Constraint Analysis for Code Generation: Basic Techniques and Applications in FACTS

KOEN VAN EIJK,[†] BART MESMAN,[†‡§] CARLOS A. ALBA PINTO,[†] QIN ZHAO,[†] MARCO BEKOOIJ,[†] JEF VAN MEERBERGEN,[†‡§] and JOCHEN JESS[†§]

Code generation methods for digital signal processors are increasingly hampered by the combination of tight timing constraints imposed by signal processing applications and resource constraints implied by the processor architecture. In particular, limited resource availability (e.g., registers) poses a problem for traditional methods that perform code generation in separate stages (e.g., scheduling followed by register binding). This separation often results in suboptimality (or even infeasibility) of the generated solutions because it ignores the problem of phase coupling (e.g., since value lifetimes are a result of scheduling, scheduling affects the solution space for register binding). As a result, traditional methods need an increasing amount of help from the programmer (or designer) to arrive at a feasible solution. Because this requires an excessive amount of design time and extensive knowledge of the processor architecture, there is a need for automated techniques that can cope with the different kinds of constraints *during* scheduling. By exploiting these constraints to prune the schedule search space, the scheduler is often prevented from making a decision that inevitably violates one or more constraints. FACTS is a research tool developed for this purpose. In this paper we will elucidate the philosophy and concepts of FACTS and demonstrate them on a number of examples.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation/compilers/optimization*; B.5.1 [**Register-Transfer-Level Implementation**]: Design—*Memory design*; C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures—*Parallel processors*

General Terms: Algorithms, Design

Additional Key Words and Phrases: DSP, constraint analysis, scheduling, phase coupling, foreground memory, register binding

1. INTRODUCTION

The exponential growth in the number of gates that can be integrated on a single chip has made the subject of embedded systems the central focus of many design and research groups. Next to commonly used microprocessors such as MIPS and ARM, embedded digital signal processors (DSPs) comprise the performance backbone for application domains such as communication and multimedia. About five years ago, application domain specific instruction set

Authors' addresses: [†]Department of Electrical Engineering, Eindhoven University of Technology, P.O. Box 513, NL-5600MB Eindhoven, The Netherlands; [‡]Philips Research Laboratories, Prof. Holstlaan 4, NL-5656AA Eindhoven, The Netherlands; [§]Eindhoven Embedded Systems Institute, Eindhoven University of Technology.

processors (ASIPs) [Leupers et al. 1994] were indicated as suitable architectures for satisfying the demands of embedded DSPs [Paulin et al. 1995], such as small code size, high performance, and low power dissipation. These processors often contain highly irregular datapaths and relatively few registers, resulting in a strong phase coupling (e.g., the register addresses allowed for reading operands may depend on the specific function executed). Because compiler retargetability [Leupers 1997] is considered an essential feature in the context of ASIPs, processor-specific optimizations are ideally not part of ASIP compilers. Some general-purpose tecchniques have been exploited to capture the strong phase coupling resulting from datapath irregularities [Kästner and Langenbach 1999]. These CLP [Kuchcinski 1997] or ILP formulations tend to grow large when applied to the *whole* problem, thereby inducing unacceptable overhead in schedule length and code size [Paulin and Liem 1996]. As a result, most programmers prefer to write assembly code in order to exploit the efficiency of ASIP architectures. This requires extensive knowledge of the processor architecture as well as the instruction set and can be very time-consuming.

Lately, a trend can be observed toward more orthogonal instruction sets in digital signal processors to ease high-level compilation, portability of software, and retargetability and maintenance of compiler software [TI 1997; TM 1997]. In particular, VLIW architectures have been acclaimed for the orthogonality of the associated instruction sets. The first cracks in this trend toward regularity have already appeared, however: For reasons of synthesizability, clock speed, and power, some DSP companies *partition* the (ideally) single register file in a number of files [Faraboschi et al. 1998] with a *limited* number of registers. This potentially increases local register pressure and introduces additional communication delay between *clusters* of functional units [Faraboschi et al. 1998], so it creates additional constraints and irregularities that the compiler has to deal with.

Probably the most significant obstacle for large-scale embedding of VLIW processors is the problem of code size, which creates a severe cost factor for on-chip program memory/cache and a large contribution to power consumption in handheld devices. It can therefore be expected that for a large application domain, interest in power and cost-efficient (but irregular) DSP architectures will remain, and possibly increase. Compiler research in this area is essential and should focus, among others, on supporting the combination of *constraints* arising from architectural features and the application domain, and take *phase coupling* into account in order to arrive at feasible and efficient solutions [Bashford and Leupers 1999; Mesman et al. 1999; Rau et al. 1998]. FACTS is a research tool for code generation based on these observations.

The rest of the paper is organized as follows. Section 2 gives an overview of FACTS and demonstrates the basic philosophy with an example. In Section 3 our model and search space representation are explained. Some basic constraint analysis techniques are treated in Section 4. In Section 5 we demonstrate how these basic techniques are used in a search strategy to satisfy more global constraints or to optimize with respect to some criterion. Experimental results are given in Section 6. Section 7 summarizes the main ideas presented in this paper and concludes with a short discussion of future work.

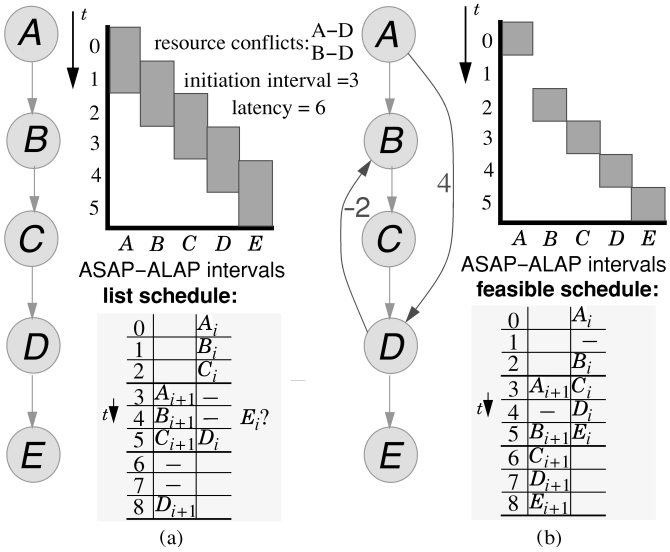

Fig. 1. A simple example to illustrate constraint analysis.

2. AN INTRODUCTION TO FACTS

In this section we first give an example of how constraint analysis can prune the schedule search space to prevent a scheduler from making decisions that lead to constraint violation. This is followed by an overview of the structure of FACTS and the techniques implemented in it.

2.1 A Pruning Example

Because scheduling under resource and timing constraints is an NP-hard problem, heuristics are needed to solve practical problem instances in a reasonable time. Most existing schedule heuristics are, however, easily "deceived" by the schedule freedom apparantly available in a problem instance and are therefore too often unable to generate a feasible schedule. Driven primarily by precedences and taking resource constraints into account on a clock cycle basis, commonly used heuristics such as list scheduling lack the scope to deliberately postpone operations in order to arrive at a feasible schedule.

We use a small example to illustrate the difficulty of handling the combination of different types of constraints. In Figure 1(a), a data flow graph with five operations is given. The ASAP–ALAP intervals are graphically depicted for each operation. Each such interval defines the as-soon-as-possible (ASAP) and the as-late-as-possible (ALAP) start time of the associated operation. Because calculating the exact intervals is again a NP-hard problem, conservative estimations of these intervals are used. A well-known approach is to perform a topological sorting of the data flow graph to calculate the maximum path delay from or to an operation. Note that this approach completely ignores how many resources are available. For our example, this results in the intervals shown in Figure 1(a).

The timing constraints state that each execution of the data-flow graph should take at most six clock cycles (i.e., the latency is 6) and that every

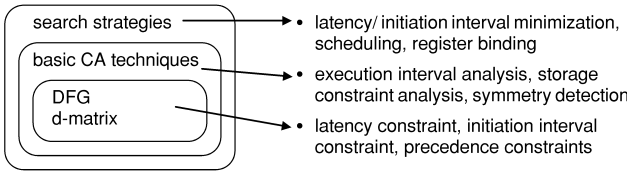

Fig. 2. The layered structure of FACTS.

three clock cycles, a new execution has to start (i.e., the initiation interval is 3). Note that this means that two consecutive executions overlap in time and that therefore a pipelined schedule has to be constructed. A list scheduler greedily schedules A, B, and C as soon as possible at clock cycles 0, 1, and 2 respectively. Since clock cycle 3 coincides with clock cycle 0, the resource conflict A–D prohibits operation D from being scheduled at clock cycle 3. Similarly, the resource conflict B–D prohibits operation D from being scheduled at clock cycle 4. As a result, D is scheduled at clock cycle 5, and consequently, there is no room left to schedule operation E. In Section 4.1 we will show how resource constraint analysis is able to identify two additional sequence constraints (given in Figure 1b) that are implied by the combination of the resource constraints and the given timing constraints. The reader can verify that no feasible solutions are excluded by these additional sequence constraints. Recalculating the ASAP–ALAP intervals reveals that there is actually no schedule freedom at all: The schedule shown in Figure 1(b) is the only feasible schedule.

2.2 FACTS Overview

The structure of FACTS consists of three layers, as depicted in Figure 2. The core layer contains the internal representations of the algorithm to be scheduled and of the schedule search space. At the intermediate layer, the basic constraint analysis techniques are provided. On top of that, search strategies are implemented.

2.2.1 *DFG and the Distance Matrix.* At the core layer of FACTS, each basic block of the algorithm to be scheduled is represented by a data-flow graph (DFG). In addition, the schedule search space is represented by a distance matrix. This distance matrix administrates the minimum and maximum difference between the start times of each pair of operations in a DFG.

The results of the constraint analysis techniques in FACTS are conceptually expressed as additional sequence constraints in the data-flow graph as in Figure 1(b). The effects of these results on the schedule search space are computed by updating the distance matrix: incrementing the minimum distance or decrementing the maximum distance between two operations. In Figure 1(b) for example, the sequence edge $A \rightarrow D$ with delay 4 indicates that operation D has to start execution at least 4 clock cycles after the start of operation A. The sequence edge $D \rightarrow B$ with delay -2 indicates that operation B cannot start execution more than 2 clock cycles before the start of operation D. Note that the combination of these two sequence constraints implies that operation B cannot start earlier than 2 clock cycles after operation A, as can be verified in

the schedule in Figure 1(b). The effect of combining two such analysis results is visible in the distance matrix by computing the *longest paths* [Cormen et al. 1990] induced by the individual sequence edges. The path $A \rightarrow D \rightarrow B$ with delay $4 + (-2) = 2$ clock cycles implies a minimal distance of 2 clock cycles between operations A and B. This feature of being able to combine the different analysis results simply by computing the longest paths between each pair of operations, is one of the main motivations for choosing the distance matrix to represent the schedule search space.

The processes that take place in FACTS at the core level can be summarized as follows:

- *Timing constraints* are *expressed* directly in the distance matrix.
- *Analysis results* are *integrated* into the distance matrix.
- *Precedence constraints* (possibly resulting from analysis) are *combined* in the distance matrix, such that all implied precedence constraints are also derived.

2.2.2 *Basic Constraint Analysis Techniques.* At the intermediate layer, the basic constraint analysis techniques are located. Most of these techniques essentially consist of *rules* triggered by the combination of one or more pairwise distances (entries in the distance matrix) and other (often architectural) constraints.

For example, in Figure 1(b), the sequence edge $A \rightarrow D$ with delay 4 is due to the following observation: In Figure 1(a), a minimum distance of three clock cycles from operation A to D is implied by the path of precedences $A \rightarrow B \rightarrow C \rightarrow D$. Suppose that operation D would be scheduled at exactly this minimum distance from A. Because the initiation interval is 3, when operation D executes, the next iteration of operation A also executes, thus violating the resource constraint A–D. As a result of the initiation interval and the resource conflict A–D, the minimum distance $A \rightarrow D$ of three clock cycles is not feasible, and, therefore, the distance $A \rightarrow D$ should be at least four clock cycles.

The essence of constraint analysis is that additional sequence edges are added that are *necessarily implied by the combination of other constraints*. In this way, the schedule search space is pruned to prevent the scheduler from finding infeasible solutions, without eliminating feasible solutions. This demand of necessarily implied sequence constraints can be relaxed, however, as is the case with symmetry analysis. That analysis technique may eliminate *equivalent* feasible solutions, provided that at least one such equivalent solution remains (see Section 4.3).

2.2.3 *Search Strategies.* At the top level of FACTS, we deal with search strategies. The objective may be to minimize some criterion such as latency or initiation interval, or to satisfy more global constraints such as a fixed capacity of a register file. These constraints are global in the sense that they have a general effect on *all* timing relations, without affecting any *specific* timing relation. For example, a constraint on the capacity of a register file limits the number of values simultaneously alive, which is determined by the timing relations. However, no single value lifetime can be identified that *necessarily* has to be serialized with some other value lifetime. It is clear that *some* value lifetimes need to be serialized in order to satisfy the capacity constraints, and therefore some *decisions* have to be made regarding the serialization of values. In

our search strategies we try to base choices on the identification of potential *bottlenecks* for satisfying the corresponding constraint. In the register file example, a potential bottleneck is identified using the *worst-case* lifetime overlap, detected by coloring a worst-case conflict graph (see Section 5.2).

The following sections discuss the three layers of FACTS in more detail and give examples of modeling and analysis techniques at each layer.

3. THE DFG MODEL AND THE DISTANCE MATRIX

In this section we will introduce the basic concepts of the core FACTS layer: the DFG model, the distance matrix, and the way to express some relevant timing constraints in the DFG and the distance matrix.

3.1 The DFG Model

An algorithmic description can be partitioned into basic blocks. Each basic block can be modeled by a data flow graph (DFG), which describes the primitive operations performed in that block, and the dependencies between those operations.

Definition 1 (Data Flow Graph). A data flow graph DFG is a triple $(V, E_d \cup E_s, w)$, where

- V is the set of vertices (operations),
- $E_d \subseteq V \times V$ is the set of data precedence edges,
- $E_s \subseteq V \times V$ is the set of sequence precedence edges, and
- $w: E_d \cup E_s \rightarrow \mathcal{Z}$ is a function describing the timing delay (in clock cycles) associated with each precedence edge.

Two (dummy) operations are always assumed to be present in the DFG: the source and the sink. They have no execution delay and represent respectively the first and the last operation to be executed. We will usually not show the source and the sink nodes when depicting a DFG.

For reasons of simplicity, we will assume in this paper that all operations have an execution delay of 1 clock cycle. Theoretically, this is not a significant restriction, because multicycle operations can be modeled using precedence constraints [Mesman et al. 1999]. In the actual implementation of FACTS, any nonnegative integer delay value is supported directly.

The task of scheduling is to assign each operation $v \in V$ a start time $s(v)$. The number of available cycles is defined by the *latency* L. In our work we also consider *pipelined* schedules [Lam 1988] that execute periodically with a period called the *initiation interval* II. The start times are constrained by the precedences. A precedence edge $(v_i, v_j) \in E_d \cup E_s$ states that

$$s(v_j) \geq s(v_i) + w(v_i, v_j) \tag{1}$$

A chain of precedence edges $v_i \rightarrow v_k \rightarrow \cdots \rightarrow v_j$ with total added weight w_{path} is called a *path*, implying that $s(v_j) \geq s(v_i) + w_{path}$.

Definition 2 (Distance). The distance $d(v_i, v_j)$ is the length of the longest path from v_i to v_j.

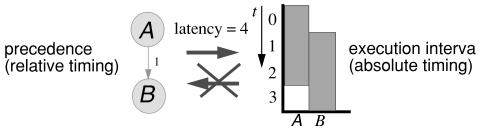

Fig. 3. The interval (or set) representation does not accurately represent relative timing.

A path in the DFG thus represents a minimum timing delay between two operations. These distances are stored in a *distance matrix*, which can be calculated using a all-pairs longest-path algorithm [Cormen et al. 1990].

3.2 Representation of the Schedule Search Space

FACTS uses the distance matrix as a representation of the schedule search space. This is not an obvious choice, since in the context of constraint satisfaction, other representations of the schedule search space are much more common. These representations are focussed on representing the schedule freedom for each *individual* operation. That is, for each operation either a set [Nuijten 1994] or an interval [Timmer and Jess 1993] is kept containing the *absolute* clock cycle numbers in which the corresponding operation can be scheduled, such as the ASAP–ALAP intervals in Figure 1. We feel that the distance matrix fits our purposes better for the following two reasons.

- The distance matrix administrates *relative* timing (order). Practically all code generation constraints have an implication on the *ordering* of operations rather than on their absolute start times (e.g., reducing register pressure can be achieved by serializing value lifetimes). Obviously, this is an ordering issue. By focusing on relative ordering information we are better able to exploit the characteristics of the scheduling problem and of the constraints relevant for code generation.
- The distance matrix is strictly more accurate than an interval representation. Any interval can be represented in terms of sequence edges. For an interval [lb; ub] of an operation A, this requires two sequence edges: A precedence source $\rightarrow A$ with weight lb and a precedence $A \rightarrow$ source with weight $-$ub. The example in Figure 3 shows that the reverse is not true: The information expressed in the distance matrix cannot be accurately expressed in terms of intervals, so the distance matrix is strictly more accurate.

An obvious drawback of maintaining a distance matrix is that it is computationally more expensive than an interval representation. However, in Section 6 we will provide experimental evidence that this overhead is quite acceptable in practice, even for larger DFGs containing hundreds of operations. Note that adding an extra sequence edge does *not* require recalculation of the entire distance matrix, but rather an incremental update.

3.3 Expressing Timing Constraints

As indicated in Figure 2, some timing constraints can directly be expressed at the level of the DFG or the distance matrix. Precedence constraints are an obvious example; a precedence $A \rightarrow B$ implies that $d(A, B) \geq w(A, B)$. Other timing constraints that can be expressed are related to the latency and the initiation interval:

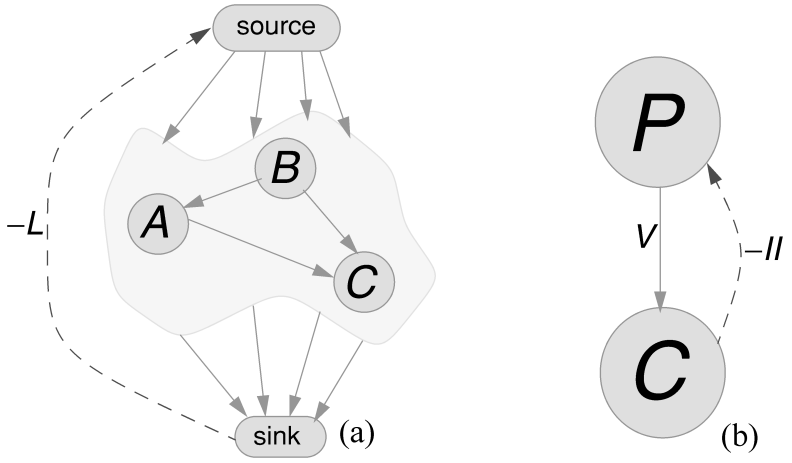

Fig. 4. Modeling a constraint on (a) the latency and (b) the initiation interval.

3.3.1 *Latency.* A constraint on the latency L can be expressed in the DFG by a sequence edge from the sink to the source with weight $-L$, as illustrated in Figure 4(a). According to inequality 1, this is interpreted as $s(\text{source}) \geq s(\text{sink}) - L$, which is equivalent to $s(\text{sink}) \leq s(\text{source}) + L$. Because the source is always scheduled in clock cycle 0, this formula expresses that the sink should be scheduled in clock cycle L or earlier. Because all other operations precede the sink, this implies that all operations have to finish their execution within the first L clock cyles.

3.3.2 *Initiation Interval.* We assume that the processor architecture contains a microcoded controller. As a consequence, the same code is executed every loop iteration. This implies that in each iteration, a value is written to the same register. Therefore, if loop iterations overlap, we have to ensure that a value belonging to the current iteration is consumed before it is overwritten by the production of that same value in the next iteration. Since consecutive productions are exactly II clock cycles apart, this means that a value cannot be alive longer than II clock cycles. So the operation C that consumes the value should execute within II clock cycles after the operation P that produces the value. Just like the latency constraint, a necessary and sufficient translation to the precedence model is that for each data precedence edge $(P, C) \in E_d$, there is an edge $C \rightarrow P$ with weight $-II$, as illustrated in Figure 4(b).

3.4 Expressing Resource Constraints

Resource conflicts are modeled in FACTS by introducing the concept of *resources* and by defining the *resource usage* of each operation. For this purpose, we associate an *operation type* with each operation. Let the set T_O be the set of operation types. Then the function $\tau: V \rightarrow T_O$ defines the operation type of each operation. In our model, each operation type is associated with a resource type, which is characterized by a delay value, a data introduction value (to support pipelined resources), and the number of instances available of that resource type.

Although our modeling of resource constraints is rather basic, it is interesting to observe that it can still handle many practical issues: Resources can also be used to model more abstract constraints, such as those arising from a fixed instruction set. Consider the case that no instruction exists for the parallel execution of operations v_i and v_j, so that the parallel execution of v_i and v_j should be prohibited. This can be accomplished by generating an *artificial* resource [Timmer et al. 1994] that is used by both v_i and v_j. A more general constraint arises from the availability of a limited set of *issue slots* [TI 1997; TM 1997] to control the data path of a processor. Both Braspenning [1999] and Eisenbeis et al. [1999] describe a method to completely replace issue slot constraints by artificial resources.

4. BASIC CONSTRAINT ANALYSIS TECHNIQUES

This section discusses some of the basic constraint analysis techniques in FACTS. In Sections 4.1 and 4.2, on resource and storage constraints respectively, sequence edges are added to prune the schedule search space without eliminating feasible solutions. In Section 4.3, symmetry analysis is discussed, which may eliminate feasible solutions, provided that at least one other "equivalent" feasible solution remains in the search space.

4.1 Execution Interval Analysis

The method we introduce in this section to analyse resource constraints is based on examining the intervals in which operations can be executed [Timmer and Jess 1993; Timmer 1995]. The method focuses on the availability of resources; it reduces the execution interval of an operation when it observes that no resource is available for executing that operation in the corresponding clock cycle. The process is illustrated in Figure 5.

In Figure 5(a), the DFG is given. The latency is 5 clock cycles, and one resource of type "add" is available for executing the operations. The initial execution intervals are determined by the ASAP–ALAP intervals as depicted in Figure 5. The reader can already verify that the timing and resource constraints restrict the execution interval of operation C to [2; 2], whereas the ASAP–ALAP interval is [1; 3].

From the initial execution intervals and the available resources (hardware modules), *module execution intervals* (MEIs) are calculated (right hand side of Figure 5c). Each MEI represents the rather abstract notion that some resource has to execute an operation. In the example, the constraints imply that the available adder has to start executing a new operation every clock cycle.

Execution interval analysis combines the execution intervals of the operations with the resource constraints by constructing a *bipartite schedule graph* in the following way. The operations and their corresponding operation execution intervals (OEIs) are shown on the left hand side. The module execution intervals are shown on the right hand side. Note that they are always ordered on their start and end times. There is an edge between an OEI and a MEI if the intervals overlap, indicating that the corresponding operation can be executed in the designated MEI. In addition, it is required that all preceding

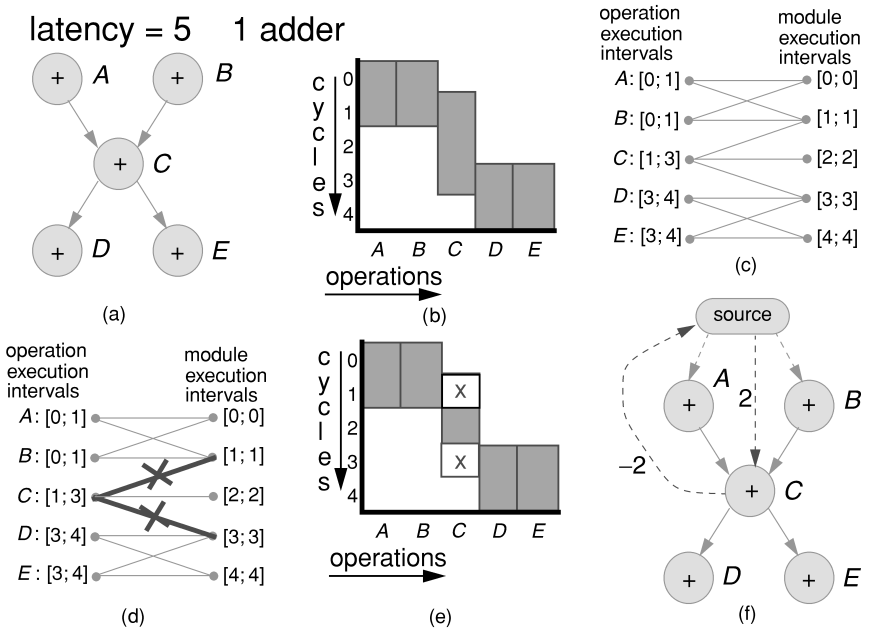

Fig. 5. Execution interval analysis: (a) DFG; (b) execution intervals; (c) bipartite schedule graph. (d) Edges that can never be part of a complete matching are eliminated. (e) Execution intervals are adjusted accordingly. (f) The analysis result is annotated in the DFG.

operations of the same operation type can be matched with preceding MEIs in the bipartite graph. A similar condition is imposed for all succeeding operations.

The key observation of the analysis is that for every feasible schedule, there exists a *complete matching* in the bipartite schedule graph between the OEIs and the MEIs. That is, every OEI is matched to exactly one MEI and vice versa. The analysis uses the algorithm of Sangiovanni-Vincentelli [1976] to identify edges in the bipartite graph that can never be part of a complete matching. The reader can verify that the bold crossed edges in Figure 5(d) are such edges. Because these edges can never be part of a matching corresponding to a feasible schedule, they are removed from the bipartite graph. As a result, operation C cannot execute in the MEIs [1; 1] and [3; 3]. The execution interval of operation C is therefore adjusted (Figure 5e). This adjustment corresponds to a pruning of the search space. The result of the analysis is annotated in the DFG (and the distance matrix) as indicated in Figure 5(f).

Execution interval analysis also contains an additional analysis: To determine the earliest possible start time of an operation, a relaxed scheduling problem involving all its predecessors is solved to determine a lower bound on the first clock cycle in which all the predecessors are completed. The scheduling problem is relaxed in the sense that the precedence constraints are essentially ignored; it is only enforced that each operation cannot start earlier than the lower bound on its start time. Similarly, all successors are analysed. The predecessors and successors are determined using the distance matrix: An operation v_i is said to precede an operation v_j iff $d(v_i, v_j) \geq 1$.

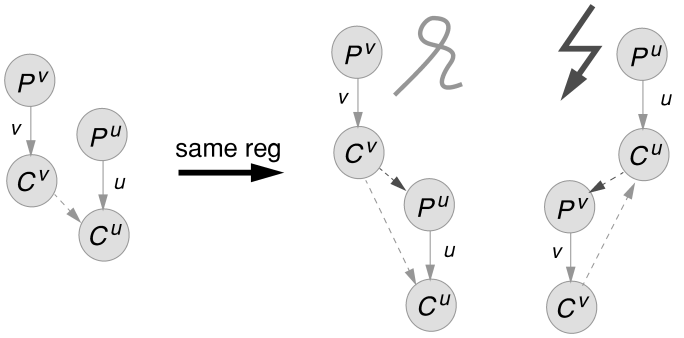

Fig. 6. To solve the register conflict, C^v has to precede P^u.

4.2 Storage Constraint Analysis

In this section we treat a relatively simply constraint arising from the decision to assign two values to the same register. This decision could be taken by a designer, by a search strategy for register binding (Section 5.2), or by necessity simply because there is, for example, only one flag register available to store a number of condition variables. Whenever two values are assigned to the same register, their corresponding lifetimes are forced to be serialized. In general this can be done in two ways (value u precedes value v or vice versa). Sometimes there already exists a precedence between the various accesses to u and v that excludes one of these possibilities. One such situation is illustrated in Figure 6.

In this situation, the sequence edge $C^v \rightarrow P^u$ is a constraint necessary and sufficient to solve the register conflict between u and v. Such situations are recognized by simple rules [Mesman et al. 1998] that examine the distance matrix for precedences.

4.3 Symmetry Detection

Many signal processing algorithms exhibit certain symmetries, for example because their structure is completely or partly regular. Practical examples include algorithms for performing a fast Fourier transform or a discrete cosine transform. These symmetries have a negative impact on the accuracy of constraint analysis techniques; these techniques are not capable of "breaking" the symmetries in order to decrease the apparent scheduling freedom because doing so would possibly remove feasible schedules. Therefore, we developed additional techniques for detecting and utilizing symmetry while preserving feasibility.

In general, the concept of symmetry is strongly related to the property of an object that it does not change under a certain transformation. The kind of transformation that is relevant for capturing symmetry in a DFG is a relabeling of the operations such that the operation types and the precedence edges are preserved. Such a transformation is called an automorphism.

Definition 3 (Automorphism). An automorphism is a bijective function α: $V \rightarrow V$ such that

- Each operation is mapped to an operation of the same type: $\forall v \in V: \tau(v) = \tau(\alpha(v))$.

- Each edge is mapped to an edge having the same weight: $\forall(v_i, v_j) \in E_d \cup E_s$: $(\alpha(v_i), \alpha(v_j)) \in E_d \cup E_s) \wedge (w(v_i, v_j) = w(\alpha(v_i), \alpha(v_j))$.

Given an automorphism, the following DFG transformation describes how an extra sequence edge can be added to break the symmetry.

FEASIBILITY PRESERVING TRANSFORMATION 1. *Given an operation v_i and an automorphism α that maps v_i to another operation, that is, $\alpha(v_i) = v_j$ with $v_i \neq v_j$. Introduce a sequence edge from v_i to v_j with weight zero.*

To illustrate this transformation, consider the DFG of Figure 5. Because there exists an automorphism that exchanges operations A and B, the transformation allows us to introduce a sequence edge with weight 0 from A to B. Similarly, it is allowed to put a sequence edge from D to E. In combination with the execution interval analysis, this associates a unique start time with each operation.

For the proof that Transformation 1 preserves feasibility, as well as for a more detailed discussion and additional techniques, we refer the interested reader to [Van Eijk et al. 1998].

5. BOTTLENECK IDENTIFICATION AND SEARCH STRATEGIES

At the top level of FACTS, we deal with search strategies. Instead of analyzing or adding constraints, at this level we actually try to find a solution to a code generation problem. It builds on the other two levels in two ways. First, decisions made by the search strategy affect the schedule search space. The techniques in the lower two levels of FACTS are responsible for pruning the search space according to such a decision. This will prevent the search strategy from making decisions that are inconsistent with the constraints and previously made decisions. Second, the constraint analysis techniques may provide information valuable for the identification of potential *bottlenecks*. In this section we discuss search strategies for the two main code generation problems: scheduling and register binding.

5.1 Scheduling

This section describes the scheduler used in FACTS. We emphasize that this scheduler is not designed to optimize some criterion, but rather to find a schedule that complies with the set of constraints. The interaction between the top-level search strategies and basic constraint analysis in our view ensures that consistency with the constraint set is maintained. That is, the scheduler makes a decision, and constraint analysis prunes the search space accordingly.

The scheduler combines a branch-and-bound approach with a scheduling strategy based on the bipartite graph model discussed in Section 4.1. The basic idea is to first fix the order of operations assigned to the same functional resource type by assigning operation execution intervals (OEIs) to module execution intervals (MEIs). This assignment does not immediately fix the start times of the operations but does already decide upon issues most relevant for finding a feasible schedule. This statement is theoretically justified by [Timmer 1995] for nonpipelined schedules that do not have to satisfy sequence constraints. For each MEI, all possible matchings to OEIs are tried until a feasible schedule is found. Only at the end of the scheduling process are the actual start times

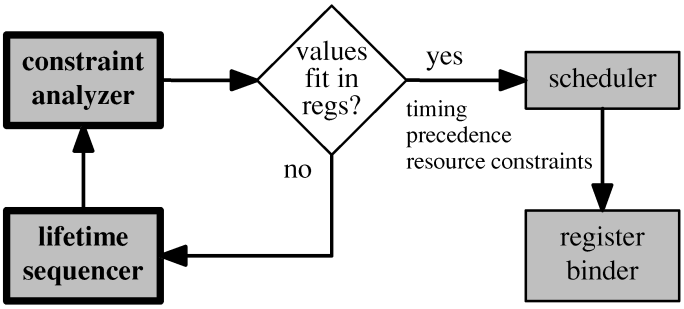

Fig. 7. Global approach for register binding.

chosen. After each scheduling decision, constraint analysis is run; whenever infeasibility is reported, backtracking is performed.

Whenever the bipartite schedule graph contains clear bottlenecks for finding a feasible schedule, those are solved first. In this context, a bottleneck is defined as a MEI of length 1 with only a small number of adjacent OEIs. If no clear bottlenecks exist, the scheduler uses heuristics similar to those for list scheduling. For more information the reader is referred to [Timmer 1995].

5.2 Register Binding

In this section we discuss our register binding approach. We assume that values are already bound to *register files* (in case of multiple register files). This binding usually follows from an assignment of operations to functional units. Furthermore, we assume that each register file has a fixed capacity that has to be respected during register binding. We intend to exploit the freedom available for scheduling to obtain a feasible register binding. The binding process is illustrated in Figure 7.

For each register file, an upper bound on the required number of registers can be computed based on the distance matrix; an exact figure is unknown because a complete schedule is not yet determined. When the upper bound for each register file already respects the file's capacity, the additional precedences are transferred to the scheduler and register binder for completion, as shown on the right hand side of Figure 7. However, in most cases and especially in the beginning of the process, the schedule freedom of the operations will be relatively large, resulting in many potential lifetime overlaps, thus inevitably violating some register file's capacity. In this case the maximum number of overlapping values has to be reduced by identifying one or more pair(s) of values that can be serialized. This value pair is selected in the *lifetime sequencer* in Figure 7. The constraint analysis techniques discussed in Section 4 subsequently calculate the effect of this serialization on the schedule freedom of all operations. This is necessary to prevent the lifetime sequencer (in subsequent iterations) from making serializations that are not possible. The process iterates until the capacity of each register file matches the worst-case requirements.

We illustrate the binding process with the example DFG given in Figure 8(a). If a greedy list scheduler is used, values a, b, d, and f end up with overlapping lifetimes, and the schedule requires four registers. It is, however, possible to

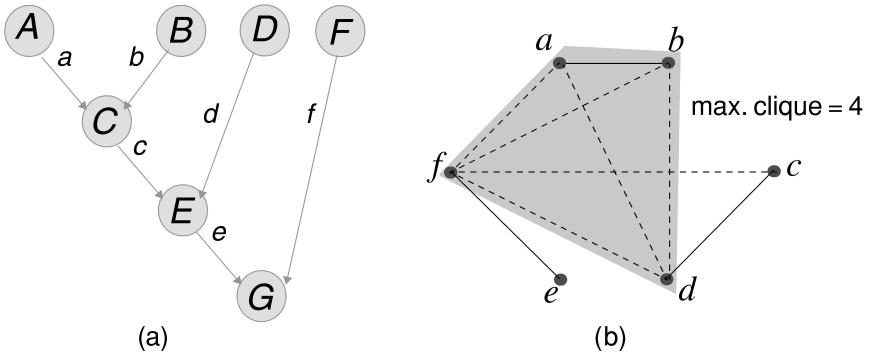

Fig. 8. (a) DFG (b) Conflict graph. Maximum clique identifies bottleneck.

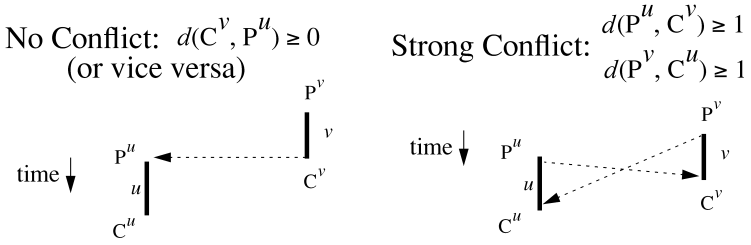

Fig. 9. Rules for determining strong and no conflicts in the conflict graph.

obtain a schedule requiring only two registers simply by postponing operations *D* and *F*.

To reduce the potential lifetime overlap, we have to detect potential conflicts between pairs of values before a complete schedule is known. The distance matrix is used to determine the "worst-case" and "best case" lifetime overlap between values. This is illustrated in Figure 9. In the left hand figure the condition is given for the case that value lifetimes never overlap. For example, values *a* and *e* have no conflict. In the right hand figure the condition is given for the case that value lifetimes surely overlap. We say that values *u* and *v* have a *strong* conflict. This is, for example, the case for values *a* and *b*. If neither condition holds, we say that *u* and *v* have a *weak* conflict, like values *b* and *d*. Values with a weak conflict can be serialized to reduce the register pressure. Strong and weak conflicts between values are annotated in a *conflict graph* as respectively, drawn and dashed edges between nodes representing values. The conflict graph for the DFG in Figure 8(a) is depicted in Figure 8(b).

By coloring the conflict graph with a minimum number of colors, bounds can be calculated on the required number of registers. By only considering the strong conflicts, a lower bound is obtained, while considering all conflicts results in an upper bound.

A potential bottleneck is identified by the largest clique (complete subgraph) in the conflict graph: Because every pair of values in this clique has a conflict, the number of registers required is at least as large as the size of this clique. Indeed, the maximum clique in Figure 8(b) consists of the potentially overlapping values *a*, *b*, *d*, and *f*. We have to reduce this potential bottleneck until the capacity of the register file is respected. We do this by eliminating conflicts

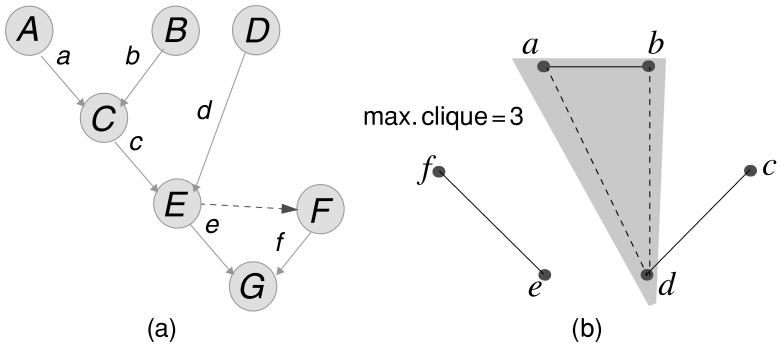

Fig. 10. (a) DFG after serializing $E \rightarrow F$ (b) Conflict graph.

in the largest clique. A weak conflict is selected between values with maximum *saturation numbers*; in a colored conflict graph, the saturation number of a node is defined as the number of different colors assigned to its neighbors. The details of this selection process are explained in Alba-Pinto et al. [1999] and Mesman et al. [1999]. The method selects values d and f to be serialized. The basic constraint analysis rule in Figure 6 implements this decision by serializing operations $E \rightarrow F$, as depicted in Figure 10(a). This reduces the potentially required number of registers to three, as shown in Figure 10(b). The new bottleneck is identified by the clique $\{a, b, d\}$. Either conflict a–d or b–d is selected for serialization, and the sequence edge $C \rightarrow D$ is added to obtain the final schedule requiring two registers.

6. EXPERIMENTAL RESULTS

In this section, we present some experimental results obtained with FACTS. The experiments are divided into five sets, each demonstrating a specific aspect. All experiments are run on a machine with a 233 MHz Pentium II processor and 64 MB of memory.

6.1 Constraint Analysis

The first experiment focuses on the accuracy of the constraint analysis techniques. This is evaluated using the fast discrete cosine transform example, which is well known from the high-level synthesis domain, under various sets of constraints. We use *mobility* as our metric for schedule freedom. The mobility is defined as the average difference between the as-late- as-possible (ALAP) start time and the as-soon-as-possible (ASAP) start time of the operations: $\frac{1}{|V|} \sum_{v_i \in V} \mathrm{ALAP}(v_i) - \mathrm{ASAP}(v_i)$.

The results are given in Table I. We compare three approaches: the mobility as calculated by a conventional topological sorting, the mobility as calculated by the basic constraint analysis techniques in FACTS, and the mobility as calculated by these techniques after running symmetry detection as a preprocessing step. The results clearly show the improvements obtained by carefully analyzing resource and timing constraints in combination.

Table I. Results of constraint analysis on example FDCT ($|V| = 42$, $|E_d| = 34$).

Constraints			Topological	Basic Analyses		+ Symm. Detect.	
mul	alu	L	Mobility	Mobility	Time (s)	Mobility	Time (s)
8	4	8	1.43	0.57	0.1	0.43	0.1
8	3	9	2.43	infeas.	0.1	infeas.	0.1
5	4	10	3.43	3.14	0.1	2.57	0.1
5	3	10	3.43	2.33	0.1	2.07	0.1
4	3	11	4.43	2.62	0.1	1.50	0.1
4	2	13	6.43	5.95	0.1	5.57	0.1
3	2	14	7.43	6.57	0.1	5.59	0.1
2	2	18	11.43	9.52	0.1	8.62	0.1
2	1	26	19.43	16.81	0.1	15.26	0.1
1	1	34	27.43	23.00	0.1	17.90	0.1

Table II. Results of scheduling and latency minimization on the FDCT example.

Constraints			Scheduling			Min. Latency	
mul	alu	L	Solved	Time (s)	Dec./Inf.	L	Time (s)
8	4	8	yes	0.2	12/0	8	0.3
8	3	9	yes	0.1	0/0	10	0.6
5	4	10	yes	0.5	30/0	10	0.5
5	3	10	no	1.2	—/100	11	1.7
4	3	11	yes	0.4	20/1	11	0.5
4	2	13	yes	0.5	22/25	13	0.5
3	2	14	yes	0.6	27/10	14	0.6
2	2	18	yes	0.5	26/0	18	0.5
2	1	26	yes	0.6	32/2	26	0.6
1	1	34	yes	0.7	34/4	34	0.8

6.2 Scheduling

The second experiment considers scheduling and latency minimization. Table II shows the results obtained for the same problem instances as used in the first experiment.

For the scheduler, the table lists whether a feasible schedule is found, the required run time, and two numbers that indicate the efficiency of the search space traversal: the number of decisions taken to arrive at the feasible solution (i.e., the depth of the search tree at the point where this solution is found), and the number of backtracks performed. The fourth problem instance is actually infeasible, and therefore the scheduler stops when the number of backtracks hits a user-defined limit.

Latency minimization is performed by first estimating the minimum latency. This is followed by repeatedly trying to find a feasible schedule, and if this fails, increasing the latency with 1. In all cases, the minimum latency is found.

Table III shows the results of latency minimization for two larger, pipelined examples. For both examples, a range of initiation interval constraints is tested. L represents the latency of the resulting schedule, while L_{lb} is a lower bound estimate of the minimum latency. In all cases, the difference is at most two cycles.

6.3 Run-Time Efficiency

The third experiment evaluates the performance of FACTS on larger examples. For that purpose, we use two scalable examples: a biquad filter with the number

Table III. Results of latency minimization on pipelined examples.

Example	$\|V\|$ / $\|E_d\|$ / $\|E_s\|$	II	L	L_{lb}	Time (s)
macDT0	210 / 192 / 153	26	infeas.		0.1
		27	35	34	54.0
		28	33	33	24.3
		29	32	32	24.8
		30	31	31	22.5
chen_ref	138 / 87 / 72	25	infeas.		0.3
		26	31	31	14.9
		27	32	30	82.2
		28	32	30	70.4
		29	31	31	17.1
		30	32	31	110.4
		31	32	31	45.0

Table IV. Efficiency on larger examples.

			Constraint Analysis		Latency Minimization	
Example	$\|V\|$	L	Mobility	Time (s)	L	Time (s)
biquad_10	100	52	47.9 → 45.4	0.2	52	6.7
biquad_20	200	102	97.8 → 95.3	0.5	102	67.1
biquad_30	300	152	147.8 → 145.3	1.5	152	280.9
fdct_2s	84	34	19.9 → 11.9	0.2	34	5.0
fdct_3s	126	50	28.0 → 12.8	0.4	50	20.1
fdct_4s	168	66	36.1 → 13.3	0.8	66	53.2
fdct_5s	210	82	44.1 → 13.6	1.4	82	115.0
fdct_6s	252	98	52.1 → 13.8	2.2	98	211.4

of stages as a parameter, and serial compositions of the FDCT example. Table IV shows the results for applying constraint analysis and for minimizing latency.

The process of applying all constraint analysis techniques until no progress is obtained, runs in seconds. For the biquad example, a relatively small improvement in mobility is obtained because the schedule freedom is actually very large. For the FDCT example, a significant reduction is obtained. The results for latency minimization show that also a search strategy on top of constraint analysis, still has quite reasonable run times. In all cases, the minimum latency is found.

6.4 Register Binding

The fifth and last experiment demonstrates our register binding strategy under fixed register file capacity constraints. The results are shown in Table V. For each problem instance, the table lists the register file capacity constraints, the number of registers actually used, the required run time, and the impact of serialization on the mobility of the operations (the numbers before and after the arrow respectively denote the mobility before and after serialization). The experimental results clearly show that our method is steered by the individual register file constraints; despite the presence of tight timing and resource constraints, it is able to generate different schedules depending on the settings of the individual capacity constraints.

Table V. Results of scheduling with register file capacity constraints.

Example	$\|V\|/II/L$	RF Caps	RF Sizes	Time (s)	Mobility
fft256	30/4/13	infinite	3, 3, 1, 2	0.1	
		1, 4, 1, 2	1, 4, 1, 2	0.1	$0.7 \rightarrow 0.3$
		2, 2, 1, 2	2, 2, 1, 2	0.4	$2.3 \rightarrow 0.0$
		2, 3, 1, 1	2, 3, 1, 1	0.8	$2.1 \rightarrow 0.0$
		3, 2, 1, 1	3, 2, 1, 1	0.9	$2.1 \rightarrow 0.0$
		4, 1, 1, 2	4, 1, 1, 2	0.1	$0.7 \rightarrow 0.4$
FDCT	42/18/18	infinite	9, 4	0.1	
		9, 4	7, 4	2.3	$9.5 \rightarrow 4.0$
		6, 4	6, 4	2.7	$9.5 \rightarrow 2.0$
		8, 2	8, 2	0.9	$9.5 \rightarrow 1.4$
loeffler	56/26/28	infinite	8, 4, 10	0.4	
		8, 4, 10	8, 4, 9	3.5	$14.4 \rightarrow 3.1$
		4, 3, 8	4, 3, 8	4.9	$14.4 \rightarrow 1.0$

7. CONCLUSIONS AND FUTURE WORK

In this paper we discussed the code generation tool FACTS. The techniques implemented in FACTS are based on the observation that traditional code generation methods require too much help and expertise from a designer to satisfy the combination of timing, resource, and storage constraints encountered when mapping DSP applications onto embedded processors. FACTS is *guided* rather than *hampered* by these constraints: By using the constraints to prune the schedule search space, the scheduler is often prevented from making a decision that inevitably violates one or more constraints.

We have argued that the problem of phase coupling cannot be ignored when constraints are tight and efficient solutions are desired. Traditional methods that perform code generation in separate stages are often not able to find an efficient or even a feasible solution. In our approach, the problem of phase coupling is addressed by letting all analyses work on a single unified representation of the schedule search space, the *distance matrix*. This is an effective representation because it administrates relative timing, which is important for solving the scheduling problem. The results of the analyses discussed in this paper are expressed in terms of sequence constraints and combined in the distance matrix simply by computing the longest paths between all pairs of operations.

The efficiency of implementing a strong interaction between several code generation stages is supported by the experimental results that feature reasonable run times for DSP applications that are constrained in the timing, resource, and storage domain. In the current implementation, constraint analysis is performed by applying all techniques consecutively until no further progress is obtained; the performance can be further improved by developing more efficient strategies to combine the various techniques.

In this paper, we have not described in detail which features of current DSP architectures can be captured by FACTS. Rather, we have focused on the abstract model used. This model fits VLIW architectures quite well. However, it is definitely not restricted to only that class of architectures. Our future work will focus on extensions to the current approach. We intend to implement predicated execution in the register binding model [Zhao and van Eijk 1999], which

helps to reduce register pressure in case of aggressive global scheduling. Another extension that we are considering is a method to assign operations to functional resources and values to register files in the context of a clustered architecture with a restricted connection network; this requires FACTS to deal with communication delays that are unknown before assignment. We also intend to analyze storage models such as fifos and rotating register files, which could help to reduce code size and/or decrease the initiation interval of pipelined schedules.

REFERENCES

ALBA-PINTO, C., MESMAN, B., AND VAN EIJK, C. 1999. Register files constraint satisfaction during scheduling of dsp code. In *Symposium on Integrated Circuits and Systems Design* (Natal, Brazil, Oct. 1999).

BASHFORD, S. AND LEUPERS, R. 1999. Constraint driven code selection for fixed-point dsps. In *Proceedings of the 36th ACM / IEEE Design Automation Conference* (1999). ACM and IEEE Computer Society, 817–822.

BRASPENNING, R. 1999. Modeling issue slot constraints with resources. Technical report (May), Eindhoven University of Technology.

CORMEN, T., LEISERSON, C., AND RIVEST, R. 1990. *Introduction to algorithms*. MIT Press.

EISENBEIS, C., CHAMSKI, Z., AND ROHOU, E. 1999. Flexible issue slot assignment for vliw architectures. In *4th International Workshop on Software and Compilers for Embedded Systems* (St. Goar, Germany, Sept. 1999).

FARABOSCHI, P., DESOLI, G., AND FISHER, J. 1998. Clustered instruction-level parallel processors. Technical Report HPL-98-204, Hewlett-Packard.

KÄSTNER, D. AND LANGENBACH, M. 1999. Code optimization by integer linear programming. In *ACM Conference on Compiler Construction* (1999).

KUCHCINSKI, K. 1997. Embedded system synthesis by timing constraints solving. In *International Symposium on System Synthesis* (Antwerp, Sept. 1997).

LAM, M. 1988. Software pipelining: An effective scheduling technique for vliw machines. In *SIGPLAN Conference on Programming Language Design and Implementation* (June 1988).

LEUPERS, R. 1997. *Retargetable code generation for digital signal processors*. Kluwer Academic Publishers.

LEUPERS, R., SCHENK, W., AND MARWEDEL, P. 1994. Microcode generation for flexible parallel architectures. In *Working Conference on Parallel Architectures and Compiler Technology* (1994).

MESMAN, B., ALBA-PINTO, C., AND VAN EIJK, C. 1999. Efficient scheduling of dsp code on processors with distributed register files. In *International Symposium on System Synthesis* (San Jose, Nov. 1999).

MESMAN, B., STRIK, M., TIMMER, A., VAN MEERBERGEN, J., AND JESS, J. 1998. A constraint driven approach to loop pipelining and register binding. In *Proceedings of the Design Automation and Test in Europe* (Paris, 1998). IEEE Computer Society Press.

MESMAN, B., TIMMER, A., VAN MEERBERGEN, J., AND JESS, J. 1999. Constraint analysis for dsp code generation. *IEEE Transactions on Computer-Aided Design 18*, 1 (Jan.), 44–57.

NUIJTEN, W. 1994. Time and Resource Constrained Scheduling. Ph.D. thesis, Eindhoven University of Technology.

PAULIN, P. AND LIEM, C. 1996. Embedded systems: Tools and trends, tutorial. In *Proceedings of the European Design and Test Conference* (Paris, Mar. 1996). IEEE Computer Society Press.

PAULIN, P., LIEM, C., MAY, T., AND SUTARWALA, S. 1995. Dsp design tool requirements for embedded systems: a telecommunications industrial perspective. *Journal of VLSI Signal Processing 9*, 1.

RAU, B., KATHAIL, V., AND ADITYA, S. 1998. Machine-description driven compilers for epic processors. Technical Report HPL-98-40, Hewlett Packard research labs.

SANGIOVANNI-VINCENTELLI, A. 1976. A note on bipartite graphs and pivot selection in sparse matrices. *IEEE Transactions on Circuits and Systems CAS-23*, 12, 817–821.

TI. 1997. *TMS320C60xx CPU and Instruction Set Reference Guide*. Texas Instruments.

TIMMER, A. 1995. From Design Space Exploration to Code Generation. Ph.D. thesis, Eindhoven University of Technology, The Netherlands.

TIMMER, A. AND JESS, J. 1993. Execution interval analysis under resource constraints. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design* (Santa Clara, Nov. 1993). IEEE Computer Society Press, 454–459.

TIMMER, A., STRIK, M., VAN MEERBERGEN, J., AND JESS, J. 1994. Conflict modelling and instruction scheduling in code generation for in-house dsp cores. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (1994). ACM and IEEE Computer Society.

TM. 1997. *Trimedia TM-1 Media Processor Data Book*. Philips Semiconductors, Trimedia Product Group.

VAN EIJK, C., JACOBS, E., MESMAN, B., AND TIMMER, A. 1998. Identification and exploitation of symmetries in dsp algorithms. In *Proceedings of the Design Automation and Test in Europe* (Munich, 1998). IEEE Computer Society Press, 602–608.

ZHAO, Q. AND VAN EIJK, C. 1999. Register binding for dsp code containing predicated execution. In *Proceedings of the Workshop on Circuits, Systems and Signal Processing* (Mierlo, the Netherlands, Nov. 1999). IEEE Benelux & ProRisc.

Received November 1999; Accepted February 2000