.

[11] G. Hadjiyiannis. ISDL: Instruction Set Description Language Version 1.0. Technical report, MIT RLE, 1998.

[12] A. Halambi, P. Grun, V. Ganesh, K. A., N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. *DATE*, 1999.

[13] S. Hanono and S. Devadas. Instruction Scheduling, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *Proceedings of the DAC*. ACM, 1998.

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

[15] D. Kästner. *Retargetable Code Optimization by Integer Linear Programming.* PhD thesis, Saarland University, 2000. To appear.

[16] D. Kästner. TDL: A Hardware and Assembly Description Language. Technical Report TDL1.4, TFB 14, Saarland University, 2000.

[17] D. Kästner and M. Langenbach. Integer Linear Programming vs. Graph-Based Methods in Code Generation. Technical report, Saarland University, 1998.

[18] D. Kästner and M. Langenbach. Code Optimization by Integer Linear Programming. In *Proceedings of the CC*, pages 122 – 136, 1999.

[19] Kästner, D. and Wilhelm, R. Operations research methods in compiler backends. *Mathematical Communications*, 1999.

[20] M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical Report CRL1, TFB 14, Saarland University, Nov. 1998.

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

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

[23] S. Novack and A. Nicolau. Mutation scheduling: A Unified Approach to Compiling for fine-grain Parallelism. In *Languages and Compilers for Parallel Computing*, pages 16–30. Springer LNCS, 1994.

[24] L. Nowak. Graph Based Retargetable Microcode Compilation in the MIMOLA Design System. *20th Annual Workshop on Microprogramming*, pages 126 – 132, 1987.

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

[26] J. Ruttenberg, G. Gao, A. Stoutchinin, and W. Lichtenstein. Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler. *Proceedings of the PLDI*, pages 1 – 11, 1996.

[27] M. Saghir, P. Chow, and C. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *Proceedings of the ASPLOS*, 1996.

[28] A. Sudarsanam. *Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors.* PhD thesis, University of Princeton, 1998.

[29] Texas Instruments. *TMS320C62xx Programmer's Guide*, 1997.

[30] R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley, 1995.

[31] H. Williams. *Model Building in Mathematical Programming.* John Wiley and Sons, 1993.

[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 Integrated Systems for Signal Processing*, 1994.