Constraint analysis for DSP code generation

Constraint Analysis for DSP Code Generation

Bart Mesman, Adwin H. Timmer, Jef L. van Meerbergen, and Jochen A. G. Jess

***Abstract*— Code generation methods for digital signal-processing (DSP) applications are hampered by the combination of tight timing constraints imposed by the performance requirements of DSP algorithms and resource constraints imposed by a hardware architecture. In this paper, we present a method for register binding and instruction scheduling based on the exploitation and analysis of the combination of resource and timing constraints. The analysis identifies implicit sequencing relations between operations in addition to the preceding constraints. Without the explicit modeling of these sequencing constraints, a scheduler is often not capable of finding a solution that satisfies the timing and resource constraints. The presented approach results in an efficient method to obtain high-quality instruction schedules with low register requirements.**

***Index Terms*—Code generation, register binding, scheduling.**

I. Introduction

DIGITAL signal-processing (DSP) design groups and embedded processor users indicate the increasing use of application-domain-specific instruction-set processors (ASIP's) [1] as a significant trend [2]. ASIP's are tuned toward specific application domains and have become popular due to their advantageous tradeoff between flexibility and cost. This tradeoff is present neither in application-specific integrated circuit (ASIC) design, where emphasis is placed on cost, nor in the design of general-purpose DSP's, where emphasis is placed on flexibility. Because of the importance of time-to-market, software for these ASIP's is preferably written in a high-level programming language, thus requiring the use of a compiler. In this paper, we will address some of the compiler issues that have not been addressed thoroughly yet: the problems of register binding and scheduling under timing constraints. Note that we do not consider resource binding. Although resource binding can have a major effect on the quality of the code, much work has been done on this subject [3]. Furthermore, ASIP's mostly have such irregular architectures that there is often little choice for mapping an operation. For example, addresses are calculated on a dedicated unit complying with the desired bit width, there is usually only one functional unit performing barrel shifting, etc. Because we consider distributed register-file architectures where a register file usually provides input for only one functional unit, the resource binding induces a binding of values to register files. In our experiments (Section IX), resource binding has been done by the Mistral2 toolset [4] for a very long instruction word (VLIW) architecture.

The reason that register binding and scheduling under timing constraints have not yet been addressed thoroughly is that most of the currently available software compiling techniques were originally developed for general-purpose processors (GPP's), which have characteristics different from those of ASIP's.

- GPP's most often have a single large register file, accessible from all functional units, thus providing a lot of freedom for both scheduling and register binding. ASIP's usually have a distributed register-file architecture (for a large access bandwidth) accompanied by special-purpose registers. Automated register binding is severely hampered by this type of architecture.
- ASIP's are mostly used for implementing DSP functionality that enforces strict real-time constraints on the schedule. GPP compilers use timing as an optimization criterion but do not take timing constraints as a guideline during scheduling.
- Designing a compiler comprises a tradeoff between compile time and code quality. Typically, GPP software should compile quickly, and code quality is less important. For embedded software (that is, for an ASIP), however, code quality is of utmost importance, which may require intensive user interaction and longer compile times.

As a result of these characteristics, compiling techniques originating from the GPP world are less suitable for the mapping problems of ASIP architectures. The field of high-level synthesis [5], concerned with generating application-specific hardware, has also been engaged in the scheduling and register-binding problem. Because the resource-constrained scheduling problem was proven NP-complete [6], most solution approaches from this field have chosen to maintain the following two characteristics.

- Decomposition in a scheduling and register allocation phase. Because these phases have to be ordered, the result of the first phase is a constraint for the second phase. A decision in the first phase may lead to an infeasible constraint set for the second phase.
- The use of heuristics in both phases.

Heuristics for register binding and operation scheduling are runtime efficient. When used in an ASIP compiler, however, they are unable to cope with the interactions of timing, resource, and register constraints. The user often has to provide

Manuscript received April 1, 1998; revised August 27, 1998. This paper was recommended by Associate Editor G. Borriello.

B. Mesman and J. L. van Meerbergen are with Philips Research Laboratories, Eindhoven 5656 AA The Netherlands and the Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands.

A. H. Timmer is with Philips Research Laboratories, Eindhoven 5656 AA The Netherlands.

J. A. G. Jess is with the Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands.

Publisher Item Identifier S 0278-0070(99)00810-6.

pragmas (compiler hints) to help the scheduler in satisfying the constraints. Furthermore, in order to obtain higher utilization rates for the resources and to satisfy the timing constraints, software pipelining [7], also called loop pipelining or loop folding, is required. In Section III, we will show that a heuristic-like list scheduling is already unable to satisfy the timing and resource constraints on a very simple pipeline example.

We discuss related work in Section II. In Section III, the dataflow graph (DFG) model is introduced with some definitions. An example of a tightly constrained schedule problem will demonstrate why traditional heuristics are not suitable to cope with the combination of different types of tight constraints. In Section IV, the problem statement is given and a global solution strategy is proposed. Sections V–VII focus on analysis. In Section VIII, complexity issues are discussed. Section IX shows some experimental results.

II. Related Work

Code generation for embedded processors has become a major trend in the CAD community. Most active in this area are the group of Paulin with the FlexWare environment [8], Marwedel's group [9], IMEC with the Chess environment [10], and Philips [11]. Because of the pressure for small instructions, mostly irregular processor architectures are used. A structural processor model for these irregular architectures, combined with the demand for *retargetability*, caused a great emphasis on code selection [12]. Compilers for these platforms have produced rather disappointing results when compared to manually written program code. Therefore, we choose to model the instruction-set irregularities procedural as hardware conflicts during the scheduling phase. This reduces the dependencies between the different code-generation phases and enables the expression of all different constraints (instruction-set irregularities, resource constraints, timing and throughput constraints, precedence, register binding, etc.) as much as possible in a single model.

Software pipelining has been the subject of many research projects. The modulo scheduling scheme by Rau [13] has inspired many researchers. His approach is essentially a list-scheduling heuristic. Backtracking is used when an operation cannot be scheduled.

Many more approaches are based on the list-scheduling heuristic, notably the work of Goossens [7] and Lam [14].

The group of Nicolau [15] devised a heuristic that often finds an efficient schedule with respect to timing. It does not take constraints on the timing into account, however, and the latency and initiation interval are difficult to control. Because implicit unrolling is performed until a steady state has been reached, code duplication occurs frequently, resulting in possibly large code sizes. These are intolerable for embedded processors with on-chip instruction memory, especially for VLIW architectures.

Integer linear programming (ILP) approaches to finding pipelined schedules started with the work of Hwang [16]. A considerable amount of constraints caused most formal methods to generate intolerable runtimes for DFG's containing more than about 20 operations.

Rau *et al.* [17] successfully performed register binding tuned to pipelined loops. They mention that for better code quality, "concurrent scheduling and register allocation is preferable," but for reasons of runtime efficiency they solve the problem of scheduling and register binding in separate phases.

Some approaches have been reported that perform scheduling (with loop pipelining) and register binding simultaneously. Eichenberger *et al.* [18] solve some of the shortcomings of the approach used by Govindarajan *et al.* [19], but both try to solve the entire problem using an ILP approach, which is computationally too expensive for practical instances of the problem depicted above. Following is a summary of these points.

- On one hand, the combination of timing, resource, and register constraints does not describe a search space that can be suitably traversed by simple heuristics.
- On the other hand, practical instances of the total problem are too large to be efficiently solved with ILP-based methods.

Therefore, we will try a different approach based on the analysis of the constraints without exhaustively exploring the search space. Timmer *et al.* [20] successfully performed constraint analysis on a schedule problem using bipartite matching, but this work is difficult to extend to register constraints.

III. Definitions

In this section, we will introduce the general high-level synthesis scheduling problem. The difficulty of solving this problem when the constraints are tight is illustrated with a simple example. A perspective is introduced to understand the reasons why this is a difficult problem to solve for traditional methods.

A. High-Level Synthesis Scheduling

A DSP application can be expressed using a DFG [21].

Definition 1—DFG: A DFG is a five-tuple (V, $E_d \cup E_s$, Y, val, w), where:

- V is the set of vertices (operations);
- $E_s \subseteq V \times V$ is the set of sequence precedence edges;
- $E_d \subseteq V \times V$ is the set of data precedence edges;
- Y is a set of values;
- val: $E_d \rightarrow Y$ is a function describing which value is communicated over a data precedence edge;
- w: $E_s \cup E_d \rightarrow \mathbb{Z}$ is a function describing the timing delay associated with a precedence edge.

In Fig. 13(a), for example, the set of operations $V =$ source, a, b, c, d, e, sink. The set of sequence precedence edges E_s = {(source, a), (b, c), (d, e), (e, sink)}, and the set of data precedence edges $E_d = \{$(a, b), (c, d)$\}$. The set of values $Y = \{v, w\}$. Furthermore val (a, b) $= v$, and val (c, d) $= w$. Every edge $(v_i, v_j) \in E$ has $w(v_i, v_j) = 1$ except w(source, a) $= 0$.

Two (dummy) operations are always (implicitly) part of the DFG: the source and the sink. They have no execution delay, but they do have a start time. The source operation is the "first" operation, and the sink operation is the "last" one.

A DFG describes the primitive actions performed in a DSP algorithm and the dependencies between those actions. A *schedule* defines when these actions are performed.

Definition 2: A schedule s: $V \rightarrow \mathbb{Z}$ describes the start times of operations.

For $v \in V$, $s(v)$ denotes the start time of operation v. We also consider *pipelined* schedules: in a loop construction, the *loop body* is executed a number of times. In a traditional schedule, iteration $i+1$ of the loop body is executed strictly after the execution of the ith iteration. Goossens [7] demonstrates a practical way to overlap the executions of different loop-body iterations, thus obtaining potentially much more efficient schedules. The pipelined schedule is executed periodically.

Definition 3—Initiation Interval (II): An II is the period between the start times of the execution of two successive loop-body iterations.

A schedule has to satisfy the following constraints. The *precedence constraints,* specified by the precedence edges, state that

$$\forall (v_i, v_j) \in E\text{: } s(v_j) \geq s(v_i) + w(v_i, v_j).$$

Furthermore, the source and sink operations have an implicit precedence relation with the other operations

$$\forall v_i \in V\text{: } s(v_i) \geq s(\text{source}).$$

When a DFG is mapped on a hardware platform, we encounter several resource limitations. These *resource constraints* are given by the function $rsc(v_i, v_j)$: $V \times V \rightarrow \{0, 1\}$, defined by

$$rsc(v_i, v_j) = \begin{cases} 0, & \text{if } v_i \text{ and } v_j \text{ have a conflict} \\ 1, & \text{otherwise.} \end{cases}$$

A conflict can be anything that prevents the operations v_i and v_j from executing simultaneously. For example, they are executed on the same functional unit, transport the result of the computation over the same bus, or there is no instruction for the parallel execution of v_i and v_j [20]. A resource constraint $rsc(v_i, v_j)$ thus states that

$$rsc(v_i, v_j) = 1 \Rightarrow s(v_i) \neq s(v_j).$$

For loop-pipelined schedules, the implication of a resource constraint is

$$rsc(v_i, v_j) = 1 \Rightarrow s(v_i) \neq s(v_j) \bmod \text{II}.$$

For reasons of simplicity, we assume that all operations have an execution delay of one clock cycle. In Section V-A, we will show how pipelined or multicycle operations are modeled using precedence constraints. The general high-level synthesis scheduling problem (HLSSP) is formulated as follows.

Problem Definition 1—HLSSP: Given are a DFG, a set of resource constraints $rsc(v_i, v_j)$, an II, and a constraint on the latency l (completion time). Find a schedule s that satisfies the precedence constraints $E_d \cup E_s$, the resource constraints, and the timing constraints II and l.

In Section V-A, we will introduce some additional constraints that characterize our specific problem. We note that HLSSP is NP-hard [6].

B. Schedule Freedom

In the previous subsection, we introduced the high-level synthesis scheduling problem. In order to solve this problem (and the extended scheduling problem from Section IV), it is convenient to describe the set of possible solutions: the *solution space.* In this subsection, we will describe the solution space as a range of possible start times for each operation. Because this set of feasible start times is as difficult to find as it is to find a schedule, we will approximate it by the "as soon as possible/as late as possible" (ASAP–ALAP; Definitions 8 and 9) interval, the construction of which is solely based on the precedence constraints $E_d \cup E_s$. By generating additional precedence constraints that are implied by the combination of all constraints, the ASAP–ALAP interval provides an increasingly more accurate estimate of the set of feasible start times.

We start with a description of the solution space.

Definition 4: The set of feasible schedules S is the set of schedules such that each schedule $s \in S$ satisfies the precedence constraints, the resource constraints, and the timing constraints.

An operation thus has a range of feasible start times, each corresponding to a different schedule.

Definition 5: The actual schedule freedom of a DFG is the average size of the set of feasible start times minus one

$$\frac{1}{|V|} \sum_{v_i \in V} (|T(v_i)| - 1).$$

The actual schedule freedom quantifies the amount of choice for making schedule decisions. For traditional schedule heuristics, a large actual schedule freedom is advantageous because it gives the scheduler more room for optimization. The actual schedule freedom is defined by the application (the DFG and the timing constraints) and the available hardware platform. A large actual schedule freedom is not guaranteed, and we have to deal with a tightly constrained scheduling problem.

Because of the complexity of finding the set of feasible start times, a conservative ASAP–ALAP estimate is more practical. For the definition of the ASAP–ALAP interval, we need the notion of immediate predecessors and successors.

Definition 6: The immediate predecessors, successors

$$\forall (v \in V)\text{: } \begin{array}{l} pred(v) = \{u \in V | (u, v) \in E\} \\ succ(v) = \{u \in V | (v, u) \in E\}. \end{array}$$

The ASAP value is recursively defined as follows.

Definition 7—ASAP Value:

$$\text{ASAP}(v) = \begin{cases} 0, & \text{if } pred(v) = \emptyset \\ \max\limits_{u \in pred(v)} \\ \quad \cdot (\text{ASAP}(u) + w(u, v)), & \text{otherwise.} \end{cases}$$

The latest possible start time is called the ALAP value. Let l denote the latency constraint. Then ALAP(sink) $= l$, and for all other operations, the following holds.

Definition 8—ALAP Value:

$$\text{ALAP}(v) = \begin{cases} l - w(v, \text{sink}), & \text{if } succ(v) = \emptyset \\ \min\limits_{u \in succ(v)} \\ \quad \cdot \text{ALAP}(u) - w(v, u), & \text{otherwise.} \end{cases}$$

The start time of each operation must lie in between the ASAP and ALAP values, inclusively

$$\forall (v \in V)\text{: ASAP}(v) \leq s(v) \leq \text{ALAP}(v).$$

Therefore, the ASAP–ALAP interval is a conservative estimate of (contains) the set of feasible start times.

In this paper, we will extract sequencing constraints that are necessarily implied by the combination of all constraints. These sequencing constraints are then explicitly added to the DFG as precedence constraints. Because the ASAP–ALAP interval is based solely on the precedence constraints, it provides an increasingly more accurate estimate of the set of feasible start times. For most scheduling methods, either the ASAP–ALAP intervals or the precedence constraints are an extremely important guideline: these methods take the precedence or the ASAP–ALAP interval explicitly as a basis. Schedule choices are made with respect to the available resources. When the ASAP–ALAP interval does not reflect the actual schedule freedom very accurately, there will often come a point in the schedule process where there are no available resources for an operation, and the operation cannot be scheduled. In this way, the precedence constraints and the resulting ASAP–ALAP interval implicitly represent the "search scope" of the scheduler. Therefore, we also define the "apparent freedom," also called mobility or slack.

Definition 9—Apparent Schedule Freedom (Mobility, Slack): The apparent schedule freedom is the average size of the set of ASAP–ALAP intervals

$$\frac{1}{|V|} \sum_{v_i \in V} \text{ALAP}(v_i) - \text{ASAP}(v_i).$$

Because the precedence and the ASAP–ALAP interval form the basis for making schedule decisions, the performance of a scheduler depends largely on the accuracy of the interval. When the ASAP–ALAP interval is an accurate estimate of the set of feasible start times $T(v_i)$, the mobility is an accurate estimate of the actual schedule freedom and vice versa. Therefore, we will use the mobility before and after the constraint analysis as a performance measure of the analysis.

C. A Small Example

Often a schedule heuristic is "deceived" by the apparent schedule freedom and is unable to generate a feasible schedule. A combination of several types of constraints is responsible for the fact that the actual schedule freedom is smaller than the apparent freedom. A small example illustrates the difficulty of handling the combination of different types of constraints.

In Fig. 1, a precedence graph of five operations is given (the arrows indicate a precedence relation). The [ASAP, ALAP] interval is printed directly left of the corresponding operation. In order to meet the constraint of three clock cycles on an

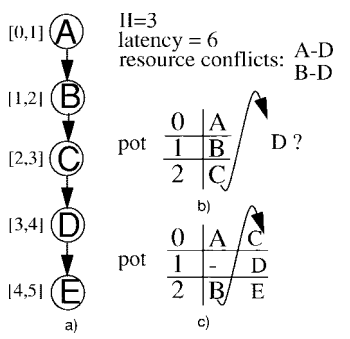


Fig. 1. Example with loop folding: (a) precedence graph, (b) list-schedule, and (c) only feasible schedule in six clock cycles.

II, loop folding has to be applied [indicated by the arrow in Fig. 1(b) and (c)]. Because folding introduces extra code, we do not want to fold more than once, which constrains the latency to six clock cycles. In Fig. 1(b), the result of a list scheduler is shown. The left column contains the *time potential* (schedule time modulo II). The list scheduler greedily schedules A, B, and C as soon as possible (ASAP), and concludes that D cannot be scheduled. In Fig. 1(c), a feasible schedule is given. The key to obtaining this schedule is to postpone B one clock cycle relative to its ASAP value. In Fig. 1, the apparent freedom or mobility equals one clock cycle per operation. The reader can verify that the combination of precedence, resource, latency, and throughput constraints leaves no actual schedule freedom at all: the schedule in Fig. 1(c) is the only possible schedule in six clock cycles. The [ASAP, ALAP] estimate of the schedule interval was not accurate enough, and the other constraints should have been considered as well. In Section V-B, we will show how the analysis of the combination of all constraints provides the most accurate ASAP–ALAP intervals (equal to the actual schedule freedom) for the schedule problem of Fig. 1.

IV. Problem Statement and Global Approach

In the previous section, we introduced the general HLSSP. In this section, we define our characteristic scheduling problem and combine it with the problem of finding a register binding. We will decompose the problem and construct a block diagram of the global approach. Our characteristic problem statement for finding a feasible schedule and register assignment is as follows.

Problem Definition 2—Register Binding and Operation Scheduling Problem: Given a cyclic DFG, the resource constraints $rsc(v_i, v_j)$, a binding of values to register files, an II, and a constraint on the latency l, find an assignment of values to registers and a schedule s that satisfies the precedence constraints $E_d \cup E_s$, the resource constraints, and the timing constraints II and l.

Because it is difficult to make a register binding and a schedule simultaneously, we decompose the problem in separate phases, as depicted in Fig. 2. First, an initial register binding (discussed in Section VII-A) is constructed in a simple manner. The II for each hierarchical level is also fixed prior to the analysis. Most often, it is set by the designer. Otherwise, we start with a lower bound based on loop-carried dependen-

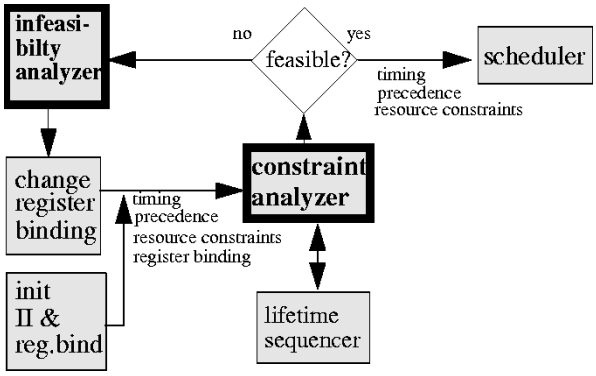


Fig. 2. Global approach.

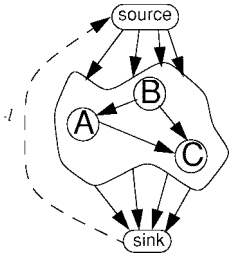


Fig. 3. Modeling the latency.

cies [22] and available resources. When this II is not feasible, it is incremented by one clock cycle. Profiling suggests that the optimal II is usually only one or two clock cycles away from the lower bound.

The central part, the constraint analyzer (discussed in Sections V and VI), generates additional precedence constraints that are implied by the combination of all constraints, including the given register binding. These additional precedence refine the ASAP–ALAP intervals, thus providing a much more accurate estimate of the set of feasible start times. They will guide the scheduler and often prevent it from making schedule decisions leading to infeasibility.

The new precedence constraints are such that the register binding is guaranteed: all lifetimes between values residing in the same register have been sequentialized. The constraint analyzer (and the lifetime sequencer) thus replaces the register-binding constraints completely by precedence constraints. When the constraint set leaves some room for different lifetime sequentializations, the *lifetime sequencer*, discussed in Section VI-C, chooses between several alternatives. When the constraint set is tight, as is the case in most of the benchmarks of Section IX, only one or two choices are made by the lifetime sequencer. A branch and bound algorithm is therefore runtime-efficient enough for the lifetime sequencer.

The added precedence may cause violation of the constraint set (including the register binding). An *infeasibility analysis* (discussed in Section VII-B) uses the administrative bookkeeping done by the constraint analyzer to find the bottleneck in the constraint set and the register binding. The "change register binding" block in Fig. 2 tries to solve this bottleneck by rebinding a value to a different register. This scheme is iterated until the constraint set and the register binding are feasible. Last, the precedence generated by the constraint analyzer is fed to a simple external schedule heuristic.

An advantage of this new approach is that in practice, a simple off-the-shelf scheduler can be used to complete the schedule. Although the existence of a schedule is not strictly guaranteed after the constraint analyzer, a schedule was found for all problem instances. As the scheduler and its heuristics are not critical in this approach, we will not focus on them in this paper.

Note that a main characteristic of our approach is that we perform register binding prior to schedule analysis. The primary reason for this is our goal to obtain an efficient register binding given the timing and resource constraints. Therefore, we first want to fix the register binding, and constrain the schedule accordingly without violating the other constraints. The additional precedence constraints will guide the scheduler more accurately toward a feasible solution.

When violation of the constraint set does occur, the infeasibility analyzer should be able to find the bottleneck.

A. Problem Statement for the Constraint Analyzer and the Infeasibility Analyzer

It is necessary that the constraint analyzer does some administrative bookkeeping, such that the infeasibility analysis is able to indicate a bottleneck in the register binding. The problem statement for the constraint analyzer and the infeasibility analyzer is therefore as follows.

Problem Definition 3—Operation Ordering and Bottleneck Identification Problem (OOBIP): Given a cyclic DFG, a register binding, a set of resource constraints $rsc(v_i, v_j)$, an II, and a constraint on the latency l, find either a partial order of operations satisfying the register binding (if the constraint set is feasible) or a smallest infeasible subset of value conflicts.

B. Effect of the Scheduler

The question rises as to whether or not the scheduler is always able to find a solution when the constraint set passes the infeasibility check. We distinguish two situations: pipelined and nonpipelined schedules.

For nonpipelined schedules, the scheduler is always able to find a solution that complies with the register binding. Experience shows, however, that the latency constraint may not always be satisfied in the final schedule.

A pipelined schedule is more difficult to obtain; sometimes a decision is made that inevitably violates the constraint set. It is therefore wise to alternate between scheduler and constraint analyzer; first the scheduler makes a schedule decision. The decision is then modeled in terms of precedence relations (Section V-A), and the constraint analyzer computes the effect of this decision on the mobility of the other operations. In this way, the search space of the scheduler is reduced according to decisions previously made in the schedule process. Although there is still no absolute guarantee that a solution is found in this way, a solution was found on all problem instances tried so far. If the scheduler fails after all, the infeasibility analyzer will indicate which value conflicts, resource conflicts, and schedule decisions are responsible for this failure. The

designer himself will then have to enforce a different schedule decision or partial rebinding.

The following sections comprise a solution to OOBIP. Section V is concerned with the analysis of resource conflicts, precedence, and timing constraints. Section VI extends the analysis to a given register binding. In Section VII-B, we will demonstrate the infeasibility analyzer based on the results of Sections V and VI.

V. Resource-Constrained Analysis

In the previous section, we introduced a block diagram of our global approach. This section will focus on part of the constraint analyzer [23]. Section V-A models the different constraints as much as possible in terms of precedence. Section V-B analyzes the resource constraints, and generates precedence as well, so that most of the constraint set is expressed in a unified model (the DFG). The analysis is illustrated on the example from Section III-C. In Section VI, the analysis is extended to handle *value conflicts* that result from a given register binding.

A. Modeling the Constraints

We start this section by showing how some of the constraints can be represented in the DFG model introduced in Section III.

- *Latency:* A constraint l on the latency is translated to an arc (sink, source) with $w = -l$, as illustrated in Fig. 3. This is interpreted as $s(\text{source}) \geq s(\text{sink}) - l$, which is equivalent to $s(\text{sink}) \leq s(\text{source}) + l$, meaning the last operation may not be executed more than l clock cycles after the start of the first operation.
- *Microcoded controller and loop folding:* We assume that the architecture contains a microcoded controller. As a consequence, the same code is executed every loop iteration. This implies that a communicated value is written in the same register each iteration. When loop iterations overlap, we have to ensure that a value is consumed before it is overwritten by the next production. Since subsequent productions are exactly II clock cycles apart, a value cannot be alive longer than II clock cycles. So the operation C that consumes a value must 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 dependency (P, C), there is an arc (C, P) with $w = -$II. Lemma 8 gives conditions when this timing constraint can be tightened.
- *Pipelined executions and multicycle operations:* Pipelined executions and multicycle operations can be modeled by introducing an operation for each stage of the execution. Subsequent stages are linked in time using two sequence edges, as indicated in Fig. 4. For multicycle operations, A and B occupy the same resource.
- *Scheduling decisions:* When schedule decisions are taken during the process, the schedule intervals of other operations are affected. Therefore, it is desirable to be able to express a schedule decision in the DFG so that its effect can be analyzed in the context of the other constraints. Scheduling decisions may take different forms. A timing relation between two operations can be directly translated to a sequence edge. When an operation v is fixed at a certain clock cycle c, we need two sequence edges, as indicated in Fig. 5.
- *Resource conflicts and instruction-set conflicts:* We use method [20] to model instruction set conflicts as resource conflicts $rsc(v_i, v_j)$, introduced in Section III-A.

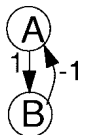


Fig. 4. Modeling pipelined and multicycle operations.

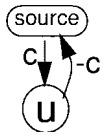


Fig. 5. Modeling a schedule decision.

B. Resource-Constraint Analysis

We now come to the point of explaining the analysis process. By observing a combination of constraints, we can reduce the search space. This reduction is made explicit by adding precedence constraints (sequence edges). In this section, a lemma will be given that observes the interaction between resource conflicts, precedence, and timing constraints. The next section demonstrates lemmas to incorporate register conflicts. All the lemmas used in our approach rely on the concept of a path between operations.

Definition 10—Path: A path of length d from operation v_i to operation v_j is a chain of precedence $v_i \rightarrow v_k \rightarrow \cdots v_l \rightarrow v_j$ that implies $s(v_j) \geq s(v_i) + d$.

Definition 11—Distance: The distance $d(v_i, v_j)$ from operation v_i to v_j is the length of the longest path from v_i to v_j.

A path in the graph thus represents a minimum timing delay. For example, in Fig. 1, the path A $\rightarrow$ B $\rightarrow$ C indicates a minimum timing delay of two clock cycles between the start times of A and C. The first lemma presented below affects the timing relation between conflicting operations. It is based on the fact that two operations with a resource conflict cannot be scheduled at the same potential. The *time potential* associated to a time t is t mod II. So if the distance between these operations would cause them to be scheduled at the same potential, the distance has to be increased by at least one clock cycle.

Lemma 1: If $d(v_i, v_j)$ mod II $= 0$ and $rsc(v_i, v_j) = 1$, we can add a sequence precedence edge (v_i, v_j) with weight $d(v_i, v_j) + 1$ without excluding any feasible schedules.

This lemma will help us to solve the schedule problem in Fig. 1. Remember that the key decision to obtaining a feasible

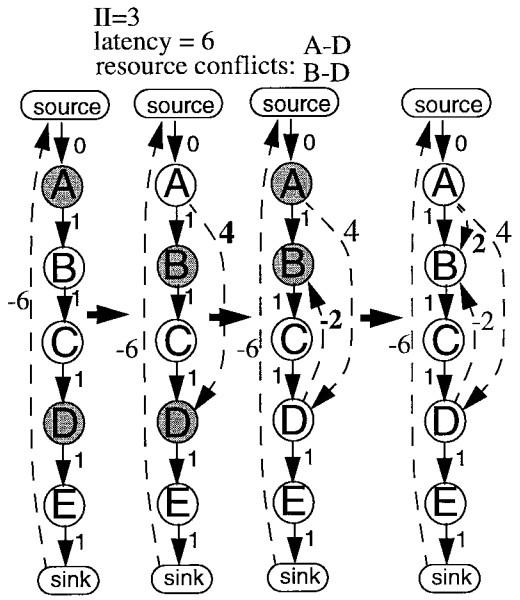


Fig. 6. Derivation of a schedule for Fig. 1.

schedule is to put a gap of one clock cycle between A and B. So our goal is to derive that $d(\mathrm{A}, \mathrm{B}) = 2$. This derivation is given in Fig. 6. Fig. 6(a) represents the DFG model of Fig. 1(a). In Fig. 6(a), we see a path A $\rightarrow$ B $\rightarrow$ C $\rightarrow$ D of length 3 mod II $= 0$ from A to D. According to Lemma 1, we can add a sequence edge A $\rightarrow$ D of weight $3 + 1 = 4$ because A and D have a resource conflict. This edge is drawn in Fig. 6(b). Next, there is a path D $\rightarrow$ E $\rightarrow$ sink $\rightarrow$ source $\rightarrow$ A $\rightarrow$ B of length $1+1-6+0+1 = -3$ clock cycles. Because of the resource conflict D–B, this length has to be increased by one clock cycle. This gives a sequence edge D $\rightarrow$ B of weight -2, as given in Fig. 6(c). We conclude by finding a path of length $4 - 2 = 2$ clock cycles. In Fig. 6(d), the associated sequence edge (A, B) of weight two is explicitly drawn. The precedence relations now completely fix the schedule. The reader can verify that the [ASAP, ALAP] intervals based on the extended DFG of Fig. 6(d) all contain just one clock cycle, and the estimated schedule freedom equals zero.

VI. Register-Constraint Analysis

The previous section introduced the methodology used in the constraint analyzer of Fig. 2. In this section, we will extend the techniques to analyze value conflicts that result from a given register binding [24]. This will be done by introducing lemmas similar to Lemma 1 in the previous section. These lemmas provide necessary conditions (in terms of precedence relations) to guarantee a given register binding. Section VI-A is restricted to nonfolded schedules in order to explain the concept more clearly. The lemmas will be generalized in Section VI-B for register conflicts that cross loop boundaries, which occur when folded schedules are considered.

A. Nonfolded Schedules

In this subsection, two lemmas observe the combination of a given register binding, precedence, and timing constraints for nonfolded schedules. Their use is demonstrated with a small example. In all given examples, a path is indicated using a dashed arc labeled with the length of the path. Sequence edges are dotted. Standard delay (if not labeled) for a sequence edge is zero clock cycle; for a data dependence, it is one clock cycle.

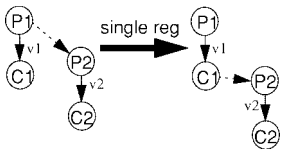


Fig. 7. Lemma 2 for sequentialized value lifetimes.

Lemma 2: Let value $v1$, produced by operation P1 and consumed by C1, and value $v2$, produced by operation P2 and consumed by C2, reside in the same register. If d(P1, P2) ≥ 0, we can add a sequence precedence edge (C1, P2) with weight zero without excluding any feasible schedules.

Lemma 2 is illustrated in Fig. 7. The values $v1$ and $v2$ are bound to the same register. If there is a path of positive length from P1 to P2, then the whole lifetime of value $v1$ has to precede the lifetime of $v2$. This is made explicit by adding a sequence edge from the consumer C1 to the producer P2. A similar lemma is valid when there is a path between the consumers of the values.

Lemma 3: Let value $v1$, produced by operation P1 and consumed by C1, and value $v2$, produced by operation P2 and consumed by C2, reside in the same register. If d(C1, C2) ≥ 0, we can add a sequence precedence edge (C1, P2) with weight zero without excluding any feasible schedules.

When there is a path between the producer of one value and the consumer of the other, we can only exclude a possibility if the delay of the path is strictly greater than zero. Otherwise, the alternative sequentializations, C2 $\rightarrow$ P1, could still yield a feasible schedule when P1 and C2 are scheduled in the same clock cycle.

Lemma 4: Let value $v1$, produced by operation P1 and consumed by C1, and value $v2$, produced by operation P2 and consumed by C2, reside in the same register. If d(P1, C2) ≥ 1, we can add a sequence precedence edge (C1, P2) with weight zero without excluding any feasible schedules.

Lemma 4 is illustrated in Fig. 8. The overall method of analysis is demonstrated in Fig. 9. In this figure, values $v1$ and $v2$ reside in the same register, as do values $w1$ and $w2$. Because operation 1 consumes value $v1$ and operation 7 consumes value $v2$, the lifetime of $v1$ has to precede the lifetime of $v2$ as a result of the precedence $1 \rightarrow 7$ (Lemma 3 applies). Therefore, the sequence edge $1 \rightarrow 8$ is added. Now there is a path $2 \rightarrow 1 \rightarrow 8$ from the consumer of $w1$ to the consumer of $w2$, and Lemma 3 applies again. The sequence edge $2 \rightarrow 9$ is added as a result. Any schedule heuristic can now find a schedule without violating the register binding, which is not the case if the sequence edges were not added.

B. Folded Schedules

In this section, we extend the lemmas from Section VI-A for sequentialized value lifetimes to handle pipelined loop schedules. An example demonstrates the use of the extended lemmas.

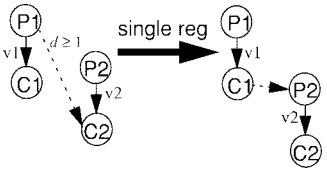


Fig. 8. Lemma 4 for sequentialized value lifetimes.

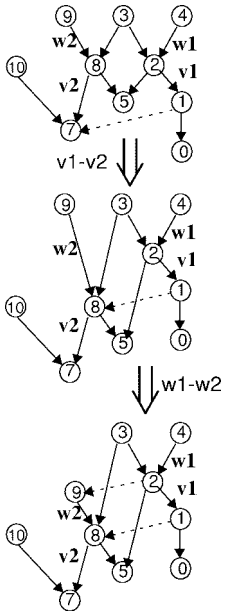


Fig. 9. Example demonstrating the use of Lemma 3.

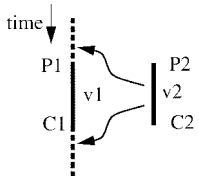


Fig. 10. Timing perspective of serializing alternatives.

When schedules are not folded, it is relatively simple to avoid overlapping lifetimes of values residing in the same register. Only two alternatives have to be considered, as depicted in Fig. 10, where the solid lines indicate the occupation of the register. When loop iterations overlap in time, we also have to take care that the ith lifetime of value $v2$ does not overlap with the $i+1$st (and the $i-1$st) lifetime of value $v1$, as depicted in Fig. 11. Applying the lemmas in this section will eliminate some alternatives, but it is not guaranteed that only one alternative remains. In this case, the lifetime sequencer in Fig. 2 will have to make a decision in order to avoid overlapping lifetimes. This is the subject of Section VI-C.

Sequentialized value lifetimes that belong to different loop iterations pose a problem for the graph model because it makes no difference between operation A_i and A_{i+1} (where A_i denotes the ith execution of A). This suggests that a timing relation between A_i and B_{i+1} has to be translated to a timing relation between A_i and B_i. This translation is straightforward: $s(\mathrm{B}_{i+1}) = s(\mathrm{B}_i)+\mathrm{II}$, so that the relation $s(\mathrm{A}_i) \geq s(\mathrm{B}_{i+1})+d$ is translated to the relation $s(\mathrm{A}_i) \geq s(\mathrm{B}_i) + \mathrm{II} + d$, which

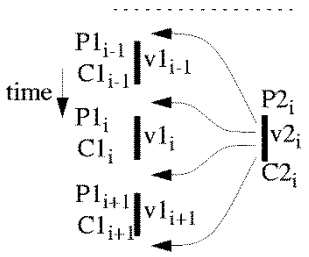


Fig. 11. Serializing alternatives when folding once.

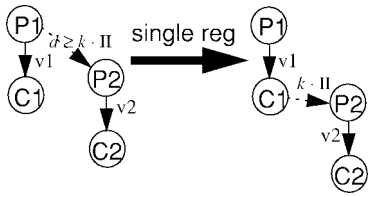


Fig. 12. Lemma 5 for sequentialized value lifetimes.

is equivalent to a sequence edge B $\rightarrow$ A with delay II $+d$. Lemmas 2 and 3 are now easily generalized to Lemmas 5 and 6.

Lemma 5: Let value $v1$, produced by operation P1 and consumed by C1, and value $v2$, produced by operation P2 and consumed by C2, reside in the same register. If d(P1, P2) $\geq k \times$ II, we can add a sequence precedence edge (C1, P2) with weight $k \times$ II without excluding any feasible schedules.

Lemma 6: Let value $v1$, produced by operation P1 and consumed by C1, and value $v2$, produced by operation P2 and consumed by C2, reside in the same register. If d(C1, C2) $\geq k \times$ II, we can add a sequence precedence edge (C1, P2) with weight $k \times$ II without excluding any feasible schedules.

Lemma 5 is illustrated in Fig. 12. Lemma 4 is generalized to Lemma 7.

Lemma 7: Let value $v1$, produced by operation P1 and consumed by C1, and value $v2$, produced by operation P2 and consumed by C2, reside in the same register. If d(P1, C2) $\geq k \times \mathrm{II}+1$, we can add a sequence precedence edge (C1, P2) with weight $k \times$ II without excluding any feasible schedules.

The last lemma we introduce with respect to folded schedules does not serialize lifetimes like the previous lemmas but restricts the lifetime of a value when there exist other values assigned to the same register.

Lemma 8: Let W be the set of values that reside in a register r, and let minlt(v) denote the minimal lifetime of value v (the distance from the producer of v to the last consumer of v). Then each value $u \in W$ has a maximum lifetime equal to

$$\mathrm{II} - \sum_{v \in W/u} \mathrm{minlt}(v).$$

Initially, all values have a minimum lifetime of one clock cycle. The lifetime expression in Lemma 8 is then simplified to II$-(k-1)$, where k equals the number of values assigned to register r. When, for example, II $= 4$, and there are two values in register r, each of these values has a maximum lifetime of $4-(2-1)=3$ clock cycles. When three values reside in r, the maximum lifetime becomes two clock cycles. This

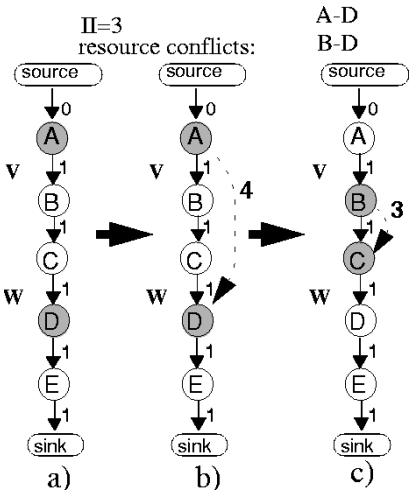


Fig. 13. Derivation of a partial schedule.

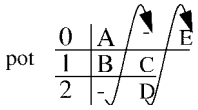


Fig. 14. Folded ASAP schedule for Fig. 13.

maximum lifetime is modeled as a sequence edge with weight-maxlt from the consumer to the producer of the value, similar to modeling the latency.

We illustrate the use of these lemmas with the example in Fig. 13. It is similar to the example of Fig. 1, but it is extended with a register binding. Value v, communicated from operation A to B, and value w, communicated from operation C to D, are bound to the same register. The same resource conflicts and the same initiation interval are used, but there is no constraint on the latency. The first step from (a) to (b) is the same as the first step in Fig. 6.

From Fig. 13(b) to (c), the value v is produced by A and consumed by B. Value w is produced by C and consumed by D. Because of Lemma 7 and d(A, D) $\geq 4 = 1 \times \text{II} + 1$, we can add a sequence edge (B, C) with weight $1 \times \text{II} = 3$ without excluding any feasible schedules.

In Fig. 14, a folded ASAP schedule is given that satisfies the newly added precedence constraints, and thus also the resource constraints and the register binding. In Fig. 14, the leftmost column indicates the time potential (schedule time modulo II), so operation C is scheduled in clock cycle 4, D in clock cycle 5, etc. Notice that the constraints have forced a gap of two clock cycles between operations B and C. A greedy scheduling approach does not put gaps between operations and would never have found a schedule that satisfies all constraints.

In Fig. 15, it is proven that operations A, B, C, and D are actually fixed at their schedule times given in Fig. 14. Fig. 15(a) shows a sequence edge (C, B) with weight $-\text{II} = -3$ as a result of modeling the loop-folding constraint as given in Section V-A. It is also a special case of Lemma 8, where r contains only one value.

From Fig. 15(a) to (b), the sequence edge generates a path from C (producer of value w) to B (consumer of value v) with distance $-3 \geq -2 \times \text{II} + 1 = -5$. Because of Lemma 7, we

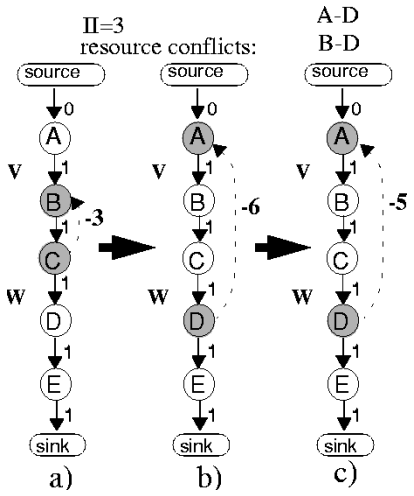


Fig. 15. Derivation of Fig. 13 continued.

can now add a sequence edge from D (consumer of value w) to A (producer of value v) of weight $-2 \times \text{II} = -6$.

From Fig. 15(b) to (c), there is now a path from D to A of distance $-6 = -2 \times \text{II}$. Because A and D have a resource conflict, Lemma 1 states that the distance is increased by one clock cycle. Accordingly, a sequence edge (D, A) with weight -5 is added.

As a result of this last sequence constraint, operation D cannot be scheduled further than five clock cycles from operation A, which is also the minimum distance because of the sequence edge from B to C of weight three. The intermediate operations (B and C) are also fixed in this way. Only operation E can be scheduled at clock cycle 6, 7, or 8.

We have now covered the basic techniques used in the constraint analyzer of Fig. 2. Note that these techniques do not guarantee that every conflict is solved (that all lifetimes of values in the same register are serialized); especially when the schedule is not pipelined, the constraints are often not sufficient to eliminate every conflict. In such a case, a schedule decision has to be made to serialize two value lifetimes, which is the subject of the next section.

C. Lifetime Sequencing

Suppose we have a value conflict between value $v1$, produced by operation P1 and consumed by C1, and value $v2$, produced by operation P2 and consumed by C2. We distinguish two situations:

- nonpipelined schedules;
- pipelined schedules.

In the first situation, the lifetime sequencer has to solve a value conflict by choosing either C2 $\rightarrow$ P1 or C1 $\rightarrow$ P2. In the pipelined situation, the iteration index must be considered as well: the alternatives are $\text{C2}_i \rightarrow \text{P1}_{i+k}$ and $\text{C1}_{i+k} \rightarrow \text{P2}_i$ for possibly more than one value of k. This is illustrated in Fig. 11 for $k \in \{-1, 0, 1\}$.

Nonpipelined blocks are sometimes large (>1000 operations), and constraints are not tight. This has two effects:

- many decisions have to be made;
- a lot of schedule freedom is available.

An actual branch and bound approach does not seem appropriate in this case: the number of decisions are too large to guarantee reasonable runtimes, and because of all the available schedule freedom, a heuristic approach suffices. Although it is not guaranteed that a feasible schedule is found in this way (in this case the values are simply separated), we have not yet encountered infeasibility in practice. Therefore, we choose one of the sequentializations by reusing the schedule procedures applied in the actual scheduler (in Fig. 2) so that our approach is maximally tuned to the existing design flow. Since our approach is being integrated in the Mistral2 [4] compiler, the ASAP values of P1 and P2 determine the highest priority. In Section IX, we included an experiment showing the effects of sequencing lifetimes for a nonpipelined schedule.

For pipelined schedules, the reverse is true: pipelined loops consist of relatively few operations (typically $<$200), and the constraints are much more tight (all lifetimes are restricted to II, resource constraints and value conflicts cross the loop iteration boundaries, etc.). As a result, only a few actual schedule decisions have to be made (typically $<$10). The pipelined benchmarks in Section IX required at most two decisions. In this case, a branch and bound approach is runtime efficient. Such an approach is also required because in the context of different types of very tight constraints, the effects of a schedule decision are very difficult to anticipate, and we are likely to make a "wrong" decision. For the same reason, we do not want to reuse the schedule procedures of the actual scheduler in Fig. 2: the constraints are simply too tight to take any optimization criteria into account. Instead, our first choice is determined by the alternative ($C2_i \rightarrow P1_{i+k}$ or $C1_{i+k} \rightarrow P2_i$ for some k) that reduces the mobility of P1, C1, P2, and C2 the least. Note that there is no actual "cost function" involved in this branch and bound approach: the detection of infeasibility (violation of the constraints) determines when to backtrack.

The infeasibility analysis is able to assist in selecting the decision to backtrack: in Section VII-B, it is explained in detail that infeasibility is detected as a positive delay cycle in the precedence graph. A decision is backtracked *only* when it is part of such a cycle, because only then may it be inconsistent with the constraints or previously made decisions.

VII. Register Binding

In this section, we cover two blocks from our global approach of Fig. 2 that are related to the register binding. The first is the initial binding, addressed in Section VII-A, and the second is the infeasibility analyzer, addressed in Section VII-B.

A. Initial Binding

It is clear from Fig. 16 that an initial register binding has to be made to start the iteration of the constraint analyzer, given the binding of values to register files. We choose the binding such that each register file holds one register. In this way, all values bound to a registrable r need to have their lifetimes sequentialized. This choice is made for two reasons. First, it produces the least hardware when ASIC's are concerned, and

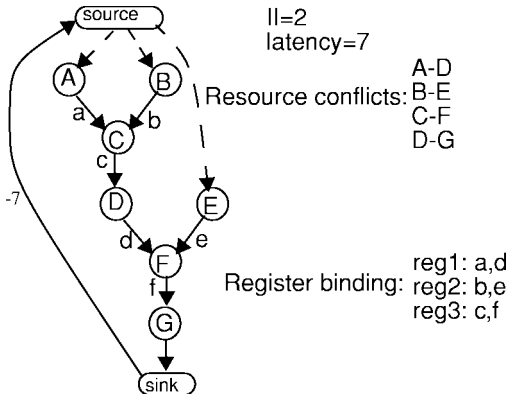


Fig. 16. Example of a precedence graph.

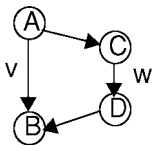


Fig. 17. v and w cannot be in the same register.

provides useful user feedback when programmable platforms are concerned. Second, when the constraints are more tight, the constraint analyzer generates more precedence constraints, so it is better able to guide the scheduler toward a feasible solution.

Starting from this minimum binding, some changes can be made trivially based on the hierarchy of basic blocks. For example, if value v is produced before loop l and consumed after loop l, it occupies a register during the entire execution of loop l. During the analysis of loop l, a register is therefore reserved for value v. Another trivial decision is based on dataflow. For example, in the precedence graph in Fig. 17, values v and w cannot reside in the same register because the value lifetimes cannot be sequentialized.

B. Infeasibility Analysis

The schedule analysis is often capable of detecting that the register binding together with the constraint set yields an infeasible result. In order to make a sensible change in the register binding, we want the infeasibility analyzer to identify the bottleneck in the register binding. More precisely, we want the analyzer to give a *smallest infeasible subset of value conflicts*, that is, a subset of value conflicts (two values residing in the same register) that together cause infeasibility. Identifying such a subset of decisions is tightly related to detecting infeasibility. The constraint analyzer detects infeasibility based on longest path information in the following way: when the longest path algorithm finds a path from an operation v to itself (a cycle in the precedence graph) and this path has a positive length, the operation v is forced to execute strictly before its own start time, which is clearly not possible. So a precedence cycle of strictly positive length indicates infeasibility.

The bottleneck lies directly in the way that the positive length cycle came into existence. For example, if in Fig. 13 the latency was constrained to six clock cycles, there was a sequence edge from the sink to the source with a delay of

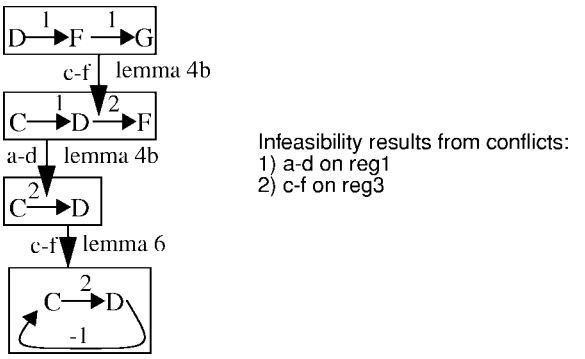


Fig. 18. Infeasibility analysis for Fig. 16.

-6 clock cycles. In Fig. 13(c), that would yield a positive delay cycle. Most edges in the precedence cycle involve data precedence, one involves the latency, and one involves a register conflict. The sequence edge B $\rightarrow$ C is a result of two components: 1) the register conflict $v - w$ and 2) a path of length four from A to D. The path from A to D consists of one sequence edge that is added as a result of the resource conflict A–D and a path A $\rightarrow$ D of length three that consists entirely of data precedence. We can thus conclude that infeasibility is caused as a result of the following combination of factors:

1) a register conflict $v - w$;
2) a resource conflict A–D;
3) the latency constraint;
4) data precedence.

When all constraints are fixed except for the register binding, we conclude that the decision to put the values v and w together in a single register is the cause of infeasibility.

Another example is the graph depicted in Fig. 16. The constraint set is infeasible with the register binding, which is derived as follows. The infeasibility analysis is graphically depicted in Fig. 18. Each block represents a path, and each downward arrow represents an inference. The derivation is top down. The path D $\rightarrow$ G of length two (= II) and register conflict $c - f$ lead to the sequence edge D $\rightarrow$ F of weight II $= 2$ as a consequence of Lemma 6 (where $k = 1$). The downward arrows show that this sequence edge is part in the path underneath. The second block from the top indicates a path C $\rightarrow$ F of length three. Together with the register conflict $a - d$, this yields a sequence edge C $\rightarrow$ D of weight two as a result of Lemma 6. In the block at the bottom of Fig. 18, the sequence edge D $\rightarrow$ C of weight -1 is generated as a result of Lemma 8. (value c and f in the same register limits their lifetimes to II$-(2-1) = 2-1 = 1$ clock cycles). The same block shows that this sequence edge causes a positive precedence cycle C $\rightarrow$ D $\rightarrow$ C with a delay of $2 + (-1) = 1$ clock cycle. As a result of this positive precedence cycle, we conclude that the register binding is infeasible.

The infeasibility analysis is done in bottom-up fashion to identify exactly those sequence edges and conflicts that have contributed to the positive precedence cycle. The combination of register conflicts that yield infeasibility is identified as:

1) $a - d$ on register 1;
2) $c - f$ on register 3.

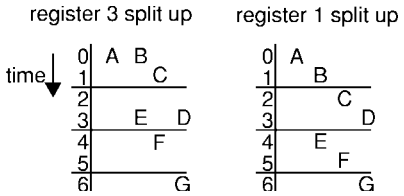


Fig. 19. The only two feasible schedules for Fig. 16 with changes in the register binding.

Note that the conflict $b - c$ on register 2 did not contribute to the infeasibility, and thus it is useless to put the values b and c in separate registers. Instead, we have to choose to split either register 1 or register 3. Both decisions yield a feasible schedule, as depicted in Fig. 19.

C. Rebinding

The infeasibility analysis generates a list of value conflicts. A conflict arises between two values. The list of value conflicts is ordered on a number of criteria.

- The number of times the conflict appears in the conflict list. When a conflict occurs more often in the list, the conflict contributes more extensively to the bottleneck.

For ASIC's, it is ordered on the following.

- The type of the values; we prefer allocating an additional 6-bit register to an additional 28-bit register.
- Addressability; when a register file contains four registers, allocating an additional register requires an additional addressing bit in the instruction word. We prefer to extend a register file with three or five to seven registers.

For ASIP's, it is ordered on the following.

- Availability of registers; we prefer to move values within a register file that contains more spare registers.

After the conflict list is ordered, only the top conflict is chosen. One of the two conflicting values is then allocated to the next register. In this way, convergence is guaranteed. The disadvantage is that the same value conflicts may arise in subsequent iterations of the scheme in Fig. 2. Because this has not proven to be a problem on our problem instances, no work has been done to overcome the disadvantage.

VIII. Complexity

In this section, we analyze the runtime complexity and memory requirements of our approach. The two major contributions to runtime are:

- finding the longest paths and updating the paths as a result of applying the lemmas from Sections V and VI;
- infeasibility analysis and changing the register binding.

A. Finding and Updating the Longest Paths

We will first consider the complexity of the former contribution. In our implementation, the longest path between each pair of operations is administrated. The memory requirements thus have order $O(V^2)$.

If a new edge is added, the impact on the current longest paths has to be calculated. Therefore, the complexity of adding a sequence edge is the dominant factor in runtime. This complexity is essentially determined by the number of paths that need to be updated as a result of the new sequence edge. Because we are only interested in the longest paths found so far, the number of updates equals V^2 in the worst case. In most cases, the addition of a sequence edge will affect a few paths. In cases where many paths need to be updated, the estimates of schedule intervals will also be improved substantially.

An upper bound on the number of path updates (as a result of adding a sequence edge) can be derived as follows. A path can have a length between $-l$ and $+l$, where l is the constraint on the latency. Because a path is updated only if its length is increased (by at least one clock cycle), the number of times a path can be updated is at most $2l$. Since the maximum number of paths we keep track of equals V^2, the number of path updates can be at most $2l \cdot V^2$. A single path update takes constant time, so the runtime of the constraint analysis is polynomially bounded.

B. Infeasibility Analysis and Rebinding

As the reader may have noticed in the examples, the infeasibility analysis requires a lot of administrative bookkeeping. Almost every path constructed during the longest path analysis has to be kept in memory for reference. A feasible implementation requiring a limited amount of memory to run an implementation of our method is only guaranteed if the storage of a path has a memory cost of $O(1)$. This is possible with the use of an *adjacency matrix* [25], which is based on the following fact of longest paths: if the longest path from A to C travels through B, then the part B to C is the longest path from B to C. As a result, the only administration necessary for the path from A (row of the matrix) to C (column of the matrix) is the first node on the path after A. To facilitate the infeasibility analysis, we also administrate the first edge traversed on the path A to C. Each sequence edge on its turn has a pointer to a register conflict (if there is one) and the matrix entry representing the path that gave rise to the edge. The complexity of the infeasibility analysis is thus bounded by $O(E \cdot \log E)$. We assume, however, that the longest paths have already been calculated in the constraint analyzer.

The complexity of rebinding is determined by the procedure of ordering the conflict list as explained in Section VII-C. Because a value conflict gives rise to a sequence edge, the number of conflicts in the list cannot exceed the number of edges in the precedence graph. Therefore, the complexity of rebinding is bounded by $O(E \cdot \log E)$.

We conclude that the complexity of one iteration of the scheme in Fig. 2 equals $O(2l \cdot V^2 + E \cdot \log E)$. In the worst case, the number of iterations is bounded by $|Y|$, the number of values in the dataflow graph. In the results section, we will also depict the iteration count for the different applications.

IX. Results

Our implementation on an HP 9000/735 has been tested on the inner loops from four different real-life industrial examples that have been mapped on a VLIW architecture with distributed register files. The results are shown in Table I. The fourth column represents the number of iterations over the constraint analyzer (see Fig. 2) before a feasible solution was found. The last two columns indicate the mobility of the operations in terms of average number of clock cycles per operation (Definition 10). The sixth column indicates the mobility before the analysis; the last column, after analysis (what is left for the scheduler to fill in). With respect to the numbers in Table I, no comparison could be made to other approaches because the register allocator and the schedulers available to us (several list schedulers) are unable to find any solution for the given constraints.

TABLE I
Results of Experiments

experiment	|V|	II	#iterations	run time	mobility before analysis	mobility after analysis
IIR	23	6	3	0.2 s	2.70	0.13
FFTa	40	4	11	17 s	4.46	0.46
FFTb	60	8	20	25 s	6.85	0.52
Rad4	81	4	1	0.8 s	4.93	1.38

The first experiment concerns an infinite impulse response (IIR) filter of 23 operations, including fetching the coefficients and data from memory. The minimum latency is ten clock cycles, which equals the latency constraint. The other experiments concern fast Fourier transform (FFT) applications, the largest of which holds 81 operations. Note in Table I that the runtimes are mainly determined by the number of iterations over the constraint analyzer. The number of iterations is a measure of the difficulty of finding a register binding because it reflects the number of changes made to the original binding in order to get a feasible schedule. In these experiments, the schedule generated by our method provided a more efficient register binding than a handmade schedule. Analyses of the minimal value lifetimes suggested that little or no improvement could be made on the generated register binding.

The mobility is decreased by a factor ranging from 3.6 (Rad4) to 13.2 (FFTb) as a result of the schedule analysis. Because this decrease of mobility is due to the constraints, it is a measure for the analyzers' capability of directing the scheduler and preventing it from making schedule decisions that violate the constraints.

We have included one more experiment to test the performance of our method on a problem instance that was not constrained with respect to timing. It is a preliminary test executed by Frontier Design, who are integrating our method within the Mistral2 toolset. The benchmark, Par2, contains 91 operations. The original schedule, generated by the Mistral2 toolset, counts 61 clock cycles. As a result of the available parallelism and the number of memory accesses, the register binder required six registers at the address-generation unit. The schedule generated by our method counts only 56 clock cycles and requires only one register at the address-generation unit. Because of the schedule freedom, a total of 111 schedule decisions had to be made by the lifetime sequencer. Runtime was less than a second. The efficient register binding of the

new schedule was expected (it was enforced), unlike the reduction in the number of clock cycles. This reduction is explained as follows: because of the serialization of the address lifetimes, the precedence graph became more regular. It is a well-known fact that heuristics such as the list scheduling are able to find more efficient schedules when the precedence graph contains more regularity.

X. Conclusions and Further Research

In this paper, we presented an approach for register binding and scheduling in the context of loop pipelining, based on the analysis of precedence, timing, and resource constraints. By expressing as much of the constraints as possible in a graph model and calculating the longest paths, we are able to see the interaction between the different constraints and compute the effect on the mobility available to a scheduler. When the combination of constraints and the register binding is infeasible, an efficient infeasibility analyzer is able to indicate a change in the binding that is necessary to obtain a feasible schedule. The results in Section IX show that our method is able to find a register binding and a pipelined schedule in short runtimes for industrially relevant designs. We also showed that the obtained reduction in mobility really prevents a greedy scheduler from making a wrong decision. When constraints are not very tight, we are still able to find more efficient schedules than heuristics. We conclude that analysis tools such as our implementation are needed in order to obtain a feasible schedule when facing resource constraints, register constraints, and tight timing constraints. Our method is being integrated in the Mistral2 toolset by Frontier Design.

Further research will focus on the analysis of other register-file models, such as first-in, first-out and stacks.

Acknowledgment

The authors would like to thank M. Strik, K. van Eijk, and P. Lippens for their support and constructive discussions.

References

[1] R. Leupers, W. Schenk, and P. Marwedel, "Microcode generation for flexible parallel architectures," in *Proc. Working Conf. Parallel Architectures and Compiler Technology*, 1994.

[2] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, "DSP design tool requirements for embedded systems: A telecommunications industrial perspective," *J. VLSI Signal Process.*, vol. 9, no. 1, 1995.

[3] P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors.* Boston, MA: Academic, 1995.

[4] M. T. J. Strik, "Efficient code generation for application domain specific processors," Eindhoven University of Technology, The Netherlands, Tech. Rep. 90-5282-390-1, 1994.

[5] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*. Anaheim, CA: ACM and IEEE Computer Society, 1988, pp. 330–336.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, CA: Freeman, 1979.

[7] G. Goossens, J. Vandewalle, and H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems," in *Proceedings of the 26th ACM/IEEE Design Automation Conference.* Las Vegas, NV: ACM and IEEE Computer Society, 1989, pp. 826–831.

[8] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala, "FlexWare: A flexible firmware development environment for embedded systems," in P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors.* Boston, MA: Academic, 1995.

[9] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor descriptions," *Design Automation Embedded Syst.*, vol. 3, no. 1, 1998.

[10] D. Lanneer, J. van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, "Chess: Retargetable code generation for embedded DSP processors," in P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors.* Boston, MA: Academic, 1995.

[11] M. T. J. Strik, J. L. van Meerbergen, A. H. Timmer, and J. A. G. Jess, "Efficient code generation for in-house DSP-cores," in *Proceedings of the European Design and Test Conference*. Paris, France: IEEE Computer Society Press, 1995, pp. 244–249.

[12] C. Liem, T. May, and P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation," in *Proceedings the European Design and Test Conference*. Paris, France: IEEE Computer Society Press, 1997, pp. 31–37.

[13] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proc. Ann. Workshop Microprogramming*, Oct. 1981, pp. 183–198.

[14] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1988, p. 328.

[15] A. Aiken, A. Nicolau, and S. Novack, "Resource-constrained software pipelining," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 1248–1270, Dec. 1995.

[16] C. T. Hwang, Y. C. Hsu, and Y. L. Lin, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 464–475, Apr. 1991.

[17] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," in *Proc. SIGPLAN Conf. Programming Language Design and Implementation*, June 1992, pp. 283–299.

[18] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham, "Optimum modulo schedules for minimum register requirements," in *Proc. Int. Conf. Supercomputing*, Barcelona, Spain, July 1995, pp. 31–40.

[19] R. Govindarajan, E. R. Altman, and G. R. Gao, "Minimizing register requirements under resource-constrained rate-optimal software pipelining," in *Proc. Symp. Microarchitecture*, Nov. 1994, pp. 85–94.

[20] A. H. Timmer, M. T. J. Strik, J. L. van Meerbergen, and J. A. G. Jess, "Conflict modeling and instruction scheduling in code generation for in-house DSP cores," in *Proceedings of the 32nd ACM/IEEE Design Automation Conference*. San Francisco, CA: ACM and IEEE Computer Society, 1995.

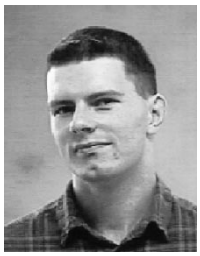
[21] D. C. Ku and G. De Micheli, Eds., *High Level Synthesis of ASIC's Under Timing and Synchronization Constraints.* Norwell, MA: Kluwer Academic, 1992.

[22] R. Reiter, "Scheduling parallel computation," *J. ACM*, vol. 15, pp. 590–599, 1968.

[23] B. Mesman, M. T. J. Strik, A. H. Timmer, J. L. van Meerbergen, and J. A. G. Jess, "Constraint analysis for DSP code generation," in *Proc. Int. Symp. System Synthesis*, Antwerp, Sept. 1997.

[24] ———, "A constraint driven approach to loop pipelining and register binding," in *Proceeding of the Design Automation and Test in Europe*. Paris, France: IEEE Computer Society Press, 1998.

[25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: MIT Press, 1990.



Bart Mesman received the Electrical Engineering degree (with honors) from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1995, where he currently is pursuing the Ph.D. degree.

His doctoral work is on the subject of scheduling for embedded DSP processor architectures with the explicit goal of codesigning processor architectures and a code-generation methodology based on constraint analysis. Since 1995, he has been a Member of both the Digital VLSI Group at Philips Research, Eindhoven, and the Information and Communication Systems Group of the Electrical Engineering Department at the University of Technology, Eindhoven. His research interests include high-level synthesis, ASIP architectures, and code generation for embedded DSP's.

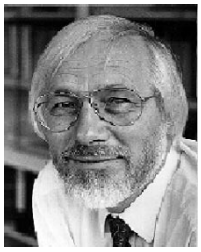
Adwin H. Timmer received the Electrical Engineering and Ph.D. degrees from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1990 and 1996, respectively.

In 1995, he joined Philips Research Laboratories, Eindhoven. In 1998, he was a Visiting IC Architect with the Philips Semiconductors WSG business line, Mountain View, CA. His current interests are in IC architectures for high-performance signal-processing applications, system-level design methods, hardware/software codesign, and compilation techniques for embedded DSP's.

Jef L. van Meerbergen received the Electrical Engineering and Ph.D. degrees from the Katholieke Universiteit Leuven, Belgium, in 1975 and 1980, respectively.

In 1979, he joined Philips Research Laboratories in Eindhoven, The Netherlands. He was engaged in the design of MOS digital circuits, domain-specific processors, and general-purpose digital signal processors. In 1985, he began working on application-driven high-level synthesis. Initially, this work was targeted toward audio and telecom DSP applications. Later, the application domain shifted toward high-throughput applications (Phideo). His current interests are in system-level design methods, heterogeneous multiprocessor systems, and reconfigurable architectures. He is a Philips Research Fellow and, since 1998, a Professor at the Eindhoven University of Technology. He is an Associate Editor of *Design Automation for Embedded Systems.*

Dr. van Meerbergen received the Best Paper Award at the 1997 ED&TC conference.



Jochen A. G. Jess received the master's and Ph.D. degrees from Aken University of Technology, Germany, in 1961 and 1963, respectively.

He became a Full Professor of electrical engineering at the Eindhoven University of Technology, Eindhoven, The Netherlands, in 1971. For a number of years, he had various research and teaching appointments at Karlsruhe University of Technology, where he was one of the founders of the Computer Science Department. During 1968–1969, he spent a sabbatical year at the University of Maryland, College Park. In Eindhoven, he was involved in founding and running the Design Automation Section. This task implied devising a long-term research program in the area of VLSI design automation and complementing it with the necessary curricular components. His is a coauthor of about 125 papers. He guided 31 Ph.D. students to graduation. In 1985, he joined IBM T. J. Watson Laboratories, Yorktown Heights, NY, for a short period to contribute to a silicon compilation path for the 801 RISC pipeline in a project guided by R. K. Brayton and R. Otten. Recently, his interest has focused on hardware platforms for multimedia systems and the problems of compilation for performance when mapping high-performance, real-time tasks onto those platforms. He is member of the board of the European Design Automation Association and has been its Chairman for a number of years. He is a Cofounder of the Design Automation and Test in Europe conference. He was Program Chair and General Chair, respectively, of ICCAD-93 and ICCAD-94.