

Phase-Coupled Mapping of Data Flow Graphs to Irregular Data Paths *

STEVEN BASHFORD

bashford@ls12.cs.uni-dortmund.de

RAINER LEUPERS

leupers@ls12.cs.uni-dortmund.de

Department of Computer Science 12, University of Dortmund, Germany

Editor: ...

Abstract. Many software compilers for embedded processors produce machine code of insufficient quality. Since for most applications software must meet tight code speed and size constraints, embedded software is still largely developed in assembly language. In order to eliminate this bottleneck and to enable the use of high-level language compilers also for embedded software, new code generation and optimization techniques are required. This paper describes a novel code generation technique for embedded processors with irregular data path architectures, such as typically found in fixed-point DSPs. The proposed code generation technique maps data flow graph representation of a program into highly efficient machine code for a target processor modeled by instruction set behavior. High code quality is ensured by tight coupling of different code generation phases. In contrast to earlier works, mainly based on heuristics, our approach is constraint-based. An initial set of constraints on code generation are prescribed by the given processor model. Further constraints arise during code generation based on decisions concerning code selection, register allocation, and scheduling. Whenever possible, decisions are postponed until sufficient information about a good decision has been collected. The constraints are active in the "background" and guarantee local satisfiability at any point of time during code generation. This mechanism permits to simultaneously cope with special-purpose registers and instruction level parallelism. We describe the detailed integration of code generation phases. The implementation is based on the constraint logic programming (CLP) language ECLiPSe. For a standard DSP, we show that the quality of generated code comes close to hand-written assembly code. Since the input processor model can be edited by the user, also retargetability of the code generation technique is achieved within a certain processor class.

Keywords: code generation, embedded processors, phase coupling, constraint logic programming

1. Introduction

Embedded systems are frequently implemented as mixed hardware/software systems. The software parts of a system are executed on embedded processors, while hardware parts are mapped to dedicated non-programmable hardware (ASICs). The assignment of pieces of system functionality to either software or hardware is guided by given time, area, or power consumption metrics.

As compared to ASICs, embedded processors show significant advantages. First of all, processors offer a high degree of flexibility. This allows for accommodating late changes in the system specification by re-programming. Moreover, software implementation facilitates reuse of system components: Software written in a high-level language (HLL) may be reused for other applications. If a processor exists

* This research was funded by the Deutsche Forschungsgemeinschaft DFG under grant Ma 943/6

in the form of a core model (e.g., in VHDL or as a physical layout), also the core itself is reusable. Due to these benefits, a general goal in hardware/software codesign systems is to implement as much of the system functionality as possible in software, whenever flexibility and reuse are of primary concern.

Whether or not a software implementation is possible, i.e., a software implementation meets the speed and size constraints, critically depends on the quality of the machine code. If an HLL like C is used, the compiler is responsible for code quality. Unfortunately, currently available compilers for embedded processors (in particular for DSPs and microcontrollers) generate code of insufficient quality. This has been confirmed by numerous system designers as well as by empirical studies [65, 53]. As a consequence, a large part of embedded software is still written manually at the assembly-language level. Due to the growing complexity of both embedded software and embedded processors, this approach obviously will no longer be viable in the future, but the use of HLL compilers will become compulsory. Therefore, more efficient compilers for embedded processors are highly desirable. If most of a system is implemented in software rather than in hardware, then a good compiler may be more important for the overall costs of a system implementation than a good hardware synthesis tool.

Besides code quality also the flexibility of compilers is an important issue. Since embedded processors in general must be very efficient, even standard, off-the-shelf processors exist in a large variety. In addition, there are more and more application-specific processors (ASIPs), intended to serve only a very narrow range of different applications. In order to avoid the necessity of a huge number of different compilers for embedded processors, the concept of retargetability has been introduced. Retargetable compilers are capable of mapping HLL code to different target machines within a certain processor class. Retargetable compilers facilitate the migration from one target processor to another. Furthermore, retargetable compilers are important in HW/SW codesign, since they permit a more fine-grained exploration of the HW/SW trade-off.

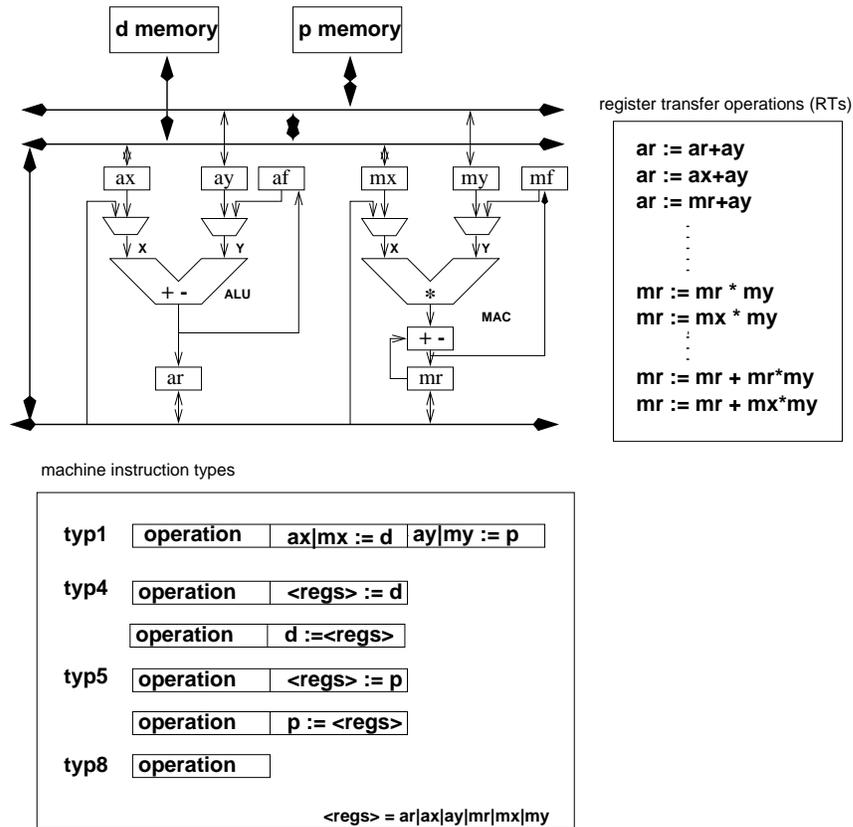
Usually, retargetability and high code quality are contrary goals. Nevertheless, at least a certain degree of retargetability should be a goal in compiler construction, since developing target-specific code optimization techniques for each different ASIP is not economically reasonable.

In the following, we will outline some general difficulties in code generation for a specific class of embedded processors, namely fixed-point DSPs, and we will present a novel solution approach. Since this approach is based on a target processor modelling formalism, retargetability is achieved for a large range of processors.

1.1. Constraints in irregular data paths

One main challenge in high quality code generation for DSPs is caused by the irregular data path architecture of these processors, i.e., there are distributed and specialized *register files* (RFs) and *functional units* (FUs). This causes strong mutual dependencies between RFs and FUs, due to restricted interconnections between these resources. In turn, this leads to strong dependencies between different code

Figure 1. ADSP-210x data path, RTs, and instruction types



generation phases. Furthermore, a certain degree of *instruction level parallelism* (ILP) is usually available which should be exploited in order to generate high-quality code. On the other hand, ILP in DSPs is often restricted to parallelization of only certain instructions due to strongly encoded machine instruction formats. Currently, there is a lack of techniques and tools capable of simultaneously handling restricted ILP and irregular data paths, because it is difficult to obey the resulting large amount of constraints during code generation.

Throughout the paper, we will generally talk of *storage resources* (SRs), which comprise RFs and memories. We only talk of RFs, if it is important to distinguish between register files and memories.

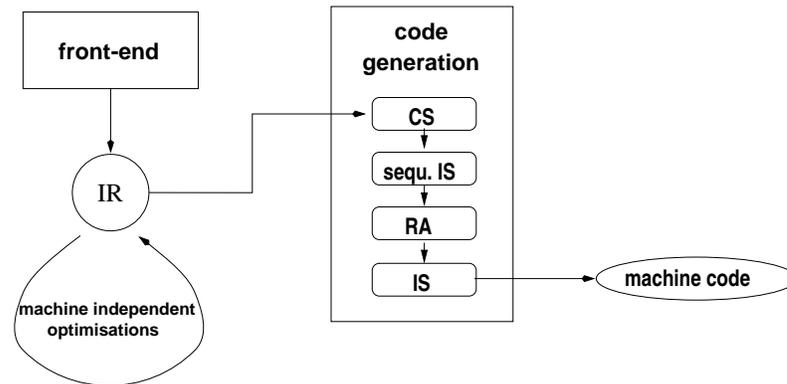
EXAMPLE: In order to exemplify typical constraints, we consider the ADSP-210x fixed-point DSP [16]. The primitive entities of instruction behavior in our model are *register transfers* (RTs). A RT reflects the operation (+, -, *, ...) performed, the SR where the result of the operation is stored, and the SRs where the operands of

the operation have to reside. In fig. 1 a partial data path of the ADSP-210x and the corresponding RTs are shown. Furthermore, a classification of *instruction types* is given¹. Each type accounts for a certain permissible combination of RTs (operations and moves) within one machine instruction. The data path consists of two FUs (ALU and MAC), and the SRs consist of the set $\{ar, ax, ax, af, mr, mx, my, mf\}$ of RFs, and the two memories d and p . The operands of RTs are required to be in special SRs. For an addition, for instance, the left operand is required to be in one of the SRs ax, ar, mr and the right operand has to be in ay or af . The result is stored either in the accumulator ar or the feedback register af .

On the ADSP-210x, at most one FU operation and two move operations from memory to registers can be executed in parallel (this corresponds to the machine instruction type "typ1" in fig. 1). This requires, that one of the move values be located in p and the other one in d . The destination of the move from d must be ax or mx and the destination of the move from p must be ay or my . The destination of the result of the operation, executed on one of the FUs ALU and MAC, is then restricted to the accumulators ar or mr , respectively. \square

1.2. Phase coupling problems

Figure 2. Code generation phases

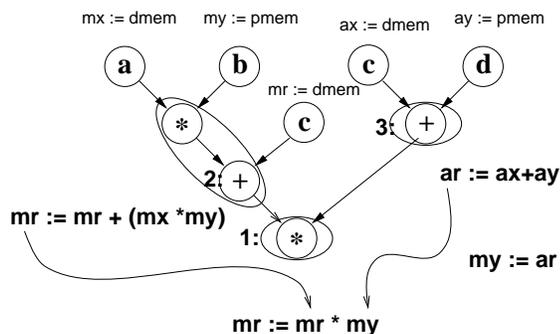


In fig. 2 a very coarse grain structure of a compiler is illustrated. The *front-end* checks the source program for syntactical and semantical correctness and generates an *intermediate representation* (IR) of the program. Code generation (frequently called the *back-end*) is concerned with mapping the IR to a sequence of *machine instructions* (MIs) of the target processor, typically in assembly or binary code. Each machine instruction comprises a set of one or more RTs that are executed in parallel.

Traditional code generators execute a set of phases strictly sequentially. Fig. 2 shows a possible ordering of the basic code generation phases:

- *Code selection* (CS) is typically performed as the first phase. It is concerned with mapping the operations in the IR to RTs of the target machine.
- *Register allocation* (RA) decides which values should reside in registers and which should reside in memory at each point of time during program execution.
- *Instruction scheduling* (IS) in one sense it is concerned with finding a sequential ordering for the selected RTs. Goals are to reduce register usage or to prevent pipeline stalls in RISC like architectures. In the context of ILP, IS is also used to denote the parallelization of RTs, so as to increase code speed (frequently called *code compaction*). Unless mentioned explicitly, we will use the latter meaning of instruction scheduling in the remainder of this paper.

Figure 3. A DFT cover



The disadvantage of a sequential phase ordering is that the decisions made in one phase can severely constrain the following phases. As an example, consider the following dependence between CS and IS: Most current code selectors are based on *tree covering* and operate on data flow tree (DFT) based IRs of basic blocks. RTs are represented by *tree patterns*, where each pattern is assigned a *cost value*. Code selection is performed by covering DFTs with tree patterns, while minimizing the total costs. An example of a covered DFT is shown in fig. 3. Tree coverers are able to select *locally optimal covers*, but optimality refers to a sequential execution model. The underlying computation model for determining optimal covers does not permit modeling of ILP. The selection of a cover implies a certain selection of SRs for the operands and results of the selected RTs. ILP can be prevented by unfavorable CS decisions. In fig. 4 an optimal cover of the example DFT by ADSP-210x RTs (or tree patterns) is shown. However, the selected RTs can hardly be parallelized, because they conflict with the constraints on ILP for the ADSP-210x. As will be shown later, a shorter schedule can be achieved, if a different DFT cover were selected during CS.

Figure 4. Optimal DFT cover when neglecting ILP

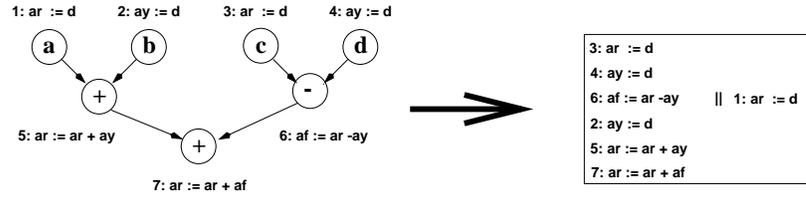


Figure 5. Representation of multiple RTs by a FRT

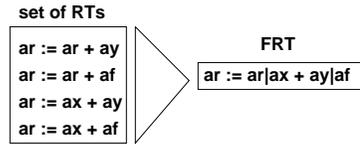
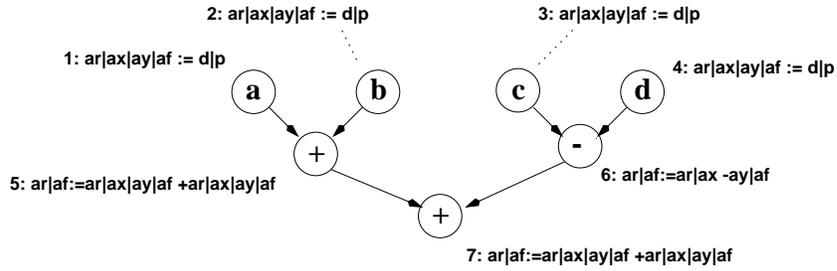


Figure 6. Representation of covers by FRTs



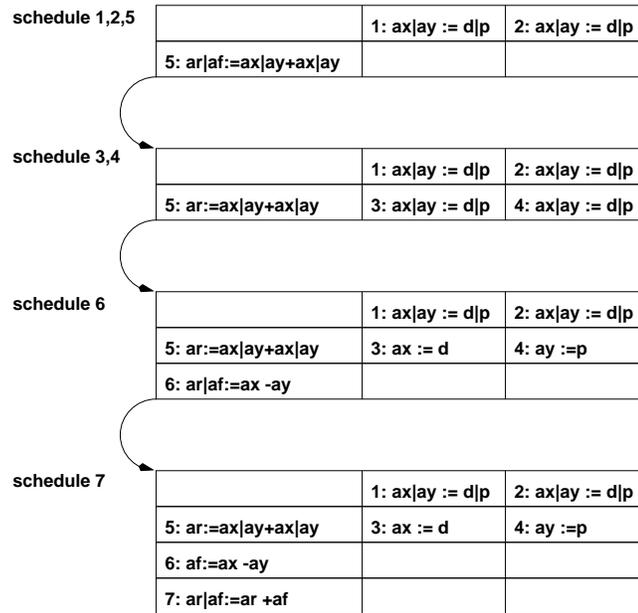
1.3. Our phase coupling approach

In order to improve code quality for irregular architectures, a tight integration (or *phase coupling*) of the code generation phases is required. For good exploitation of potential ILP, many decisions have to be postponed until IS. Thus, code selection and register allocation should be integrated into instruction scheduling. In order to enable the integration of code selection we use a covering technique that computes the set of all possible covers for a given data flow graph (DFG) representation of basic blocks efficiently in a single pass. We do not reduce the set of covers to a locally optimal solution, but retain a set of alternative covers in the remaining code generation process.

To represent the covers, all RTs matching a node in the DFG are combined to a *factorised RT* (FRT) (fig. 5). FRTs concisely represent a set of "similar" RTs. In

FRTs, the locations for the result and the operands are given as a *set of alternative SRs* (separated by a ”|” character). The set of all alternative covers are represented by an *FRT cover* of a DFG, which we denote by DFG^{FRT} . Fig. 6 shows the DFG^{FRT} for the DFT from fig. 4. There are different options for constructing the DFG^{FRT} : We have implemented techniques to generate *optimal* covers with respect to a sequential execution model. We may also use faster techniques, that partially reduce the set of initial covers for code generation: An example is to enforce the use of chained operations (e.g. the MAC operation) whenever possible.

Figure 7. Integrated code generation based on FRTs



Based on the DFG^{FRT} representation, we have developed a fully integrated technique for code selection, register allocation and instruction scheduling. The basic concepts are illustrated in fig. 7, where the scheduling of FRTs of the DFG^{FRT} from fig. 6 for the ADSP-210x (fig. 1) is shown.

EXAMPLE: After scheduling the data transfers for variables a and b in the first machine instruction, the corresponding FRTs have been *reduced*. Reducing a FRT means to delete members of the set of RTs represented by the FRT, so as to meet the constraints on ILP according to the restrictions for the resources feasible for the instruction types. On the ADSP-210x, *typ1* is the only one allowing parallel execution of two data transfers from memory into registers (cmp. fig. 6). Only RTs with sources d or p and destinations ax or ay are feasible for the $+$ operation. Therefore, the SRs af and ar are eliminated from the destinations of the

FRTs. The remaining flexibility is due to the commutativity of the addition (see schedule 1,2,5). The decision, which of the variables a and b will be loaded to ax or ay , is postponed. Generally, this may generate higher flexibility for subsequent decisions. The scheduling for variables c and d again leaves some freedom (schedule 3,4) for their locations but *reduces the possible locations* of the addition result to ar in order to meet the ILP conditions. The locations of variables c and d are determined immediately after scheduling the subtraction (schedule 6). Schedule 7 finally determines the result location of the subtraction (node 6). \square

For the implementation of our code generator we have used the constraint logic programming (CLP) language ECLiPSe [52].

1.4. Overview of the paper

The following section is concerned with related phase coupling approaches. Section 3 gives an overview of our constraint-based programming framework and of our code generation system ICG. In section 4 the internal machine model of the code generator and the retargeting mechanism are described. Section 5 describes the concepts of tree covering and our approach of FRT covering for generating the sets of DFG covers. The integrated register allocation technique is described in section 6 followed by its integration into instruction scheduling in section 7. This section also comprises a complexity analysis of our approach. Section 8 describes post-processing steps required to generate executable code as well as experimental results. The last section gives conclusions and mentions future work.

2. Related work

Traditional standard techniques for homogeneous architectures are described in the "Dragon Book" [6]. Advanced techniques for general purpose processors are presented in [45]. An early contribution emphasizing phase ordering problems in the context of microcode generation is [60].

In order to compare existing phase coupling approaches, we give a classification of phase coupling approaches according to the following cases:

- Only regular architectures are considered; we further distinguish whether or not ILP is considered. In case of ILP, orthogonal VLIW-like parallelism is assumed, and mutual dependencies between SRs and FUs are neglected.
- Irregular architectures are of primary concern. We will differentiate the following classes:
 - Approaches neglecting ILP.
 - Approaches considering ILP based on heuristic techniques.
 - Approaches considering ILP based on exact techniques.

As distributed resources are of concern, we will further classify these approaches by the sets of resources which are simultaneously captured. This will reflect the coupling degree of CS, RA and IS.

As tree covering is very important in the context of phase coupling problems for DSPs, we first give an overview of existing approaches. We then describe phase coupling approaches according to the given classification.

2.1. Tree covering techniques

Much effort has been put into the design of efficient tree pattern matchers and efficient code selector generators. Initiated by the work of Graham–Glanville [27] LR-parsing techniques were used for pattern matching, where the target machine specification was defined by a context free grammar. A parser generator was used for generating the code selector². The basic problem of these approaches is the potential ambiguity of the required instruction-set grammars. Tree pattern matching with dynamic programming [3, 2] constituted a solution to this problem. Code selector generators map a specification of the target instruction set to a dedicated code selector. The specification techniques used are based on weighted regular tree grammars [23]. Tree grammars are represented by a set of tree reduction rules³ of the form $X \leftarrow \textit{pattern}[\textit{cost}] : \textit{action}$. The action part defines the output of the code selector. The generated code selectors make two passes over expression trees. The first pass is bottom-up and finds a covering with minimum costs. The second pass is top-down and emits the code. Examples for code generator generators based on this model are: *BEG* [18], *Twig* [2], *burg* [21], *iburg* [20]. *BEG*'s and *iburg*'s tree pattern matchers are hard coded and mirror the tree patterns like recursive descent parsers mirror their input grammars. *Twig* matchers use a table-driven variant of string matching. Other table driven approaches, based on finite tree automata [23], move dynamic programming to compile-compile time [55, 32, 30, 31, 23].

The tree pattern matching technique is capable of handling distributed SR and FUs. Tree pattern matchers are capable of computing the set of all covers, which is required for delayed resource binding. There are several DSP related approaches based on the application of code selector generators [19, 4, 5, 37].

CBC [19] is based on a machine description in the hardware description language *nML*. The machine specification is transformed to an *iburg* specification and the code selector is generated by an extended version of *iburg*. *CBC* has been developed for irregular architectures. Instruction level parallelism is supported by delayed binding of FUs. *CBC* tries to take into account common subexpressions that cross basic block boundaries by means of *heuristic node duplication*. A control data flow graph is modified in order to create more complex RTs across basic block boundaries by node duplication. In [5] edges of DFGs are pruned in order to provide faster code selection for the DFTs of a basic block for architectures satisfying the so-called "RTG criterion". Code selectors are generated by the code selector generator *OLIVE* [8]. A technique for constructing spill free schedules for machines satisfying the "[1, ∞] model" and the RTG criterion, is given in [4].

2.2. Phase coupling for regular architectures

Early approaches of coupling code selection, sequential instruction scheduling and register allocation are given in [3] yielding optimal solutions for trees and architectures with a homogeneous register file and without ILP.

The following approaches are concerned with coupling register allocation with scheduling. They consider ILP but neglect irregularities in the target architecture. Register allocation tries to reuse as many registers as possible, therefore adding many additional false dependencies that constrain the instruction scheduler's ability to reorder machine instructions. Instruction scheduling tries to parallelise as many RTs as possible. This results in high register pressure which drastically increases the amount of interferences.

Some approaches are based on the exchange of information between phases. Goodman and Hsu [26] manipulate the scheduler's data dependence graph, such that its width is not larger than the number of registers available. In a second method (*integrated pre-pass scheduling* (IPS)) a local scheduler is restricted to use a fixed number of registers for local values. If *register limits* are reached, the scheduler tries to free some of the registers or to increase the register limit. The subsequent local register allocation can generate spill code which enforces rescheduling. Bradlee compares two strategies with graph coloring followed by scheduling [9, 12, 11]. The first strategy is an improvement of IPS. The second one called RASE first performs initial passes of the instruction scheduler for estimating local schedule costs, first for a very limited number of registers and then with the maximum number of available registers. The computed estimations are used in the priority scheme of graph coloring, followed by a local list scheduler. Freudenberger [22] describes a method that integrates greedy register allocation into trace scheduling in order to provide global optimisation. In Bersons approach [10] the data dependence graph is incrementally sequentialised with respect to global aspects of over-utilised and under-utilised regions of resource requirements. Register allocation is performed on-the-fly, together with appropriate spilling. Approaches like [41, 51, 46] start with an initial register allocation. During instruction scheduling false dependencies are eliminated using dynamic renaming [15].

There are several works based on graph coloring register allocation while considering aspects of parallelism: Norris and Pollok [48] add edges to the interference graph to estimate the re-ordering effect of instruction scheduling. Pinter [54] also constructs an interference by adding additional edges. She first constructs a graph from the data dependence graph, where the transitive closure of all dependence edges are placed into a graph as undirected edges. Target machine resource conflicts are added that restrict the parallel execution of machine operations. From this resulting graph, the graph's complement is constructed and the union with the register allocator's interference graph is constructed (called the *parallel interference graph*). Brasier [13] proposes a method based on late register allocation and limits the additional interferences to false dependencies that will limit the instruction scheduler. Only if spilling becomes necessary during late register allocation it is switched back to early register allocation. The "early" interference graph is

augmented with edges that are exclusively found in the "late" interference graph and which are colored with the same color in the early interference graph. Further works based on graph coloring are [1, 49, 50].

Mutation scheduling [47] takes into account distributed FUs. It integrates code selection and register allocation into instruction scheduling. Each value in the program is associated with a set of alternative (functionally equivalent) expressions called "mutations", each using a different set of resources of the target architecture. During scheduling one of the alternatives is selected. If the resources are occupied, another mutation is selected. The mutation sets can change dynamically during scheduling. When a value is moved to a register, a reference to that register is added to the mutation set. If a value is spilled, a *load* entry with the corresponding location is added. Initial register allocation is performed like in [41, 51, 46]. But in contrast to these approaches spilling is also integrated. Also re-computation of a value is considered as a mutation [47]. In contrast to rematerialization every equivalent expression can be selected. Issues of irregular register sets and ILP conditions are not considered.

2.3. Phase coupling for irregular architectures

2.3.1. ILP not considered In [36] an approach for combined code selection and register allocation for DFGs is represented. The approach is based on binate covering and addresses accumulator-based architectures. It yields optimal results in acceptable time for small basic blocks. The approach of Paulin [56] performs tree pattern matching and dynamic programming, but delays the binding of locations for result and operands (due to register class specifications associated with the patterns). Interdependencies between the locations are not considered. Register allocation is performed by using the left edge algorithm. If a free location for a value is found, this location is bound. Register allocation does not take ILP into account. Instead, there is a postpass compaction phase.

2.3.2. ILP and heuristic approaches Data routing incorporates register allocation for distributed register files and instruction scheduling. The aim of data routing is the selection of good routing paths for values. The BULLDOG Compiler of Ellis [17] performs local scheduling together with greedy register allocation on-the-fly. Data routing is performed in order to support ILP. Good spill decisions are not considered. An approach combining delayed binding of FUs with consideration of different data routes is proposed in [29]. Distributed register sets together with fine-grain parallelism are taken into account. Storage resources are composed to more abstract storage resources, so as to enable the delayed binding of SRs. The code selector performs traditional tree pattern matching with dynamic programming. A trace scheduler is generated from a machine specification to guide the order of the choices the trace scheduler has to make with respect to the requirements of the target machine. The trace scheduler performs register allocation on-the-fly.

Rimey and Hilfinger [57] perform local scheduling together with greedy binding of FUs and register allocation on-the-fly. Pattern matching of RTs is performed during scheduling. For each RT they try to find data routes from the definitions of uses of the RT, such that the RT can be scheduled in the current instruction cycle. The data routes must be compatible with the current schedule. Hartmann's approach [28] is a refinement of Rimey's and Hilfinger's work, extended by deadlock detection. There, FUs are bound in advance. The approach proposed in [35] determines data routes, taking ILP into account. It examines various data routes while also considering global spilling and re-computation. Selection of data routes for each value is guarded by a cost model based on distribution graphs. The FUs for each operation have to be bound in advance.

The retargetable code generator *MSSQ* [40] embedded in the *MIMOLA* software system simultaneously performs binding of FUs and local instruction scheduling, thereby considering a set of machine instruction types. Variables are pre-allocated to certain SRs, defined by the user. Temporary values are allocated to register cells on-the-fly during pattern matching. Algebraic transformation rules are considered, while spill code generation is not.

2.3.3. ILP and exact approaches The following approaches are based on strategies yielding optimal solutions. Code generation phases are described in the form of constraints (generally linear equations and inequations). The complete solution space is explored while all constraints are considered simultaneously, leading to a complete phase integration. The approach in [39] performs instruction scheduling based on an Integer Programming (IP) approach yielding optimal solutions. Code selection and register allocation are performed in advance, based on iburg-generated code selectors. This results in the binding of SRs. Binding of FUs is delayed to scheduling. Complete integration of code selection, register allocation and instruction scheduling based on IP are given in [62, 24]. Wilson's approach [62] leads to very high runtimes due to large IP models. The approach of Gebotys [24] takes advantage of describing constraints based on horn clauses. This can be mapped to Linear Programming problems, which can be solved more efficiently than IP problems. However, only restricted classes of architectures can be handled efficiently by these approaches. Architectures comprising register files with multiple registers, like the ADSP210x, typically lead to an explosion of the generated models. In [34] a covering approach for DFGs for finding a minimal set of (VLIW) instructions (taking ILP into account) is specified. The approach is based on binate covering. Detailed register allocation has to be performed in a post processing phase.

2.4. Non-phase-coupling approaches for irregular architectures

An exact scheduling method for architectures with highly constrained ILP, based on bipartite graph matching, is described in [59]. The search space is pruned in advance – but without eliminating any feasible schedules – such that for many benchmarks results are found in acceptable time. All resources are fixed in advance,

so that the quality of generated code depends on earlier phases. In [44] an approach for loop pipelining based on constraint analysis is described. Resources are bound in advance. Resource constraints are mapped to dependency relations and if a feasibility check holds, schedules can be generated by any conventional scheduler. Infeasibility leads to a re-binding of registers (by means of register assignment) if possible, and to the generation of new sequencing constraints. The approach iterates until a feasible schedule is found or no re-binding of registers is possible. An approach for DFG covering based on BDDs is given in [64]. Covering is extended to yield a certain binding of operations to FUs. A single covering, where all resources are fixed, is selected. No underlying cost criterion for the selection is given. In a second phase a list scheduling algorithm, extended by spill code insertion, generates the final schedule. Due to fixed resource binding, this phase heavily depends on the covering phase.

2.5. Main differences of our approach

We address irregular architectures with ILP, as well as register files with capacity larger than one. Our code selection approach generates all DFG covers with respect to a given set of RTs (actually: FRTs). These covers also contain associated resource information and all mutual dependencies between resources in the form of constraints. The basic difference between our approach and the work mentioned above is a new concept for delayed SR and FU binding. In contrast to the existing approaches we do not fix resources unless we are forced to. We keep alternatives solutions as long as possible and reactivate decision procedures only when more information has become available. We have also integrated a flexible spilling strategy. In this approach spilling is not restricted to loads and stores from/to memory. In order to avoid exhaustive runtimes, we do not consider the *complete* solution space like IP based approaches. However, as will be shown in the following, there is a tight integration of code generation phases, which allows for generation of high quality code.

3. The constraint-based code generator ICG

This section begins with an overview of constraint programming and an outline of our constraint-based implementation. Then, an overview of the components of our code generator (the ICG system), their functionality and the data flow between them is given. The section ends with some remarks on the benchmark set we used for evaluating the system.

3.1. Constraint programming

We model phase coupling of code generation in the form of *constraint satisfaction problems* (CSPs) [33]. Therefore, a short and informal explanation is given here. CSPs are represented by a set of variables and a set of constraints, describing

dependencies between the variables. Variables are associated with certain domains, i.e., sets of values. A CSP solution is a mapping of each variable to a certain member of its domain, which meets all constraints. The goal is to find one valid solution or an optimal solution according to an objective function. An algorithm for determining the satisfaction for a set of constraints is called a *constraint solver*. For certain application domains there exist dedicated solvers. A general technique used in many solvers is to rewrite constraints into so called "solved form", for which it is clear, that a solution exists (e.g. by Gauss-Jordan elimination for linear arithmetic constraints).

ICG is written in the constraint logic programming (CLP [43]) language ECLiPSe [52, 63]. ECLiPSe is based on PROLOG and comes along with a set of domains together with dedicated constraints, solvers, and search- and optimization strategies. A basic technique used in CLP systems is that of constraint propagation [61]. Constraints, occurring in a constraint logic program, are collected in constraint store when executing the program. Adding new constraints generally leads to the reduction of certain domains. Collected constraints operate in the background, and they locally check the feasibility of the domains and also eliminate members of the domains, which will lead to invalid solutions. As an example we consider the variables X and Y , both associated with the domain $\{1, \dots, 10\}$. If we now impose the constraint $X < Y$, the domain of X is reduced to $\{1, \dots, 9\}$, because there is no legal assignment of Y meeting the constraint, if X is set to 10. As there is no feasible solution if $Y = 1$, the domain of Y is reduced to $\{2, \dots, 10\}$. A setting of X to 5 leads to a reduction of Y 's domain to $\{6, \dots, 10\}$. In the following we will call variables like X and Y *domain variables*. If the domain of a variable is reduced, we will call this a reduction of the variable. Solving is performed in a second phase and consists of *labeling* the variables with certain members of their domains. Again the constraints guide the feasible labeling of variables. An assignment which does not meet the given set of constraints is rejected and leads to backtracking.

ECLiPSe is designed for solving difficult combinatorial problems in the area of planing, scheduling and resource allocation. The ECLiPSe system offers several predefined search strategies, w.r.t. the ordering of selecting variables and members of the domains, and provides direct control over search strategies for the user. In addition to constraint propagation, techniques based on Mixed Integer Programming (MIP), and stochastic techniques (like Simulated Annealing) are supported. ECLiPSe enables the user to mix these techniques in order to build hybrid search strategies. It provides powerful mechanisms to define new domains and constraints, and new search strategies and solvers directly in ECLiPSe. In our implementation we made effective use of the finite domain (FD) library (the alternative resource sets are implemented as finite domains). The library provides predicates for setting up the domains together with predefined constraints over finite domains. Also access to the domains of variables is given together with predicates for creating new domains from existing ones (via well known set operations) and for updating domains of variables. These features were extensively used in the specification of new constraints, needed for our system. New, complex constraints were defined

- based on a set of existing elementary constraints, provided by the library,

- using low level predicates for direct domain access and modification, and
- making usage of a generalized propagation approach (cf. library manuals of ECLiPSe [52]).

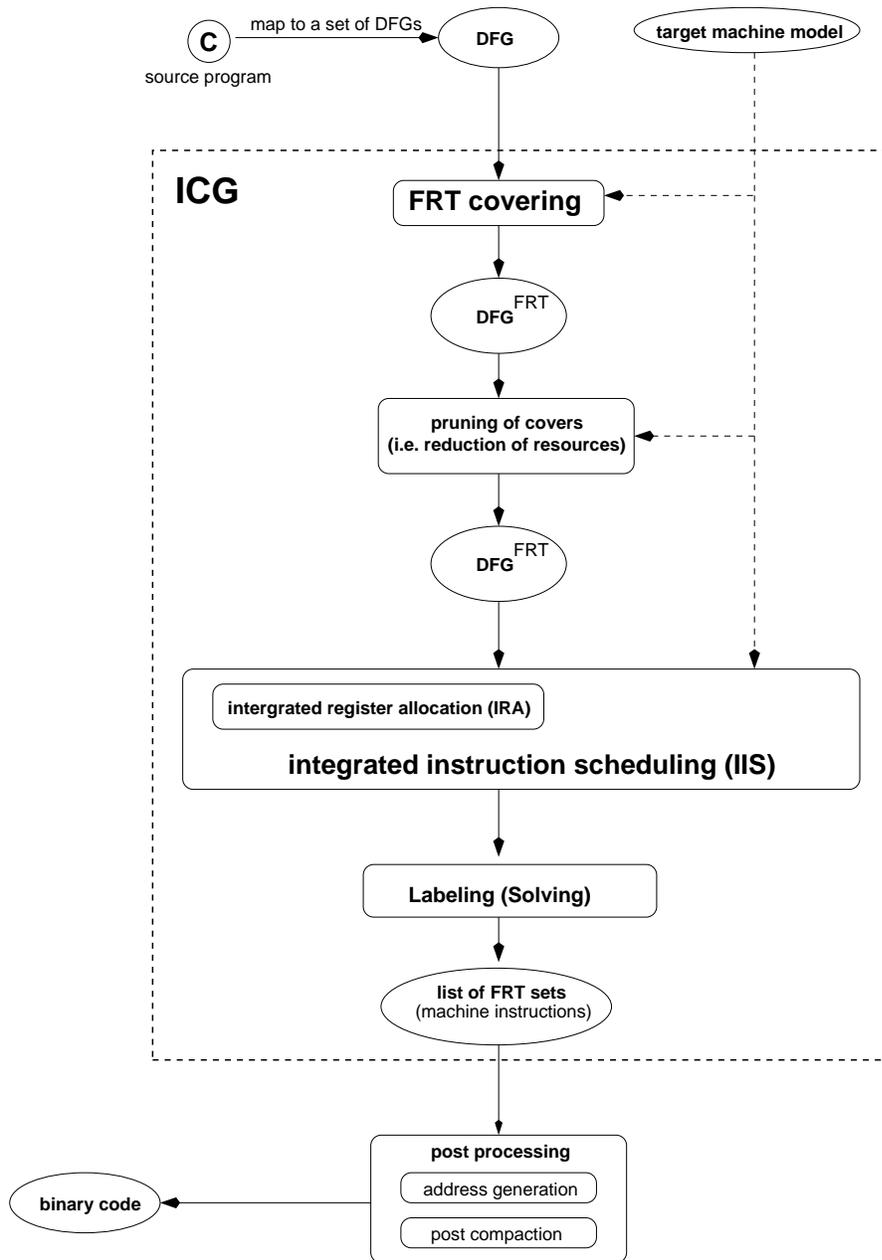
ECLiPSe allows specifying conditions and priorities for reactivating constraints, in order to prevent unnecessary reactivations.

3.2. ICG System Overview

An overview of the components of our code generation system ICG and the communication between these components is shown in fig. 8. C source programs are mapped to control data flow graphs (CDFGs). For these, basic blocks of a program are represented by data flow graphs (DFGs). In the current version, only data flow oriented parts of a program are supported. Therefore we will focus on DFGs as the input of ICG in this paper. The second input is a model of the target machine for which code is to be generated. The output of ICG is a sequence of machine instructions in the form of a set of parallel FRTs. The system comprises the following basic components:

1. *FRT covering*: A DFG is mapped to the initial DFG^{FRT}, representing the set of all covers of the input DFG. This comprises setting up the domains of the variables, representing the alternative resources of variables, and generating the constraints specifying mutual dependencies between resources. These constraints guide the concise reduction of variables throughout the remaining code generation phases, leading to only legal covers (section 5).
2. *Pruning of covers*: A set of strategies is provided, which allow to prune the set of all covers in advance. This includes reduction to the set of optimal covers for DFTs and also for DFGs. It also allows to reduce certain domains in advance, e.g., it is possible to specify the locations allowed for common subexpressions (CSEs). These strategies include labeling and optimization strategies. The result is a smaller set of alternative covers, in the form of reduced domains.
3. *Integrated register allocation (IRA)*: Integrates code selection and register allocation, which comprises techniques for data routing and spilling (section 6). *Integrated instruction scheduling (IIS)* extends the concepts developed in IRA to instruction scheduling with regards to ILP (section 7). IIS maps a DFG^{FRT} to sequence of compacted machine instructions represented as sets of FRTs. Both, IRA and IIS set up new domain variables and constraints. For data routing and spilling, data moves have to be inserted. Constraints, which guarantee that the data moves are feasible in the context of the current covers, and provide guidance such that no register file resources are exceeded, are inserted. For ILP, constraints are added that guide the concise reduction of variables in order to meet the ILP conditions.
4. *Labeling*: Generates a certain mapping of variables to resources. This also serves as a feasibility check. Since constraints only check local feasibility, it may still

Figure 8. System overview



happen, that there exists no solution which satisfies all constraints. In case there exists no solution, we allow certain constraints to be violated and we have to insert correction code⁴.

Additionally, there are two post-processing phases: address generation and post-compaction. In these phases, in which executable code is generated, we apply techniques developed earlier in the RECORD compiler project [37].

3.3. Remarks on benchmarks and run-times

To evaluate our system we have used benchmarks from the *DSPStone benchmark set* [65] which evolves to a standard benchmark set in the DSP compiler community. Since ICG is currently only able to generate code for the data flow components of programs (i.e. DFGs of basic blocks), we have selected programs which are basically data flow oriented (complex multiply, complex update, and iir filter, biquad and biquad_n and a lattice filter). The basic blocks of the selected benchmarks represent realistic and representative examples with regards to other basic blocks in the DSPStone benchmark set. Additionally, a set of internal benchmarks was tested, with very large basic blocks (DFGs with upto 100 nodes and 200 edges).

We have compared the generated sequences of machine instructions of the DSPStone benchmarks with hand written code and the results of the dedicated GNU C compiler for the ADSP-210x. As a metric we have used code size. For basic blocks, this also indicates code performance⁵.

The primary goal of our approach is to generate high quality code. Compilation speed is of secondary concern as long it is in an acceptable range. This is feasible, since compilation times are of minor concern for embedded software, and even run-times of hours are acceptable in certain cases.

4. Internal machine model

This section describes the internal model of the target machine used for the code generator. This model is based on FRTs. Understanding of our internal machine model is quite essential for the understanding of the code generation techniques described in the following section. The internal machine model comprises the description of the instruction set in the form of the available set of RTs, constraints on ILP, and information about address generation units and specific control functions. In this contribution we concentrate on the set of RTs and the modeling of ILP. Our code generator can be re-targeted by exchanging the internal machine model, since none of the techniques is dedicated to a certain processor model.

There are several approaches for modeling instruction sets with strongly constrained ILP and with provisions for retargeting all major phases of code generation [40, 58, 59, 37, 34, 64]. Our approach is based on constraints and allows a very concise and compact specification methodology. We introduce a new concept for modeling highly constrained ILP based on instruction types⁶ which allows a very intuitive specification method.

We use the following notations: The set \mathcal{SR} denotes the available SRs. \mathcal{FU} is the set of FUs, and the set \mathcal{T} denotes a set of machine instruction types, used for modeling ILP.

4.1. Register transfers

The model for RTs are *RT patterns* together with *extended resource information* (ERI). A RT pattern reflects the operation and the SRs where the result is stored and the SRs where the operands are expected. For example, if $\mathcal{SR} = \{a, b, c\}$, then the RT $c := a + b$ denotes an addition where the first and second operand must reside in SRs a and b , respectively. The result is stored in SR c . We call c the *definition* of the RT and a, b the *uses*. We define two functions def and use such that for a RT rt given as $D := op(U_1, \dots, U_i, \dots, U_n)$, $def(rt)$ denotes D , and $use(rt, i)$ denotes U_i ⁷ (also written as use_i). *Transfer operations* (TOs) are specific RTs of the form $c := a$. They denote the transfer of a value from SR a to SR c . In section 1 (fig. 1) some of the RTs of the ADSP-210x have been shown.

The triple $ERI = (t, fu, c)$ of extended resource information denotes the type $t \in \mathcal{T}$, the consumed FU $fu \in \mathcal{FU}$, and the cost c of a RT. Costs are given by means of instruction cycles necessary for executing the RT. The ERI is very important for meeting the scheduling constraints of RTs in the resulting code. Types and FUs are used to model potential parallelism between RTs. Two RTs can be only executed in parallel, if they can be encoded in the same instruction type $t \in \mathcal{T}$ and if they do not consume the same $fu \in \mathcal{FU}$.

4.2. Factorised RTs

The usage of *factorised RTs* (FRTs) in our code generator is twofold. As explained in the introduction, FRTs are used to represent the set of covers for DFGs (cf. section 1.2). We will also use FRTs as a concise specification and internal representation for the set of RTs of the target machine, from which the matching FRTs are derived during covering. We consider a subset of RTs implementing the addition in the ADSP-210x: $\{ar := ar + ay, ar := mr + ay, ar := ax + ay\}$. There are three alternatives for the location of the first operand, i.e. ar, mr, ax . We combine these RTs to a single representation by means of *factorising*, in this example use_1 . The result is the FRT $ar := ar|mr|ax + ay$, where use_1 represents a set of alternative SRs.

We model FRTs by an *RT-pattern* and a set of *constraints*. In the RT pattern the definition and each use are representing *domain variables*. The domain variables are associated with a certain set of SRs. The set of constraints define the domains of the variables and certain dependencies between them. For instance, the FRT $ar := ar|mr|ax + ay$ is given by $(D := U_1 + U_2, C)$, where C is the set of constraints $\{D = ar, U_1 \in \{ar, mr, ax\}, U_2 = ay\}$. In the following we will call a domain variable of a RT-pattern a *location*, and $dom(L)$ denotes the domain of the domain variable L (e.g. $dom(U_1) = \{ar, mr, ax\}$). A *substitution* for the locations of a FRT is an assignment of a certain SR to each location.

The feasible RTs represented by a given FRT can now be obtained by the set of substitutions of a FRT. We say that a FRT is *orthogonal* if any substitution of its locations yields a feasible RT. A FRT is called *constrained*, if there exist dependencies between the locations. Consider the set of RTs $\{c := a + b, a := c + b\}$ and the FRT $(D := U_1 + U_2, D \in \{c, a\}, U_1 \in \{c, a\}, U_2 = b)$. We need a constraint in order to describe the dependencies between D and U_1 . This could be formulated as $D = c \equiv U_1 = a$ ($D \neq U_1$ would also be sufficient). If we now set D to c , this leads to the reduction of U_1 to a , in order to meet the constraint. By adding new constraints to a FRT, the domains of variables may be reduced. In turn, this may imply a further reduction of the set of RTs.

We will use the following notation for FRTs: $D^d :=_C op(U_1^{d_1}, \dots, U_n^{d_n})$, where C is the set of constraints and $d = dom(D)$, $d_1 = dom(U_1), \dots, d_n = dom(U_n)$. We will also use $d := op(d_1, \dots, d_n)$ if we are not interested in denoting the variables.

We apply factorization also to the ERI. This is achieved by adding constraints for the domain variables T, R , and C . This also allows to define dependencies between locations and resource information. In this way we obtain a very compact specification of target machine models. We also allow to compose a FRT from a set of FRTs having the same RT pattern.

4.3. Factorised machine instructions

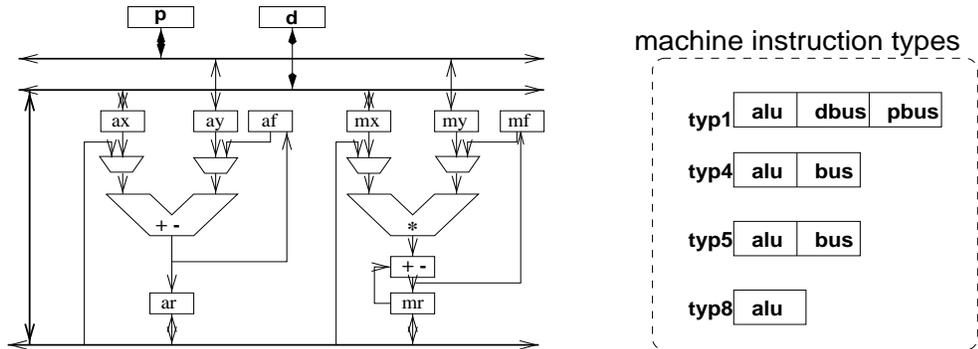
A machine instruction is a set of RTs which can be executed in parallel. For a given set of RTs R , a RT rt is called *compatible* to R , if $R \cup rt$ is a legal machine instruction. The concept of factorization is extended to machine instructions by means of representing these as a set of FRTs and a set of constraints guiding reduction of FRTs, so as to obtain only feasible machine instructions.

4.4. Example: Analog Devices ADSP-210x

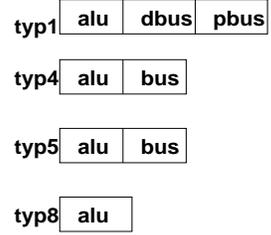
EXAMPLE: In fig. 9 the FRTs of the ADSP-210x are shown in the form of tree patterns. The FRTs for $+$ and $-$ are merged since they are isomorphic. The factorised extended resource information is given for the types and consumed resource (FU). Costs are not mentioned since the costs are equal to 1 for each of the shown FRTs. The RTs modeled by *frt1..3* can be executed in any machine instruction of type *typ1..4*. The given constraints model the ILP condition for *typ1*, which enforces, that the result of the RT will be located in an accumulator (and not in a feedback register *af* or *mf*). The constraint also denotes, that if the result of a RT is located in a feedback register, the RT cannot be included in a machine instruction of *typ1* (i.e. $T \neq typ1$). The FRTs modeling the parallel transfer operations are explicitly given by *frt4* and *frt5*.

ILP is modeled by the extended resource information plus a resource description for each type of machine instruction. The resources are specified according to the instruction field consumed by a RT in a machine instruction (machine instruction types in fig. 9). Thus the resources for the instruction type *typ1* are given by

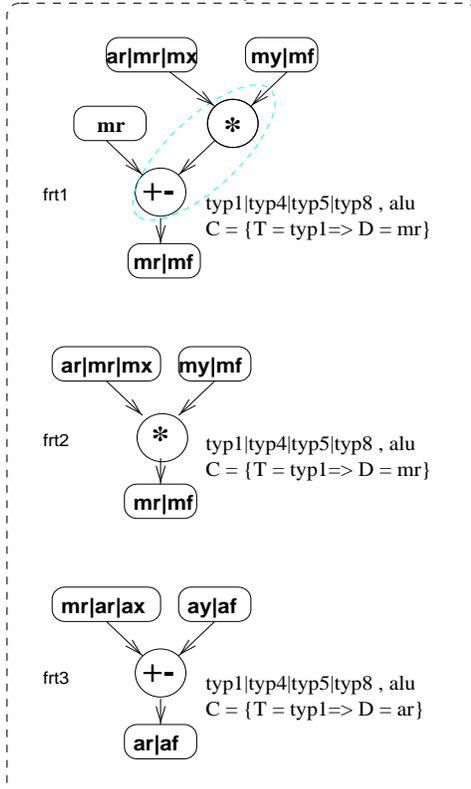
Figure 9. FRTs of the ADSP-210x



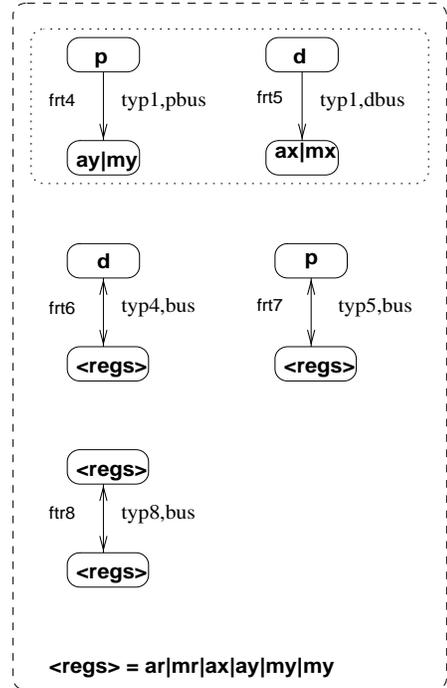
machine instruction types



FRTs of operations



FRTs of transfer operations



alu|dbus|pbus. It is sufficient to model both FUs in a single resource *alu*, as RTs executed on FUs cannot be performed in parallel on the ADSP-210x. \square

4.5. Retargeting of the code generator

A specification of the machine model is given as an input to the code generator. Parts of the specification of the ADSP-210x are shown in fig. 10. Each FRT is specified by the quintuple $(O, D, [U_1, \dots, U_n], ERI, Constraints)$: O, D, U_1, \dots, U_n are domain variables denoting the operation O , the definition D , and the uses $[U_1, \dots, U_n]$; $ERI = (Typ, FU, Cost)$ are the domain variables denoting the extended resource information, and $Constraints$ is the set of constraints specifying the dependencies of the variables⁸. Transfer operations are given explicitly. Resource declarations permit to introduce macros for sets of resources (e.g., `mems(X)`) and define the sizes of SRs. The machine type declaration part relates the machine types with the available FUs for each type.

Retargeting of the code generator is performed by generating a set of constraints from the specification, which eventually yields the internal machine model:

- From the FRTs a constraint $match(o, D, Uses, T, R, C)$ is derived. For a given operation o it will derive the domains and the constraints for the input variables. The operands $Uses$ are given in form of a list $[U_1, \dots, U_n]$ of domain variables.
- The relation $S \rightarrow^1 D$ is derived from the transfer operations. $S \rightarrow^1 D$ holds, if a value can be transferred from S to D by a single transfer operation.
- The binary relation \rightarrow^* denotes the reflexive, transitive closure of transfer operations. Therefore, $S \rightarrow^* D$ holds, if $S = D$ or a value can be transferred from S to D by a sequence of transfer operations.
- $S \rightarrow^+ D$ holds, if a value can be transferred from S to D , by a sequence of at least one transfer operations.
- The constraints *intf* and *ilp* are derived from the declarations of machine instruction types. They guide the correct reduction of machine instructions (given by a set of FRTs), in order to meet the ILP conditions of the target machine.

Furthermore, functions yielding resource information (e.g., sizes of SRs) are provided as part of the machine specification. All techniques of our code generator make use of these constraints and functions as the only interface to the internal machine model. So far we are able to handle target machine classes with single cycle instructions.

The specification of FRTs in fig. 10 almost represents the high-level specification methodology for defining constraints provided by ECLiPSe (except that certain mathematical symbols are represented by strings available in most text editors; e.g. \in is denoted by "`:::`"). For most of the derived constraints, used by FRT covering, we used a *generalized propagation approach* [52]. The constraints *intf* and *ilp* are generic and make extensive usage of low level predicates for domain access which is also provided by the finite domain library of ECLiPSe.

Figure 10. Partial internal machine model of the ADSP-210x

Operations

```

frt(Operation,Result,[X,Y].(Typ,alu,1), C = {
    Operation ∈ [+,-],
    Result    ∈ [ar, af],
    X         ∈ [ax, ar, mr],
    Y         ∈ [ay, af],
    Instr     ∈ [typ1, typ4,typ5,typ8],
    Result=af #=> Typ≠typ1 }).

frt('*',Result,[X,Y].(Typ,alu,Cost), C = {
    Result ∈ [mr,mf,'*(X,Y)'],
    X      ∈ [mx, ar, mr],
    Y      ∈ [my, mf],
    Cost   ∈ [0,1],
    Typ    ∈ [typ1,typ4,typ5,typ8],
    Result=mf ⇒ Typ≠typ1,
    Result='*(X,Y)' ⇔ Cost=0}).
    Virtual SR'*(X,Y)' for MAC operation.
    Cost=0 for MAC operation.

frt(Operation,mr,[mr,'*(X,Y)'],(Typ,alu,1), C = {
    Operation ∈ [+,-],
    Typ       ∈ [typ1,typ4,typ5,typ8]}).

frt(Operation,Result,[X,'1'].(Instr,alu,1), C = {
    Operation ∈ [+,-],
    Result    ∈ [ar,af],
    X         ∈ [ay, af],
    Typ       ∈ [typ1,typ4,typ5,typ8],
    Result=af ⇒ Typ≠typ1}).
...

frt(const(1),'1',[],(Typ,alu,0),Typ ∈ [typ1,typ4,typ5,typ8]).
frt(const(_).Result,[],(Typ,bus,1),Typ ∈ [typ4,typ5],regs(Result)).
...

```

Transfer Operations

```

transfer(Src,Dst,typ4,1,{Dst=d,regs(Src)}).
transfer(Src,Dst,typ5,1,{Dst=p,regs(Src)}).
transfer(Src,Dst,typ8,1){regs(Dst),regs(Src)}.
...

```

Resource Declarations

```

all_srs({af,mf,ar,mr,ax,ay,mx,my,p,d}).  Resource classes used in FRTs.
regs({ar,mr,ax,ay,mx,my}).
mems({p,d}).
...
size(ar,1).
size(ax,2).
...

```

Machine Instruction Declaration

```

instr(typ1,[alu,dbus,pbus]).
instr(typ4,[alu,bus]).
instr(typ5,[alu,bus]).
instr(typ8,[alu,bus]).
instr(typ6,[bus]).

```

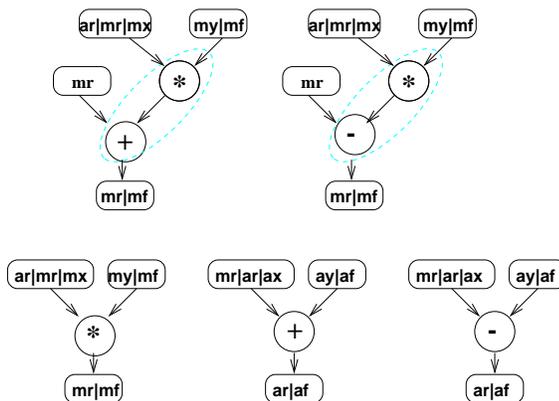
5. Code selection and FRT covering

In this section we first review the traditional approaches of code selection by means of tree pattern matching and dynamic programming. Then we introduce our concept of generating and representing the set of covers by means of *FRT covering*. FRT covering is one of the central issues of ICG. The goal is to provide a representation of the set of alternative covers, so as to be able to integrate code selection into the other code generation phases. So our goal is not to select a certain cover, but to generate a set of alternative covers, which should enable more flexibility for making up good decisions for the subsequent phases. Our approach is extended to DFGs, by means of taking into account the data routes between the definitions of CSEs and their multiple uses. Additionally, we have also derived techniques for pruning the set of alternative covers. One strategy is the generation of optimal covers for DFTs as well as for DFGs, when neglecting ILP. These techniques are based on the labeling and optimization strategies of ECLIPSe. We also used information of optimal covers to partially reduce the initial set alternative covers. An overview is given at the end of the section.

5.1. Tree pattern matching and dynamic programming

Most of the current tree covering approaches are based on *tree pattern matching* combined with *dynamic programming*. Tree pattern matchers are used to determine a set of possible covers for a DFT. In order to select a (locally) cheap cover from the set of possible covers, tree pattern matching is combined with dynamic programming.

Figure 11. Tree patterns



RTs are represented as tree patterns. In fig. 11 the tree patterns of the ADSP-210x are shown in a factorised form. Edges of tree patterns are associated with

the SRs required for the result and operands. These SRs have to be taken into account during covering. Consider the operation of node 1 in fig. 3. The nodes 2 and 3 are matched by the tree patterns rt_1 and rt_2 . Generally, a tree pattern $D := op(U_1, \dots, U_n)$ matches a node with operation op , if $def(rt_i) = U_i$ or if there exists a sequence of transfer operations to move the value in $def(rt_i)$ to U_i ($i \in \{1, \dots, n\}$). Consider the tree pattern depicted for node 1. The matching for the second operand is feasible as there exists a data transfer from ar to my ($my := ar$).

Most approaches are based on bottom-up tree pattern matching. In a bottom-up tree traversal all possible covers w.r.t. a given set of RT patterns can be constructed simultaneously. Each node of the tree is labeled with a set of matching tree patterns.

Selection of a certain cover from the set of possible covers is mostly done by dynamic programming. This is a very efficient technique to create locally optimal code for DFTs. A nice feature of such code selectors is that they can be generated automatically by code selector generators (CSGs). Thus, retargetability is supported. CSGs require a formal description of the target instruction set given as a tree grammar⁹. Each tree pattern is represented as a rule of the tree grammar, where the SRs are represented by the nonterminals of the grammar. Each rule is associated with a cost and an action. The action describes the emission of code corresponding to the rule¹⁰.

The main drawback of approaches based on dynamic programming is that ILP cannot be taken into account, since this would contradict the paradigm of dynamic programming: Optimal solutions for a tree must be computed from the optimal solutions of its subtrees, and solutions for subtrees must be independent of each other. Additionally, in the case of inhomogeneous SRs, spill costs cannot be taken into account.

5.2. FRT covering

In this subsection the generation of the set of all alternative covers for DFGs is described. Our pattern matching approach is based on labeling each DFG node i with a FRT that denotes the set of all RTs matching i . In the following we will denote the operation associated with node i of a DFG as op_i . We will refer to the value generated by this operation as *value i* , or as *operand i* (if i is an argument of another operation op_j). The FRT associated with node i is denoted as frt_i . Each frt_i is represented by the set of domain variables $DV_i = \{D_i, U_{i,j_1}, \dots, U_{i,j_n}\} \cup ERI_i$. D_i denotes the definition of frt_i . $U_{i,j}$ denotes a use of frt_i , i.e. the required location of operand j . ERI_i is the set of domain variables $\{T, R, C\}$, corresponding to the extended resource information of frt_i .

The set of matching RTs is determined based on the FRTs of the internal machine model. We perform matching at a node i by applying the constraint *match* (cf. section 4.5) which yields a FRT frt_i . This is performed through a bottom-up traversal of the DFG. For each operand j of frt_i we apply constraint $D_i \rightarrow^* U_{j,i}$, in order to take into account distributed SRs. This ensures, that there exists a path from each definition D_i to a use $U_{j,i}$. The constraint eliminates each d from $dom(D_i)$ and each u from $dom(U_{j,i})$, for which no sequence of operations for moving

a value from s to d exists. The reduction of domains may lead to further reductions of f_{rt_i} or f_{rt_j} , dependent on the associated constraints.

There is special handling of DFG nodes for program variables (leaves) and *common subexpressions* (CSEs). With each node denoting a program variable a set of possible initial locations at the beginning of a basic block is associated. Therefore, during covering, no FRT is associated with these nodes, but only a domain variable D denoting the location. This location is then considered as the definition D in the remaining covering process. With each common subexpression CSE_i we associate an extra variable D'_i . Covering of common subexpressions rooted at node i is handled by defining the \rightarrow^* relation between D'_i and each use $U_{k,i}$. The machine specification formalism allows to define certain allocations for the values of CSEs given by a special constraint $cse_loc(L)$. $D_i \rightarrow^* D'_i$ and $cse_loc(D'_i)$ are generated for each CSE_i . Uses of CSEs are handled like any other uses of operands. For each f_{rt_i} using CSE_j , $D'_j \rightarrow^* U_{i,j}$ is generated.

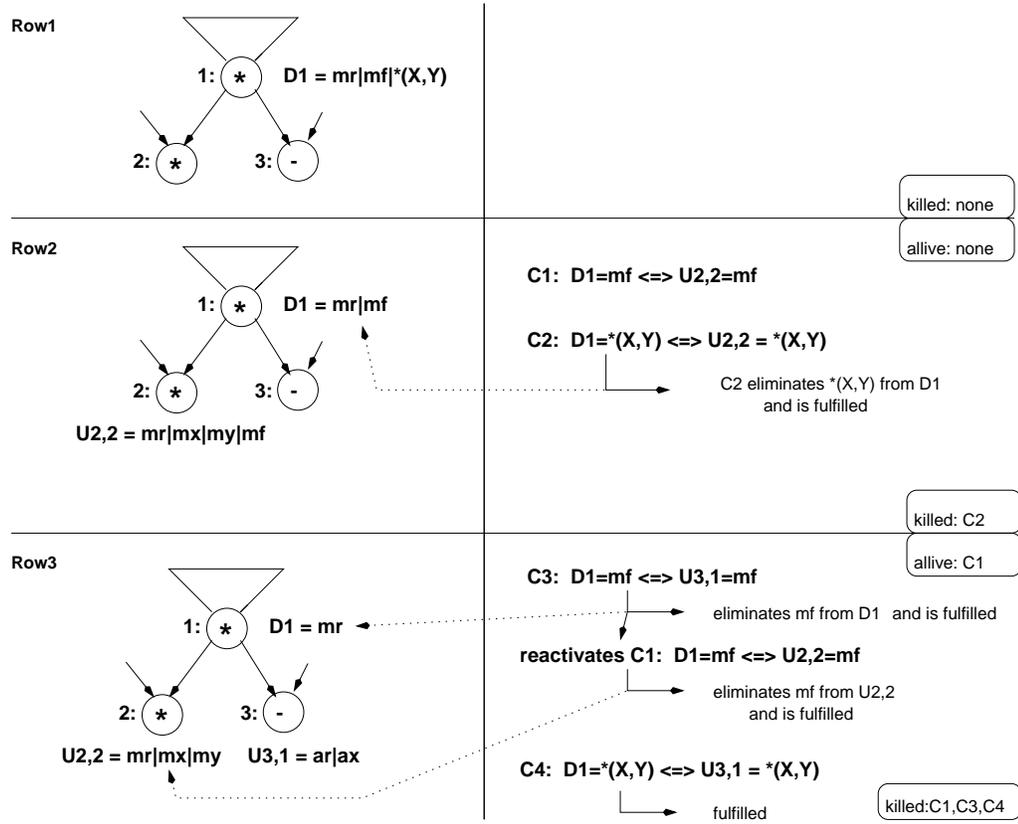
Only legal covers are left after pattern matching. Code generation now consists in reducing the domains of the domain variables such that each FRT reduces to a single RT. The constraints associated with the FRTs guide the correct reduction of domains, in order to yield only feasible RTs. In our CLP framework ECLiPSe, constraints are activated automatically whenever domains of variables are being reduced.

EXAMPLE: In fig. 12 the covering and reduction process is exemplified. The reduction of a partial DFG with some of the essential domain variables is shown in the left column of fig. 12. New generated constraints and their re-activations are shown in the right column and the effects of the constraints are described. The rows depict the matching process for the DFG nodes 1,2 and 3, respectively. For sake of simplicity, only data move constraints between the CSE and its uses are shown, which are generated by the application of \rightarrow^* . The handling of the extra variable D' of CSEs is also omitted. Constraints which are known to be fulfilled and which will not cause any further reduction on variables never have to be reactivated again and are "killed".

Row1 shows the situation after matching node 1. The result location of node 1 may be one of the locations $\{mr, mf, *(X, Y)\}$, where $*(X, Y)$ denotes the virtual resource to handle chained operations (cf. section 4.5). The second row Row2 shows the situation after matching node 2. New data move constraints C1 and C2 were generated. C1 specifies that there exists no possibility to move data from mf to any other SR. If D_1 is mapped to mf , then all uses also have to be mapped to mf . C2 denotes a constraint used for modeling chained operations (cf. section 4.5). Since the equality in C2 cannot hold for $U_{2,2}$, $*(X, Y)$ is removed from D_1 's domain, in order to meet the equivalence constraint of C2. Row3 shows the results of matching node 3. The new generated data move constraint C3 eliminates mf from D_1 since mf is no member of the alternative SRs of $U_{3,1}$. The modification of D_1 's domain reactivates C1, which now removes mf from $U_{2,2}$'s domain. Both C3 and C1 are fulfilled now and are "killed". C4 is fulfilled, since $*(X, Y)$ is not a member of D_1 and $U_{3,1}$.

□

Figure 12. FRT covering



In table 1 the run-times of FRT covering for the selected DSPStone benchmarks are shown. All run-times in this paper are given in SPARC-20 CPU seconds. The runtime data indicate that FRT covering is efficient. FRT covering for our internal benchmarks was in the range of a few seconds. The table also shows some characteristics of the benchmarks: the number of DFG nodes, DFG edges, and the number of CSEs.

In the following a coarse worst case analysis of the complexity of FRT covering is given. The complexity depends on the following factors. We assume we have N DFG nodes and that for each node, a maximum of C constraints are activated. Each constraint uses at most V variables and the maximal size of the domains is D . The complexity of the constraints we used for specifying FRTs depends on the number of variables and domain sizes, therefore denoted as $c(V, D)$. In most cases $c(V, D) \leq O(V * D)$ holds. C , V and D are of limited sizes. V is in the range $1 < V < 8$. C and D depend on the machine model: C also depends on the maximal uses of CSEs but was less than 10 on average for the benchmarks used so far. D is limited to the maximal amount of FUs, SRs and instruction types. The constraints we use and generate are not disjunctive. Reactivation of constraints is triggered by the reduction of domain variables. Thus, each constraint can be reactivated no more than $V * D$ times. In total we have $(V * D) * (C * N) * c(V, D)$. If we take the maximal ranges for V and C into account and assume that $c(V, D) = O(V * D)$ then we get a complexity of $O(N * D^2)$, where D is constant for a given machine model. *Another important result* of this analysis is, that a generally exponential number of alternative covers is given by a representation of linear size (w.r.t. to the number of DFG nodes N), by means of FRT covers.

Table 1. Run-times for FRT covering

source	runtime	DFG Nodes	DFG Edges	CSEs
complex multiply	0.85	13	14	4
complex update	0.96	17	18	4
iir filter	0.62	17	19	2
lattice filter	1.52	23	27	8

5.3. Pruning the set of alternative covers

As a special application of FRT covering, we have developed techniques for pruning the set of alternative covers. This comprises:

- Reduction of certain variables, e.g., the SRs for CSEs can be specified the target processor.
- Generation of *optimal* DFG covers when neglecting ILP. The FRT covering procedure can thus also be used as a stand-alone optimal code selection technique for DFTs and also for DFGs. It should be noted that the result is the set of all

optimal covers, in contrast to the traditional approaches, where only a single optimal cover is produced.

- Using information from optimal covers in order to partially reduce the set of all alternative covers. For the ADSP-210x we copy all chained operations (MAC operations) occurring in the optimal covers into the initial set of all alternative covers.

The techniques for generating optimal covers for DFTs and DFGs can consume exponential run-times in the worst case. This is due to the fact, that a FRT cover may represent an exponential number of alternative covers (w.r.t. to number of DFG nodes). In worst case, the complete search space has to be explored in order to find an optimal solution. Experiments have shown that optimal DFT covers can be generated efficiently and that they are not time critical at all. It should be noted, that techniques based on dynamic programming are faster by one order of magnitude in average. Experiments for DFGs have shown that optimal DFG covers can be generated for small to medium size (up to 25-30 nodes) DFGs within reasonable amounts of computation time (cf. table 2). We have developed a suboptimal strategy for larger DFGs, which leads to acceptable run-times. In this strategy, the labeling of domain variables of the roots of DFTs, which are CSEs or last uses in the basic block, is postponed until its first usage is labeled. This generally leads to better results compared to the accumulated optimal costs of DFTs, and comes close to the optimal solution for DFGs. Results are shown in table 2. The costs shown in this table denote the accumulated costs for all DFTs occurring in the basic blocks. The column denoted *DFT* shows the results for optimal covering for each DFT, column *CSEs* shows the suboptimal and column *DFGs* the optimal results for DFGs. Run-times are all given in CPU seconds. Since uses of CSEs may occur in different DFTs, different orderings in optimizing the DFTs of a basic block also lead to different results (this only occurred in some internal test cases). All internal benchmarks were sub-optimally covered within a minute for the *DFT* strategy and within five minutes for the *CSE* strategy. Optimal covers for DFGs could not be computed for the larger internal benchmarks in acceptable time.

In our experiments we made use of the suboptimal strategy in order to determine chained operations. The resources of the corresponding DFG nodes were only reduced as much as necessary.

Table 2. Optimization strategies

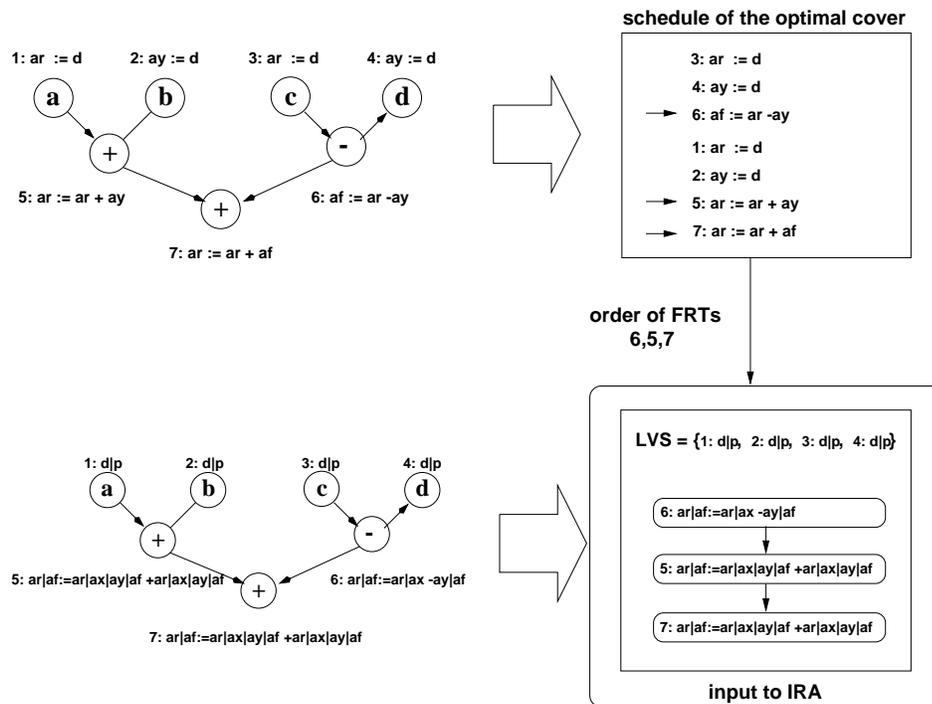
source	DFTs	time1	CSEs	time2	DFGs	time3
complex multiply	14	0,17	10	0,46	10	6,90
complex update	16	0,30	12	0,95	12	4,80
iir filter	16	0,41	13	0,58	13	1,73
lattice filter	39	0,75	25	4,49	24	235,37

6. Integrated register allocation

In this section the integration of code selection and register allocation is described. This includes the tasks of *spill code generation* and *data routing*:

- *Spilling* is required, when the number of live values located in a certain SR exceeds the SR capacity. In this case values have to be selected, which are moved to other SRs. Spill code has to be inserted by means of transfer operations to the spill location, and for reloading it for later uses.
- *Data routing* determines the SR paths which a value will take from its definition to its uses. This includes also the selection of resources and transfer operations to implement the paths.

Figure 13. Sequentialization of DFG^{FRT}



IRA expects a sequence of FRTs, i.e., a sequential schedule of a given DFG^{FRT}. Note that this sequence does not yet contain transfer operations (cf. fig. 13). These are inserted only by the register allocator by means of data routing. The second task of the register allocator is to include necessary spill code. The register allocator has to keep track of the locations of live values. This information is given

by means of a *live value set* (LVS). Each $lv \in \text{LVS}$ is a tuple (i, pl) where pl denotes the current location of value i . The initial LVS of the register allocator contains the locations of the variables live at the beginning of the basic block (cf. fig. 13). With respect to a given live value set LVS, we denote an FRT as *data ready*, iff all operands j have an entry in LVS, and *use ready*, iff all operands are in their required locations (denoted by the predicate $use_ready(frt, LVS)$). All use ready FRTs are also data ready by definition.

The sequence of FRTs is generated based on a schedule of an optimal cover of the DFG. The schedule is generated by heuristic (sequential) instruction scheduling technique ([37]) for irregular data paths. As this approach is tree based, the optimal DFG^{FRT} is decomposed into a sequence of FRT trees. Instruction scheduling is performed subsequently on each of these trees. This ordering is then reconstructed for the DFG^{FRT} representing the set of alternative covers (fig. 13).

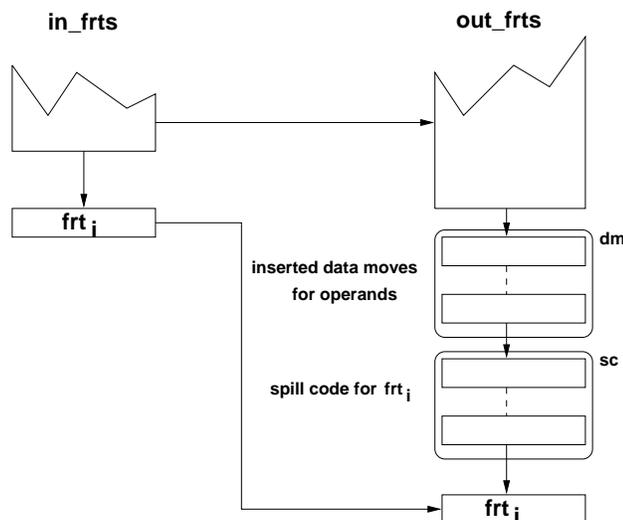
The register allocator traverses the FRT sequence. For each frt_i it inserts a new live value for the definition to LVS, in order to provide the subsequent FRTs with the locations of their operands. If an operand is not in its location, transfer operations have to be inserted. The new locations have to be inserted to the LVS (see also fig. 14). We now give an informal description of the algorithm of IRA. A more detailed description is given in the following subsections. Sequences are represented as lists $l = [l_1, l_2, \dots, l_n]$, where @ denotes the concatenation of lists. [] represents the empty list. $head(l) = l_1$ gives the first element of l and $tail(l) = [l_2, \dots, l_n]$:

- *input*: the FRT sequence $frts_{in}$ of a basic block and the initial set LVS, containing the locations of the variables live at the beginning of the basic block.
- *output*: the FRT sequence $frts_{out}$.
- *algorithm*:
 - While** $frts_{in} \langle \rangle []$ **do**
 - 1. $frt_i := head(frts_{in})$
 - 2. **operands use ready**: For each operand j of frt_i , which is not use ready, generate the necessary transfer operations to_j . This may induce spilling and we have to insert spill code in the form of a list of transfer operations sc_j . This results in the list of transfer operations $dm_j := sc_j @ to_j$ for operand j . The final list of transfer operations for operands is given by $dm := dm_{j_1} @ \dots @ dm_{j_n}$.
 - 3. **last uses**: Eliminate all last uses from LVS. These are all live values which are not referenced by any of the FRTs in frt_{in} and which are not live at the end of the basic block.
 - 4. **add definition**:
 - Check if spilling is necessary when inserting a new live value for the definition of frt_i .
 - If spilling is required generate the necessary spill code sc .
 - Insert a new live value $(i, def(frts_i))$ for the definition of frt_i to LVS.

5. $frts_{out} := frts_{out}@dm@sc@[frt_i]$
6. $frts_{in} := tail(frts_{in})$

The algorithms in this paper are represented in an imperative style¹¹. For sake of simplicity, they do not reflect the backtracking features given by the original implementation in ECLIPSe. Aspects of backtracking will be mentioned when required in the following descriptions.

Figure 14. Insertion of transfer operations and spill code

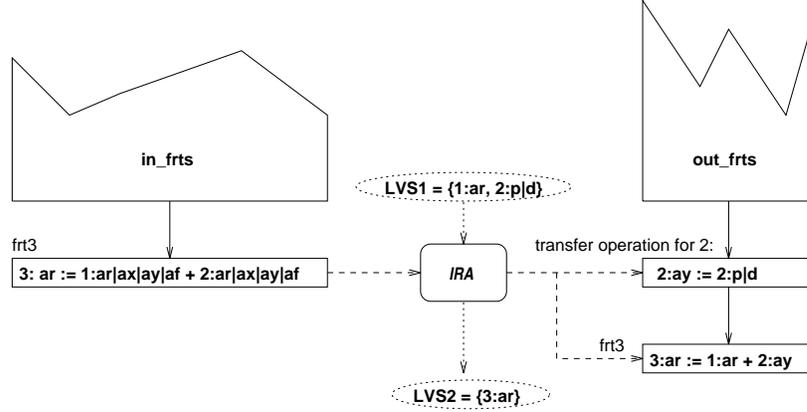


6.1. Data routing of operands

The current location L_j of an operand j is given by the LVS. The required locations for the operands of frt_i are given by the the list of uses $U_{i,j_1}, ..U_{i,j_n}$. The register allocator performs the following strategy to check if operands are in their required locations:

1. First all uses are determined for which the operands are in the required locations, i.e. for which the condition $L_j = U_{i,j}$ holds.
2. For operands j not in the required location, transfer operations from L_j to $U_{i,j}$ have to be inserted. This may result in a sequence of transfer operations $to_1, .., to_n$, where to_n is the FRT $U_{i,j} := L'_j$. For each $to_i = d := s$, the single step relation $s \rightarrow^1 d$ must hold.

Figure 15. Illustration of data routing



EXAMPLE: We consider the example in fig. 15. The register allocator has reached frt_3 and checks the locations of the operands. The locations of the operands are given by LVS_1 . The location of operand 1 is L_1^{ar} which means that it is located in the accumulator ar . Operand 2 is located in one of the memories (p, d) . It is first checked if any operand is in its required location. This is the case for operand 1, since the condition $L_1 = U_{3,1}$ holds. This equality leads to the reduction of $dom(U_{3,1})$ to $\{ar\}$. In order to yield only feasible RTs, $dom(U_{3,2})$ is reduced to $\{af, ay\}$. Now operand 2 is not in the required location. Thus a transfer operation has to be inserted. Therefore, the single step relation $L_2 \rightarrow^1 U_{3,2}$ must hold. In order to meet this constraint, the location $U_{3,2}^{ay|af}$ is reduced to $U_{3,2}^{ay}$ as af cannot be reached in a single move from either memory p or memory d .

The equality $L = L'$ is a constraint which leads to the reduction of both domains of the locations. In our example both domains of L_1 and of $U_{3,1}$ are reduced to ar . Constraints like the equality have to hold throughout code generation. Therefore, they are attached to the locations and guide all domain reductions. In case of the equality $L_i = L_j$, any reduction of one of the domains will enforce the same reduction for the other domain. \square

For each data transfer a new live value must be inserted to the LVS. The insertion of new live values can lead to spilling. Therefore, spill code has to be inserted before the transfer operation which induced the spilling. The generation of spill code for a transfer operation to of the form $d := s$ has to save the value in location d . Thus a spill operation $d' := d$ has to be performed before to , where d' is the new location for the spilled value. For irregular data paths, *spilling may enforce the spilling of other values*. Thus a sequence *spills* of spill operations might need to be generated. The code induced by a data move may have the form $spills@[d := s]$.

After checking the operands, all last uses are eliminated from LVS. Last uses are the live values, which have no further use in the remaining sequence of FRTs

considered by the register allocator. In case that values are still live at the end of the basic block (because they are used in subsequent basic blocks), they are kept in LVS.

The above mechanism provides very flexible data routing, because it leaves as much freedom for the subsequent decisions as possible. Therefore, locations are only reduced as much as necessary. If we have a use with the location $U_{i,j}^{ar|ax|ay|af}$ and the current location of operand j is $L_j^{ax|ay}$, then it is sufficient to reduce both locations to the domain $\{ax|ay\}$ (achieved by the equality constraint $U_{i,j} = L_j$). The same holds for any inserted transfer operation, e.g. $ar|ax|ay := p|d$. As a result, our data routing technique yields a *set of alternative data routes*.

We use a specific data routing for CSEs. CSEs are required in more than one SR, due to their multiple uses. Therefore, for CSEs more than one entry in LVS may occur. This may support paths with common prefixes in the paths of transfer operations.

6.2. Spilling

When a new live range is inserted into LVS, it must be checked, that the SR capacities will not be exceeded (only for RFs). Otherwise, spilling of a live value of LVS has to be performed. As there is no fixed assignment of values to SRs, whether or not capacities will be exceeded cannot be determined by simply counting the values in each SR. To determine necessary spilling we use an extended notion of *interference*, known from graph coloring register allocators [14, 7]. We say that the live values L_j and L_i interfere, if they have overlapping life times and if the intersection of the domains of locations is not empty. Interfering live values have to be assigned to different registers.

If we insert a new live range nlr , the *necessary condition for spilling* is, that the number of interfering members satisfies the condition $|lr_i \in LRs| \geq |dom(nlr)|$. The new live value in fig. 16a) does not interfere with the live value in LVS, as the intersection of the domains is empty, i.e. their locations can never be reduced to the same register. In the second example 16b) the two live values interfere but they can be still reduced to different SRs. In order to guarantee this, an inequality constraint¹² is generated for these live values. In example 16c) spilling has to be performed, as any substitution of the three locations will result in at least two identical locations.

The set of live values which interfere with a new live value nlv is called the set of *spill candidates*. If the *necessary spilling condition* holds we apply the two following strategies:

1. *Reduction*: The necessary condition for spilling will not always enforce spilling. There are three situations when spilling can be prevented by means of reducing the domains of locations.
 - (A) Try to reduce the domains of the locations of the spill candidates, such that the necessary condition for spilling is violated, by reducing the number of interfering live values.

- (B) Try to reduce the domain of the location of the new live value, such that the necessary condition for spilling is violated.
- (C) Select spill candidates with multiple entries in LVS (live values of CSEs).
2. *Spilling*: If reduction cannot be performed, select a spill candidate lv_{spill} and a spill location, and generate spill code. Spilling is not necessarily performed to memory, but *also free registers can be used for spilling*.

Figure 16. Interference of live values

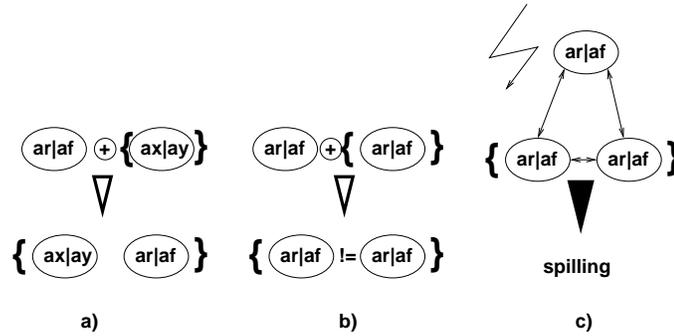
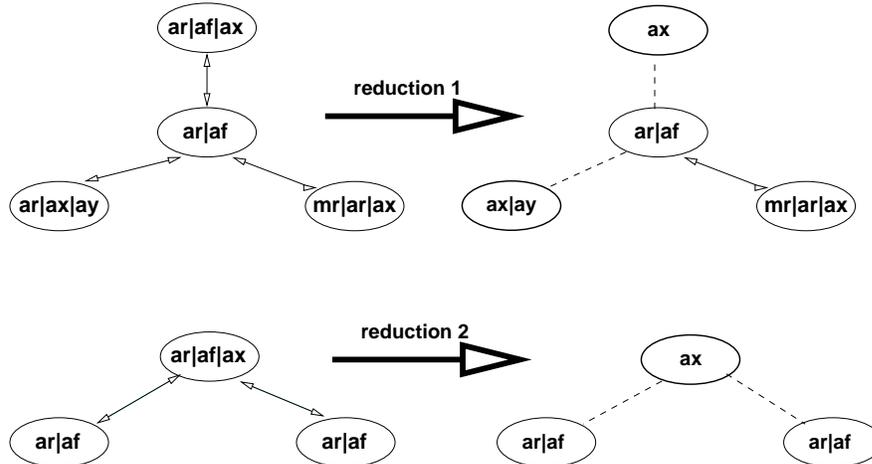


Figure 17. Prevention of spilling by reduction



6.2.1. Reduction strategy The basic idea of the reduction strategies is to reduce the domains of live values, such that the intersection of the domains is empty. This leads to reduction of interfering live values and may therefore violate the necessary condition for spilling. We try three reduction strategies for a new live value lv :

- The first strategy applies, if there exists a spill candidate lv_s with $dom(lv) \subset dom(lv_s)$. In this case we reduce the domain of the corresponding spill candidate to the difference $dom(lv)/dom(lv_s)$. An example is shown in fig. 17 (Reduction1).
- The second strategy applies, if there exist spill candidates lv_s with $dom(lv_s) \subset dom(lv)$. In this case we reduce the domain of the new live value lv . An example is shown in fig. 17 (Reduction2).
- Select a live value of a CSE j which has multiple entries in LVS and which is not in one of its required locations.

The strategies to select a candidate for reduction are very simple. The first candidate in the list of spill candidates, whose domain can be reduced, is selected. The same applies to multiple LVS entries. The development of more powerful selection functions is a potential for further improvements.

Backtracking guarantees, that if one of the strategies cannot be applied the next one is automatically tried. Furthermore, if the code generation process in the subsequent phases fails, backtracking enforces the selection of an alternative strategy.

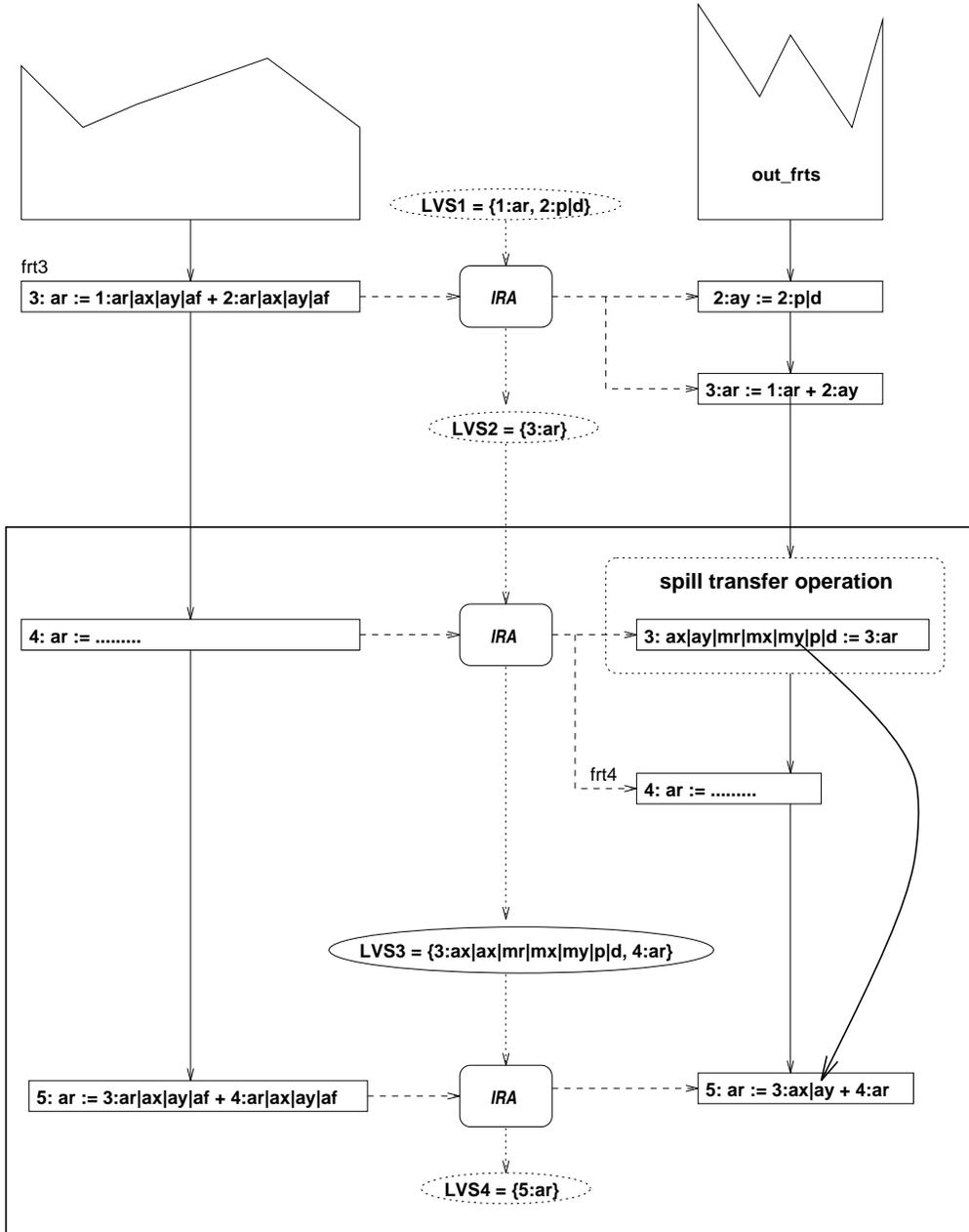
6.2.2. Spilling strategy The selection of a spill candidate is driven by the following order of criteria:

1. Select a candidate which is not in one of the required locations of one of its uses. In this case no overhead is produced, as for these candidates a transfer operation is necessary anyway.
2. Select a candidate with latest next use according to the occurrence in the remaining sequence frt_{in} of FRTs.
3. Select a candidate with the smallest number of remaining uses in frt_{in} .

Spilling is not necessarily performed to memory. A factorised transfer operation is inserted denoting all possible locations that can be reached from the actual location of lr_{spill} in one move (fig. 18). The spilling to RFs has the advantage, that one transfer operation can be saved, if the spill location L_j and a use of the operand $U_{i,j}$ satisfy the condition $L_j = U_{i,j}$. Further advantages will be mentioned later in section 6.5.

Spilling may induce further spilling of values. In order not to spill one of the required operands, we provide a mechanism to prevent certain live values in LVS from being spilled. In case spilling is impossible due to constraints or locking of live values, backtracking is enforced, leading to the selection of new spill candidates. Also, the generation of certain transfer operations can be rejected.

Figure 18. Spill routes



6.3. Live values and the interference graph

We assume that insertion of live values is only performed, if this does not require spilling of other values. Thus decisions for spilling have to be performed in advance. Inserting a new live value lv with location L to LVS always implies generating the constraint $intf(L, [L_1, \dots, L_n])$ for each location L_j of an interfering live value $lv_j \in$ LVS. These constraints can be considered as an implicit representation of an interference graph. The constraints ensure, that the number of interfering live values will never exceed the available register resources. Also, the reduction strategies described above are integrated in this constraint, in order to enable the satisfaction of the constraints (cf. the remarks on complexity in section 7.6).

It should be mentioned that in IRA two classes of deadlocks can occur:

1. Deadlocks caused by situations, where there exists no labeling of the domain variables, such that all constraints are fulfilled.
2. Deadlocks due to data routing, as mentioned in [28]. Examples are deadlocks, where spilling of values is required, which are located in SRs from which spilling is impossible (e.g. SR mf).

To prevent deadlocks due to constraints which cannot be met, we perform relaxation of constraints. We delete interference constraints and perform a reallocation where spill code has to be inserted for the corresponding values.

Deadlocks due to data routing can be solved by backtracking. However, we only allow backtracking up to the decisions made for a single FRT (in the context of IRA, these are the decisions made in one iteration of the main loop). We are currently developing a deadlock model to identify deadlock conditions for certain machine classes. The goal is the definition of constraints, that prevent the occurrence of deadlocks, by means of reducing certain potential SRs which could denote deadlock candidates. We have developed a set of constraints which prevent deadlocks, but we still have to classify the set of all deadlocks which may occur. However, we did not encounter deadlocks for all practical benchmarks considered so far.

6.4. Example for IRA

EXAMPLE: The FRT sequence generated by IRA still contains a large amount of flexibility. The following example shows the generated FRT sequence for the example from the introduction, generated from the FRT sequence shown in fig. 13:

```

3: ar|ax|ay|mr := d|p
4: ar|ax|ay|mr := d|p
6: af|ar      := - ax|ay|ar|mr:c_3 ax|ay|ar|mr:d_4
1: ar|ax|ay|mr := d|p
2: ar|ax|ay|mr := d|p
5: af|ar      := + ax|ay|ar|mr:a_1 ax|ay|ar|mr:b_2
7: ar|af      := + af|ar:tmp_5 af|ar:tmp_6

```

This sequence actually represents a set of possible sequences. It also contains that one which can later be transformed into the optimally compacted sequence of machine instructions.

The insertion of another FRT $af := \dots$ would lead to the reduction of the live value for node 7 to ar , thus preventing the spilling of live value 7.

The result of labeling will lead only to feasible sequence of RTs, e.g.:

```

3: ar:= d
4: ay:= d
6: af := - ar:c_3 ay:d_4
1: ar:= d
2: ay:= d
5: ar := + ar:a_1 ay:b_2
7: ar := + ar:tmp_5 af:tmp_6

```

□

6.5. Results of IRA

Techniques for data routing consider different data routes between definitions and uses in order to prevent spilling. In our method, data routing is given by FRTs. The domains of the locations represent alternative routes. This is also the case for inserted data moves and spill code. Furthermore, alternative routes are kept as long as possible, so as to yield freedom for subsequent decisions.

The spilling strategy we propose not necessarily spills values to memory, **but also to free registers**. One result is the reduction of data transfers, as shown in section 6.2.2. There are some further advantages implied by the reduction of memory accesses:

- The amount of required memory is reduced as certain values can exclusively be kept in registers.
- Reducing the amount of memory accesses in general also leads to reduced power dissipation.
- The amount of code needed for memory address computations is also reduced.

By introducing flexible spilling and data routing for the register allocator, the code was improved up to 20 % compared to the code generated for optimal covers without flexibility. The combined data routing and spilling strategy reduced the amount of memory accesses drastically, as much as 70 %. The results of IRA are given in table 3. The results given in columns S1 and S2 show the lengths of the generated FRT sequences of IRA applied on the optimal cover with spilling only to memory and for the same schedule of FRTs with data routing and flexible spilling, respectively. Columns Mem1 and Mem2 represent the transfer operations with accesses to memory, and columns RT1 and RT2 represent the transfer operations

without memory accesses. In columns `time1` and `time2` the run-times (in SPARC-20 CPU seconds) for both approaches are shown (also compare the analysis of the worst case complexity in section 7.6).

Table 3. Results of IRA

source	S1	S2	Mem1	Mem2	RT1	RT2	time1	time2
complex multiply	10	9	7	6	0	0	1.02	1.25
complex update	13	11	7	6	1	0	1.24	1.51
iir filter	16	13	11	8	0	0	1.39	1.71
lattice filter	29	24	12	3	1	5	2.40	3.67

In the following section we consider the integration of code selection and register allocation into the instruction scheduling phase. In this phase, also ILP is taken into account for optimization.

7. Integrated instruction scheduling (IIS)

IIS denotes the integration of code selection, register allocation and instruction scheduling. The goal is exploitation of potential ILP, so as to reduce the amount of instruction cycles for basic blocks and to prevent unnecessary generation of spill code. Our approach is based on list scheduling, extended by the integration of data routing and spilling techniques of integrated register allocation. The IIS algorithm maps a DFG^{FRT} to a sequence of machine instructions. Like other list scheduling algorithms, IIS makes use of a data ready set (DR) and selection functions for choosing the next candidate to be scheduled. The selection of the next candidate is based on the goal of minimizing spill code.

We make use of the following predicates: $compatible(frt, mi)$ denotes the compatibility of FRT frt and machine instruction mi . $spilling(frt, LVS)$ denotes, that frt requires spill code if a new live value for the definition is inserted to LVS. This predicate fails, if reduction, according to section 6.2.1 is feasible. $no_resources_free(mi)$ denotes, that all resources of a machine instruction are consumed by its set of FRTs. The IIS algorithm is given informally in fig. 19.

7.1. Compatibility of FRTs and machine instructions

The concept of factorised RTs is extended to factorised machine instructions. Machine instructions are represented by a set of FRTs. Constraints between the FRTs of a machine instruction guide the correct reduction of FRTs, in order to meet the ILP constraints of the target processor. A new constraint $ilp(frt_i, mi)$ is generated if a new FRT is inserted into a machine instruction. The test whether a new FRT is compatible with mi can also be performed by constraint ilp (cf. section 4.5). The concept of factorised machine instructions permits to postpone the assignment

Figure 19. IIS algorithm

- *input*: DFG^{FRT} and the initial live value set LVS.
- *output*: The machine instruction sequence *mis*.
- *algorithm*:

```

DR = initial set of FRTs which are data ready
mi = new machine instruction
while DR ≠ {} do
  a) if no_resources_free(mi) then
    Append mi to mis and generate a new empty machine instruction:
    mis = mis@[mi];
    mi = new machine instruction;
  b) scheduling:
  1. Select a compatible RT from DR which induces no spilling, iff
     ∃frt ∈ DR.compatible(frt, mi) ∧ uses_ready(RT, LVS) ∧ ¬spilling(frt, LVS)
     Insert frt into mi, generate the constraints and update
     LVS.
  2. Generate compatible data moves:
     2.1 Select a FRT which is not use ready, generate the transfer
         operations, and insert them into DR, iff
         ∃frt ∈ DR.¬uses_ready(RT, LVS)
     2.2 If not 2.1 then select a required spill candidate which is use
         ready and insert it into DR, iff
         ∀frt ∈ DR.spilling(frt, LVS) ∨
         ∃frt ∈ DR.spilling_required(frt, LVS)
  3. If not 2 then generate a new machine instruction:
     mis = mis@[mi];
     mi = new machine instruction;

```

of a certain machine instruction type. Again, this leaves more freedom for the subsequent decisions.

7.2. Selection of a compatible FRT from DR

We select a compatible FRT from the DR, which can be scheduled into the current instruction cycle. A FRT is selected which induces no spilling. This comprises FRTs which can be reduced according to the reduction strategies of section 6.2.1. The FRT with the longest path in the DFG is selected first. The insertion of a FRT to the current machine instruction *mi* yields the the new machine instruction $mi \cup \{f_{rt_i}\}$. The following tasks have to be performed:

1. $mi := mi \cup \{f_{rt_i}\}$ and generate a new *ilp* constraint.
2. Eliminate the last uses of f_{rt_i} from LVS and insert a new live value for D_i into LVS.

3. Insert all f_{rt_j} to DR which become data ready due to the insertion of D_i to LVS.

7.3. Generating compatible data moves

If there are no compatible and spill free FRTs in the data ready set DR, and there are still free resources for transfer operations in mi , we try to generate transfer operations for data ready but not use ready FRTs.

7.3.1. Generate moves for not use-ready FRTs A FRT which is data ready but not use ready is selected. Only compatible transfer operations are generated by means of data routing as introduced in section 6.1. The generated moves will be inserted into DR and will be selected in the next iteration of the main loop of the IIS algorithm. The FRT with the longest path in the DFG will be selected first.

7.3.2. Generating required spill code If spilling can be prevented by scheduling a non-compatible FRT into the next machine instruction, and if there is no FRT for which spilling has to be performed anyway, then no transfer operation for spilling is generated. Therefore, the conditions for generating spill code are:

1. All use ready FRTs $f_{rt_i} \in DR$ imply $spilling(f_{rt_i}, LVS)$, or
2. there exists an FRT $f_{rt_j} \in DR$ for which spilling has to be performed.

Here, the same spilling strategies as for IRA are used.

7.4. Machine instruction sequence for the example

EXAMPLE: IIS generates the following sequence of machine instructions for our running example, which still retains a certain amount of freedom:

```

=====
Instr => Typ: typ1  RLS: [dbus|pbus, dbus|pbus] Cycle: 0
1: ax|ay:= dmem|pmem
2: ax|ay:= dmem|pmem
=====
Instr => Typ: typ1  RLS: [dbus|pbus, dbus|pbus, alu] Cycle: 1
3: ax|ay:= dmem|pmem
4: ax|ay:= dmem|pmem
5: ar := + ax|ay:a_1 ax|ay:b_2
=====
Instr => Typ: Instr1=typ4|typ5|typ8  RLS: [alu] Cycle: 2
6: af := - ax|ay:c_3 ax|ay:d_4
=====
Instr => Typ: Instr1=typ1|typ4|typ5|typ8  RLS: [alu] Cycle: 3
7: ar := + r:tmp_5 af:tmp_6
=====

```

□

7.5. Results of IIS

In table 4, the number of generated machine instructions for our set of benchmarks is shown. The numbers of machine instructions for the first three benchmarks are the same as in the hand crafted code of DSPStone, which are optimal when neglecting address computations. For our small running example we also obtain the optimal result shown in the last row. The run-times for IIS are given in the last column (in SPARC-20 CPU seconds). Run-times for internal benchmarks were all within a minute. The following subsection 7.6 comprises an analysis of the worst case complexity of IIS. In section 8 results for executable code including address computations will be shown.

Table 4. Results of IIS

source	# instructions	runtime
complex multiply	6	1.11
complex update	9	1.76
iir filter	12	1.15
lattice filter	18	6.53
example	4	0.91

Like in IRA, we allow constraints violation, to prevent deadlocks due to constraints which cannot be met. So far, in the benchmarks and in the internal test cases, this case did not occur.

7.6. Remarks on the complexity of IIS

In this section we will informally analyze the worst case complexity of our scheduling approach IIS. The essential parameters for a worst case analysis are given in table 5. In the following analysis the sizes of the sets DR and LVS will be approximated with N . The overall worst case complexity of IIS is given by $WC = C_l * C_b * C_c$.

Table 5. Essential parameters for complexity analysis

C_l	number of main loop iterations
C_b	worst case complexity of possible backtracking steps in a single main loop iteration
C_c	worst case complexity for constraint re-/activations
N	number of DFG nodes
V_{max}	the maximum number of variables in occurring in a constraint
C_{max}	the maximum number of constraints generated for each DFG node
D_{RC}	number of register cells in the target architecture
D_{FU}	number of FUs

7.6.1. Main loop iterations (C_l): We can assume, that the while loop terminates within $C_l = O(N)$ iterations, since in every iteration either one node is scheduled or some new data move nodes are generated. The amount of generated data moves is finite and generally proportional to N .

7.6.2. Backtracking (C_b): Like in IRA, IIS does not comprise complete backtracking. We only allow backtracking over the possible solutions in a single main loop iteration of IIS¹³. The worst case effects of backtracking may result in testing the maximal number of solutions possible for:

- *Selecting a candidate for scheduling:* For a $d \in DR$ we check if the uses are ready in LVS. The complexity of all solutions is $O(|DR| * |LVS|)$ which can be approximated by $O(N^2)$.
- *Data routing for FRTs which are not use ready:* We search for a $d \in DR$ which is not uses ready in LVS. The worst case complexity is $O(N^2)$.
- *Selecting a spill candidate:* depends on how complex the strategy is, which is $O(N^2)$ in the current version of IIS.

The total complexity of backtracking therefore results in $C_b = O(N^2)$

7.6.3. Re-activations of constraints (C_c): In section 6 a reduction strategy was introduced, in order to prevent the generation of spill code. This strategy is also integrated in the constraints *intf*, and a similar reduction strategy is incorporated in the constraint *ilp*.

First we assume, that the reduction strategy is disabled. Then, only non-disjunctive constraints are generated in our system. If we resume the complexity analysis of covering in section 5, the total number of possible re-activations of constraints occurring in a DFG was given by $O(N)$. This still holds for C_c for the following reasons:

- The number of new constraints is limited and the maximum of generated constraints can be still given by a constant C . For each scheduled node, the constraints *intf* and *ilp* are generated. For the new generated data moves, the matching constraints described in section 5 are generated.
- The complexity of the generated constraints *intf* and *ilp* depends D_{RC} and D_{FU} . The complexity of each reactivation of *intf* is in the worst case $O(D_{RC}^2)$, and $O(D_{FU}^2)$ for *ilp*. This does not affect the overall complexity of re-activations, since D_{RC} and D_{FU} are constant for a given target architecture.

If we enable the reduction strategies, then *intf* and *ilp* become disjunctive, which can lead to exponential run-times: there are $D = \max(D_{RC}, D_{FU})$ possibilities for reducing variables at each node. Since the reduction strategy may indicate the reactivation of further reductions we may yield D^N possible combinations for reduction, which could all be tried due to failure and backtracking over the constraints.

7.6.4. Total complexity If the reduction strategy is disabled, in the worst case we get a run-time of $O(N^4)$. Otherwise, exponential run-times are possible in the worst case, but this case never occurred in our experiments. Concerning the reactivation of constraints, only very few constraints are reactivated and failures are detected very early. This also holds for the reduction strategy, where in general a feasible reduction or the failure of reduction is found very early. Our experiments show that our approach is practicable for real life examples (with large basic blocks), even if reduction is activated.

7.6.5. Complexity of Labeling Labeling serves two purposes. The first one is to yield the final code from the FRT machine instructions, by means of a final resource binding. The second goal is to check for global feasibility of the final schedule, since the generated constraints only guarantee local feasibility. If labeling succeeds we also know that there exists a feasible solution. In case no feasible solution exists, one problem is, that the feasibility check is an NP-complete problem, which may cause exponential run-times. In our experiments the labeling procedure did not fail so far, and a labeling was found very fast (within a second). To prevent cases of exponential run-times, user defined time outs are possible for labeling. If labeling fails, we allow certain constraints to be violated. Strategies to select constraints are very simple (since so far there was no necessity to apply them). Repair code has to be inserted, in form of extra spill code or of splitting up MIs into a set of sequential MIs.

7.6.6. Concluding remarks of complexity For the case that the proposed strategies fail, due to unacceptable run-times and time outs, we provide standard techniques, in order to yield code for every source program. When considering modern compiler technology, compilers incorporate strategies which make usage of alternative optimization strategies (often in parallel), and select the one (or even a set) with the best results. In this context we think, that techniques which may fail in certain cases but normally lead to a strongly improved code quality, are not only permissible, but also constitute a real enrichment for a compilation system.

8. Post-processing and results

The final phase in our code generator is the generation of executable code by two post-processing phases (cf. fig. 8):

1. Insertion of code needed for memory address computations: After code generation, values are bound to specific memories, but not yet to fixed addresses. Due to the specialized architecture of address generation units (AGUs) in DSPs, assignment of variables to memory addresses must be performed carefully. The goal in address assignment is to maximize the utilization of auto-increment and auto-decrement AGU operations for address computations.

2. Post-compaction: After AGU operations have been inserted into the code generated for DFG computations, a post-compaction step is required to actually parallelize DFG computations and AGU operations, so as to obtain the final machine code. Simultaneously, the post-compaction phase exploits the code selection freedom that is left in the generated factorised machine instructions. Each alternative instruction is associated with a binary encoding (opcode). From all alternative opcodes, post-compaction selects those ones which permit an optimum parallelization with generated AGU operations.

For address generation and post-compaction, we currently reuse techniques developed for the RECORD compiler [38, 39].

Table 6 shows experimental results for our set of benchmarks. Each entry gives a number of generated machine instructions including address computations. Column 1 shows results obtained with a GNU-based ADSP-210x C-Compiler. Column 2 lists results obtained with the RECORD compiler [37], and column 3 gives the length of the hand-written reference code for the DSPStone benchmarks. Finally, the results of our code generator ICG are shown in column 4.

Table 6. Comparison of generated code for benchmarks

source	GNU	RECORD	hand-written code	ICG
complex multiply	16	11	6	6
complex update	23	15	9	9
iir filter	33	21	12	12
lattice filter	-	30	-	18

In none of the compared examples optimizations like e.g. loop unrolling or software pipelining have been used, hence larger code size cannot lead to higher performance. Thus, smaller code sizes also indicate a higher performance. The GNU compiler showed an overhead of 170 % on the average compared to the hand-written code, while the code generated by RECORD showed a 70 % average overhead. An analysis of the code proved that this overhead is actually due to too many data moves between registers and poor exploitation of potential parallelism.

As can be seen in columns 3 and 4, ICG achieved the same code quality as the hand-written code for the first three benchmarks in table 6. Due to the use of address optimization techniques [38], no additional overhead was induced for address computations. The overall compilation speed of our approach, comprising all phases, is in the order of 3-5 generated instructions per CPU second.

9. Conclusions and future work

The use of processors in embedded system design demands for high-level language compilers capable of generating very efficient machine code. Current compilers hardly meet this demand. One main reason for the poor performance of current

compilers for embedded processors are irregularities in processor architectures, i.e., the presence of special-purpose registers and FUs in combination with restricted instruction-level parallelism. This type of architecture is frequently found in fixed-point DSPs. As we have exemplified for a standard DSP, tight coupling of code generation phases is necessary to overcome this problem.

Using a phase-coupled approach to code generation requires higher compilation times than in general-purpose computing. However, the tight code size and speed constraints for embedded processors in most cases make higher compilation times acceptable.

In this paper we have proposed a novel constraint-driven approach to code generation for embedded DSPs. This approach has two significant advantages as compared to previous work mostly based on heuristics: Firstly, there is no need of designing specific optimization algorithms capable of obeying all constraints that arise due to irregularities and ILP. Machine code is not *constructed*, but it is obtained by a successive *reduction of the solution space* based on the constraints imposed by the target processor. An efficient and uniform constraint satisfaction approach implemented with constraint logic programming guarantees that no constraint is violated in the emitted code. Secondly, our approach achieves a tight coupling of the main code generation phases, i.e., code selection, register allocation, and instruction scheduling. As a result, it is possible to generate machine code of very high quality, which comes close to hand-written assembly code. This has been demonstrated for benchmarks and a standard DSP.

Further improvements of our approach concern more intelligent functions for selecting candidates for reduction and spilling. This can be achieved by consideration of larger search spaces, e.g., the global interference graph augmented with information about parallelism. We will also replace the post-processing used in the current version by integrated techniques for generation of executable code.

While our approach already achieves a tight coupling of phases, at some points heuristics are still used to make decisions. We plan to eliminate these heuristics step by step, and to replace them by the labeling and optimization strategies of our CLP system. This will allow to study up to which problem complexity globally optimal code generation for a DFG will be possible within a reasonable amount of computation time. Furthermore, application of algebraic rules during code generation is another potential area for improvements.

Notes

1. The numbering of types corresponds to the ADSP-210x User's Manual [16].
2. A very good summary of first approaches using grammars and attributed grammars for specifying code selectors can be found in [25].
3. The notion *tree reduction rules* reflects a view of the tree covering process as the reduction of a certain input tree to a certain nonterminal, by using the rules of the tree grammar as rewriting rules. A pattern detected in the input tree is replaced (or substituted) by the nonterminal on the left hand side of the corresponding rule.
4. Since labeling may consume exponential run-times if no solution exists, we can specify timeouts for the feasibility checks.

5. This holds, since the compared code examples did not include software pipelining or loop unrolling optimizations.
6. Our instruction types differ slightly from the ones introduced in [58].
7. In chained operations the order of the U_i 's is given by their occurrence in the expression (left to right).
8. Chained patterns are split into a set of FRTs as follows (cf. the tree automaton construction in [23]): For each sub-pattern a new virtual resource is inserted and the FRTs which match the sub-operations are extended with the corresponding virtual resource. E.g., the MAC operation is modeled by inserting the SR $'*(X,Y)'$ as the definition of operation $'*'$ and as a use of the operation $'+'$. If the FRT for $'*'$ reduces to the MAC operation, the costs are set to 0, and the corresponding operand of the FRT of $'+'$ is constrained to $'*(X,Y)'$; the correct costs are reflected by the FRT of $'+'$.
9. Tree grammars [8] are a special case of context free grammars.
10. Each rule in a tree grammar can be associated with a sequence of RTs given by the action part. In our approach, the sequence has to be restricted to a single RT.
11. This style was chosen for better understanding of our algorithms for readers not familiar with CLP. It also enables a conversion to other (non-CLP based) constraint systems.
12. We also consider RFs with capacity > 1 . Thus, an inequality constraint on locations is not sufficient; in our approach this constraint is extended to take into account indices of SRs. Thus we generate the constraint $(L_j \neq L_i) \vee ((L_j = L_i) \wedge (I_i \neq I_j))$, where I_i and I_j denote domain variables for the possible set of indexes for addressing the SRs given by L_i and L_j .
13. If there exists no feasible solution in a certain iteration (due to that constraints cannot be met), our strategy is, to allow the violation of certain constraints. Then, repair code has to be inserted in a post-processing phase. In the benchmarks we tested, this case didn't occur so far

References

1. Wolfgang Ambrosch, Anton Ertl, Felix Beer, and Andreas Krall. Dependence conscious register allocation. In Juergen Gutknecht, editor, *Programming Languages and System Architectures*, volume 782, pages 125–136. LNCS Series, Springer-Verlag, Zurich, Switzerland, March 1994.
2. Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.
3. Alfred V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, 1976.
4. Guido Araujo and Sharad Malik. Optimal code generation for embedded memory non-homogeneous register architectures. In *Intl. Symp. on System Synthesis ISSS'95*, 1995.
5. Guido Araujo, Sharad Malik, and Mike Tien-Chien Lee. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *33rd Design Automation Conference (DAC)*. 1996.
6. Alfred V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, New York, 1986.
7. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.
8. A. Balachandran, D.M. Dhamdere, and S. Biswas. Efficient Retargetable Code Generation Using Bottom-Up Tree Pattern Matching. *Comput. Lang. vol. 15, no. 3*, 1990.
9. David G. Bradley, Susan J. Eggers, and Robert R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, Santa Clara, California, 1991.
10. David A. Berson, Rajiv Gupta, and Mary Lou Soffa. Resource spackling: A framework for integrating register allocation in local and global schedulers. *Working Conf. on Parallel Architectures and Compilation Techniques*, August 1994.

11. David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. *SIGPLAN Notices*, 26(6):229–240, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*.
12. David G. Bradlee. Retargetable instruction scheduling for pipelined processors. PhD Thesis 91-08-07, Dept. of Computer Science, Univ. of Washington, 1991.
13. Thomas S. Brasier and Phillip H. Sweany. Craig: A practical framework for combining instruction scheduling and register assignment. In *PACT'95*, Limassol, Cyprus, 1995.
14. G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markenstein. Register allocation via coloring. *Computer Languages*, 6(1):47–57, January 1981.
15. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and Kenneth F. Zadeck. Efficiently computing the static single assignment and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
16. Analog Devices. *ADSP-2101 User's Manual*. Analog Devices, 1991.
17. J.R. Ellis. *Bulldog: A compiler for vliw architectures*. The MIT Press, Cambridge, Mass., 1986.
18. Helmut Emmelmann, Friedrich-Wilhelm Schröer, and Rudolf Landwehr. BEG – A generator for efficient back ends. *SIGPLAN Notices*, 24(7):227–237, July 1989. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*.
19. Andreas Fauth, G. Hommel, A. Knoll, and C Mueller. Global code selection for directed acyclic graphs. In Peter A. Fritzon, editor, *Compiler Construction*, volume 786 of *LNCS*, pages 128–141. Springer-Verlag, Eddinburgh, U.K., April 1994. 5'th International Conference, CC'94.
20. C. Fraser, R. Henry, and Todd A. Proebsting. Engineering a Simple, Efficient Code-Generator Generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
21. C. Fraser, R. Henry, and Todd A. Proebsting. BURG – fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992.
22. Stefan M. Freudenberger and John C. Ruttenberg. “Phase Ordering of Register Allocation and Instruction Scheduling”. In Robert Giegerich and Susan L. Graham, editors, “Code Generation — Concepts, Tools, Techniques”, *Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany, 20-24 May 1991*, Workshops in Computing, pages 146–172. Springer-Verlag, 1991. ISBN 3-540-19757-5 and 3-387-19757-5.
23. Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree automata for code selection. *Acta Informatica, Springer-Verlag*, pages 741–760, 1994.
24. C.H. Gebotys. An Efficient Model for DSP Code Generation: Performance, Code Size, Estimated Energy. In *10th International Symposium on System Synthesis (ISSS)*. 1997.
25. Mahadevan Ganapathi, C.N. Fisher, and J.L. Hennessy. Retargetable compiler code generation. *Computing Surveys*, 14(4), October 1982.
26. J. Goodman and W. Hsu. Code scheduling and register allocation. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 1988.
27. S.L. Graham and R.S. Glanville. A new method for compiler code generation. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, 1977.
28. R. Hartmann. Combined scheduling and data routing for programmable asic systems. In *Proceedings of EDAC'92*, pages 486–490, March 1992.
29. Werner Heinrich. *Formal Description of Parallel Computer Architectures as a Basis of Optimizing Code Generation*. PhD thesis, TU Munich, 1993.
30. R. Henry. Algorithms for Table-Driven Code Generators Using Tree Pattern Matching. Technical Report 89-02-03, Computer Science Department, University of Washington, Seattle, WA 98195 USA, 1989.
31. R. Henry. Encoding Optimal Pattern Selection in a Table-Driven Bottom-Up Tree Pattern Matcher. Technical Report 89-02-04, Computer Science Department, University of Washington, Seattle, WA 98195 USA, 1989.
32. R. Henry. Performance of Table-Driven Code Generators Using Tree Pattern Matching. Technical Report 89-02-02, Computer Science Department, University of Washington, Seattle, WA 98195 USA, 1989.

33. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.
34. Silvina Hanono, George Hadjiyiannis, and Srinivas Devadas. Aviv: A Retargetable Code Generator Using ISDL. In *Proc. 34th DAC'97*, 1997.
35. Dirk Lanner, Marco Cornero, Gert Goossens, and Hugo De Man. Data routing: a paradigm for efficient data-path synthesis and code generation. In *Proc. 7th IEEE/ACM Int. Symp. on High-Level Synthesis*, May 1994.
36. Stan Liao, Srinivas Devadas, Kurt Kreuzer, and Steve Tjiang. Instruction Selection Using Binate Covering for Code Size Optimization. *International Conference on CAD (ICCAD)*, 1995.
37. Rainer Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, 1997.
38. R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. *ICCAD*, 1996.
39. R. Leupers and P. Marwedel. Time-Constrained Code Compaction for DSPs. *IEEE Transactions on VLSI Systems*, vol. 5, no. 1, 1997.
40. R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. In *Design Automation for Embedded Systems*, vol. 3, no. 1, 1998.
41. S. Moon and K. Ebcioglu. An efficient resource constraint global scheduling technique for superscalar and vliw processors. In *MICRO*, December 1992.
42. Peter Marwedel and Gert Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.
43. Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.
44. Bart Mesmann, Adwin H. Timmer, Jef L. van Meerbergen, and G. Jess Jochen A. Constraint Analysis for DSP Code Generation. In *Proc ISSS'97*, 1997.
45. Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
46. Steven Novack and Alexandru Nicolau. Trailblazing: A hierarchical approach to percolation scheduling. Technical Report TR-92-56, Irvine University, August 1993.
47. Steven Novack and Alexandru Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 892 of LNCS, pages 16-30. Springer-Verlag, Ithaca, NY, USA, August 1994.
48. Cindy Norris and L. Pollok. A scheduler-sensitive global register allocator. In *Proceedings of Supercomputing'93*, 1993.
49. Cindy Norris and L. Pollok. Register allocation over the program dependence graph. *SIGPLAN Notices*, 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
50. Cindy Norris and L. Pollok. Register allocation sensitive region scheduling. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'95)*, 1995.
51. Alexandru Nicolau, R. Potasman, and H. Wang. Register allocation, renaming and their impact on parallelization. In *Languages and Compilers for Parallel Computing*, volume 589. LNCS Series, Springer-Verlag, 1991.
52. IC Parc. Homepage. <http://www.icparc.ic.ac.uk/eclipse/>.
53. P. Paulin, M. Cornero, and C. Liem. Trends in Embedded Systems Technology, in: M.G. Sami, G. De Micheli (eds.): *Hardware/Software Codesign, An Industrial Perspective*. Kluwer Academic Publishers, 1996.
54. S.S Pinter. Register allocation with instruction scheduling. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 248-257, 1993.
55. Eduardo Pelegri-Llopert and Susan L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294-308, San Diego, California, January 1988.
56. P. Paulin, C. Liem, T May, and S. Sutarwala. Flexware: A Flexible Firmware Development Environment for Embedded Systems. In Marwedel and Goossens [42], chapter 4, pages 65-84.

57. K. Rimey and P.N. Hilfinger. Lazy Data Routing and Greedy Scheduling. In *MICRO*, volume 21, pages 111–115. 1988.
58. Marino T.J. Strik, Adwin H. Timmer Jef van Meerbergen, Jochen A.G. Jess, and Stefan Note. Efficient Code Generation for In-House DSP Cores. In *Proc. ED&TC'95*, 1995.
59. Adwin H. Timmer, Marino T.J. Strik, Jef L. van Meerbergen, and Jochen A.G. Jess. Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores. In *Proc. of 32nd DAC*, 1995.
60. S.R. Vegdahl. *Local code generation and compaction in optimizing compilers*. PhD thesis, Carnegie-Mellon University, Pittsburgh, 1982.
61. Mark Wallace. Constraint Programming. Contact address: IC-Parc, William Penney Laboratory, Imperial College, London SW7 2AZ, email:mgw@doc.ic.ac.uk, sep 1995. Publications at <http://www.icparc.ic.ac.uk/>.
62. T. Wilson, G. Grewal, S Henshall, and D Banerjii. An ILP-Based Approach to Code Generation. In Marwedel and Goossens [42], chapter 6, pages 103–118.
63. Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLⁱPS^e: A Platform for Constraint Logic Programming. Contact address: IC-Parc, William Penney Laboratory, Imperial College, London SW7 2AZ, email:mgw@doc.ic.ac.uk, aug 1997. Publications at <http://www.icparc.ic.ac.uk/>.
64. Masyuki Yamaguchi, Nagisa Ishiura, and Takashi Kambe. Binding and Scheduling Algorithms for Highly Retargetable Compilation. In *Proc. ASP-DAC'98*, 1998.
65. V. Zivojnovic, J.M. Velarde, C. Schlaeger, and H. Meyr. DSPStone - A DSP oriented Benchmarking Methodology. In *ICSPAT*. 1994.