A mathematical formulation of the loop pipelining problem

Jordi Cortadella, Rosa M. Badia and Fermín Sánchez
Department of Computer Architecture — Universitat Politècnica de Catalunya
08071 Barcelona, Spain
e-mail: {jordic,rosab,fermin}@ac.upc.es

Abstract

This paper presents a mathematical model for the loop pipelining problem that considers several parameters for optimization and supports any combination of resource and timing constraints.

The unrolling degree of the loop is one of the variables explored by the model. By using Farey's series, an optimal exploration of the unrolling degree is performed and optimal solutions not considered by other methods are obtained.

Finding an optimal schedule that minimizes resource and register requirements is solved by using an Integer linear programming (ILP) model. A novel paradigm called *branch and prune* is proposed to efficiently converge towards the optimal schedule and prune the search tree for integer solutions, thus drastically reducing the running time.

This is the first formulation that combines the unrolling degree of the loop with timing and resource constraints in a mathematical model that guarantees optimal solutions.

1 Introduction

Loops monopolize most execution time in programs. In many applications a few loops, if not only one, determine the throughput achievable by the implementation of a behavioral description. For example, DSP filters often consist of an infinite loop that repeatedly executes for every sample of the input stream.

In architectural synthesis, the problem of optimizing loop execution under timing and area constraints is crucial to obtain high quality architectures. The techniques that address this problem attempt to overlap the execution of different loop iterations to reduce the cycle count (*initiation interval* or *II*) per iteration. Different methods have been proposed with such a goal: *loop folding* [1], *functional pipelining* [2], *loop winding* [3] and *rotation scheduling* [4] among others. The area of fixed-rate DSP has also drawn the attention of other authors to propose techniques for loop pipelining with timing constraints [5, 6].

Loops are usually represented by means of a *data dependence graph* (DG). Figure 1 shows an example. Vertices represent operations. Unlabeled edges represent intra-loop dependences (ILDs), e.g. B_i (reads $X[i]$) depends on A_i (produces $X[i]$). Labeled edges represent loop-carried dependences (LCDs), e.g. B_{i+1} (reads $Y[i+1]$) depends on B_i (produces $Y[i+1]$). Labels on edges indicate the number of iterations traversed by the dependence. Thus, ILDs can also be represented as 0-labeled edges.

If no overlap between successive iterations is allowed to execute the loop, the total execution time is $3 \cdot I$ cycles (assuming 1-cycle operations). Within each

for $i = 0$ **to** $I - 1$ **do**
A_i: $X[i] = R[i] + S[i]$;
B_i: $Y[i+1] = Y[i] + 2 * X[i]$;
C_i: $Z[i] = 4 * Y[i+1] + T[i]$;
endfor

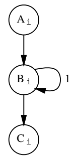


Figure 1: Loop and data dependence graph

iteration, the execution of A_i, B_i and C_i must be sequential due to ILDs. An overlapped execution (loop pipelining) takes $I + 2$ cycles to complete, as shown in Figure 2. The problem of loop pipelining is basically reduced to find a schedule (a folded loop body) that executes at the maximum rate allowed by the dependences. In the schedule, instructions from different iterations (folds) are executed (A_{i+f} denotes that the execution of A_i belongs to fold f).

In general, unrolling the loop is crucial to obtain optimal solutions. If two adders are available for the schedule in Figure 2(b), the loop requires two cycles (*II* $= 2$) to be executed, as shown in Figure 3(a). However, if the loop is unrolled twice (Figure 3(b)), every iteration executes in 1.5 cycles on average (*II* $= 3/2$).

Another important aspect to be considered is the *span* (number of folds required to obtain the schedule). As further commented in Section 4.5, smaller spans result in shorter variable lifetimes, reducing in general the schedule's register pressure.

The loop optimization problem addressed in this paper comprises a large variety of formulations with different timing and resource constraints. The two extreme cases are next described:

- **Resource-constrained loop pipelining** (RCLP). Given a set of resource constraints, to find a schedule that minimizes the execution time.
- **Time-constrained loop pipelining** (TCLP). Given an upper bound on the execution time, the objective is to find a schedule that minimizes the cost of the resources required to execute the loop.

There is a wide range of problems between RCLP and TCLP , e.g. finding a time-constrained schedule with constraints on a subset of resources.

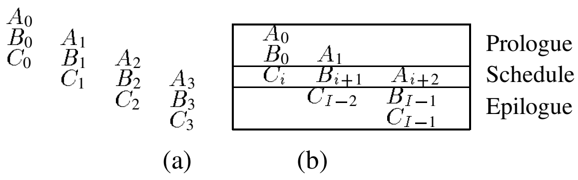


Figure 2: (a) Overlapped loop execution. (b) Schedule

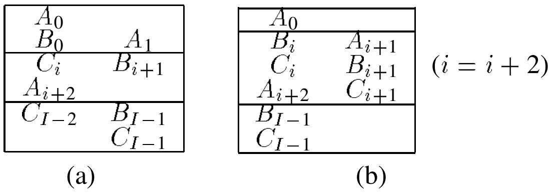


Figure 3: Schedule with resource constraints: (a) without unrolling (b) by unrolling twice the loop.

This paper presents UNRET (unrolling and retiming), a formal approach to solve RCLP. TCLP is addressed in [7]. Since the *delay decision (minimization) problem* of loop pipelining with resource constraints is NP-hard [8], several heuristics have been proposed to solve it in moderate computation time. Several authors have used linear programming to obtain optimal or quasi-optimal solutions for the problems of scheduling and allocation in architectural synthesis [9, 10]. The closest approaches related to the work presented here have been proposed in [11, 12].

The main contributions of UNRET in relation to existing formal methods are next presented:

- UNRET performs an exhaustive analysis of the unrolling degrees of the loop that can derive optimal solutions for the available resources. Unlike other methods that perform loop unrolling [13, 10], this paper presents a new approach that guarantees an optimal unrolling degree.
- Similarly to [11, 12], the number of folds for the schedule is automatically obtained by solving an ILP model for loop pipelining. The number of registers required to execute the loop is reduced by reducing the maximum number of live variables at any cycle.
- A new approach, called *Branch-and-Prune*, is proposed to solved the ILP model. The heuristics devised to explore the space of solutions allow us a rapid convergence to the optimal solution. ILP models with more than 1000 variables and 700 constraints have been solved in a reasonable running time.

The paper is organized as follows. Section 2 presents some preliminary definitions. Section 3 proposes a loop unrolling strategy to find time-optimal schedules for the RCLP problem. The ILP model to find an optimal schedule is presented in Section 4. The branch and prune strategy to efficiently solve the ILP model is described in section 5. Experimental results and conclusions are presented in Sections 6 and 7.

2 Basic definitions

For the sake of simplicity, we will first assume that all operations can be executed in any of the functional units (FUs) of the architecture in one cycle. Extensions to multiple-cycle, pipelined functional units and several types of resources can be found in [14].

2.1 Representation of a loop

A loop is represented by a labeled dependence graph, $DG(V, E)$. Vertices and edges represent operations and data dependences respectively. Labels of the DG are defined by two mappings, λ (fold) and δ (dependence distance), in the following way:

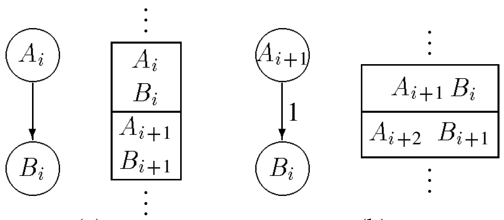


Figure 4: (a) Schedule of a loop with one ILD. (b) Schedule of the same loop with an equivalent labeling function and one LCD.

- $\lambda(u)$, defined on vertices, denotes the fold to which u belongs in the schedule ($\lambda(u) \geq 0$). $\lambda(u) = i$ will be denoted by u_i in the DG.
- $\delta(u, v)$, denotes the dependence distance (number of iterations traversed by the dependence) between operations u and v. An ILD between u and v is represented by $u_i \xrightarrow{0} v_i$ or simply $u_i \rightarrow v_i$. Similarly, an LCD between u and v with distance d is represented as $u_i \xrightarrow{d} v_i$.

Initially, a loop is represented by a DG with only one fold, i.e. $\forall u \in V$: $\lambda(u) = 0$. After finding a schedule, each operation is labeled with a fold representing the relative execution skew (in iterations) with regard to the other operations of the loop.

Equivalent labeling functions can be obtained by simple transformations. Dependence $A_{i+1} \xrightarrow{1} B_i$ (or in general $A_{i+d} \xrightarrow{d} B_i$) is equivalent to $A_i \rightarrow B_i$. This transformation can be used to pipeline the loop, as shown in Figure 4. Note that only ILDs constrain the scheduling process, and therefore DG from Figure 4(b) is more *parallel* than the one shown in Figure 4(a). ILDs can be transformed into LCDs by changing the fold assignment and shorter schedules for the loop body can be found. This transformation is analogous to the *retiming* technique proposed to minimize the clock period in synchronous systems [15].

Definition 1 : Equivalent labeling functions: *Let (λ, δ) and (λ', δ') be two labeling functions for $DG = (V, E)$. They are equivalent if $\forall\ (u, v) \in E$:*

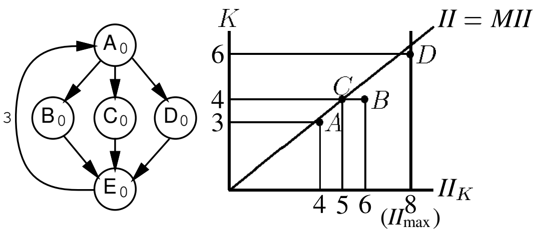
$$\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v) \quad (1)$$

2.2 Initiation Interval

As proposed by other authors [16], UNRET first calculates a lower bound on the *II* of the loop: the minimum initiation interval (*MII*). Two lower bounds on *MII* must be taken into account:

- the *minimum initiation interval imposed by resource constraints* (*ResMII*). If each iteration of the loop requires using an FU during C cycles, and the architecture has N FUs of such a type, then *II* $\geq \lceil \frac{C}{N} \rceil$. Therefore, the FU with the maximum such ratio determines a lower bound on *II*.
- the *minimum initiation interval imposed by the recurrences*[1] *of the loop* (*RecMII*). Let us consider a recurrence R. A feasible schedule must fulfill *II* $\geq \lceil \frac{ET}{D} \rceil$, where ET is the sum of the execution

[1] A recurrence R is a set of edges that form a cycle.



(a) (b)

Figure 5: (a) DG example (b) (II_K,K) pairs for 4 FUs.

times of the operations in R and D is the sum of the distances of its dependences [16]. The recurrence with the maximum such ratio determines another lower bound on *II*.

Let us define δ_R as the sum of the distance of the dependences in a recurrence R. In the example of Figure 5(a), there are three recurrences with the same value for *RecMII*. Let us take "$A_0 \rightarrow B_0 \rightarrow E_0 \xrightarrow{3} A_0$", with $\delta_R = 3$ and $|R| = 3$. This indicates that A_3 must be executed at least 3 cycles after A_0, thus resulting in $RecMII_R = 1$. The other two recurrences are isomorphic to this one.

3 Loop unrolling for RCLP

The length of a schedule is an integer number of cycles, whereas the *MII* of a loop may be a rational number. Let II_K be the initiation interval of a schedule comprising K instances of the loop body ($II = \frac{II_K}{K}$). The goal of UNRET is to find a schedule that minimizes *II*. This is done by exploring pairs (II_K,K) in increasing order of *II*, starting from *MII*. In order to bound the search space for (II_K,K), a maximum value for II_K is defined: the maximum length of the schedule ($II_{\max}$).

Figure 5(a) depicts the DG of a 5-instruction loop. If 4 resources are used for execution, then $RecMII = 1$ and $MII = ResMII = \frac{5}{4}$. The diagram in Figure 5(b) represents the pairs (II_K,K) that can be explored to find a feasible schedule for the loop. These pairs correspond to the points with integer values for II_K and K enclosed in the triangle limited by the lines $K = 0$, $II_K = II_{\max}$ and $\frac{II_K}{K} = MII$. Distinct points with the same *II* lie on the same line. Among all points lying on the same line, those with smaller values for II_K (or K) are preferred (they produce shorter schedules).

In the example, the first point to be explored is $C = (5, 4)$, meaning that 4 iterations must be executed in 5 cycles (*II*=*MII*=1.25). However, no feasible schedule with such characteristics exists. In [16], II_K is incremented by 1 when no schedule is found and, thus, the point $B = (6, 4)$ is next explored. This results in a suboptimal solution. A time-optimal solution (for $II_{\max} = 8$) is found if the point $A = (4, 3)$ is explored after $C = (5, 4)$. But, is there any efficient strategy to explore all points in increasing order of *II* ?

3.1 Farey's series

For a fixed integer $D > 0$, the sequence of all reduced fractions with nonnegative denominator $\leq D$, arranged in increasing order of magnitude, is defined by the *Farey's series* of order D (F_D).

Let $\frac{x_i}{y_i}$ be the ith element of the series. F_D can be generated by using the following equations[17]:

- The first two elements are $\frac{x_0}{y_0} = \frac{0}{1}$; $\frac{x_1}{y_1} = \frac{1}{D}$
- The generic term $\frac{x_i}{y_i}$ can be calculated as:

$$x_i = \left\lfloor \frac{y_{i-2} + D}{y_{i-1}} \right\rfloor x_{i-1} - x_{i-2} \qquad y_i = \left\lfloor \frac{y_{i-2} + D}{y_{i-1}} \right\rfloor y_{i-1} - y_{i-2}$$

For a more detailed explanation of how Farey's series are explored see [18].

3.2 Loop unrolling

Every pair (II_K, K) obtained from Farey's series denotes a different unrolling degree for the loop (K) and a target initiation interval (II_K). Unrolling a DG K times generates another DG in which each vertex v is instantiated K times ($v^0, v^1, \ldots, v^{K-1}$). Besides, data dependence distances must be changed according to the unrolling degree. A dependence $u \xrightarrow{d} v$ in the original DG is represented by K dependences $u^i \xrightarrow{\left\lfloor \frac{i+d}{K} \right\rfloor} v^{(i+d) \bmod K}$ in the unrolled DG. A complete description can be found in [18].

3.3 UNRET algorithm for RCLP

The strategy followed by UNRET is the following:

1. Calculate *MII*.
2. Find the first unrolling degree (K) and expected initiation interval (II_K) that minimizes *II* such that $II \geq MII$.
3. Unroll the loop K times.
4. Find a schedule with length II_K (ILP approach explained in Section 4).
5. If no schedule with II_K cycles is found, generate new values for K and II_K that minimize *II* (*II* is explored in increasing order by using Farey's series) and go to step 3.

4 Loop pipelining: An ILP approach

The problem of loop pipelining can be reduced to two interrelated subproblems that can be simultaneously solved by using an ILP model: folding the DG (finding the functions λ and δ) and finding a schedule for the folded DG subject to the data dependences and resource constraints.

4.1 Preliminaries

Initially, a DG obtained by unrolling the original DG K times will be given. An objective II (II_K) will fix the number of cycles of the schedule. Hereafter, and for the sake of brevity, we will use *II* instead of II_K. $C = \{0, \ldots, II - 1\}$ will denote the set of cycles of the schedule.

All variables used in the model are nonnegative integers. The *least nonnegative residue system* $(0, 1, \ldots, k-1)$ will be used when *modulo k* (mod k) operations are performed on constants.

4.2 Loop folding constraints

The following variables are defined for the labeling functions:

$$\lambda_u, \quad \forall u \in V \quad \text{(fundamental variables)}$$
$$\delta_{u,v}, \quad \forall (u,v) \in E \quad \text{(auxiliary variables)}$$

Folding the loop means finding an equivalent labeling function for the DG. According to equation (1), the auxiliary variables $\delta_{u,v}$ are defined as follows:

$$\delta_{u,v} = \lambda_u - \lambda_v + \lambda(v) - \lambda(u) + \delta(u,v), \quad \forall (u,v) \in E \tag{2}$$

where $\lambda(u)$, $\lambda(v)$ and $\delta(u,v)$ denote the initial labeling of the DG (usually $\lambda(u) = \lambda(v) = 0$).

4.3 Scheduling and data dependence constraints

The following variables are defined $\forall u \in V,\ i \in C$:

$$s_{u,i} = \begin{cases} 1 & \text{if } u \text{ starts executing at cycle } i \\ 0 & \text{otherwise} \end{cases}$$

The following constraint guarantees that an instruction is scheduled at only one cycle of the schedule:

$$\sum_{i \in C} s_{u,i} = 1, \quad \forall u \in V \tag{3}$$

For simplicity, the auxiliary variable c_u will be used to denote the cycle at which u is scheduled. Hence,

$$c_u = \sum_{i \in C} i \cdot s_{u,i}, \quad \forall u \in V \tag{4}$$

Data dependences are honored by the constraint:

$$c_v \geq c_u + T(u) - II \cdot \delta_{u,v}, \quad \forall (u,v) \in E \tag{5}$$

where $T(u)$ is the *execution time* of u.

4.4 Resource constraints

The following constraints guarantee that no more than F_t functional units are used at each cycle:

$$\sum_{u \in A(t)} \sum_{j=i-L(u)+1}^{i} s_{u,(j \bmod II)} \leq F_t, \quad \forall t \in M, i \in C \tag{6}$$

where $A(t)$ is the number of available F_t. An instruction u uses an FU during the cycles $c_u \ldots c_u + L(u) - 1$ (mod *II*). In case the latency of the operation is longer than *II*, the execution of several instances of the same operation may overlap and, therefore, more than one FU may be required.

4.5 Register requirements

The maximum number of variables whose lifetimes overlap at any cycle, MAXLIVE , is the minimum number of registers required for a schedule [19]. For an edge $u \rightarrow v$, the variable lifetime spreads from the completion of u (cycle $c_u + T(u)$) to the cycle in which the FUs executing v does not required the input data anymore (cycle $c_v + L(v) - 1$). The following auxiliary variables are defined to specify whether an operation reads or writes a result before a given cycle:

$$\text{RB}_{u,c} = \sum_{i=0}^{c-1} s_{u,(i-L(u)+1) \bmod II}$$

$$\text{WB}_{u,c} = \sum_{i=0}^{c-1} s_{u,(i-T(u)+1) \bmod II}$$

The auxiliary variable $r_{u,v,c}$ defines the number of registers required to store a result in cycle c produced by operation u and consumed by operation v:

$$\begin{aligned} r_{u,v,c} \quad = \quad & \delta_{u,v} - \left(\left\lfloor \frac{T(u)-1}{II} \right\rfloor + \text{WB}_{u,(T(u)-1) \bmod II} \right) + \\ & \left(\left\lfloor \frac{L(v)-1}{II} \right\rfloor + \text{RB}_{v,(L(v)-1) \bmod II} \right) + \\ & \text{WB}_{u,c} - \text{RB}_{v,c} \end{aligned} \tag{7}$$

The expressions in brackets determine a correction on $\delta_{u,v}$ produced by the execution time of u and the *latency* of v ($L(v)$), i.e. no register is required to store the value while u executes, whereas a register is required during the first $L(v)$ cycles of v's execution.

A variable can be the input of more than one instruction. However, there is no need to allocate different registers for the same variable. The number of registers required is the maximum from all edges with the same source. Therefore,

$$r_{u,v,c} \leq r_{u,c}, \quad \forall u \in V, c \in C \tag{8}$$

It can be easily proved that if operation v is the last use of u's result then $r_{u,v,c} = r_{u,c}$ for any cycle c [12]:

$$\sum_{u \in V} r_{u,c} \leq ML, \quad \forall c \in C \tag{9}$$

4.6 Objective function

Assume we have a cost vector $A = (A_1, \ldots, A_m)$ for the FUs of the architecture and a cost A_r for each register[2]. We formulated the objective function as:

$$min \ \text{Area} = \sum_{t \in M} A_t \cdot F_t + A_r \cdot ML \tag{10}$$

F_t and *ML* can be variables or constants, according to the initial constraints. If all of them are constants, a feasible solution will be found. Here we have considered *ML* as the number of registers required by the schedule. Although this might not be true, it is a realistic assumption for many practical cases [19].

4.7 Complexity of the model

Table 1 describes the fundamental variables and constraints of the model. V and E denote the number of nodes and edges of the DG respectively, whereas m is the number of different FU types of the architecture. Some of the variables, e.g. F_t and *ML*, may become constants if the number of resources of the architecture is defined in advance.

[2]Piecewise linear cost functions for registers can also be incorporated, as proposed in [20].

variable	number
λ_u	V
$s_{u,i}$	$V \cdot II$
F_t	m
$r_{u,c}$	$V \cdot II$
ML	1
total	$V(2II+1)+m+1$

constraint	number
(3)	V
(5)	E
(6)	$m \cdot II$
(7,8)	$V \cdot II$
(9)	II
total	$V(II+1)+E+II(m+1)$

Table 1: Variables and constraints of the ILP model

5 Branch and Prune

The most popular method to solve ILP models is the combination of branch-and-bound techniques with a linear-programming solver such as *simplex* [21]. Having integer variables in the model increases complexity from polynomial to exponential. The running time for ILP is highly influenced by the exploration of the branch-and-bound tree. For an ILP solver insensible to the problem, the tree of solutions is "blindly" explored and finding valid integer solutions may require solving an excessive number of LP problems.

We have implemented *branch and prune*, an ad-hoc solver for loop pipelining that takes advantage of the information known a priori about the problem. The solver uses a branch-and-bound paradigm to explore the space of integer solutions that allows us a rapid pruning when enough information has been captured to evaluate the objective function. The order in which integer variables are explored is also crucial to reduce the running time.

Loop pipelining is decomposed in two different problems: Retiming (finding the loop fold λ for each operation) and scheduling (assigning operations to cycles). After solving retiming, loop pipelining is reduced to the scheduling of a basic block in which not all dependences must be taken into account. According to this idea, variables corresponding to *retiming* are explored before variables corresponding to *scheduling*.

5.1 Retiming

Recurrences are the most stringent constraints for retiming, since the sum of their edges is a constant [14].

The order of the variables is selected as follows: first explore the nodes belonging to the most stringent recurrences. Inside each recurrence, first explore those nodes that also belong to other recurrences.

For nodes not belonging to any recurrence, the exploration order is defined according to a neighboring criteria to nodes in recurrences. In these cases the number of branches to be explored may be unbounded, since no recurrence limits the value of λ. For this reason, an early calculation of a lower bound for register requirements is done at each node of the tree.

5.2 Scheduling

After having defined the folds of the operations, a scheduling problem is posed for each of the leaves of the search tree. At this moment, some dependences of the DG can be eliminated from the model (if $T(u) - II \cdot \delta_{u,v} \leq 0$) and the critical path of the retimed DG can be calculated. In case the critical path is longer than *II*, no feasible schedule can be found and the exploration is pruned.

The exploration order of the scheduling variables ($s_{u,i}$) is also crucial. We have chosen one of the well-known algorithms for scheduling (force-directed scheduling (FDS) [22]) to assist the solver in defining an efficient strategy to generate the search tree. FDS performs a stepwise assignment of operations to cycles according to criteria that attempt to balance the utilization of resources over all cycles of the schedule.

This strategy leads the solver to a near-optimal solution very soon, thus allowing an efficient pruning of the tree for the other branches. Similarly to branch-and-bound, the LP solver is invoked at each branch of the tree to prune those solutions with a cost greater than the best integer solution found.

6 Experimental results

The techniques presented in the previous sections have been implemented by using the package *lp_solve* [23]. In this section, several results are reported to show the main features of the method. The types of resources used in the schedules are adders (1 cycle) and multipliers (2 cycles).

6.1 Optimal unrolling degree

Table 2 presents the results obtained by exploring two different unrolling degrees for the example depicted in Figure 5. MAXLIVE is also minimized for the optimal *II* found by the model. The columns V and E indicate the number of nodes and edges of the DG after unrolling the loop. #Vars and #Const indicate the number of variables and constraints of the ILP model. SPAN is calculated as $\lambda_{max} - \lambda_{min} + 1$. the schedule). *MII*is 1.25 for all three cases.

The first row presents the result for point $C = 5, 4$ in Figure 5), which is an infeasible problem. The second row presents the result for point $A = (4, 3)$. This case obtains optimal results in MAXLIVE and *II* in front of the schedule found for the third point ($B = (6, 4)$), which is the one explored by other techniques that perform sub-optimal loop unrolling.

II	K/II_K	V	E	#Vars	#Const	CPU	SPAN	MAXLIVE
1.25	4/5	15	21	167	121	225	-	Non feasible
1.33	3/4	10	14	82	72	199	4	5
1.5	4/6	15	21	192	138	22	4	6

Table 2: Results for the example of Figure 5(a).

6.2 ILP model for RCLP

A significant part of the computational cost for solving the ILP model is spent in guaranteeing that the final solution is optimal. For this reason, we also report the best solution obtained after 1 minute of CPU[3]. MAXLIVE is represented as ML.

The results presented in Tables 3 and 4 show that benchmarks with a large number of operations can be solved with moderate computational cost. The method can also be used for heuristic search by limiting the maximum CPU time and providing the best solution found at that moment (e.g. one minute). The results demonstrate that the optimal solution can be often obtained by only using a small fraction of the total computational cost, although a proof of optimality cannot be given in this case. Optimal solutions have been found for models with more than 1000 variables and 700 constraints.

Resources		II	#Vars	#Const	CPU (secs)	ML	SPAN	ML (60 secs)
X	+							
8	15	2	118	97	7	16	5	-
4	8	4	210	149	10	8	3	-
2	4	8	394	253	38	4	2	-
2	3	10	486	305	77	4	2	4
1	2	16	762	461	1252	3	2	3

Table 3: Results for the 16-point FIR filter ($V = 34 and E = 22$)

[3]We only report the value of MAXLIVE after 1 min, which is the critical variable in the optimization of the objective function.

Resources			II	#Vars	#Const.	CPU (secs)	ML	SPAN	ML (60 secs)
$\times_p$	$\times$	$+$							
2		4	16	1125	684	201	9	2	9
2		3	16	1125	684	204	9	2	9
1		3	16	1125	684	188	10	2	10
1		2	17	1193	721	433	9	2	9
	3	3	16	1125	684	203	9	2	9
	2	3	16	1125	684	223	10	2	10
	2	2	17	1193	721	472	9	2	9
	1	2	19	1329	795	1924	9	2	9

Table 4: Results for the 5th-order elliptic wave filter ($\times_p$ and $\times$ stand for pipelined and non-pipelined multipliers). $V = 34$ and $E = 58$

6.3 Solutions for large examples

Although the branch-and-prune strategy has proved to be efficient for models with 10^3 variables/constraints, the exponential nature of the method becomes critical in some cases. Rather than discarding this method, we claim that it can still be used to obtain near-optimal solutions by limiting the CPU time of the search.

Tables 5 and 6 present results on the Cytron's and FDCT loops obtained by limiting the CPU time to 10 minutes. Given the status of the search at the time the solution was delivered, we suspect that the results are optimal (although we could not prove it).

#R	*MII*	*II*	K/II_K	V	E	#Vars	#Const	SPAN	ML
3	5.66	5.66	3/17	51	63	1787	1015	3	12
4	4.25	4.25	4/17	68	84	2382	1342	4	18
5	3.4	3.4	5/17	85	105	2977	1669	5	19

Table 5: Results for the Cytron example

Resources			*MII*	*II*	K/II_K	#Vars	#Const	SPAN	ML
$+$	$-$	$\times$							
4	4	4	4	4	1/4	382	279	3	12
2	2	2	8	8	1/8	718	463	2	16
3	3	3	5.33	5.33	3/16	4162	2365	2	38

Table 6: Results for the FDCT 1st and 2nd cases: $V = 42, E = 53$. Third case: $V = 126, E = 159$

7 Conclusions

In this paper we have presented a mathematical model that can be solved by using ILP. However, efficient techniques to wisely explore the space of integer solutions are required to avoid a blind navigation through the branch-and-bound tree.

We have presented a new strategy called *branch and prune* that takes advantage of the information known a priori about the problem to seek near-optimal solutions as fast as possible. Decomposing loop pipelining into two sub-problems (retiming and scheduling) and using force-directed scheduling to assist the search have been essential heuristics to guarantee optimal solutions in moderate CPU times.

We have also demonstrated that exploring the unrolling degree is necessary to find optimal solutions for loop pipelining. A mathematical formulation based on Farey's series has been proposed for such a goal.

References

[1] T.-F. Lee, A.C.-H. Wu, Y.-L. Lin, and D.D. Gajski. A transformation-based method for loop folding. *IEEE Trans. Computer-Aided Design*, 13(4):439–450, April 1994.

[2] C.-T. Hwang, Y.-C. Hsu, and Y.-L. Lin. Scheduling for functional pipelining and loop winding. In *Proc. of the 28th Design Automation Conf.*, pages 764–769, June 1991.

[3] E.F. Girczyc. Loop winding: A data flow approach to functional pipelining. In *Proc. Int. Symp. Circuits and Systems*, pages 382–385, May 1987.

[4] L.-F. Chao, A. LaPaugh, and E.H.-M. Sha. Rotation scheduling: a loop pipelining algorithm. In *Proc. of the 30th Design Automation Conf.*, pages 566–572, June 1993.

[5] S.M. Heemstra de Groot, S.H. Gerez, and O.E. Herrmann. Range-chart-guided iterative data-flow graph scheduling. *IEEE Transactions on Circuits and Systems–I*, 39(5):351–364, May 1992.

[6] F. Sánchez and J. Cortadella. Time constrained loop pipelining. In *Proc. Int. Conf. Computer-Aided Design*, pages 592–596, November 1995.

[7] J. Cortadella, R.M. Badia, and F. Sànchez. A mathematical formulation of the loop pipelining problem. Technical Report UPC-DAC-95-36, Dept. of Computer Architecture, Univ. Politècnica de Catalunya, November 1995.

[8] R. Cytron. *Compiler-time scheduling and optimization for asynchronous machines*. PhD thesis, University of Illinois at Urbana-Champaign, 1984.

[9] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. Computer-Aided Design*, 10(4):464–475, April 1991.

[10] M. Rim and R. Jain. Lower-bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. Computer-Aided Design*, 13(4):451–458, April 1994.

[11] R. Govindarajan, E.R. Altman, and G.R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, November 1994.

[12] A.E. Eichenberger, E.S. Davidson, and S.G. Abraham. Optimum modulo schedules for minimum register requirements. In *Proc. of 9th ACM the International Symposium on Supercomputing*, pages 31–40, June 1995.

[13] K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Trans. Computers*, 40(2):178–195, February 1991.

[14] J. Cortadella, R. M. Badia, and F. Sánchez. A mathematical formulation of the loop pipelining problem. Technical Report UPC-DAC-1995-36, Department of Computer Architecture (UPC), October 1995.

[15] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[16] B.R. Rau and C.D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[17] M.R. Schroeder. *Number theory in science and communication*. Springer-Verlag, 1990.

[18] F. Sánchez. *Loop Pipelining with Resource and Timing Constraints*. PhD thesis, Universitat Politècnica de Catalunya (Spain), 1995.

[19] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, November 1994.

[20] C.H. Gebotys and M.I. Elmasry. Global optimization approach for architectural synthesis. *IEEE Trans. Computer-Aided Design*, 12(9):1266–1278, September 1993.

[21] C.H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, 1982.

[22] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Trans. Computer-Aided Design*, 8(6):661–679, June 1989.

[23] M.R.C.M. Berkelaar. *lp_solve version 2.0: a public domain ILP solver*, 1995. available at ftp.es.ele.tue.nl.