A CONSTRAINT DRIVEN APPROACH TO LOOP PIPELINING AND REGISTER BINDING

Bart Mesman[1,2], Marino Strik[1], Adwin H. Timmer[1], Jef L. van Meerbergen[1] and Jochen A.G. Jess[2]

[1]Philips Research Laboratories, WAY4, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
[2] Section ICS, Department of Electrical Engineering, Eindhoven University of Technology, The Netherlands

Abstract

Code generation methods for 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 resource- and timing constraints. The analysis identifies sequencing constraints between operations additional to the precedence constraints. Without the explicit modeling of these sequencing constraints, a scheduler is often not capable of finding a solution that satisfies the timing, resource and register constraints. The presented approach results in an efficient method of obtaining high quality instruction schedules with low register requirements.

1 Introduction

In recent surveys [14], the most significant trend indicated by DSP design groups and embedded processor users is the increasing use of application domain specific instruction set processors (ASIPs) [7] as a key design building block. ASIPs are tuned towards specific application domains and have become popular due to their advantageous trade-of between flexibility and cost. Because of the importance of time-to-market, software for these ASIPs is preferably written in a high-level programming language, thus requiring the use of a compiler In this paper we will address one of the compiler issues that have not been addressed thoroughly yet: the problem of register binding and scheduling under timing constraints. The reason is that most of the currently available software compiling techniques have originally been developed for General Purpose Processors (GPPs), which have characteristics different from those of ASIPs:

- GPPs most often have a single large register file, accessible from all functional units, thus providing a lot of freedom for both scheduling and register allocation. ASIPs usually have a distributed register file architecture (which increases access bandwidth) accompanied by special-purpose registers. Register allocation is severely hampered by this type of architecture.
- ASIPs are mostly used for implementing DSP functionality that enforce 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 making a trade-of between compile time and code quality. Typically, GPP software should compile quickly and code quality is of minor importance. 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 from 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 run-time 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 pragmas to help the scheduler satisfy the constraints. Furthermore, in order to obtain higher utilization rates for the resources and to satisfy the timing constraints, software pipelining [2], also called loop pipelining or loop folding, is required. Previously [15], we showed that a heuristic like list scheduling for loop pipelining is unable to satisfy the timing and resource constraints even for simple examples.

Rau et al. [11] successfully perform register binding tuned to pipelined loops. They mention that for better code quality “Concurrent scheduling and register allocation is preferable”, but for reasons of run-time 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. [12] solve some of

the shortcomings of the approach used by Govindarajan et al. [13], but both try to solve the entire problem using an ILP approach, which is computationally too expensive for practical instances of the scheduling and register allocation problem. Summarizing,

> 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, and 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. [4] successfully performed constraint analysis on a schedule problem using bipartite matching, but this work is difficult to extend to register constraints. Instead, this paper extends our previous work [15]. This work is based on finding the longest paths in the precedence graph. Necessary timing constraints are added as a result of resourceconflicts. In this paper we provide necessary additonal timing constraints to sequentialize value lifetimes.

In Section 2 the problem statement is given, and a global solution strategy is proposed. In Section 3.1 we describe the method of analysis for non-folded schedules. Section 3.2 generalizes the analysis to include loop folding. Section 4 shows how the results of analysis are used to determine a register binding and in Section 5 some results will be presented.

2 Problem statement and approach

Before the problem is stated, let us briefly discuss the assumptions made:

- All operations have been mapped to functional units. This is often the case because instruction selection is done prior to the scheduling phase (see for example [10]), thus providing a resource binding.
- All values have been mapped to register files. In ASIP-architectures, a register file is often bound to a functional unit or to a specific use, both of which are fixed after instruction selection. Within a register file there are multiple registers however, and the assignment of values to these registers remains to be performed.
- The controller is microcoded. One consequence is that in a folded loop a value cannot reside in a certain register for a period longer than the *initiation interval*, which is the period of initiating the schedule for a loop iteration. Another restriction is that a loop-body execution is the same for each loop index. This is not the case in e.g. the Phideo toolset [3], where potentially better schedules can be obtained.
- The initiation interval II for each hierarchical level is fixed prior to scheduling. Most often it is set by the designer. Otherwise, we start with a lower bound based on loop-carried dependencies [9] 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.
- A DSP algorithm is represented as a hierarchical cyclic directed precedence graph, as given in Figure 1. It consists of a set V of vertices representing operations, and a set $A \subseteq V \times V$ of arcs, called precedence edges or sequence edges, representing precedence relations between operations. For $\in$ let $s(v)$ denote the start time of operation v. An arc (v_i, v_j) with weight d indicates that $\geq$. Two dummy operations are added to the precedence graph: a source and a sink. The source operation is always the first operation to execute, and the sink the last one. In this graph model, a restriction of II clock cycles on the lifetime of a value x, produced by operation A and consumed by operation B is represented using an edge B->A with delay -II. Similarly, a restriction l on the latency is modelled using an edge from sink to source with delay equal to -l (Figure 1). With these edges the precedence graph is cyclic and connected. Details can be found in [15].

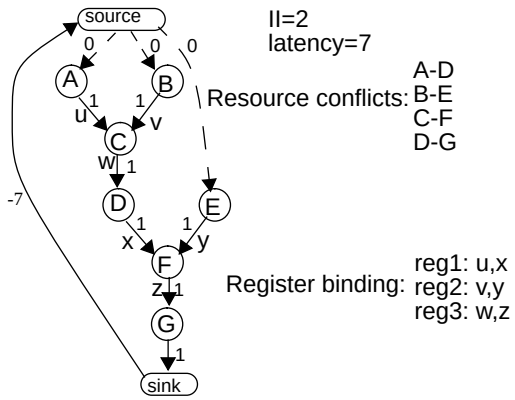


Figure 1 Example of a precedence graph

Our general problem statement for finding a feasible schedule and register assignment, is as follows.

> **Problem 1**: Given a cyclic signal flow graph (SFG), a binding of operations to functional units, a set of resource conflicts (including bus access, memory access, and instruction conflicts), a binding of values to register files, a latency, and an initiation interval (II), find a schedule and an assignment of values to registers.

Because it is difficult to make a register binding and a schedule simultaneously, we decompose the problem in three separate phases as depicted in Figure 2. The central part, the schedule analyzer, generates additional precedence constraints that are implied by the combination of all constraints. The new precedence constraints are such that the register binding is guaranteed: all lifetimes between values residing in the same register have been sequentialized. We have thus completely replaced the

register-binding constraints by precedence constraints. 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 schedule analyzer, a schedule has always been found in practice. As the scheduler and its heuristics are not critical in this approach, we will not focus on them in this paper.

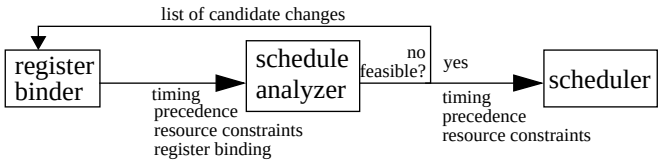


Figure 2 Global approach

Note that a main characteristic of our approach is that we perform register binding prior to schedule analysis. The reason for this is related to the mechanics of our constraint satisfaction approach: when more constraints are provided to the analysis, it becomes more accurate, and a register binding provides additional constraints to the analyzer. Furthermore, the accuracy is increased most when the method of analysis can exploit the interactions between the various types of constraints. Therefore we want to analyze these interactions in a single model, so all the different types of constraints are integrated in one single model (the precedence graph).

There is however a problem when the register binding is performed prior to the schedule analyzer: because the value lifetimes are not yet fixed, a binding decision may be taken that inevitably yields an infeasible result. It is therefore necessary that the schedule analyzer is able to indicate a change in the register binding that may yield a feasible constraint set. The problem statement for the schedule analyzer is therefore as follows.

Problem 2: Given a cyclic SFG, a register binding, a set of resource conflicts, a latency, and an initiation interval, find either a partial order of operations satisfying the register binding (if the constraint set is feasible) or a smallest infeasible subset of register-binding decisions.

The schedule analysis is based on finding the longest paths in the precedence graph.

Definition: A path of length d from operation v_i to operation v_j is a chain of precedences v_i -> v_k ->...v_l ->v_j that imply ≥

A path in the graph thus represents a minimum timing delay. For example, in Figure 1 the path A->C->D indicates a minimum timing delay of 2 clock cycles between the execution times of A and D. When the minimum timing delay is not feasible as a result of a resource conflict, a new timing constraint (a sequence edge) is subsequently added to the graph model. For example, D cannot execute exactly 2 clock cycles (= II) after A because the second execution of A would coincide with the first execution of D, whereas A and D have a resource conflict. Therefore the minimum delay between A and D must equal three clock cycles. In previous work [15], we showed how this approach often prevents a scheduler from making wrong decisions. In this paper we wish to extend the scope of the approach to incorporate conflicts as a result of register bindings, and solve problem 2.

In Section 3 we will show how to analyze the register binding constraints. Section 4 provides a method for finding a smallest infeasible subset of register-binding decisions when the binding turns out to be infeasible.

3 Register constraint analysis

3.1 Non-folded schedules

In this section we will show how sequence edges are used to provide necessary and sufficient timing constraints for the scheduler to satisfy the given register binding. In the following, we will give some lemmas that indicate when a sequence edge is necessary to solve a register conflict. These lemmas rely on the concept of distance in the precedence graph.

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

In the following examples a path is indicated using a dashed arc labelled with the length of the path. Sequence edges are dotted. Standard delay (if not labelled) for a sequence edge is zero clock cycles, for a data dependence it is 1 clock cycle.

Lemma 1: Let variable v1, produced by operation p1 and consumed by c1, and variable v2, produced by operation p2 and consumed by c2, reside in the same register. If ≥ we can add a sequence edge (c_1,p_2) with weight 0 without excluding any feasible schedules.

Lemma 1 is illustrated in Figure 3. The variables 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 variable 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 variables.

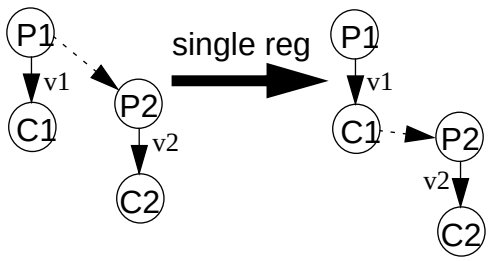


Figure 3 Lemma 1 for sequentializing variable lifetimes

When there is a path between the producer of one variable 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 sequentialization (c2->p1 with delay 0) could still yield a feasible schedule when P1 and C2 are scheduled in the same clock cycle.

Lemma 2: Let variable v1, produced by operation p1 and consumed by c1, and variable v2, produced by operation p2 and consumed by c2, reside in the same register. If ≥ we can add a sequence edge (c_1,p_2) with weight 0 without excluding any feasible schedules.

Lemma 2 is illustrated in Figure 4. The overall method of analysis is demonstrated in Figure 5. In this figure, variables A1 and A2 reside in the same register, as do values B1 and B2. Because operation 1 consumes value A1 and operation 7 consumes value A2, the lifetime of A1 has to precede the lifetime of A2 as a result of the precedence 1->7. Therefore the sequence edge 1->8 is added. Now there is a path 2->1->8 from the consumer of B1 to the consumer of B2. The sequence edge 2->9 is added as a result. Any schedule heuristic can now find a schedule without violating the register binding, which is not true if the sequence edges were not added.

3.2 Folded schedules

When schedules are not folded it is relatively simple to avoid overlapping lifetimes of variables residing in the same register. When loop iterations overlap in time, we also have to take care that the i^{th} lifetime of value v does not overlap with the i+1th lifetime of value w. This means we have to sequentialize value lifetimes belonging to different loop iterations. The graph model however, makes no difference between operation A_i and A_{i+1} (where A_i denotes the i^{th} execution of A), because it has no notion of loop iteration. 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 because , so that the relation ≥ is translated to the relation , which is equivalent to a sequence edge B->A with delay II+d. Lemma 1 is now easily generalized to lemma 3:

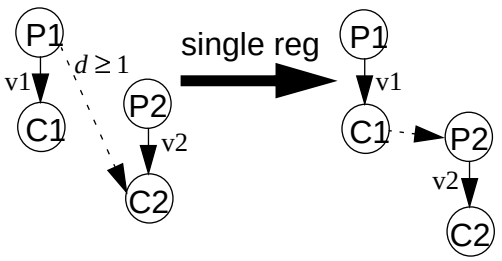


Figure 4 Lemma 2 for sequentializing variable lifetimes

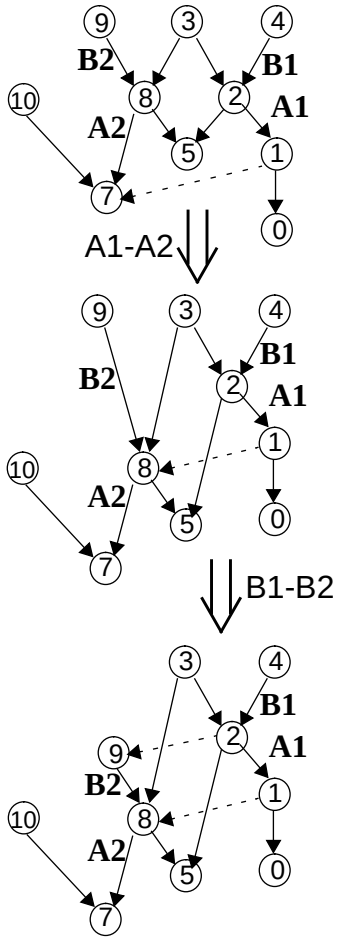


Figure 5 Example demonstrating lemmas 1 and 2

Lemma 3: Let variable v1, produced by operation p1 and consumed by c1, and variable v2, produced by operation p2 and consumed by c2, reside in the same register. If ≥ · we can add a sequence edge (c_1,p_2) with weight ' · '' without excluding any feasible schedules.

Lemma 3 is illustrated in Figure 6.

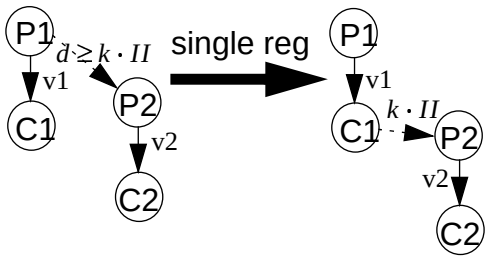


Figure 6 Lemma 3 for sequentializing variable lifetimes

Lemma 2 is generalized to lemma 4:

Lemma 4: Let variable v1, produced by operation p1 and consumed by c1, and variable v2, produced by operation p2 and consumed by c2, reside in the same reg-

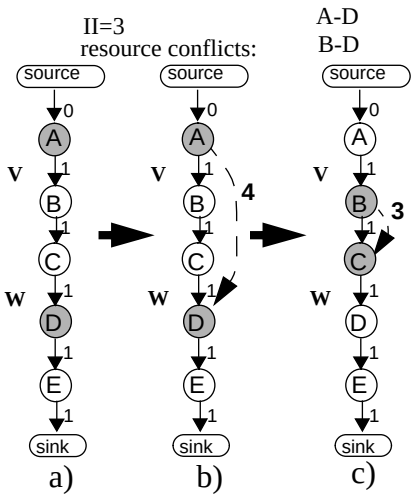


Figure 7 Derivation of a partial schedule

ister. If $d(p_1, c_2) \geq k \cdot II + 1$ we can add a sequence edge (c_1,p_2) with weight ' · without excluding any feasible schedules.

In Figure 7, a partial schedule is derived using lemma 4 for the register conflicts, and a lemma from [15] for the resource conflicts. In this figure, value V is communicated from operation A to B, and value W is communicated from operation C to D. We bound V and W to the same register. The derivation of the schedule is as follows:

from a to b: If the minimum distance of 3 clock cycles between operations A and D is maintained in the schedule, A_1 would coincide with D_0, while they have a resource conflict. Therefore the minimum distance from A to D cannot equal 3 clock cycles, but must be at least 4 (see lemma 1 in [15]). Therefore the sequence edge A->D is drawn.

from b to c: Value V is produced by A and consumed by B. Value W is produced by C and consumed by D. Because of lemma 4 and ≥ = 1 · II + 1 we can add a sequence edge (B,C) with weight · = 3 without excluding any feasible schedules.

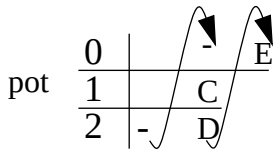


Figure 8 Folded ASAP-Schedule for Figure 7

In Figure 8 a folded ASAP schedule is given that satisfies the newly added precedence constraints, and thus also the resource constraints and the register binding. In Figure 8, the leftmost column indicates the *time potential* (schedule time modulo II), so operation C is scheduled in clock cycle 4, D in 5 etc. Notice that the constraints have forced a gap of 2 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.

4 Register binding

4.1 Initial binding

It is clear from Figure 1 that an initial register binding has to be made to start the iteration of the schedule analyzer, given the binding of values to register files. We choose the binding such that each register file holds 1 register. In this way, all values bound to a register file need to have their lifetimes sequentialized. This choice is made for two reasons: first, it produces the least hardware when ASICs are concerned, and provides useful user feedback when programmable platforms are concerned. Second, the schedule analyzer produces more accurate results when the constraints are more severe.

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, value v occupies a register during the entire execution of loop l. Because the analysis is performed blockwise, the register binder reserves a register for value v during the analysis of loop l. Another trivial decision is based on data flow. In the precedence graph in Figure 9, values v and w cannot reside in the same register because the value lifetimes cannot be sequentialized.

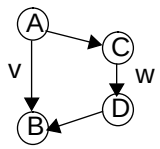


Figure 9 v and w cannot be in the same register

4.2 Infeasibility analysis

When the schedule analyzer detects that the register binding together with the constraint set yields an infeasible result, it should be able to indicate how the register binding must be changed. More precise, we want the analyzer to give a *smallest infeasible subset of register-binding decisions*. That is, a subset of register decisions that together cause infeasibility. Identifying such a subset of decisions is tightly related to detecting infeasibility. The schedule 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 execution time, which is clearly not possible. So a precedence cycle of strictly positive length indicates infeasibility.

The cause of infeasibility lies directly in the way that the positive length cycle came into existence. For example, if in Figure 7 the latency was constrained to 6

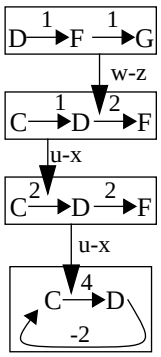
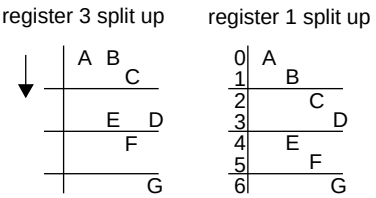


Figure 10 Infeasibility analysis for Figure 1

clock cycles, there was a sequence edge from the sink to the source with a delay of -6 clock cycles. In Figure 7c that would yield a positive delay cycle. Most edges in the precedence cycle involve data precedences, one involves the latency, and one involves a register conflict. The sequence edge B->C is a result of two components: 1) the register conflict V-W, and 2) a path of length 4 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->D of length 3 that consists entirely of data precedences. 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, and 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 Figure 1. The constraint set is infeasible with the register binding, which is derived as follows. The infeasibility analysis is graphically depicted in Figure 10. Each block represents a path, and each downward arrow represents an inference. The derivation is top down. The path D->G of length 2 (=II) and register conflict w-z lead to the sequence edge D->F of weight II=2 as a consequence of a variation of lemma 1 (the path is between the consumers of the conflicting values). The downward arrow show that this sequence edge is part in the path underneath. The second block from the top indicates a path C->F of length 3. Together with the register conflict a-d this yields a sequence edge C->D of weight 2 as a result of the consumers variation of lemma 1. In the third block the conflict u-x is used again with the C->F of length 4 to add the sequence edge C->D of weight 4. The block at the bottom shows that this sequence edge causes a positive precedence cycle C->D->C with a delay 4 + (-2) = 2 clock cycles. The edge D->C with delay -2 is added because the lifetime of each value (in this case value c) cannot exceed II clock cycles, so the consumer (D) must execute within 2 clock cycles after the producer (C). 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 which have contributed to the positive precedence cycle. The combination of register conflicts that yield infeasibility is identified as 1) a-d on register 1 and 2) c-f on register 3. Note that the conflict b-e on register 2 did not contribute to the infeasibility, and thus it is useless to put the values b and e 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 Figure 11.



In our approach a simple heuristic chooses the register conflict to be solved based on the availability over registers in a certain register file, the number of times the conflict appears in the conflict-list, etc.

As the reader may have noticed on 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* [16], 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).

5 Results

Our implementation on a HP 9000/735 has been tested on the inner loops from 4 different real life industrial examples. The results are shown in Table 1. The fifth column represents the number of iterations over the schedule analyzer (see Figure 2) before a feasible solution was found. The last 2 columns indicate the schedule freedom [4] or mobility of the operations in

terms of average number of clock cycles per operation. It is calculated as ALAP (as late as possible) minus ASAP (sa soon as possible), based on the precedence graph and the latency constraint. The 7th column indicates the mobility before the analysis, the last column after analysis (what is left for the scheduler to fill in). W ith respect to the numbers in Table 1 no camparison could be made to other approaches, because the re gister allocater and the schedulers available to us (several list schedulers) are unable to find any solution for the given constraints.

The first experiment concerns an IIR filter of 23 operations, including fetching the coef ficients and data from memory. The minimum latency is 10 clock cycles, which equals the latenc y constraint. The other e xperiments concern FFT applications, the largest of which holds 81 operations. Note in Table 1 that the run-times are mainly determined by the number of iterations over the schedule analyzer. The number of iterations is a measure of the difficulty of finding a re gister binding because it reflects the number of changes made to the original binding in order to get a feasible schedule. In these experiments, the register binding provided by our method improved upon a hand-made schedule. Analyses of the minimal v alue lifetimes suggested that little or no improvement could be made on on the generated register binding.

Table 1 Results of experiments

exper-iment	# oper-ations	II	la-tency	# iter-ations	Run-time	mobility before analysis	mobility after analysis
IIR	23	6	10	3	0.2 s	2.70	0.13
FFTa	40	4	13	11	17 s	4.46	0.46
FFTb	60	8	18	20	25 s	6.85	0.52
Rad4	81	4	11	1	0.8 s	4.93	1.38

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 pre venting it from making schedule decisions that violate the constraints.

6 Conclusions and further research

In this paper, we presented an approach for re gister binding and scheduling in the context of loop pipelining, based on the analysis of precedence, timing and resource constraints. By making all constraints explicit 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 schedule freedom (mobility) available to a scheduler . When the combination of constraints and the re gister binding are 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 5 show that our method is able to find a register binding and a pipelined schedule in short run times for industrially rele vant designs. We also showed that the obtained reduction in mobility really prevents a greedy scheduler from making a wrong decision. We conclude that analysis tools such as our implementation are needed in order to obtain a feasible schedule when f acing resource constraints, register constraints, and tight timing constraints.

Further research will focus on inte grating speculative execution in the model.

References

[1] P.G. Paulin and J.P. Knight, "Force-directed scheduling for the behavioural sythesis of ASIC's", *IEEE Transactions on Computer-Aided Design*, pp. 680-685, June 1989

[2] G. Goossens, J. V andewalle and H. De Man, "Loop optimization in register-transfer scheduling for DSP-systems", *Proc. 26th DAC*, pp. 826-831, 1989

[3] W.F.J Verhaegh, P.E.R. Lippens, E.H.L. Aarts and J.L. van Meerbergen,"Multidimensional periodic scheduling: A solution approach", *Proc. ED&TC 1997*, pp. 468-474, March 1997

[4] A.H. Timmer, M.T.J. Strik, J.L. v an Meerbergen and J.A.G. Jess, "Conflict modelling and instruction scheduling in code generation for in-house DSP cores", *Proc. 32nd DAC*

[5] M. C.SJ. McF arland, A.C. P arker and P . Camposano, "Tutorial on High-level synthesis", *Proc. 25th DAC*, pp. 330-336, 1988

[6] M.R. Garey, D.S. Johnson, "Computers and intractability: A guide to the theory of NP-completeness", Freeman, 1979

[7] R. Leupers, W. Schenk and P . Marwedel, "Microcode generation for flexible parallel architectures", *Proc. Working Conf. Parallel Archit. and Compil. Techn.*, North-Holland, 1994

[8] D.C. Ku and G. De Micheli, "High-le vel synthesis of ASICs under timing and synchronization constraints", Kluwer Academic Publishers, 1992.

[9] R. Reiter, "Scheduling parallel computation", *Journal of the ACM*, vol.15, pp. 590-599, 1968

[10] C.Liem, T.May and P. P aulin, "Instruction-set matching and selection for DSP and ASIP code generation", Proc. ED&TC pp. 31-37, Paris, Feb. 1994

[11] B.R. Rau, M. Lee, P .P. Tirumalai and M.S. Schlansker, "Register allocation for software pipelined loops", Proc. of the SIGPLAN '92 conf. on Programming language design and implementation, pp. 283-299, June 1992

[12] A.E. Eichenberger, E.S. Da vidson and S.G. Abraham, "Optimim modulo schedules for minimum re gister requirements", Proc. of the int. conf. on Supercomputing, pp. 31-40, Barcelona, Spain, July 3-7 1995

[13] R. Govindarajan, E.R. Altman, and G.R. Gao, "Minimizing register requirements under resource-constrained rate-optimal softw are pipelining", Proc. of the 27th int. Symp. on Microarchitecture, pp. 85-94, November 1994

[14] P.G. Paulin, C. Liem, T.C. May, and S. Sutarw ala, "DSP design tool requirements for embedded systems: a telecommunications industrial perspecti e", *J. VLSI Signal Processing*, Vol.9, No.1, 1995

[15] B. Mesman, M.T .J. Strik, A.H. T immer, J.L. v an Meerbergen, and J.A.G. Jess, "Constraint analysis for DSP code generation", *Proc. ISSS'97*

[16] T.H. Cormen, C.E. Leiserson, R.L. Rivest, "Introduction to Algorithms", MIT Press 1990