UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

6

UNRET: LOOP PIPELINING WITH RESOURCE CONSTRAINTS

6.1 INTRODUCTION

Some software pipelining techniques [SDX86, AN88a, PLNG90] find a loop schedule by assuming infinite resources. Following this, the schedule is reorganized in order to satisfy resource constraints. Other approaches consider resource constraints during the scheduling process [RG81, Lam88, Rim93]. These later techniques are more suitable for loop pipelining in real architectures, and in general they obtain better results than the former.

The number of resources available in parallel architectures (superscalar processors or VLIW processors) is limited and depends on the architecture of the processor. In the case of HLS systems, the area of the chip is limited and therefore the number of resources (FUs, buses, etc.) is finite. Since resource-constrained scheduling is an NP-hard problem [GJ79], an optimal schedule for these architectures might not be found by a known polynomial algorithm.

The throughput of a schedule is the ratio between the unrolling degree of the loop and the length of the schedule. The variations in both magnitudes determine the *solution space* (all possible throughputs) for a given loop. When an optimal schedule is not found, the *solution space* must be examined and a criterion must be established to determine when an "acceptable" solution has been found.

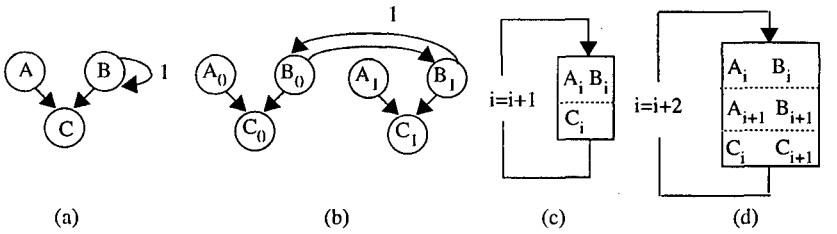

Figure 6.1 Different schedules of a loop
(a) π
(b) π^2
(c) Sub-optimal schedule of π
(d) Time-optimal schedule of π^2

This chapter presents *UNRET* (*UNrolling and RETiming*), a new approach for software pipelining with resource constraints which takes resources into account during the scheduling process. *UNRET* computes the *MII* of the loop and tries to find a schedule which executes the loop in *MII* cycles. Since *MII* may not be an integer, loop unrolling is used to find a multiple-instanced loop with an integer *MII*. Figure 6.1 shows an example of how the potential parallelism achievable for a loop schedule may be increased by previous unrolling. For the sake of simplicity, we assume that all instructions in the figure are sums which can be executed in a single cycle, and two adders are available.

A loop schedule in *MII* cycles may not exist for some loops (*MII* is calculated by considering resources and recurrences, but other factors, such as the topology of the dependences, may also impose constraints to the scheduling). Even if the schedule exists, the loop pipelining approach may not be able to find it.

Previous software pipelining approaches [RG81, Lam88], including those that previously consider loop unrolling [SDX86, BC90, Rim93], explore the *solution space* in one dimension, i.e. the expected II of the schedule is increased when a schedule with the previous expected initiation interval is not found.

Conversely, *UNRET* explores the *solution space* in both dimensions the unrolling degree of the loop (K) and the expected initiation interval for the schedule of the loop unrolled K times (II_K). To do so, pairs (II_K, K) are generated in decreasing order of the expected throughput of the schedule until a schedule is found. For each pair (II_K, K), the loop is unrolled K times, repeatedly transformed (by means of *dependence retiming*) and scheduled. If no schedule of π^K is found in II_K cycles, a new pair is explored. Finally, once a schedule has been found, the number of registers required is reduced. This step will be explained in detail at Chapter 7. The flow diagram depicted in Figure 6.2 shows the previously detailed strategy.

Section 6.2 describes how the pairs (II_K, K) are generated. Section 6.3 describes how a π-graph is transformed by means of *dependence retiming*. Section 6.4 presents the algorithm used to find a schedule, as well as the heuristics proposed to transform the π-graph and to decide when to stop the search. Section 6.5 shows three simple examples to illustrate how *UNRET* works. Some results are reported in Appendix B.

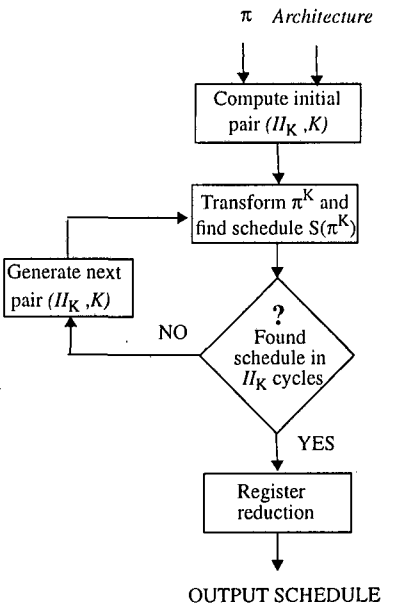

Figure 6.2 General overview of *UNRET*

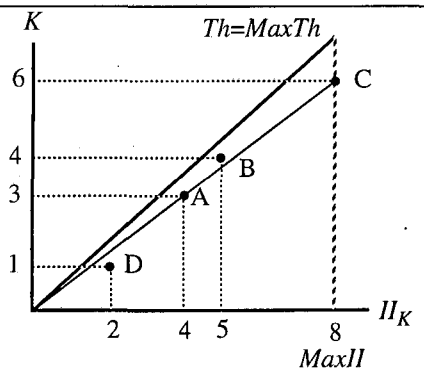

Figure 6.3 Representing throughput in a diagram

6.2 EXPLORING THE SOLUTION SPACE

6.2.1 Throughput diagram

The throughput of any schedule of a loop can be represented in a diagram, as shown in Figure 6.3. The y axis represents the unrolling degree of the loop (K), and the x axis represents the number of cycles of the schedule (II_K). A point (II_K, K) in the diagram represents a schedule[1]

[1] This schedule may not exist.

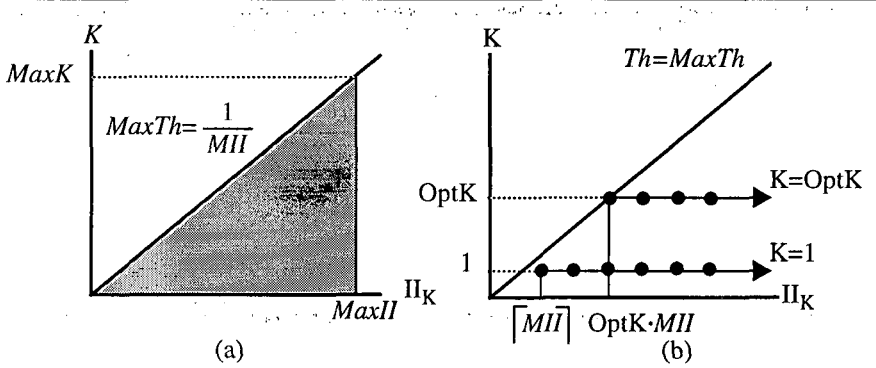

Figure 6.4 (a) Solution space for *UNRET*
(b) Solution space for other approaches

of K iterations of the loop in II_K cycles. The throughput of such a schedule is $\frac{K}{II_K}$ iterations per cycle.

All points representing schedules with the same throughput fall in a line. For example, points A and C in Figure 6.3 represent two different schedules with the same throughput $Th = 0.75$. The throughput of the schedules represented by points B and D is $Th = 0.8$ and $Th = 0.5$ respectively. Note that point B is above the line which includes points A and C, since the throughput of B is greater than the throughput of A and C. On the other hand, point D is below this line because it represents a schedule with less throughput.

As shown in Chapter 3, the maximum throughput ($MaxTh$) achievable by a schedule is bounded by the recurrences of the loop and the set of resources of the architecture ($MaxTh = \frac{1}{MII}$). All the schedules achieving maximum throughput fall in the line $Th = MaxTh$. Any feasible schedule of the loop must be represented by a point (with integer values for K and II_K) below this line. We are interested in exploring the points in the diagram in decreasing order of throughput. Since the number of points below the line $Th = MaxTh$ is infinite, we will limit them by constraining the maximum number of cycles of any schedule. This bound is called $MaxII$.

The points to explore are included in the triangle formed by the II_K axis and the lines $II_K = MaxII$ and $Th = MaxTh$. The shadowed triangle in Figure 6.4(a) represents the solution space. Figure 6.4 compares the solution space explored by *UNRET* to the solution space explored by other approaches.

- Software pipelining techniques which do not perform loop unrolling explore the points shown in the arrow represented on the line $K = 1$. Note that the first point to explore may not be on the line $Th = MaxTh$, and therefore may not represent a time-optimal schedule (its initiation interval is $\lceil MII \rceil \geq MII$).
- Software pipelining approaches considering loop unrolling explore the points shown in the arrow represented on the line $K = OptK$. $OptK$ is the optimal unrolling degree of the loop. Given a π-graph π with $MII(\pi) = \frac{a}{b}$, $OptK = \frac{b}{gcd(a,b)}$ [JC92b, JA90]. Therefore, a theoretical optimal-time schedule may exist in $OptK \cdot MII$ cycles. For a given value of K, the distance between any two points is always an integer value. In general, the distance between two

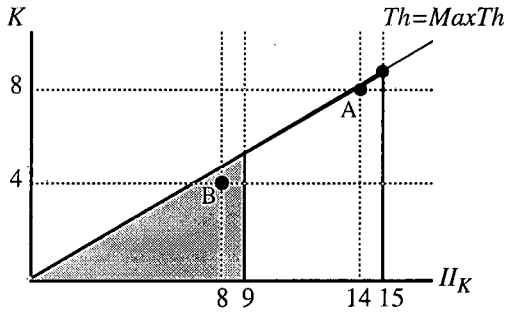

Figure 6.5 Triangles delimited by $MaxII = 9$ and $MaxII = 15$

consecutive points is one unit, but this depends on the approach (for example, in [Huf93] the proposed increment is $\max(\lfloor 0.04 \cdot II \rfloor, 1)$. This is done to avoid spending an excessive amount of time compiling large complex loops).

- Whilst the previous approaches explore the solution space in only one dimension (they change the expected initiation interval for a fixed unrolling degree), *UNRET* explores the solution space in two dimensions (it changes both the expected initiation interval and the unrolling degree), as shown by the shadowed triangle from Figure 6.4(a). Exploring different unrolling degrees may increase the throughput in face to only explore different initiation intervals. Figure 6.5 depicts a diagram with two triangles corresponding to $MaxII = 9$ and $MaxII = 15$. The point $A = (14, 8)$ belongs to the triangle delimited by the line $MaxII = 15$, whilst the point $B = (8, 4)$ belongs to both triangles. Although A represents a potential schedule of 14 cycles and B a potential one of only 8 cycles, the schedule throughput of A ($Th = \frac{8}{14} = 0.57$) is higher than that of B ($Th = \frac{4}{8} = 0.50$).

6.2.2 Farey's series

For a fixed $n > 0$, the sequence of all the reduced fractions with nonnegative denominator $\leq n$ arranged in increasing order of magnitude is called the *Farey's series* of order n, and denoted by F_n [Sch90]. For example, F_5 is the series of fractions:

$$\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \cdots$$

Figure 6.6 shows a diagram representing such a sequence[2]. Note that point (4,2) is not in the sequence, since it is not reduced (it represents the same fraction as point (2,1), as the line that crosses both points in Figure 6.6 shows).

The throughput of a schedule represented in Figure 6.3 is a fraction with a denominator lower than or equal to *MaxII*. Therefore, the Farey's series of order *MaxII* forms the sequence of points to

[2] In order to compare such a diagram to the throughput diagram from Figure 6.3, numerators have been represented in the y axis and denominators in the x axis.

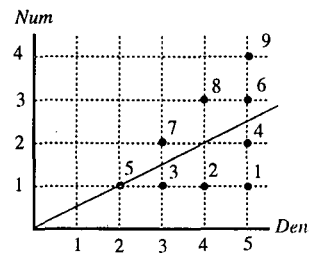

Figure 6.6 Representing Farey's series F_5 in a diagram

be explored in the throughput diagram. We are interested in exploring the elements of F_{MaxII} in decreasing order, starting at the first element E_i fulfilling $E_i \leq MaxTh$. An approach to perform such exploration, based on maximizing the resource utilization of the architecture, has previously been proposed in [SC93a, SC93b]. In the next section, we will explain a simpler way to perform such an exploration.

6.2.3 Exploring Farey's series in decreasing order of magnitude

By using the equations from [Sch90], we derive new equations to recursively generate the elements from Farey's series in decreasing order of magnitude. The ith element of the Farey's series F_{MaxII}, $E_i = \frac{X_i}{Y_i}$, can be recursively calculated by using the two following elements as follows:

$$X_i = \left\lfloor \frac{Y_{i+2} + MaxII}{Y_{i+1}} \right\rfloor \cdot X_{i+1} - X_{i+2} \qquad Y_i = \left\lfloor \frac{Y_{i+2} + MaxII}{Y_{i+1}} \right\rfloor \cdot Y_{i+1} - Y_{i+2} \tag{6.1}$$

Note that two consecutive elements of Farey's series F_{MaxII} are required to recurrently compute the previous one. We will now describe the way to compute the two first elements to be considered. Such elements must be as near as possible to the fraction representing maximum throughput, which is the first fraction that may represent a feasible schedule. The throughput of the schedule represented by the first element must be $MaxTh$ whenever possible.

Theorem 6.1 *The lowest fraction with denominator $MaxII$, with a value greater than or equal to $MaxTh$ is:*

$$\frac{\left\lceil \frac{MaxII}{MII} \right\rceil}{MaxII} \tag{6.2}$$

Proof:

Let us consider the throughput diagram from Figure 6.7. Point Y is in the line $Th = MaxTh$, but it cannot be considered because it is not represented by integer values of II_K and K. For

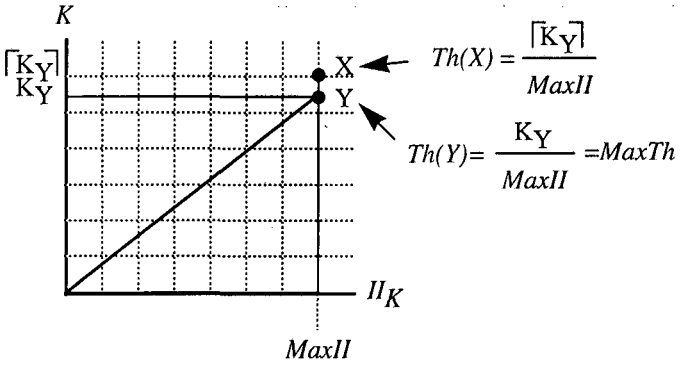

Figure 6.7 First element of Farey's series to be considered

a given $MaxTh$ and $MaxII$, we are interested in finding the fraction corresponding to point X. Since point Y fulfills $Th = \frac{1}{MII}$, we have:

$$Th(Y) = \frac{1}{MII} = \frac{K_Y}{MaxII} \Rightarrow K_Y = \frac{MaxII}{MII}$$

Since Farey fractions are formed by integer numerators and denominators, we are interested in finding $\lceil K_Y \rceil$. Point $X = (MaxII, \lceil K_Y \rceil)$ represents the first point which we are looking for, with a throughput:

$$Th(X) = \frac{\lceil K_Y \rceil}{MaxII} = \frac{\lceil \frac{MaxII}{MII} \rceil}{MaxII}$$

Note that point X may not represent a feasible schedule, since $Th(X)$ may be greater than $MaxTh$.

□

The fraction expressed by Equation (6.2) is the first to be analyzed. However, to use the recurrence from Equation (6.1) we need two consecutive fractions. By using theorems from [HW79], we derive a recurrent equation to compute a fraction from F_{MaxII} by using the next one:

$$X_i = x + X_{i+1} \cdot \left\lfloor \frac{MaxII - y}{Y_{i+1}} \right\rfloor \qquad Y_i = y + Y_{i+1} \cdot \left\lfloor \frac{MaxII - y}{Y_{i+1}} \right\rfloor \tag{6.3}$$

where x and y are two integers satisfying the relation $\gcd(-Y_i, X_i) = (-Y_i) \cdot x + X_i \cdot y$. The coefficients x and y can easily be computed by using the *extended gcd* [HW79]. Algorithm 6.1 calculates the second (previous) Farey fraction by using Equation (6.3). In the algorithm, called *Previous_fraction*, function ExtendedGCD(n, m) returns the list $[g, x, y]$, where g is $\gcd(n, m)$, and x and y are two integers satisfying the relation $g = nx + my$. Once both fractions have been obtained, we can recurrently use Equation (6.1) to compute the remainder elements of the series.

function *Previous_fraction*(X_k,Y_k,Z);
{Read the fraction $\frac{X_{k+1}}{Y_{k+1}}$ of the Farey series F_{MaxII} and
computes the previous fraction $\frac{X_k}{Y_k}$ }
$[g, x, y]$:=ExtendedGCD($-Y_{k+1}$,X_{k+1});
$r := \lfloor \frac{MaxII-y}{Y_{k+1}} \rfloor$;
return $\frac{x+r \cdot X_{k+1}}{y+r \cdot Y_{k+1}}$

Algorithm 6.1 Algorithm to compute a Farey fraction by using the next one

The first fraction which may correspond to a feasible schedule is the fraction $E_j = \frac{x_j}{y_j}$, such that $\frac{x_j}{y_j} \leq MaxTh$.

Since the fractions are reduced, not all points in the throughput diagram are represented by the elements of Farey's series (an example is given in Figure 6.6. Point (4,2) is not represented in the series, since it is not reduced). However, if a schedule is not found for a given point (II_K, K), this does not mean that a schedule for the point $(n \cdot II_K, n \cdot K)$ does not exist[3] for any integer $n > 0$. Therefore, for each Farey fraction we generate all the equivalent ones with a denominator lower than or equal to *MaxII*. Such equivalent fractions are generated in increasing order of denominator, since we are interested in first exploring the shortest schedules (for schedules with the same throughput). For example, if we are interested in the series F_5 in decreasing order of magnitude, and the maximum throughput achievable by a schedule is $MaxTh = \frac{4}{5}$, the fractions to explore are:

$$\frac{4}{5}, \frac{3}{4}, \frac{2}{3}, \frac{3}{5}, \frac{1}{2}, \frac{2}{4}, \frac{2}{5}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}$$

In summary, the algorithm to compute all the pairs in decreasing order of throughput is as follows:

1. Compute the lowest reduced fraction $\frac{x_k}{MaxII}$ such that $\frac{x_k}{MaxII} \geq MaxTh$. This is the first fraction $\frac{x_k}{y_k}$ of the series in which we are interested. Equation (6.2) shows such a fraction.
2. Compute the previous fraction, $\frac{x_{k-1}}{y_{k-1}}$, by using Equation (6.3). Generate all the fractions equivalent to $\frac{x_{k-1}}{y_{k-1}}$ with a denominator lower than or equal to $MaxII$.
3. Compute the remainder fractions by recurrently using Equation (6.1). For each fraction, generate all the equivalent fractions with a denominator lower than or equal to $MaxII$. This calculation is faster than using step 2 for each element.

[3]Despite not finding any such case in our experiments, we have not been able to prove the contrary.

6.2.4 Reducing the solution space

Number of points to explore

The number of points to explore may be quite large. Since only integer values for K and II_K are explored, the number of points inside the triangle which forms the solution space is in the order of the area of the triangle (see Figure 6.4(a)): *Number of points* $\approx \frac{MaxK \cdot MaxII}{2}$.

$MaxK$ corresponds to the number of iterations involved into a schedule with maximum throughput, whose length is $MaxII$ cycles. The throughput of such a schedule is $\frac{1}{MII}$ iterations per cycle. Since $\frac{MaxK}{MaxII} = \frac{1}{MII}$, we conclude that $MaxK = \frac{MaxII}{MII}$.

Therefore, the number of points in the throughput diagram is in the order of:

$$Number\ of\ points \approx \frac{MaxK \cdot MaxII}{2} = \frac{\frac{MaxII}{MII} \cdot MaxII}{2} = \frac{MaxII^2}{2 \cdot MII}$$

We conclude that the number of points is proportional to the square of $MaxII$. Thus, reducing $MaxII$ quadratically decreases the number of points in the solution space.

Reducing MaxII

The maximum number of cycles of a schedule is determined by the designer. However, when *MaxII* is a large value the solution space is too large. In this cases, it is useful to look for a *good* solution rather than an optimal solution if this significantly reduces the solution space.

Let us consider two solution spaces SS_1 and SS_2, such that $SS_2 \subset SS_1$. We define SS_2 as a *good* subspace to explore SS_1 if, for each point (II_K, K) representing a schedule with throughput $\frac{K}{II_K}$, there exists a point (II'_K, K') such that $\frac{K'}{II'_K} \geq x \cdot \frac{K}{II_K}$, where $x \in (0, 1]$ and x approaches to 1.

$$\forall (II_K, K) \in SS_1,\ \exists (II'_K, K') \in SS_2 \text{ such that } \frac{K}{II_K} \geq \frac{K'}{II'_K} \geq x \cdot \frac{K}{II_K}$$

The previous equation indicates that, for each point belonging to SS_1, a point exists in SS_2 with a similar throughput.

We assume that, if a schedule exists for a point (II_K, K), a schedule also exists for any point (II'_K, K') such that $\frac{K'}{II'_K} \leq \frac{K}{II_K}$. We cannot demonstrate this assumption, but we have not found in our experiments any case which indicates the contrary.

The solution space can be reduced in two ways, as shown in Figure 6.8:

- By limiting the maximum number of iterations of the loop (K)

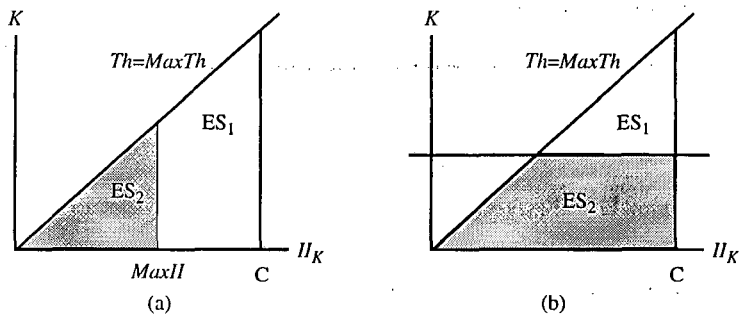

Figure 6.8 Reducing solution space
(a) By limiting *MaxII*
(b) By limiting K

- By limiting the maximum number of cycles of the schedule (*MaxII*)

Limiting *MaxII* also produces a reduction in the number of iterations to unroll. Furthermore, it is better than a simple reduction in K as far as reducing the number of registers required by the schedule is concerned (in general, the greater the number of cycles, the greater the register requirements). Thus, for a given maximum number of cycles C determining a solution space SS_1, our objective is to find a $MaxII < C$ determining a solution space SS_2 such that, for any schedule S in SS_1 there exists a schedule in SS_2 with at least x per cent of the throughput of S.

The number of fractions in F_C is greater than the number of fractions in F_{MaxII}. The fractions in a Farey's series are not uniformly distributed. Therefore, the number of fractions belonging to F_C which are between two consecutive fractions from F_{MaxII} is not constant. We are interested in studying two consecutive fractions such that the difference between them is maximum. Note that this is the worst case for our proposal, which is finding a *good* subspace determined by *MaxII*.

For a given Farey's series F_C, the maximum difference between two consecutive fractions corresponds to fractions $\frac{1}{1}(\frac{C}{C})$ and $\frac{C-1}{C}$ (there are other consecutive fractions with the same difference). Such a difference is $\frac{1}{C}$. Figure 6.9 shows the number of fractions from F_C which are between two consecutive fractions from F_{MaxII} between which there is the largest difference.

The point labelled $\frac{MaxII-1}{MaxII}$ in Figure 6.9 must represent a schedule with at least x per cent of the throughput of the point labelled $\frac{x}{y}$. Therefore, we have:

$$x \cdot (1 - \frac{1}{C}) \leq \frac{MaxII - 1}{MaxII}$$

and, by operating with the former equation, we obtain:

$$MaxII \geq \frac{1}{\frac{x}{C} + 1 - x}$$

$$MaxII = \left\lceil \frac{1}{\frac{x}{C} + 1 - x} \right\rceil$$

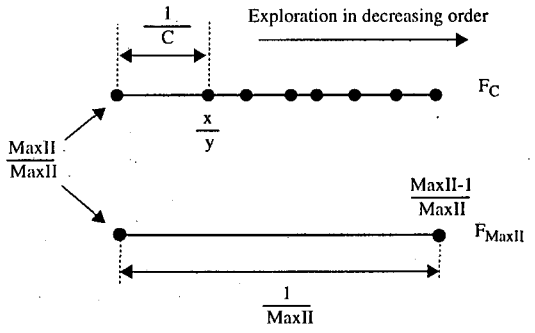

Figure 6.9 Comparing number of points in F_C and F_{MaxII}

In order to illustrate the effectivity of reducing the solution space, we will show the *MaxII* calculated for $C = 10$, $C = 50$, $C = 100$ and $C = 200$, by assuming $x = 0.95$.

$$C = 10 \Longrightarrow MaxII = 7$$
$$C = 50 \Longrightarrow MaxII = 15$$
$$C = 100 \Longrightarrow MaxII = 17$$
$$C = 200 \Longrightarrow MaxII = 19$$

We conclude that using a *small MaxII* is possible to achieve a *good* solution. Reducing the solution space may significantly reduce the time required to find a *good* schedule.

6.2.5 Figures of merit

This section presents a figure of merit to evaluate how good a schedule is.

Definition 6.1 : Time-optimal schedule

A schedule S is time-optimal *when its throughput is $Th(S) = MaxTh$.*

Definition 6.2 : ε (Schedule Efficiency Ratio)

We define the efficiency ratio *of a schedule S, ε, as:*

$$\varepsilon = \frac{Th(S)}{MaxTh}$$

The *efficiency ratio* is a real number belonging to the interval (0..1], which measures the time efficiency of a schedule with respect to a (theoretical) time-optimal one. The efficiency ratio of a time-optimal schedule is $\varepsilon = 1$. This figure of merit is independent from the technique used for loop pipelining.

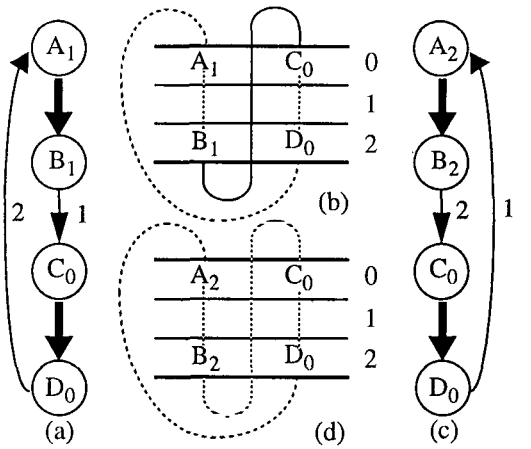

Figure 6.10 $MPP(\pi) \leq II$ does not guarantee a schedule in II cycles
(a) Example of π-graph π
(b) An incorrect schedule in 3 cycles of π
(c) π-graph π' equivalent to π
(d) A correct schedule in 3 cycles of π'

6.3 RETIMING DEPENDENCES

6.3.1 Range for retiming dependences

Theorem 6.2 *A necessary condition for a π-graph π to be scheduled in II cycles is $MPP(\pi) \leq II$.*

Proof: All instructions in the a positive path must start their execution inside the II cycles of the schedule (see definition 4.8). Thus, if the length of the critical path is longer than II, such a path cannot be scheduled in II cycles. □

In order to obtain a schedule in MII cycles, dependences must be retimed until an equivalent π-graph π' is found such that $MPP(\pi') \leq MII(\pi')$.

Figure 6.10 shows an example of how the condition presented in Theorem 6.2 is not a sufficient condition to find a schedule. In the example, we assume $L_A = L_B = L_C = 2$ and $L_D = 1$. Therefore, $MII(\pi) = \frac{7}{3} = 2.33$. Bold edges indicate the *positive scheduling dependences* (PSDs), for an expected initiation interval $II = 3$, and $MPP(\pi) = 3$. Figure 6.10(c) depicts a π-graph π' equivalent to π. The length of the critical path is 3 in both π-graphs. Although $MPP(\pi) \leq II$, no schedule in three cycles exists for π, as Figure 6.10(b) shows. However, a schedule in three cycles exists for π', as shown in Figure 6.10(d). Note that the distance of dependence (B, C) in π' is greater than the distance of the same dependence in π.

Given that only PSDs are taken into account to compute the MPP, transforming PSDs into NSDs (*negative scheduling dependences*) will reduce the length of the MPP. However, NSDs also impose constraints on the scheduling process, as schedule in Figure 6.10(b) shows. Transforming NSDs into FSDs (*free scheduling dependences*) makes the scheduling task easier (see Figure 6.10(d)).

```
function increase_distance(π, e); {e = (u, v)}
    S := {v ∈ V | there is a path from u to v};
    P := V_c − S; { V_c= connected component which u belongs to }
    for each x ∈ V_c do
        if x ∈ P then λ(x) := λ(x) + 1;
    for each (x, y) ∈ E_c do
        if x ∈ P and y ∈ S then δ(x, y) := δ(x, y) + 1;
    return π';
```

Algorithm 6.2 Algorithm to increase the distance of a dependence not belonging to any recurrence without decreasing the distance of any other dependence.

No advantage is achieved by increasing the distance of an FSD. Therefore, since the range for a dependence $e = (u, v)$ to be an FSD is $\delta(e) \geq \frac{L_u + II - 1}{II}$, the range of variation for the distance of a dependence is:

$$0 \leq \delta(e) \leq \left\lceil \frac{L_u + II - 1}{II} \right\rceil$$

6.3.2 Retiming dependences not belonging to recurrences

Theorem 6.3 *The distance of a dependence not belonging to any recurrence can be increased without decreasing the distance of any other dependence.*

Proof:

Algorithm 6.2 (*increase_distance*) increases the distance of a dependence not belonging to any recurrence without decreasing the distance of any other dependence. We only consider nodes belonging to the same connected component as u (the rest of nodes is not influenced by dependence retiming). In the algorithm, V_c and E_c represent respectively the set of nodes and edges of the connected component which u belongs to, S represent the set of nodes v such that a path exists from u to v and P represents the set of nodes w such that a path exists from w to some node $v \in S$

For a given dependence $e = (u, v)$, *increase_distance* increments $\delta(e)$. In order to build an equivalent π-graph π', *increase_distance* also increments $\lambda(x)$ for all nodes $x \in P$. We will demonstrate that such an algorithm produces an equivalent π-graph.

Figure 6.11 shows how *increase_distance* operates on a π-graph. Transformations on the labelling of vertices (λ) and edges (δ) are indicated by the symbol "+1".

All the edges $(u, v) \in E_c$ fall into three categories[4]:

1. $u, v \in P$:
   $\lambda'(u) = \lambda(u) + 1$ and $\lambda'(v) = \lambda(v) + 1$. Therefore $\delta'(u, v) = \delta(u, v)$.

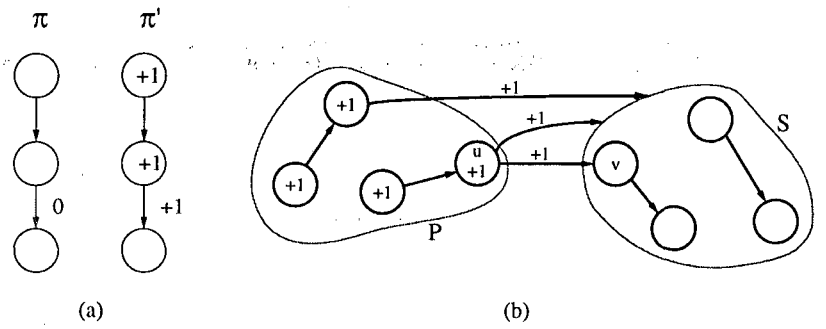

Figure 6.11 Effect of *increase_distance* in a π-graph
(a) An example of *increase_distance* in a π-graph without recurrences.
(b) Representation of the three types of edges.

2. $u \in P$ and $v \in S$:
   $\lambda'(u) = \lambda(u) + 1$, $\lambda'(v) = \lambda(v)$, and $\delta'(u,v) = \delta(u,v) + 1$.

3. $u \in S$ and $v \in S$:
   $\lambda'(u) = \lambda(u)$ and $\lambda'(v) = \lambda(v)$. Therefore $\delta'(u,v) = \delta(u,v)$.

Figure 6.11(b) shows the three categories of edges. For the three cases Equation (3.1) holds. Therefore, we conclude that π and π' are equivalent π-graphs. □

S can be built in $O(V+E)$ time by using a *depth-first search* algorithm [CLR90]. The connected components in a π-graph can be found in $O(V+E)$ time [CLR90]. Therefore, P can also be built in $O(V+E)$ time. In the worst case, *increase_distance* modifies the iteration index of all vertices in the π-graph and the distance of all dependences. Therefore, we conclude that *increase_distance* executes in $O(V+E)$ time.

6.3.3 Retiming dependences belonging to recurrences

Let us consider a π-graph π and a recurrence $R \in \pi$. Let us assume that *dependence retiming* is performed in a dependence $e \in R$, obtaining a π-graph π'. Let δ_R and δ'_R be the sum of the distances of the dependences belonging to recurrence R in π and π' respectively.

Lemma 6.1 $\delta_R = \delta'_R$

Proof: Let us consider a recurrence R composed of n edges: $R = \{e_1, e_2, \ldots, e_n\}$.

$$\delta_R = \sum_{e \in R} \delta(e) = \delta(e_1) + \delta(e_2) + \cdots + \delta(e_{i-1}) + \delta(e_i) + \cdots + \delta(e_n)$$

[4] The case where $u \in S$ and $v \in P$ is not possible. If $u \in S$ and $(u,v) \in E_c$, then $v \in S$.

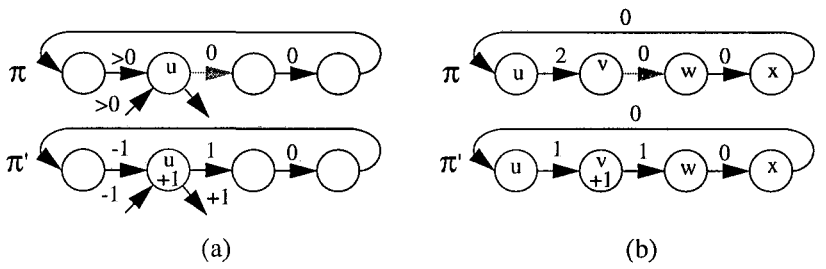

Figure 6.12 *Dependence retiming* performed in a recurrence

with $i \leq n$.

Let us assume that *dependence retiming* is performed in dependence $e_i \in R$. According to the rules of retiming described in Section 3.5.1, $\delta'(e_i) = \delta(e_i) + 1$ and $\delta'(e_{i-1}) = \delta(e_{i-1}) - 1$. The distance of the remainder dependences does not vary. Therefore:

$$\delta'_R = \sum_{e \in R} \delta'(e) = \delta'(e_1) + \delta'(e_2) + \cdots + \delta'(e_{i-1}) + \delta'(e_i) + \cdots + \delta'(e_n) =$$

$$\delta(e_1) + \delta(e_2) + \cdots + \delta(e_{i-1}) - 1 + \delta(e_i) + 1 + \cdots + \delta(e_n) =$$

$$\delta(e_1) + \delta(e_2) + \cdots + \delta(e_{i-1}) + \delta(e_i) + \cdots + \delta(e_n) = \sum_{e \in R} \delta(e) = \delta_R$$

□

Figure 6.12(b) shows how *dependence retiming* allows to distribute the distance of the edges in a recurrence. This feature may reduce the length of the longest positive path in the recurrence. Retiming a dependence belonging to a recurrence may vary the distance of some dependences not belonging to the recurrence, as shown in Figure 6.12(a).

Theorem 6.4 *The distance of a dependence belonging to a recurrence cannot be increased without decreasing the distance of another dependence in the recurrence.*

Proof:

Since the sum of the distances of the edges in a recurrence is constant, increasing the distance of an edge in t units always implies decreasing the distance of other edges in a total of t units. Otherwise, lemma 6.1 would not be fulfilled. □

6.4 FINDING A SCHEDULE WITH MAXIMUM THROUGHPUT

6.4.1 Quality of a π-graph

Given two equivalent π-graphs, π and π', we are interested in deciding which π-graph is more appropriate for scheduling before scheduling both π-graphs. We assume that π-graphs more in accord with the heuristics defined in Chapter 5 for the scheduling algorithm will probably be more suitable.

Therefore, we are interested in reducing the size of the critical path in a π-graph and transforming as many dependences as possible into FSDs. Since PSDs impose more constraints on the scheduling process than NSDs, we assume that π-graphs with a fewer number of dependences (by considering PSDs before NSDs) of each type are better for scheduling.

Definition 6.3 : $min\_cycles(u, v)$

$Min\_cycles(u, v)$ is the number of cycles that v must be scheduled after u when v is ASAP scheduled.

$$min\_cycles(u, v) = L_u - II \cdot \delta(u, v)$$

$Min\_cycles(u, v)$ is a quantitative way of measuring the constraints imposed by a dependence e on the scheduling. Note that $min\_cycles(e) > 0$ if e is a PSD, whilst $min\_cycles(e) \leq 0$ if e is an NSD. Therefore, for a PSD, the greater $min\_cycles(e)$, the greater the constraints imposed by e. On the other hand, for an NSD, the greater $min\_cycles(e)$, the lower the constraints imposed by e.

Definition 6.4 : $PSD\_constraints(\pi)$, $NSD\_constraints(\pi)$

The total constraints imposed by the PSDs (NSDs) on the scheduling of π, $PSD\_constraints(\pi)$ ($NSD\_constraints(\pi)$), is the average of $min\_cycles(e)$ for all PSDs (NSDs) in π.

$$PSD\_constraints(\pi) = \frac{\sum_{e \in E^+} min\_cycles(e)}{Number\ of\ PSDs}$$

$$NSD\_constraints(\pi) = \frac{\sum_{e \in E^-} min\_cycles(e)}{Number\ of\ NSDs}$$

We will use $PSD\_constraints(\pi)$ and $NSD\_constraints(\pi)$ to decide between π-graphs containing the same number of dependences of a given type. Note that $PSD\_constraints(\pi)$ is a positive integer, whilst $NSD\_constraints(\pi)$ is a negative integer. Finally, according to the heuristics defined in Chapter 5, the number of nodes ready to be scheduled on each π-graph will also be considered.

Definition 6.5 : Quality of a π-graph

Let π and π' be two equivalent π-graphs. We define that quality(π) > quality(π') if (in order of priority):

1. $MPP(\pi) < MPP(\pi')$
2. *Number of PSDs in* π < *Number of PSDs in* π'
3. $PSD\_constraints(\pi) < PSD\_constraints(\pi')$
4. *Number of NSDs in* π < *Number of NSDs in* π'
5. $NSD\_constraints(\pi) < NSD\_constraints(\pi')$[5]
6. *Number of nodes ready to be scheduled in* π > *Number of nodes ready to be scheduled in* π'.

If $quality(\pi) > quality(\pi')$, we assume that π is *better* for scheduling than π'.

The execution time of comparing the *quality* of two π-graphs is $O(V + E)$.

6.4.2 Finding a schedule in II cycles

Improving the quality of a π-graph

Algorithm 6.3 (*retiming_and_scheduling*) tries to find a schedule of a loop body in a previously known number of cycles. *Retiming_and_scheduling* successively transforms a π-graph by means of *dependence retiming*, looking for an equivalent π-graph with better *quality*. The π-graph is scheduled after each transformation. The algorithm halts when a schedule in the expected number of cycles is found or the *quality* of the π-graph cannot be further improved.

The first objective of *retiming_and_scheduling* is reducing the length of the *MPP*. According to the definition of *quality*, when the *MPP* cannot be further reduced, the number of PSDs and NSDs must be then reduced.

The length of the *MPP* may be reduced by transforming dependences belonging to a critical path (PSDs) into NSDs or FSDs. In order to achieve this purpose, the function *select_edge* selects the appropriate edge to be retimed. An edge can only be marked if it belongs to a recurrence. Otherwise, it can always be transformed into an FSD without decreasing the distance of any other dependence (see Lemma 6.3). The criteria for selecting and edge, in order of priority, are as follows:

1. The *head_edge* of an MPP. This edge is selected because dependence retiming increases its distance without creating any dependence with a distance lower than zero. When the distance

[5] $NSD\_constraints(\pi)$ is a negative integer which indicates maximum timing constraints. Therefore, decreasing $NSD\_constraints(\pi)$ increases the time frame for scheduling some nodes in π.

```
function retiming_and_scheduling(π, II);
    π' := π;
    unmark(π');
    Repeat
        S :=scheduling(π', II);
        if schedule found then return S;
        e:=select_edge(π'); {e = (u, v)}
        if no unmarked edge can be selected
            then return no schedule found;
        π' := dependence_retiming(π', e);
        if e is an FSD then mark(e);
        if quality(π') > quality(π) then
            π := π';
            unmark all edges in π';
    Forever
```

Algorithm 6.3 Algorithm to find a schedule in a given number of cycles

of an edge e has increased until e becomes an FSD, e is marked in order not to repeat transformations on it (otherwise, the algorithm would never end).

2. The *tail_edge* of an *MPP*. The distance of the *tail_edge* is increased by using a transformation similar to dependence_retiming. The *head_edge* and the *tail_edge* are the only edges over which retiming (or the similar transformation) can be used without creating any dependence with a distance lower than zero. As in the former case, the *tail_edge* is marked when it becomes an *FSD*.

3. The *head_edge* or the *tail_edge* from a positive path not maximal.

4. An NSD not marked. *Dependence_retiming* is performed to attempt to transform it into an FSD.

If no edge can be selected by using the previous criteria, we assume that the *quality* of the π-graph cannot be further increased. In this assumption, the algorithm stops the search without finding a schedule.

In some cases, a π-graph of worse *quality* than the previous one may be produced after a transformation. Since we are interested in continuously improving the *quality*, the π-graph with the best *quality* found is always stored, and the new π-graphs obtained are compared to it. However, *dependence_retiming* is performed in the current π-graph, even though its *quality* may be worse than the *quality* of the *best* π-graph found. This kind of search allows the algorithm to escape from local maxima of *quality*.

Although experiments prove that *quality* compares two π-graphs quite well, it cannot be guaranteed that a π-graph with better *quality* is actually easier to schedule than a π-graph with worse *quality*. For this reason, scheduling is done after each transformation (instead of only once by using the *best* π-graph found).

Figure 6.13 shows an example in which the π-graph (b) has better quality[6] than π-graph (a), but a shorter schedule may be found for (a) by using five FUs.

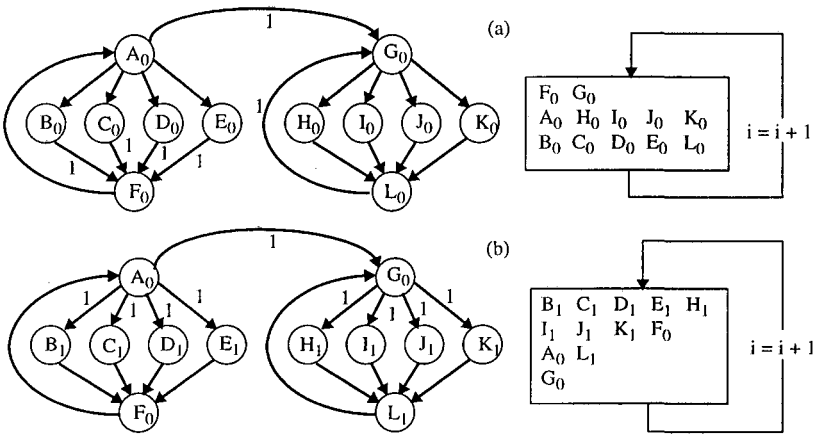

Figure 6.13 (a) has a shorter schedule than (b) despite quality(b)>quality(a)

Theorem 6.5 *Algorithm* retiming_and_scheduling *halts.*

Proof: All the π-graphs representing the same loop are completely ordered by the *quality*. Moreover, although the number of π-graphs is infinite, the number of different equivalent π-graphs explored by the algorithm is finite (an upper bound exists for the distance of a dependence, as shown in Section 6.3). Therefore, there is a π-graph with better quality than any other. *Retiming_and_scheduling* continuously stores the π-graph with the best *quality* found. If all the markable edges have been marked without improving the *quality* of the π-graph, *retiming_and_scheduling* assumes that the π-graph with the best *quality* has been found, and consequently the algorithm halts. □

Computational complexity

We will now study the computational complexity of *retiming_and_scheduling*.

- The scheduling algorithm executes in $O(V^2 + V \cdot E)$ time (see Chapter 5).
- The number of times a dependence (u, v) is retimed depends on $\delta(u, v)$ and L_u. Therefore, it does not depend on $|E|$ and $|V|$, and thus it does not influence the computational complexity of the algorithm. In the worst case, all dependences will be retimed. Therefore, we conclude that the execution time of *retiming_and_scheduling* is:

$$O(V^2E + VE^2)$$

[6] Despite the MPPs from both π-graphs are equal, π-graph (a) has 13 PSDs, whereas π-graph (b) has only 10

```
function retiming_and_scheduling(π,II);
    π' := π;
    unmark all edges in π';
    α = |E|/8;
    i := 0;
    Repeat
        i := i + 1;
        if (i mod α = 0) then S :=scheduling(π', II);
        if schedule found then return S;
        e:=select_edge(π');
        if no unmarked edge can be selected
            then return no schedule found;
        π' := dependence_retiming (π',e);
        if e is an FSD then mark(e);
        if quality(π') > quality(π) then
            π := π'
            unmark all edges in π';
    Forever;
```

Algorithm 6.4 Optimized *retiming_and_scheduling* algorithm

Reducing the execution time of retiming_and_scheduling

Scheduling is the task with the highest computational complexity in *retiming_and_scheduling*. The scheduler is called after each retiming transformation, and the number of retiming transformations performed before finding a π-graph with a feasible schedule may be very large. In order to reduce the execution time of *retiming_and_scheduling*, we are interested in reducing the number of times the scheduler is called. A way to do this is by executing the scheduler once out of α transformations, instead of executing it after each retiming transformation. The parameter α is defined as a function of the size of the π-graph. After several experiments, we have found that $\alpha = \frac{|E|}{8}$ is an appropriate value for α.

Although not scheduling the π-graph after each transformation could, in principle, produce sub-optimal results, comparison with the results obtained with $\alpha = 1$ shows no detectable influence. On the other hand, a significant CPU-time reduction has been achieved. The optimized *retiming_and_scheduling* algorithm described in lines above is the Algorithm 6.4.

As can easily be calculated, the computational complexity of the optimized *retiming_and_scheduling* algorithm is $O(V^2 + VE)$.

6.4.3 General algorithm

The objective of *UNRET* is finding the shortest schedule for a loop in a given architecture. Algorithm 6.5 presents the general *UNRET* algorithm. The algorithm always halts, since:

algorithm *UNRET*;
Calculate *MII*;
if $MII < MaxII$ **then exit**; {scheduling impossible}
Repeat
Generate pair (K, II_K);
$\pi^K := unroll(\pi, K)$; {builds π^K}
schedule=retiming_and_scheduling(π^K, II_K);
until a schedule is found **or** *there are no more pairs to generate*;
if not schedule found
then return schedule not found;
else return schedule;

Algorithm 6.5 *UNRET* Algorithm

- a schedule is found by *retiming_and_scheduling* or
- all the points in the throughput diagram are generated without finding a correct scheduling

Figure 6.14 depicts a detailed flow diagram of *UNRET*.

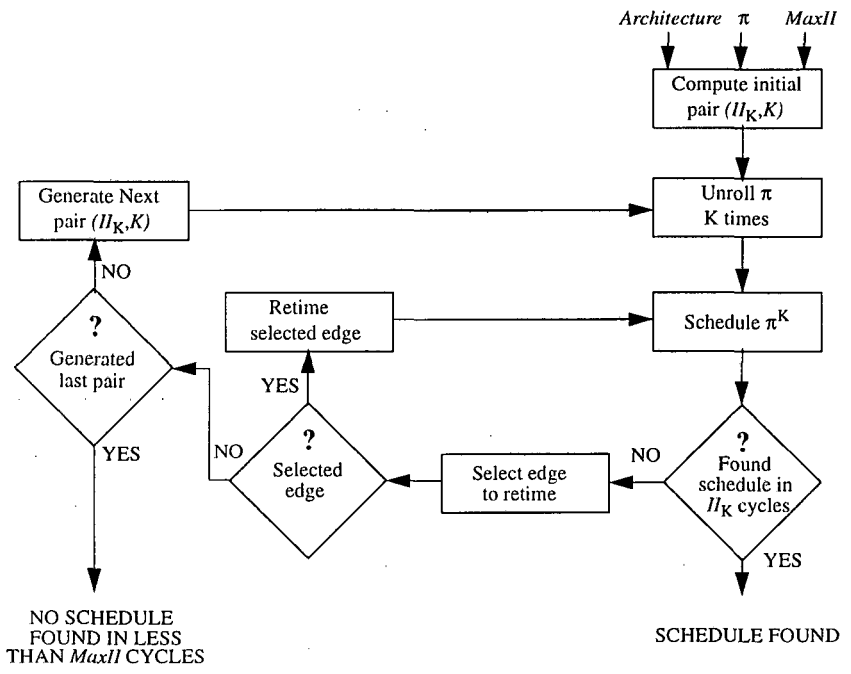

Figure 6.14 Flow diagram of *UNRET*

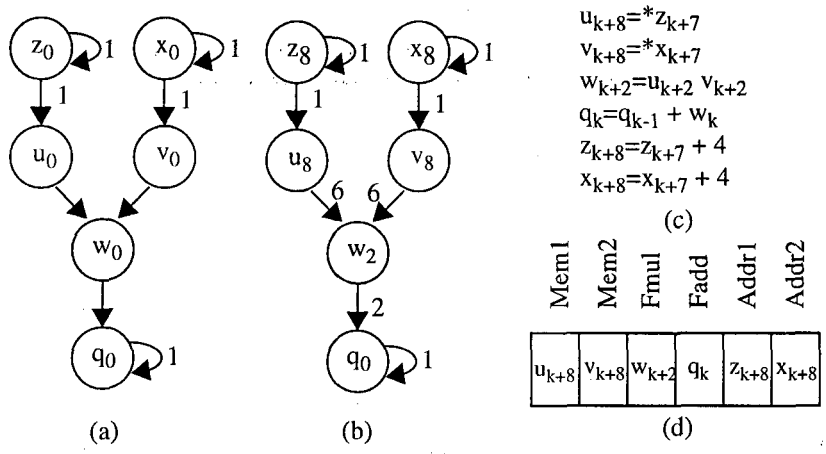

Figure 6.15 Example 1. Schedule for the inner product in 1 cycle
(a) Initial π-graph
(b) Transformed π-graph
(c) New loop body
(d) Schedule in the Cydra 5

The computational complexity of *UNRET* is dominated by the computational complexity of *retiming_and_scheduling*. Our results show that only a few points are generated in the worst cases, and in most cases a schedule with maximum throughput is found for the first point explored.

Once a schedule has been found, the number of required registers is reduced. This step will be explained in detail at Chapter 7.

6.5 EXAMPLES

We will use three different examples to illustrate how *UNRET* works. Each of the examples shows a particular feature of the algorithm.

6.5.1 Example 1

We will use the inner product example to illustrate how *UNRET* may find a schedule with maximum throughput. Figure 6.15(a) shows the π-graph representing the compiled *inner product*.

When the loop executes in the Cydra 5 architecture defined in Chapter 3, we have $MII(\pi) = 1$ (and therefore $MaxTh(\pi) = 1$). That is, a schedule of one iteration in one cycle may exist.

Let us assume $MaxII = 5$. The first point in the throughput diagram which corresponds to a schedule with maximum throughput is $(II_K, K) = (1, 1)$. Therefore, *UNRET* retimes the loop until an equivalent π-graph which has a schedule in one cycle is found. Figures 6.15(b) and 6.15(c)

cycle	MEM1	MEM2	ADDR1	ADDR2	FMUL	FADD
1	$u_1 = *z_0$	$v_1 = *x_0$	$z_1 = z_0 + 4$	$x_1 = x_0 + 4$		
2	$u_2 = *z_1$	$v_2 = *x_1$	$z_2 = z_1 + 4$	$x_2 = x_1 + 4$		
3	$u_3 = *z_2$	$v_3 = *x_2$	$z_3 = z_2 + 4$	$x_3 = x_2 + 4$		
4	$u_4 = *z_3$	$v_4 = *x_3$	$z_4 = z_3 + 4$	$x_4 = x_3 + 4$		
5	$u_5 = *z_4$	$v_5 = *x_4$	$z_5 = z_4 + 4$	$x_5 = x_4 + 4$		
6	$u_6 = *z_5$	$v_6 = *x_5$	$z_6 = z_5 + 4$	$x_6 = x_5 + 4$		
7	$u_7 = *z_6$	$v_7 = *x_6$	$z_7 = z_6 + 4$	$x_7 = x_6 + 4$	$w_1 = u_1\ v_1$	
8	$u_8 = *z_7$	$v_8 = *x_7$	$z_8 = z_7 + 4$	$x_8 = x_7 + 4$	$w_2 = u_2\ v_2$	
9	$u_9 = *z_8$	$v_9 = *x_8$	$z_9 = z_8 + 4$	$x_9 = x_8 + 4$	$w_3 = u_3\ v_3$	$q_1 = q_0 + w_1$
10	$u_{10} = *z_9$	$v_{10} = *x_9$	$z_{10} = z_9 + 4$	$x_{10} = x_9 + 4$	$w_4 = u_4\ v_4$	$q_2 = q_1 + w_2$
	...	...	...	...	...	...
15	$u_{15} = *z_{14}$	$v_{15} = *x_{14}$	$z_{15} = z_{14} + 4$	$x_{15} = x_{14} + 4$	$w_9 = u_9\ v_9$	$q_7 = q_6 + w_7$
16					$w_{10} = u_{10}\ v_{10}$	$q_8 = q_7 + w_8$
17					$w_{11} = u_{11}\ v_{11}$	$q_9 = q_8 + w_9$
18					$w_{12} = u_{12}\ v_{12}$	$q_{10} = q_9 + w_{10}$
19					$w_{13} = u_{13}\ v_{13}$	$q_{11} = q_{10} + w_{11}$
20					$w_{14} = u_{14}\ v_{14}$	$q_{12} = q_{11} + w_{12}$
21					$w_{15} = u_{15}\ v_{15}$	$q_{13} = q_{12} + w_{13}$
22						$q_{14} = q_{13} + w_{14}$
23						$q_{15} = q_{14} + w_{15}$

Figure 6.16 Overlapped execution of inner product

show such a π-graph and the associated loop body. The found schedule is shown in Figure 6.15(d). This schedule has maximum throughput, and therefore the schedule *efficiency ratio* is $\varepsilon = 1$.

In [DHB89] a schedule in 1 cycle is also found by using the same architecture. The software pipelining technique used overlaps the execution of consecutive iterations of the loop until a repetitive pattern is found. The detailed execution of the loop is shown in Figure 6.16. Since the result is the same obtained by *UNRET*, Figure 6.16 allow us to show the shape of the prologue (cycles 1st to 8th) and the epilogue (cycles 16th to 23rd) for this loop schedule.

6.5.2 Example 2

This example shows how unrolling a loop may increase the execution throughput of the schedule. We will use the first example proposed in Section 2.3. Figure 6.17(a) shows the initial π-graph. We will illustrate here the way to achieve the schedules shown in Figures 6.19(a) and 6.19(b) by assuming that all instructions are additions which can be executed in an adder in a single cycle.

Scheduling by using 4 adders

First, we will assume an architecture with four adders. The objective is finding a schedule no longer than 15 cycles ($MaxII = 15$). The lower and upper bounds for the loop are $MII(\pi) = \frac{3}{4}$ and $MaxTh(\pi) = \frac{4}{3}$, indicating that four iterations may be executed each 3 cycles.

The first point generated by using Farey's series F_{15} is $(II_K, K) = (5, 4)$, labelled X in the throughput diagram from Figure 6.17(b). If no schedule exists for this point, the next points to

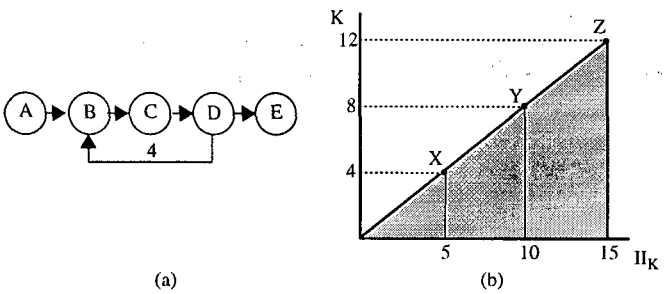

Figure 6.17 Example 2.
(a) π-graph example
(b) Points representing schedules with maximum throughput for 4 adders

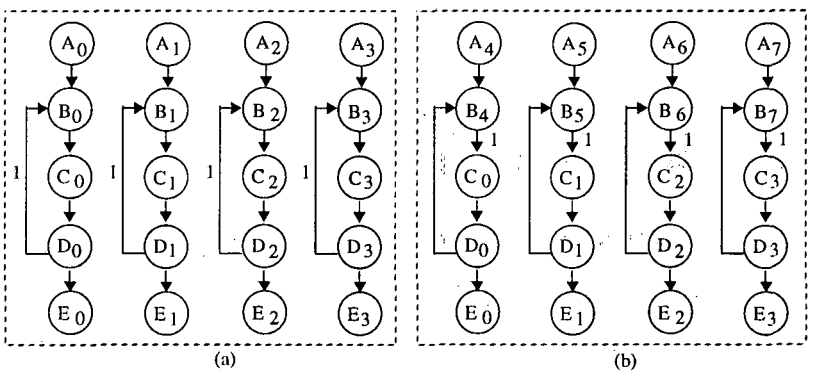

Figure 6.18 Example 2.
(a) initial π-graph unrolled 4 times
(b) π-graph unrolled 4 times after retiming several dependences

explore are the points $Y = (10, 8)$ and $Z = (15, 12)$. Note that Y and Z do not belong to Farey's series F_{15} since they represent a schedule with the same throughput as point X, but they are not reduced.

UNRET selects X as the first point to explore, unrolls the π-graph four times ($K = 4$) and attempts to find a schedule in five cycles ($\Pi_K = 5$). The initial unrolled π-graph is shown in Figure 6.18(a).

Figure 6.19(a) shows the found schedule. Note that no retiming has been necessary to find the schedule. The schedule is a time-optimal schedule for four adders ($\varepsilon = 1$).

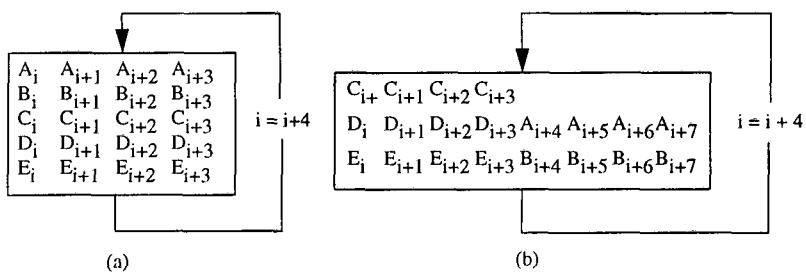

Figure 6.19 Different schedules for the loop
(a) Schedule with 4 adders
(b) Schedule with 8 adders

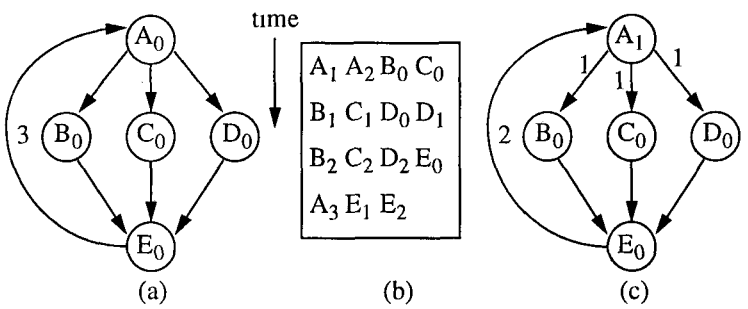

Figure 6.20 Example 3. Π-graph and schedule for 4 adders
(a) Example of π-graph
(b) Schedule found
(c) π-graph corresponding to the found schedule

Scheduling by using 8 adders

Let us now assume that the architecture has eight adders. We will use the same $MaxII = 15$. The first pair generated in this case is $(II_K, K) = (3, 4)$. *UNRET* unrolls the π-graph four times and performs dependence retiming successively until a schedule in three cycles is found. Figure 6.18(b) shows the retimed π-graph corresponding to the schedule found in three cycles and depicted in Figure 6.19(b). This schedule has $\varepsilon = 1$, and therefore it is a time-optimal schedule for the number of adders considered. In fact, this schedule achieves the maximum throughput of the loop.

6.5.3 Example 3

Finally, we will illustrate the effectiveness of using Farey's series when a time-optimal schedule is not found. We will use the second example shown in Section 2.3 to accomplish this purpose. Figure 6.20(a) shows the initial π-graph with five additions which can be executed by an adder in a single cycle. We will assume that the architecture has four adders.

We have executed *UNRET* by assuming $MaxII = 8$ and $MaxII = 15$. In both cases, the first pair explored, (II_K, K)=(5,4), represents a schedule with a throughput $Th = 0.8$. A schedule with such characteristics does not exist, as recently proved in [CBS95] by using an integer linear approach. Therefore, *UNRET* does not find a schedule for this pair and new pairs are generated until a schedule is found. The first pair for which a schedule is found is, in both cases, (II_K, K)=(4,3). The throughput of the schedule is $Th = 0.75$, and the schedule efficiency ratio is $\varepsilon = \frac{0.75}{0.8}$. This point is marked by an arrow in Figure 6.21(a).

Other approaches considering loop unrolling [RG81, SDX86, Rim93] explore the solution space in only one dimension. Since they do not find a schedule at point (II_K, K)=(5,4), they increase the expected initiation interval without modifying the current unrolling degree. Therefore, their second attempt consists of looking for a schedule for point (II_K, K)=(6,4). This point is marked X in Figure 6.21(a). The schedule associated has $Th = 0.66$ and $\varepsilon = 0.83$. Figure 6.21 compares the solution space explored by *UNRET* to the solution space explored by other techniques [RG81,

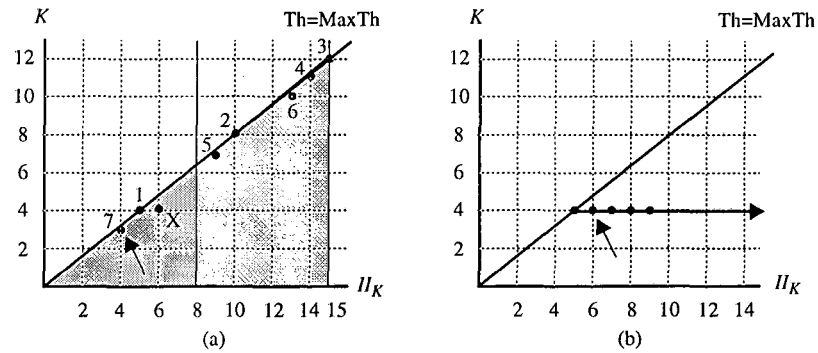

Figure 6.21 Example 3.
(a) Points to explore for F_8 and F_{15}
(b) Points to explore by other techniques [RG81, Huf93, Rau94]

Huf93, Rau94]. The arrow in Figure 6.21(b) indicates the point for which a schedule is found by such techniques.

The exploration of Farey's series allows us to search for several points representing schedules with throughput between $\frac{4}{5}$ and $\frac{4}{6}$. For $MaxII = 8$ the sequence of fractions generated has been $\frac{4}{5}, \frac{3}{4}$. The time used by *UNRET* to find the schedule was 0.46 seconds. For $MaxII = 15$, the sequence generated was $\frac{4}{5}, \frac{8}{10}, \frac{12}{15}, \frac{11}{14}, \frac{7}{9}, \frac{10}{13}, \frac{3}{4}$. The total time used to explore all points was 13.1 seconds by using *retiming_and_scheduling* Algorithm 6.3, and 12.5 seconds by using *retiming_and_scheduling* optimized Algorithm 6.4. Figure 6.21(a) depicts the points corresponding to the fractions explored for $MaxII = 15$ in the throughput diagram. Although more points were explored for $MaxII = 15$, the result obtained was the same as for $MaxII = 8$. However, some of the points would have produced a feasible schedule.

6.6 EXPERIMENTAL RESULTS

This section presents some results obtained by *UNRET*. A more complete set of results is presented in Appendix B. *UNRET* significantly improves some results obtained by other approaches, achieving optimal schedules in most cases.

6.6.1 High-level synthesis

The first columns in Tables 6.1 and 6.3 show the different number of available resources. Following the resources, the *minimum initiation interval* computed for each resource-constraints is specified. The following columns show the *initiation interval* achieved by different approaches: Percolation Based Synthesis (PBS) [PLNG90], the microcode compiler ATOMICS integrated in the CATHEDRAL II compiler (ATM) [GRVD87, GVD89, GRVD90], pipelined synthesis (PLS) [HHL91], the Theda.Fold (TF) algorithm [LWGL92, LWLG94], the Software Package for Synthesis of Pipelines from Behavioral Specifications (SEHWA) [PP88], the Force Directed Scheduling

FUs	MII	Algorithms						(II_K, K)	T
		PBS	ATM	PLS	TF	UNRET			(secs)
∞	3	3	3	3	3	3	$\varepsilon = 1$	(3,1)	0.11
6	3	3	3	3	3	3	$\varepsilon = 1$	(3,1)	0.15
5	17/5	4	4	4	4	17/5	$\varepsilon = 1$	(17,5)	2.18
4	17/4	5	5	5	5	17/4	$\varepsilon = 1$	(17,4)	1.20
3	17/3	6	6	6	6	17/3	$\varepsilon = 1$	(17,3)	0.75

Table 6.1 Cytron's example

FUs		MII	Algorithms					(II_K, K)	T
*	ALUs		FDS	ALPS	MEPS	UNRET			(secs)
3	2	6	6	6	6	6	$\varepsilon = 1$	(6,1)	0.06
2	2	6	7	7	7	6	$\varepsilon = 1$	(6,1)	0.08
2	1	6	-	-	-	6	$\varepsilon = 1$	(6,1)	0.11

Table 6.2 Differential Equation

FUs			MII	Algorithms					(II_K, K)	T
+	-	*		SEHWA	PLS	TF	UNRET			(secs)
5	5	6	8/3	-	3	3	8/3	$\varepsilon = 1$	(8,3)	13.6
4	4	4	4	4	4	4	4	$\varepsilon = 1$	(4,1)	1.43
3	3	4	13/3	5	5	5	22/5	$\varepsilon = 0.98$	(22,5)	36.5
3	3	3	16/3	6	6	6	16/3	$\varepsilon = 1$	(16,3)	14.7
2	2	2	8	10	10	8	8	$\varepsilon = 1$	(8,1)	1.06
1	1	1	16	17	17	17	16	$\varepsilon = 1$	(16,1)	1.13

Table 6.3 Fast Discrete Cosine Transform

(FDS) [PK89a], the Integer Linear Approach ALPS [HLH91, LHL89] and an approach based on the Multiple Exchange Pair Selection (MEPS) [PK91]. The schedule *efficiency ratio* obtained by *UNRET* (obtained throughput / optimal throughput) is shown on the right of the found initiation interval. The penultimate column shows the number of times the loop has been unrolled by *UNRET* (K) and the number of cycles of the found schedule (II_K). The last column shows the CPU time (T) required to find the schedule when using the *retiming_and_scheduling* algorithm without time optimization.

Results from Tables 6.1 and 6.3 show how loop unrolling may increase the throughput of the final schedule.

Third row of Table 6.3 shows how the exploration of Farey's series is useful for finding good schedules. Note that an optimal schedule is not found, but the exploration of Farey's series allows us to find a good schedule ($\varepsilon = 0.98$). For the 1st, 3rd and 4th row, *UNRET* achieves better results due to the ability of unrolling the loop. For the 6th row *UNRET* achieves the *MII* (16 cycles) without unrolling by scheduling only one iteration of the loop, while the remaining techniques require 17 cycles.

Application Program		*MII*	*UNRET*				*HRMS*		
			II	T	ε	(II_K, K)	II	T	ε
SPEC-SPICE	Loop1	1	1	0.01	1	(1,1)	1	0.01	1
	Loop2	2	2	0.13	1	(2,1)	2	0.05	1
	Loop3	6	6	0.05	1	(6,1)	6	0.05	1
	Loop4	10	10	0.41	1	(10,1)	10	0.25	1
	Loop5	2	2	0.08	1	(2,1)	2	0.01	1
	Loop6	2	2	0.13	1	(2,1)	2	0.13	1
	Loop7	1.5	1.5	0.21	1	(3,2)	2	0.13	0.75
	Loop8	1.5	1.5	0.11	1	(3,2)	2	0.03	0.75
	Loop10	3	3	0.06	1	(3,1)	3	0.02	1
SPEC-DODUC	Loop1-f	20	20	0.21	1	(20,1)	20	0.27	1
	Loop3	20	20	0.15	1	(20,1)	20	0.20	1
	Loop7	2	2	0.08	1	(2,1)	2	0.20	1
SPEC-FPPPP	Loop1	20	20	0.10	1	(20,1)	20	0.20	1
Livermore	Loop1	1.5	1.5	0.51	1	(3,2)	2	0.03	0.75
	Loop5	3	3	0.11	1	(3,1)	3	0.03	1
	Loop23	8	8	0.38	1	(8,1)	8	0.13	1
Linpack	Loop1	1	1	0.08	1	(1,1)	1	0.03	1
Whetstone	Loop1	17	17	0.33	1	(17,1)	17	0.15	1
	Loop2	6	6	0.11	1	(6,1)	6	0.13	1
	Loop3	5	5	0.15	1	(5,1)	5	0.03	1
	Cycle1	4	4	0.06	1	(4,1)	4	0.02	1
	Cycle2	2	2	0.11	1	(2,1)	2	0.03	1
	Cycle4	1.33	1.33	0.10	1	(4,3)	2	0.02	0.66
	Cycle8	1.33	1.33	0.11	1	(4,3)	2	0.03	0.66

Table 6.4 Comparison for an architecture with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Since ALPS is an integer linear programming approach, its results are time-optimal. *UNRET* finds schedules with the same initiation interval as ALPS for all cases (more results can be found in Appendix B). Moreover, some results are improved because the schedule is overlapped (see table 6.2). An example is shown in Appendix B) (see Figure B.1(b)).

6.6.2 Superscalar and VLIW processors

By using the set of 24 benchmark loops proposed in Appendix A, we compare the results obtained by *UNRET* with a modulo scheduling approach, called HRMS [LVA95]. Table B.9 shows the results obtained by using an architecture composed of 3 FP-adders, 2 FP-multipliers, 1 FP-divisor and 2 load/store units.

Table B.9 shows that *UNRET* obtains the same results as HRMS for the same unrolling degree. However, the utilization of the optimal unrolling degree improves the results achieved by HRMS in 21 per cent of all cases. This is because, in general, systems which do not perform unrolling try to find a schedule in $\lceil MII \rceil$ cycles when *MII* is not an integer. Note that optimal schedules are found in all cases. Therefore, Farey's series do not require to be explored.

6.7 SUMMARY AND CONCLUSIONS

In this chapter we present *UNRET*, a new algorithm for loop pipelining with resource constraints. *UNRET* generates pairs (K, Π_K) in decreasing order of expected throughput $(\frac{K}{\Pi_K})$, and tries to find a schedule of the loop body unrolled K times in Π_K cycles. Farey's series are used to generate the sequence of pairs.

In order to find a schedule for a given pair, a new software pipelining algorithm is proposed. The algorithm, called *unrolling_and_retiming*, successively transforms the initial loop body by means of *dependence retiming*, searching for another loop body more "suitable" for scheduling. *Quality* is defined in order to decide when a loop body is more appropriate for scheduling than another equivalent one. The *quality* of the loop body is improved until it cannot be further improved or a schedule in Π_K cycles is found. If no schedule is found for the current pair (K, Π_K), a new pair (K', Π'_K) is generated, and the previous steps are repeated until a schedule in the expected initiation interval is found or no further pairs can be generated.

The main contributions of *UNRET* are the following:

1. Unlike the software pipelining approaches proposed by other authors, *UNRET* explores the solution space in two dimensions. Current software pipelining approaches explore the solution space in only one dimension, increasing the expected initiation interval when a schedule is not found by using the previous initiation interval. *UNRET* explores the solution space in two dimensions, the initiation interval and the unrolling degree of the loop.

2. Since the minimum initiation interval of the loop is known in advance, a figure of merit, ε, is defined to measure how far the found schedule is from the theoretically optimal one.

3. A new software pipelining approach, based on *dependence retiming*, is proposed. The algorithm, called *retiming_and_scheduling*, explores several equivalent configurations of a loop body, looking for a schedule in the expected Π. This approach differs from previous ones, which usually explore a single configuration of the loop.

4. Retiming and scheduling are separated into individual tasks. Thus, different scheduling algorithms may be used.

5. By using the dependence theory described in Chapter 4, *quality* is defined to "guess" when a given configuration of a loop body is easier to schedule than another equivalent one. This enables the retiming task to guide the process independently from the scheduling algorithm.

The software pipelining approach proposed in this Chapter is appropriate to HLS systems, since it produces very promising results in less time than other approaches. On the other hand, the approach described in this chapter may help the compiler to find the best schedule for executing a loop in a given architecture. The exploration of Farey's series has proven to be very useful when a time-optimal schedule is not found. Exploring the solution space in two dimensions will produce, in general, better results than exploring it in only one dimension. Since this exploration represents little execution time in face to the time required by a software pipelining approach, it may be included by a compiler without significantly increasing the time to compile a loop.

We also show how a high percentage of the throughput can be obtained (at least) by significantly limiting the maximum number of cycles of a schedule. This limitation produces a reduction in the

number of schedules to explore, avoids code explosion and allows us to obtain schedules which use a reduced number of registers.