A new guaranteed heuristic for the software pipelining problem.

Pierre-Yves Calland, Alain Darte, Yves Robert

▶ **To cite this version:**

Pierre-Yves Calland, Alain Darte, Yves Robert. A new guaranteed heuristic for the software pipelining problem.. [Research Report] LIP RR-1995-42, Laboratoire de l'informatique du parallélisme. 1995, 2+24p. hal-02102096

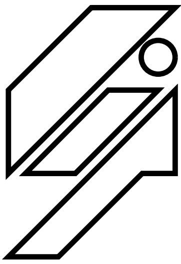
HAL Id: hal-02102096

https://hal-lara.archives-ouvertes.fr/hal-02102096

Submitted on 17 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Laboratoire de l'Informatique du Parallélisme

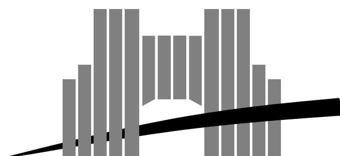
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

A new guaranteed heuristic for the software pipelining problem

Pierre-Yves Calland
Alain Darte
Yves Robert

November 1995

Research Report N° 95-42



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens–lyon.fr

A new guaranteed heuristic for the software pipelining problem

Pierre-Yves Calland
Alain Darte
Yves Robert

November 1995

Abstract

We present yet another heuristic for the software pipelining problem. We believe this heuristic to be of interest because it brings a new insight to the software pipelining problem by establishing its deep link with the circuit retiming problem. Also, in the single resource class case, our new heuristic is guaranteed, with a better bound than that of [6]. Finally, we point out that, in its simplest form, our algorithm has a lower complexity.

Keywords: Software pipelining, circuit retiming, guaranteed heuristic, list scheduling, cyclic scheduling.

Résumé

Nous présentons une nouvelle heuristique pour le problème du pipeline logiciel. Nous montrons, par cette nouvelle approche, l'existence d'un lien étroit entre le problème du pipeline logiciel et le problème de resynchronisation des circuits. De plus, nous montrons que dans le cas de ressources identiques, notre heuristique est garantie (avec une borne de garantie meilleure que celle obtenue pour l'heuristique de Gasperoni et Schwiegelshohn [6]) et, dans sa forme non optimisée, une complexité moindre.

Mots-clés: Pipeline logiciel, resynchronisation de circuits, heuristiques garanties, ordonnancement de liste, ordonnancement cyclique.

A new guaranteed heuristic for the software pipelining problem

Pierre-Yves Calland[*], Alain Darte[†] and Yves Robert[‡]

Laboratoire LIP, URA CNRS 1398
Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07
e-mail: [Pierre-Yves.Calland,Alain.Darte,Yves.Robert]@lip.ens-lyon.fr

October 1995

Abstract

We present yet another heuristic for the *SP* problem. We believe this heuristic to be of interest because it brings a new insight to the *SP* problem by establishing its deep link with the circuit retiming problem. Also, in the single resource class case, our new heuristic is guaranteed, with a better bound than that of [6]. Finally, we point out that, in its simplest form, our algorithm has a lower complexity.

Keywords Software pipelining, circuit retiming, guaranteed heuristic, list scheduling, cyclic scheduling.

1 Introduction

Software pipelining (*SP* hereafter) is a technique aimed at the efficient execution of simple loops. The main problem is to cope with both dependence and resource constraints which make the problem NP-complete in general. Consider the following example to illustrate the discussion. We will work on this example all along the paper.

Example

DO $k = 0, N$
- (op_1): $a(k) = c(k-1)$
- (op_2): $b(k) = a(k-2) * d(k-1)$
- (op_3): $c(k) = b(k) + 1$
- (op_4): $d(k) = f(k-1)/3$
- (op_5): $e(k) = \sin(f(k-2))$
- (op_6): $f(k) = \log(b(k) + e(k))$

ENDDO

[*]Supported by Region Rhône-Alpes

[†]Corresponding author. Telephone +33 72 72 83 89, Fax +33 72 72 80 80, e-mail Alain.Darte@lip.ens-lyon.fr

[‡]Supported by the ESPRIT Basic Research Action 6632 *NANA2* of the European Economic Community and by the CNRS-INRIA project *ReMaP*

The instructions $(op_1), \ldots, (op_n)$ within the loop ($n = 6$ in our example) are called *generic tasks (or operations)*. Each of them is executed $N+1$ times, where N is assumed to be very large. Instance k of operation op_i is denoted (op_i, k), and its execution is scheduled to begin at time $\sigma(op_i, k) \geq 0$ and to last $\delta(op_i)$ units of time.

The goal is to determine a schedule σ to minimize the total execution time $T = \max\{\sigma(op_i, k) + \delta(op_i), 1 \leq i \leq n, 0 \leq k \leq N\}$ subject to *resource constraints*, and while preserving the semantics of the original loop (*dependence constraints*).

Resource constraints In the most general instance of the *SP* problem, resource constraints are expressed as follows: generic tasks are partitioned into classes. To each class $\mathcal{C}$ corresponds a given (finite) number $p(\mathcal{C})$ of available processors (or resources). For example think of tasks being partitioned into additions (class $\mathcal{C}_1$) and multiplications (class $\mathcal{C}_2$); then $p(\mathcal{C}_1)$ is the number of available adders and $p(\mathcal{C}_2)$ that of multipliers. We need to ensure that the scheduling σ satisfies to the resource constraints, i.e. that at any time-step and for each resource class $\mathcal{C}$, no more than $p(\mathcal{C})$ instances of tasks belonging to class $\mathcal{C}$ are being executed.

In a simpler instance of the *SP* problem, there is a single resource class composed of p identical processors. Each operation op can be executed indifferently on any processor, with delay $\delta(op)$. Although less general, this single resource class instance has motivated a great deal of research, for at least two reasons:

- it has a great importance in practice, as it models fine-grain parallelism extraction in shared memory machines with programmable ALUs (or off-the-shelf microprocessors),
- it is the only case where guarantees exist [6] for scheduling heuristics. This is very important as it gives a sound basis for comparing heuristics (the only alternative is to multiply experiments).

Dependence constraints Dependence constraints express the fact that some computations must be executed in a specified order so as to preserve the semantics of the loop. In our example, computation (op_1, k) writes $a(k)$, hence it must precede computation $(op_2, k+2)$ which reads this value. There is also (among others) a dependence from $(op_6, k-2)$ to (op_5, k), because $f(k-2)$ must be read in (op_5, k). All dependences are usually captured into a reduced dependence graph (see Figure 1).

The *SP* problem has motivated a great amount of research. Since the pioneering work of Aiken and Nikolau [1], several authors have proposed various heuristics [6, 2, 5, 7, 8] [1]. The main contribution of this paper is to present yet another heuristic for the *SP* problem. We believe this heuristic to be of interest because it brings a new insight to the *SP* problem by establishing its deep link with the circuit retiming problem. Also, in the single resource class case, our new heuristic is guaranteed, with a better bound than that of [6]. Finally, we point out that, in its simplest form, our algorithm has a lower complexity.

The rest of the paper is organized as follows: in Section 2, we formally define the *SP* problem and we briefly survey complexity results and heuristics from the literature. In Section 3, we present the guaranteed heuristic of Gasperoni and Schwiegelshohn [6]. In Section 3.3, we show how to modify this heuristic so as to cut more edges in the dependence graph (hence expecting better results). This gives the starting point of our heuristic whose presentation is split over Sections 4 and 5. Furthermore, we show that the bound for our heuristic is better than the bound in [6]. Finally,

[1] This small list is far from being comprehensive.

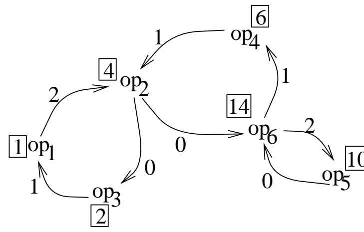


Figure 1: The reduced dependence graph G

we discuss some extensions in Section 6. We summarize our results and give some perspectives in Section 7.

2 Known results on the *SP* problem

2.1 Problem formulation

Formally, software pipelining problem instances are represented by a finite, vertex-weighted, edge-weighted directed multi-graph $G = (V, E, \delta, d)$. The vertices V of the graph model the generic tasks: $V = \{op_1, op_2, \ldots, op_n\}$. Each generic task op_i has a positive delay (or latency) $\delta(op_i)$. Vertex (task) delays δ can be rational numbers, but since the graph is finite we can always change the time unit to have integer delays. The graph G is often called the *reduced dependence graph* in the literature.

Each generic task op_i has several instances (op_i, k), $0 \leq k \leq N$. The problem is to find a schedule σ that assigns a time-step $\sigma(op_i, k)$ to begin the execution of each computation (op_i, k).

The directed edges E of the graph model dependence constraints. Let $e = (op_i, op_j) \in E$ be an edge of G with weight $d(e)$: this means that instance k of generic task op_i must be completed before the execution of instance $k + d(e)$ of generic task op_j. In other words, we have the scheduling constraints:

$$\forall e = (op_i, op_j) \in E : \sigma(op_i, k) + \delta(op_i) \leq \sigma(op_j, k + d(e))$$

Edge delays are nonnegative integers by construction. Some edge delays may be equal to zero, but there does not exist any cycle in G whose length (the sum of the edge delays) is zero (otherwise some computation depends upon itself). The multi-graph G for our example is depicted in Figure 1: operator delays are given in square boxes.

Resource constraints are expressed as follows: generic tasks are partitioned into classes. To each class $\mathcal{C}$ corresponds a given number $p(\mathcal{C})$ of available processors (or resources). We need to ensure that at any time-step no more than $p(\mathcal{C})$ instances of tasks belonging to class $\mathcal{C}$ are being executed. This translates into the following formula:

$$\forall t \geq 0, \forall \mathcal{C}, |\{(op, k), op \in \mathcal{C}, 0 \leq k \leq N, t - \delta(op) < \sigma(op, k) \leq t\}| \leq p(\mathcal{C}).$$

Resource constraints can be expressed slightly differently in the case of pipelined processors. If we assume processors of class $\mathcal{C}$ to be pipelined, then no more than $p(\mathcal{C})$ tasks of class $\mathcal{C}$ can be initiated at each unit of time.

The *SP* problem can therefore be reduced to the determination of the schedule σ subject to the above constraints. *Valid* schedules are those schedules satisfying all constraints (both dependence constraints and resource constraints). Because of the regular structure of the *SP* problem, we

usually search for a *cyclic* schedule σ: we aim at finding a nonnegative integer λ (called *the initiation interval* of σ) and constants c_i such that

$$\sigma(op_i, k) = \lambda k + c_i$$

Because the input loop is supposed to execute many iterations (N is large), we focus on the asymptotic behavior of σ. The initiation interval λ is a natural performance estimator of σ, as $1/\lambda$ measures σ's throughput. Note that if the reduced dependence graph G is acyclic and if the target machine has enough processors, then λ can be zero (this type of schedule has infinite throughput).

A variant consists in searching for a nonnegative rational $\lambda = u/v$ and to let $\sigma(op_i, k) = \lfloor \lambda k + c_i \rfloor$ (with rational constants c_i). This amounts to unroll the input loop by a factor v. Note that rational cyclic schedules are dominant in the case of unlimited resources [2].

In the following, we restrict to the simplest case, i.e. the case of **a single resource class** of **non pipelined processors**. Our heuristic does apply to many resource classes, but we are no longer able to guarantee it in this case.

2.2 Related work

Many papers on software pipelining have been published. In this section, we very briefly summarize four of the most recent ones, by Feautrier [5], Hanen and Munier [2], Gasperoni and Schwiegelshohn [6] and Wang et al. [8].

Feautrier [5] Feautrier formalizes the *SP* problem in terms of integer linear programming, in the case of different resource classes. He gives two scheduling algorithms, one for the case $p(\mathcal{C}) = 1$ (which means that all resources are considered as different) and another for the general case. In both cases, it is shown how to translate resource constraints into systems of bilinear integer constraints. This technique permits to derive optimal solutions, though in non-polynomial time.

Hanen and Munier [2] Hanen and Munier use a very interesting graph-based approach. They restrict themselves to the single resource class case. They use "tie-breaking" graphs to derive schedules. Basically, the idea is to add some edges to the reduced dependence graph. These new edges link task instances to be executed on the same group of processors. The problem is to determine how to partition processors into groups and how to add new edges. The proposed heuristics look quite powerful, although not guaranteed.

Gasperoni and Schwiegelshohn [6] Gasperoni and Schwiegelshohn tackle the single resource class case, both with pipelined and non-pipelined processors. They use yet another approach: they separate, so to speak, dependence constraints and resource constraints. They first schedule the reduced dependence graph G assuming an unlimited number of processors. Such a schedule is the basis to decide which edges to cut in G so as to make it acyclic. On the new acyclic graph G_a, they use a list-scheduling heuristic to cope with the resource constraints. The key-point is that the final scheduling is guaranteed. We explain Gasperoni and Schwiegelshohn's approach with full details in Section 3, and we build upon their results.

Wang, Eisenbeis, Jourdan and Su [8] Wang et al. have an approach very similar to that of Gasperoni and Schwiegelshohn. The main difference is in the selection criterion to cut edges in G so as to make it acyclic. They cut more edges than Gasperoni and Schwiegelshohn, thereby

expecting better results when list-scheduling the acyclic graph G_a.

For our new heuristic, we apply this idea of cutting edges too. Our goal is twofold:

- Miminize the longest path in the acyclic graph G_a so as to have the best possible performance bound, hence improving the heuristic guarantee,
- Minimize the number of edges in G_a so as to have as few constraints as possible.

As already mentioned, we first describe Gasperoni and Schwiegelshohn's approach (Section 3) before introducing our new heuristic (Section 3.3).

3 Going from cyclic scheduling to acyclic scheduling

Before going into the details of Gasperoni and Schwiegelshohn heuristic (GS for short), we recall some properties of cyclic schedules, so as to make the rest of the presentation clearer.

3.1 Some properties of cyclic scheduling

Given the dependence graph $G = (V, E, \delta, d)$, a cyclic schedule σ is a schedule of the form:

$$\sigma(op_i, k) = \lambda k + c_i \text{ where } \lambda,\ c_i,\ k \in \mathbb{N}$$

that satisfies both dependence constraints and resource constraints. Such a cyclic schedule is periodic, with period λ: the computation scheme is reproduced every λ units of time. More precisely, if instance (op_i, k) is assigned to begin at time t, then instance $(op_i, k+1)$ will begin at time $t + \lambda$. Therefore we only need to study a slice of λ clock cycles to know the behavior of the whole cyclic scheduling in steady state.

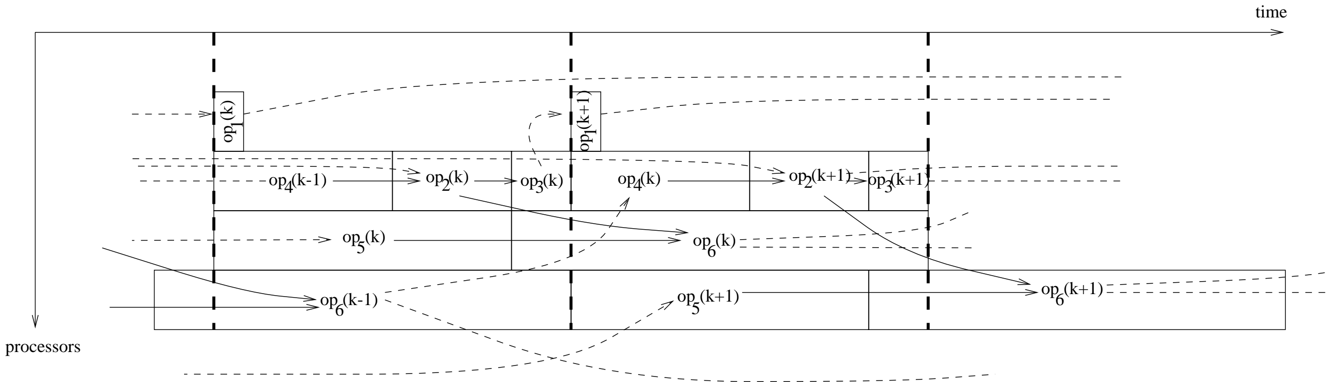


Figure 2: Successive slices of a schedule for graph G (unbroken lines represent type 1 dependences, dotted lines represent type 2 dependences).

Let us observe such a slice, e.g. the slice S_K from clock cycle $K\lambda$ up to clock cycle $(K+1)\lambda - 1$, where K is large enough so that the steady state is reached (see Figure 2 for an example with an optimal schedule and unlimited resources). Perform the Euclidean division of c_i by λ: $c_i = r_i + \lambda q_i$ where $0 \leq r_i \leq \lambda - 1$. Then

$$\sigma(op_i, k) = r_i + \lambda(k + q_i)$$

This means that one and only one instance of op_i is initiated within the slice S_K: it is instance $k = K - q_i$, started r_i clock cycles after the beginning of the slice.

If the schedule is valid, both resource constraints and dependence constraints are satisfied. The latter constraints can be satisfied because of two different reasons: either two dependent computation instances are initiated in the same slice S_K (type 1) or they are initiated in two different slices (type 2). Of course, the partial dependence graph induced by type 1 constraints is acyclic, because type 1 dependences impose a partial order on the operations, according to their apparition order within the slice.

The main idea of GS is the following. Assume that we have a valid cyclic schedule of period λ_0 for a given number p_0 of processors, and that we want to deduce a valid schedule for a smaller number p of processors. A way of building the new schedule is to keep the same slice structure, i.e. to keep the same operation instances within a given slice. Of course we might need to increase the slice length to cope with the reduction of resources. In other words, we have to stretch the rectangle of size $\lambda_0 \times p_0$ to build a rectangle of size $\lambda \times p$. Using this idea, type 2 dependences will still be satisfied if we choose λ large enough. Only type 1 dependences have to be taken into account for the internal reorganization of the slice (see Figure 3). But since the corresponding partial dependence graph is acyclic, we are brought back to a standard acyclic scheduling problem for which many theoretical results are known. In particular, a simple list scheduling technique provides a *guaranteed* heuristic (and the shorter the longest path in the graph, the more accurate the heuristic bound).

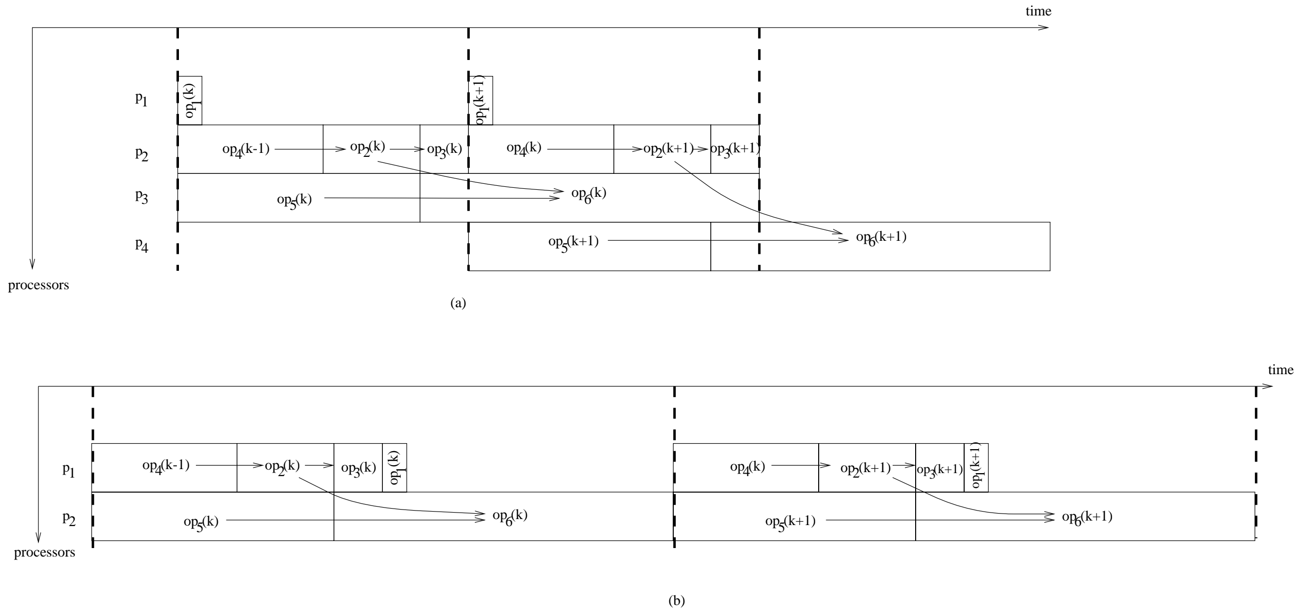


Figure 3: Two different allocations of a slice of graph G ($p_0 = 4$, $p = 2$)

Once this main principle settled, there remain several open questions:

1. How to choose the initial scheduling?
2. How to choose the reference slice? (There is no reason *a priori* to choose a slice beginning at a clock cycle congruent to 0 modulus λ_0)
3. How to decide that an edge is of type 1, hence to be considered in the acyclic problem?

These three questions are of course linked together. Intuitively, it seems important to (try to) minimize both

- the length of the longest path in the acyclic graph, which should be as small as possible as it is tightly linked to the guaranteed bound for the list scheduling,
- and the number of edges in the acyclic graph, so as to reduce the dependence constraints for the acyclic scheduling problem.

We will give a precise formulation to these questions and give a solution. Beforehand, we review the choices of GS.

3.2 The heuristic of Gasperoni and Schwiegelshohn

In this section we explain with full details the GS heuristic [6]. The main idea is as outlined in the previous section. The choice of GS for the initial scheduling is to consider the optimal cyclic scheduling for an infinite number of processors ($p_0 = \infty$), i.e. without resource constraints.

3.2.1 Optimal schedule for unlimited resources

Consider the cyclic scheduling problem $G = (V, E, \delta, d)$ without resource constraints ($p = \infty$).

Let λ be a nonnegative integer. Define from G an edge-weighted graph $G'_\lambda = (V', E', d')$ as follows:

Vertices of G'_λ Add to V a new vertex s: $V' = V \cup \{s\}$.

Edges of G'_λ Add to E an edge from s to all other vertices: $E' = E \cup (\{s\} \times V)$.

Weight of edges of G'_λ Define $d'(e) = 0$ if $e \in E' \setminus E$ and $d'(e) = \delta(op_i) - \lambda d(e)$ if $e = (op_i, op_j) \in E$.

We have the following well-known result:

Lemma 1 *λ is a valid initiation interval $\Leftrightarrow$ G'_λ has no cycle of positive weight.*

Proof

- If λ is a valid initiation interval, there is a cyclic schedule $\sigma(op_i, k) = c_i + \lambda k$ that satisfies the dependence constraints:

$$\forall e = (op_i, op_j) \in E : \ c_i + \delta(op_i) \leq \lambda d(e) + c_j \tag{1}$$

  Consider a cycle C in G'_λ. Note that C is a cycle of G, too. Summing all inequalities (1) that involve the edges of C leads to:

$$\sum_{op_i \in C} (c_i + \delta(op_i)) \leq \lambda \sum_{e \in C} d(e) + \sum_{op_j \in C} c_j \Leftrightarrow \delta(C) \leq \lambda d(C) \Leftrightarrow d'(C) \leq 0$$

- Conversely, if G' has no cycle of positive weight, one can define, for all $op \in V$, the longest path (in G') from s to op, that we denote by $t(s, op)$.

  By definition, $t(s, op)$ satisfies the following triangular inequality:

  $$\forall e = (op_i, op_j) \in E,\ t(s, op_j) \geq t(s, op_i) + d'(e)$$

  i.e.

  $$\forall e = (op_i, op_j) \in E,\ t(s, op_i) + \delta(op_i) \leq t(s, op_j) + \lambda d(e)$$

  This proves that $\sigma(op, k) = t(s, op) + \lambda k$ is a valid cyclic schedule. ■

Lemma 1 has two important consequences:

- First, given an integer λ, it is easy to determine if λ is a valid initiation interval and if yes, to build a corresponding cyclic schedule by applying Bellman-Ford's algorithm [4] on G'_λ.
- The optimal initiation interval λ_∞ is the smallest integer λ such that G'_λ has no positive cycle. Therefore, $\lambda_\infty = 0$ if G is acyclic and $\lambda_\infty = \max\{\lceil \frac{\delta(C)}{d(C)} \rceil;\ C \text{ cycle of } G\}$ otherwise.

3.2.2 Algorithm GS for p resources

As said before, in the case of p identical processors, the algorithm consists in the conversion of the dependence graph G into an acyclic graph G_a. G_a is obtained by deleting some edges of G. As initial scheduling, GS takes the optimal scheduling with unlimited resources

$$\sigma_\infty(op_i, k) = t(s, op_i) + \lambda_\infty k.$$

As reference slice, GS takes a slice starting at a clock cycle congruent to 0 modulus λ_∞, i.e. a slice from clock cycle $K\lambda_\infty$ up to clock cycle $(K+1)\lambda_\infty - 1$. This amounts to decomposing $t(s, op_i)$ into

$$t(s, op_i) = r_i + \lambda_\infty q_i \text{ where } 0 \leq r_i \leq \lambda_\infty - 1$$

In other words $r_i = t(s, op_i) \bmod \lambda_\infty$. Consider an edge $e = (op_i, op_j) \in E$. In the reference slice, the computation instance $(op_i, K - q_i)$ is performed. If $r_i + \delta(op_i) > r_j$, the computation instance of op_j which is performed within the reference slice (namely $(op_j, K - q_j)$) is started before the end of the computation $(op_i, K - q_i)$. Hence this computation instance $(op_j, K - q_j)$ is not the one that depends upon completion of $(op_i, K - q_i)$. In other words, $K - q_i + d(e) \neq K - q_j$. The two computations in dependence through edge e are not initiated in the same slice. Edge e can be safely considered as a type 2 edge, thus can be deleted from G. This is the way edges are cut in GS heuristic[2]. We are led to the following algorithm:

Algorithm 1 (Algorithm GS)

1. *Compute the optimal cyclic schedule σ_∞ for unlimited resources.*
2. *Let $e = (op_i, op_j)$ be an edge of G. Then e will be deleted from G if and only if*

   $$t(s, op_j) \bmod \lambda_\infty < t(s, op_i) \bmod \lambda_\infty + \delta(op_i) \qquad (C_1)$$

   This provides the acyclic graph G_a.

[2]However, this is not the best way to determine type 2 edges. See Section 3.3.

3. (a) *Consider the acyclic graph G_a where vertices are weighted by δ and edges represent task dependences, and perform a list scheduling σ_a on the p processors.*

   (b) *Let $\lambda = \max_{op_i}(\sigma_a(op_i) + \delta(op_i))$ be the latency of the schedule for G_a.*

4. *For all $op_i \in E$ and $k \in \mathbb{N}$:*

$$\sigma(op_i, k) \;=\; \sigma_a(op_i) + \lambda \left\lfloor \frac{t(s, op_i)}{\lambda_\infty} \right\rfloor + \lambda k$$

   is a valid cyclic schedule.

The correctness of Algorithm GS can be found in [6]. It can also be deduced from the correctness of Algorithm CDR (see Section 4.2.1).

3.2.3 Performances of Algorithm GS

GS gives an upper bound to the initiation interval λ obtained by Algorithm 1. Let λ_{opt} be the optimal (smallest) initiation interval with p processors. The following inequality is established:

$$p\lambda \leq p\lambda_{opt} + (p-1)\Phi \tag{2}$$

where Φ is the length of the longest path in G_a. Moreover, owing to the strategy for cutting edges, $\Phi \leq \lambda_\infty + \delta_{max} - 1$ (see Lemma 1 in [6]). This implies:

$$p\lambda \leq p\lambda_{opt} + (p-1)(\lambda_\infty + \delta_{max} - 1)$$

which leads to

$$\frac{\lambda}{\lambda_{opt}} \leq 2 - \frac{1}{p} + \left(\frac{p-1}{p}\right)\left(\frac{\delta_{max} - 1}{\lambda_{opt}}\right)$$

GS is the first guaranteed algorithm. We see from equation (2) that the bound directly depends upon Φ, the length of the longest path in G_a.

Example We go back to our example. Assume $p = 2$ available processors. Figure 4 (a) recalls the graph G for which $\lambda_\infty = 12$. In Figure 4 (b), we depict the graph G'_{12} that permits to compute $t(s, op)$ for all op (the different values $t(s, op)$ are given in circles on the figure) and an optimal schedule with unlimited resources $\sigma_\infty(op, k) = t(s, op) + \lambda_\infty k$. This schedule was already represented Figure 2: 4 processors are needed. Figure 4 (c) shows the acyclic graph G_a obtained by cutting edges $e = (op_i, op_j)$ such that $r_j < r_i + \delta(op_i)$ where $r_i = t(s, op_i) \bmod \lambda_\infty$. Finally, Figure 4 (d) shows a possible schedule of operations provided by a list scheduling, for which $\lambda = 25$.

3.3 Cutting edges by retiming

Let us summarize Algorithm GS as follows: first compute the values $t(s, op)$ in G'_{λ_∞} to provide the optimal scheduling without resource constraints σ_∞. Then take a reference slice starting at top $0 \bmod \lambda_\infty$:

$$\sigma_\infty(op_i, k) = r_i + \lambda_\infty(k + q_i) \text{ with } 0 \leq r_i \leq \lambda_\infty - 1$$

Finally, delete from G *some* edges that necessarily correspond to dependences between different slices: *only* those edges $e = (op_i, op_j)$ such that $r_i + \delta(op_i) > r_j$ are removed by GS.

However, edges that correspond to dependences between different slices are those such that $q_i \neq q_j + d(e)$. Indeed, within the reference slice, the scheduled computation instances are $(op_i, K - q_i)$

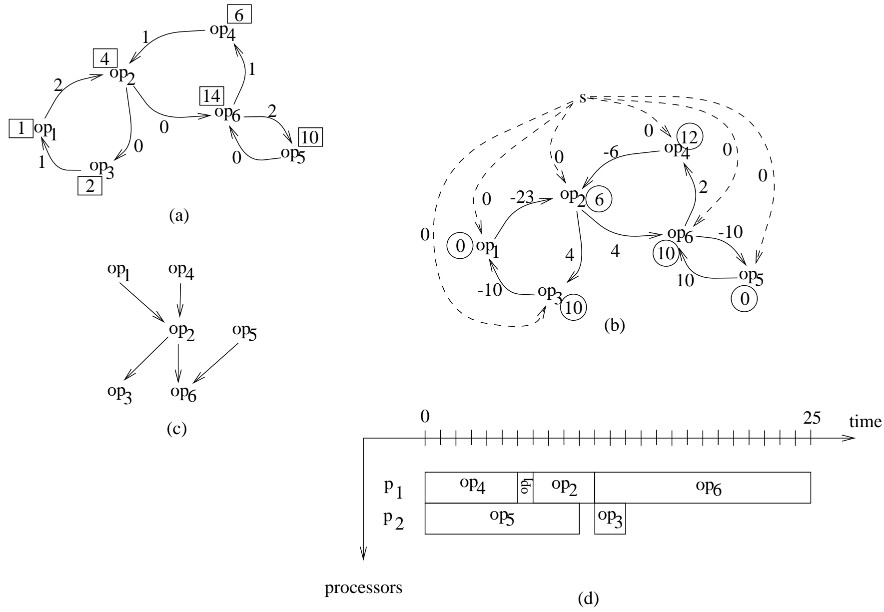


Figure 4: (a): The reduced dependence graph G: $\lambda_\infty = 12$
(b): The graph G'_{12}
(c): The acyclic graph G_a
(d): A corresponding list scheduling allocation: $\lambda = 25$

and $(op_j, K - q_j)$ for edge $e = (op_i, op_j)$. Therefore, the computation $(op_j, K - q_i + d(e))$, which depends upon $(op_i, K - q_i)$, is performed in the same slice iff $K - q_i + d(e) = K - q_j$, i.e. $q_j + d(e) = q_i$. Otherwise, it is performed in a subsequent slice, and in this case $q_j + d(e) > q_i$. Therefore, the condition for cutting edges corresponding to dependences between different slices (i.e. those we called type 2 dependences) writes $q_j + d(e) > q_i$ rather than $r_i + \delta(op_i) > r_j$.

Let us check this mathematically. Consider a valid cyclic scheduling $\sigma(op_i, k) = c_i + \lambda k$ and let $c_i = r_i + \lambda q_i$ with now $t_0 \leq r_i \leq t_0 + \lambda - 1$, where t_0 is given: we look at an arbitrary slice of length λ. For each edge $e = (op_i, op_j)$, the dependence constraint is satisfied, thus $r_i + \delta(op_i) + \lambda q_i \leq r_j + \lambda(q_j + d(e))$. Then,

$$\begin{cases} & r_i + \delta(op_i) + \lambda q_i \leq r_j + \lambda(q_j + d(e)) \\ \Leftrightarrow & (r_i - r_j) + \delta(op_i) \leq \lambda(q_j + d(e) - q_i) \\ \Rightarrow & -\lambda + 1 \leq \lambda(q_j + d(e) - q_i) \\ \Rightarrow & -1 < q_j + d(e) - q_i \\ \Rightarrow & 0 \leq q_j + d(e) - q_i \end{cases}$$

Furthermore, if $q_j + d(e) - q_i = 0$, then the dependence constraints directly writes $r_i + \delta(op_i) \leq r_j$. Conversely, if $r_i + \delta(op_i) > r_j$, then necessarily $q_j + d(e) - q_i > 0$: if an edge is cut by GS, then it is also cut by our new rule. We are led to a modified version of GS which we call mGS. Since we cut more edges in mGS than in GS, the acyclic graph mG_a obtained by mGS contains a subset of the edges of the acyclic graph G_a. See Figure 5 to illustrate this fact.

We are now ready to formulate the problem. We need neither an initial ordering nor a reference slice any longer. What we only need is to determine a function $q : V \longrightarrow \mathbb{Z}$ such that:

$$\forall e = (u, v) \in E, \; q(v) + d(e) - q(u) \geq 0$$

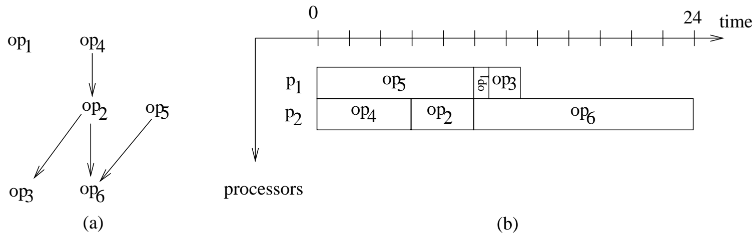


Figure 5: (a): The acyclic graph provided by Algorithm mGS
(b): A corresponding list scheduling allocation: $\lambda = 24$

Such a function q is called a *retiming* in the context of synchronous VLSI circuits [9]. Retiming is an assignment of an integer lag $q(u)$ to each vertex $u \in V$: it amounts to suppress $q(u)$ "registers" to the weight of each edge leaving u (whose tail is u) and to add $q(v)$ registers to each edge entering v (whose head is v). It leads to a new edge-weighting function d_q defined for an edge $u \xrightarrow{e} v$ by $d_q(e) = d(e) + q(v) - q(u)$. After a suitable retiming is found, we define the acyclic graph mG_a as follows: an edge $e \in E$ is kept in mG_a iff its new weight $d_q(e)$ is 0 (edge "without register"). Clearly, mG_a is acyclic (assume there is a cycle, and sum up retimed edge weights on this cycle to get a contradiction). Given mG_a, we list schedule it as a DAG whose vertices are weighted by the initial δ function.

Recall that our goal was to answer the two following questions:

- How to cut edges so as to obtain an acyclic graph G_a whose longest path has minimal length?
- How to cut as many edges as possible so that the number of dependence constraints to be satisfied by the list-scheduling of G_a is minimized?

Now, using our new formulation, we can state our objectives more precisely in terms of retiming:

Objective 1 Find a retiming q that minimizes the longest path in mG_a, i.e. in terms of retiming, that minimizes the clock period Φ of the retimed graph (see Section 4).

Objective 2 Find a retiming q so that the number of edges in mG_a is minimal, i.e. distribute registers so as to leave as few edges without registers as possible.

In Section 4, we show how to achieve the first objective (this is a well-known problem). There are several possible solutions, and in Section 5, we show how to select the best one with respect to the second objective, and we state our final algorithm. We improve upon GS for two reasons: first we have a better bound, and second we cut more edges, hence more freedom for the list scheduling.

4 Minimizing the longest path of the acyclic graph

There are well-known retiming algorithms that can be used to minimize the clock period of a VLSI circuit. In this section, we show how to use such algorithms to derive a valid value $q(op_i)$ for each operator op_i.

4.1 Retiming algorithms

Formally, a retiming of a graph $G = (V, E, \delta, d)$ is a vertex-labeling function $q : V \longrightarrow \mathbb{Z}$. q performs a transformation of the initial graph G into a new graph $G_q = (V, E, \delta, d_q)$ where d_q is defined as

follows: if $e = (u, v)$ is an edge of E then

$$d_q(e) = d(e) + q(v) - q(e)$$

Such a retiming is valid if for each edge e of E, $d_q(e) \geq 0$. Note that we assumed that in any cycle of G there is at least one edge whose weight is positive: using VLSI terminology, we say G is synchronous.

Leiserson and Saxe [9] present several algorithms to compute an *optimal* valid retiming, in the sense that the longest path of null weight in the retimed graph is as short as possible. Before presenting these algorithms with more details, we need some definitions. We denote by $u \stackrel{P}{\leadsto} v$ a path P of G from u to v, by $d(P) = \sum_{e \in P} d(e)$ the sum of the dependences of the edges of P, and by $\delta(P) = \sum_{v \in P} \delta(v)$ the sum of the delays of the vertices of P. We define D and Δ as follows:

$$D(u, v) = \min\{d(P) : u \stackrel{P}{\leadsto} v\}$$

$$\Delta(u, v) = \max\{\delta(P) : u \stackrel{P}{\leadsto} v \text{ and } d(P) = D(u, v)\}$$

D and Δ are computed by solving an all-pairs shortest-path algorithm on G where edge $u \stackrel{e}{\to} v$ is weighted with the pair $(d(e), -\delta(u))$. Finally, let

$$\Phi(G) = \max\{\delta(P) : P \text{ path of } G,\ d(P) = 0\}$$

$\Phi(G)$ is the length of the longest path of null weight in G (and is called the clock period of G in VLSI terminology).

Theorem 1 (Theorem 7 in [9]) *Let $G = (V, E, \delta, d)$ be a synchronous circuit, let λ be an arbitrary positive real number, and let q be a function from V to the integers. Then q is a legal retiming of G such that $\Phi(G_q) \leq \lambda$ if and only if*

1. *$q(u) - q(v) \leq d(e)$ for every edge $u \stackrel{e}{\to} v$ of G, and*
2. *$q(u) - q(v) \leq D(u, v) - 1$ for all vertices $u, v \in V$ such that $\Delta(u, v) > \lambda$.*

Theorem 1 provides the basic tool to establish the following algorithm (Algorithm 2) that determines a retiming such that the clock period of the retimed graph is minimized.

Algorithm 2 (Algorithm OPT1 in [9])

1. *Compute D and Δ (see Algorithm WD in [9]).*
2. *Sort the elements in the range of Δ.*
3. *Binary search among the elements $\Delta(u, v)$ for the minimum achievable clock period. To test whether each potential clock period λ is feasible, apply the Bellman-Ford algorithm to determine whether the conditions in Theorem 1 can be satisfied.*
4. *For the minimum achievable clock period found in step 3, use the values for the $q(v)$ found by the Bellman-Ford algorithm as the optimal retiming.*

This algorithm runs in $O(|V|^3 \log |V|)$, but there is a more efficient algorithm whose complexity is $O(|V||E| \log |V|)$, which is a significant improvement for sparse graphs. It runs as the previous algorithm except in step 3 where the Bellman-Ford algorithm is replaced by the following algorithm:

Algorithm 3 (Algorithm FEAS in [9]) *Given a synchronous circuit $G = (V, E, \delta, d)$ and a desired clock period λ, this algorithm produces a retiming q of G such that G_q is a synchronous circuit with clock period $\Phi \leq \lambda$, if such a retiming exists.*

1. *For each vertex $v \in V$, set $q(v)$ to 0.*
2. *Repeat the following $|V| - 1$ times:*
   (a) *Compute graph G_q with the existing values for q.*
   (b) *for any vertex $v \in V$ compute $\Delta_0(v)$ the maximum sum $\delta(P)$ of vertex delays along any zero-weight directed path P in G leading to v. This can be done in $O(|E|)$.*
   (c) *For each vertex v such that $\Delta_0(v) > \lambda$, set $q(v)$ to $q(v) + 1$.*
3. *Run the same algorithm used for step 2(b) to compute Φ. If $\Phi > \lambda$ then no feasible retiming exists. Otherwise, q is the desired retiming.*

After performing a retiming q to obtain the graph G_q with minimal clock period, we convert this graph into an acyclic one by deleting edges with positive weight. From the definition of a retiming, we see that the sum $d(C)$ along any cycle C of G remains unchanged, i.e. $d_q(C) = d(C)$. Furthermore since any cycle of the dependence graph G contains at least one edge with positive weight, the graph we obtain by deleting edges with positive weight is acyclic.

4.2 A new scheduling algorithm: Algorithm CDR

We can now give our new algorithm and prove that both resource and dependence constraints are met.

Algorithm 4 (Algorithm CDR) *Let $G = (V, E, \delta, d)$ be a dependence graph*

1. *Find a retiming q that minimizes the length Φ of the longest path of null weight in G_q (use Algorithm 2 with the improved algorithm for step 3).*
2. *Delete edges of positive weight, or equivalently keep edges $e = (u, v)$ which satisfy $q(v) - q(u) + d(e) = 0$ (i.e. edges with no registers). By this way, we obtain an acyclic graph G_a.*
3. *Perform a list scheduling σ_a on G_a and compute $\lambda = \max_{u \in V}(\sigma_a(u) + \delta(u))$.*
4. *Define the cyclic schedule σ by:*

$$\forall u \in V \quad \forall k \in \mathbb{N} \quad \sigma(u, k) \;=\; \sigma_a(u) + \lambda(k + q(u))$$

Note that the complexity of Algorithm CDR is determined by Step 1 whose complexity is $O(|V||E|\log(|V|))$. Therefore, the complexity of Algorithm CDR is lower than that of Algorithm GS whose complexity is $O(|V||E|\log(|V|\delta_{max}))$. This comes from the fact that Φ_{opt} can be searched among the $|V|^2$ values $\Delta(u, v)$ whereas λ_∞ is searched among all values between 0 and $|V|\delta_{max}$. In particular, we point out that the complexity of Algorithm CDR does not depend on δ_{max}, which makes it more robust.

4.2.1 Correctness of Algorithm CDR

Theorem 2 *The schedule σ obtained with Algorithm CDR meets both dependence and resource constraints.*

Proof Resource constraints are obviously met because of the list scheduling and the definition of λ, which ensures that slices do not overlap. To show that dependence constraints are satisfied for each $e = (u, v)$ of E, we need to verify

$$\sigma(u,k) + \delta(u) \leq \sigma(v, k + d(e))$$

$$\Leftrightarrow \sigma_a(u) + \lambda q(u) + \delta(u) \leq \sigma_a(v) + \lambda q(v) + \lambda d(e)$$

$$\Leftrightarrow \sigma_a(u) - \sigma_a(v) + \delta(u) \leq \lambda(q(v) - q(u) + d(e)) \tag{3}$$

On one hand, suppose that e is not deleted, i.e. $e \in G_a$. It is equivalent to say that the weight of e after the retiming is equal to zero: $q(v) - q(u) + d(e) = 0$. But, since σ_a is a schedule for G_a:

$$\sigma_a(u) + \delta(u) \leq \sigma_a(v)$$

Thus, inequality (3) is satisfied.

On the other hand, if e is deleted, then $q(v) - q(u) + d(e) > 0$, and thus $\lambda(q(v) - q(u) + d(e)) \geq \lambda$. But, by definition of λ we have

$$\sigma_a(u) + \delta(u) - \sigma_a(v) \leq \sigma_a(u) + \delta(u) \leq \lambda$$

Thus, inequality (3) is satisfied. ■

4.2.2 Performances of Algorithm CDR

Now, we show that our algorithm is guaranteed and we give a bound for the initiation interval λ that is better than the bound given for Algorithm GS.

Theorem 3 *Let G be a dependence graph, Φ_{opt} the minimum achievable clock period for G, λ the initiation interval of the schedule generated by Algorithm CDR when p processors are available, and λ_{opt} the best possible initiation interval for this case. Then*

$$\frac{\lambda}{\lambda_{opt}} \leq 1 + \left(\frac{p-1}{p}\right)\left(\frac{\Phi_{opt}}{\lambda_{opt}}\right)$$

Proof Let Φ be the overall time in σ_a when no more than $p - 1$ processors are busy. Since λ is the makespan of the list scheduling σ_a:

$$p\lambda \leq \sum_{u \in V} \delta(u) + (p-1)\Phi$$

(see [3] for more details). As σ_a is generated by a list scheduling algorithm, there exists a dependence path P in G_a such that $\Phi \leq \delta(P)$. By construction, Φ_{opt} is the length of the longest path in G_a, thus $\Phi \leq \delta(P) \leq \Phi_{opt}$. So, we can write:

$$p\lambda \leq \sum_{u \in E} \delta(u) + (p-1)\Phi \leq p\lambda_{opt} + (p-1)\Phi_{opt}$$

$$\text{which leads to } \frac{\lambda}{\lambda_{opt}} \leq 1 + \left(\frac{p-1}{p}\right)\left(\frac{\Phi_{opt}}{\lambda_{opt}}\right)$$

■

Now we show that the bound obtained for Algorithm CDR (Theorem 3) is always better than the bound for Algorithm GS (see Equation 2). This is a consequence of the following lemma:

Lemma 2

$$\lambda_\infty \leq \Phi_{opt} \leq \lambda_\infty + \delta_{max} - 1$$

Proof Let us apply Algorithm CDR with unlimited resources. For that, we define a retiming q such that $\Phi(G_q) = \Phi_{opt}$ and we define the graph G_a by deleting from G all edges e such that $d_q(e) > 0$. Then, we define a schedule for G_a with unlimited resources by $\sigma_a(u) = \max\{\delta(P) : P \text{ path of } G_a \text{ leading to } u\}$. The makespan of σ_a is Φ_{opt} by construction. Finally, we get a schedule for G by defining $\sigma(u,k) = \sigma_a(u) + (q(u)+k)\Phi_{opt}$. Since by definition λ_∞ is the smallest initiation interval for $p = \infty$, we have $\lambda_\infty \leq \Phi_{opt}$.

Now, consider a schedule σ for unlimited resources and initiation interval equal to λ_∞, as defined in Section 3.2.1: $\sigma(u,k) = t(s,u) + \lambda_\infty k$. Let $r(u) = t(s,u) \bmod \lambda_\infty$ and $q(u) = \lfloor \frac{t(s,u)}{\lambda_\infty} \rfloor$. As proved in Section 3.3, q defines a retiming for G, i.e. for all edges $e = (u,v)$, $q(v) - q(u) + d(e) \geq 0$. Furthermore, $q(v) - q(u) + d(e) = 0$ implies $r(u) + \delta(u) \leq r(v)$. Define G_a by deleting from G all edges e such that $d_q(e) > 0$ (as in Algorithm mGS). Let P be any path in G_a, $P = (u_1, \ldots, u_n)$. We have, for all i, $1 \leq i < n$:

$$r(u_i) + \delta(u_i) \leq r(u_{i+1})$$

Summing up these $n-1$ inequalities, we obtain:

$$\sum_{i=1}^{n-1} \delta(u_i) + r(u_1) \leq r(u_n)$$

$$\delta(P) - \delta(u_n) + r(u_1) \leq r(u_n)$$

$$\delta(P) \leq r(u_n) + \delta(u_n) \leq (\lambda_\infty - 1) + \delta_{max}$$

By construction, $\Phi(G_q)$ is the length of the longest path in G_a, thus $\Phi(G_q) \leq \lambda_\infty + \delta_{max} - 1$. Finally, we have $\Phi_{opt} \leq \Phi(G_q)$, hence the result. ■

Theorem 4 *The bound for Algorithm CDR is better than the bound for Algorithm GS.*

Proof This is easily derived from the fact that $\Phi_{opt} \leq \lambda_\infty + \delta_{max} - 1$ as shown by Lemma 2. ■

4.2.3 Link between λ_∞ and Φ_{opt}

As shown in Lemma 2, λ_∞ and Φ_{opt} are very close. However, the retiming that can be derived from the schedule with initiation interval λ_∞ does not permit to define an acyclic graph with longest path Φ_{opt}. In other words, looking for λ_∞ is not the right approach to minimizing the period of the graph. In this section, we investigate more deeply this fact, by recalling another formulation of the retiming problem given by Leiserson and Saxe [9].

Lemma 3 (Lemma 9 in [9]) *Let $G = (V, E, \delta, d)$ be a synchronous circuit, and let c be a positive real number. Then there exists a retiming q of G such that $\Phi(G_q) \leq \lambda$ if and only if there exists an assignment of a real value $s(v)$ and an integer value $q(v)$ to each vertex $v \in V$ such that the following conditions are satisfied:*

$$\begin{cases} -s(v) \leq -\delta(v) \textit{ for every vertex } v \in V \\ s(v) \leq \lambda \textit{ for every vertex } v \in V \\ q(u) - q(v) \leq d(e) \textit{ wherever } e = u \rightarrow v \\ s(u) - s(v) \leq -\delta(v) \textit{ wherever } e = u \rightarrow v \textit{ such that } q(u) - q(v) = d(e) \end{cases} \tag{4}$$

By letting $s(u) = r(u) + \delta(u)$ for every vertex u, inequalities 4 are equivalent to:

$$\begin{cases} r(v) \geq 0 \text{ for every vertex } v \in V \\ r(v) \leq \lambda - \delta(v) \text{ for every vertex } v \in V \\ q(u) - q(v) \leq d(e) \text{ wherever } e = u \rightarrow v \\ r(v) \geq r(u) + \delta(u) \text{ wherever } e = u \rightarrow v \text{ such that } q(u) - q(v) = d(e) \end{cases} \tag{5}$$

This last system permits to better understand all the techniques that we developed previously:

Optimal schedule for unlimited resources

As seen in Lemma 2, the schedule $\sigma(u, k) = t(s, u) + \lambda_\infty k$ satisfies system 5 with $r(u) = t(s, u) \bmod \lambda_\infty$ and $q(u) = \lfloor \frac{t(s,u)}{\lambda_\infty} \rfloor$ **except** the second inequation. We do have $r(v) \leq \lambda_\infty - 1$ but not necessarily $r(v) \leq \lambda_\infty - \delta(v)$ (except if $\delta_{max} = 1$ and in this case, $\lambda_\infty = \Phi_{opt}$ (see Lemma 2 for another proof)).

Schedule obtained by Algorithm CDR for unlimited resources

By construction, with $r = \sigma_a$, q the retiming such that $\Phi(G_q) = \Phi_{opt}$, $\lambda = \Phi_{opt}$, system 5 is satisfied with the smallest value for λ. Therefore, this technique leads to the better cyclic schedule with unlimited resources **for which the slices do not overlap** (because of the second inequation). Therefore, it is not always possible to find λ_∞ this way.

Schedule obtained by Algorithms CDR and GS for p resources

The schedule obtained satisfies system 5 with $r = \sigma_a$, λ the makespan of σ_a. For CDR, q is the retiming that achieves the optimal period, whereas for GS, q is the retiming defined from λ_∞ ($q(u) = \lfloor \frac{t(s,u)}{\lambda_\infty} \rfloor$). For CDR, the fourth inequation is satisfied exactly for all edges $e = (u, v)$ such $q(v) - q(u) = d(e)$. However, for GS, σ is required to satisfy the fourth inequation for more edges than necessary (actually for all edges $e = (u, v)$ such that $r(u)+\delta(u) \leq r(v)$). Note that for both algorithms, there are additional conditions imposed by the resource constraints that do not appear in system 5.

Example We can now apply Algorithm CDR to our key example (assume again $p = 2$ available processors). $\Phi_{opt} = 14$ and the retiming q that achieves this clock period is obtained in two steps by Algorithm 3 (Figures 6 (a), 6 (b) and 6 (c) show the successive retimed graphs). Figure 6 (d) shows the corresponding acyclic graph G_a and finally, Figure 6 (e) shows a possible schedule of operations provided by a list scheduling technique, whose initiation interval is $\lambda = 20$. This is better than what we found with Algorithm mGS (see Figure 5 (b)) and a fortiori with Algorithm GS (see Figure 4 (d)).

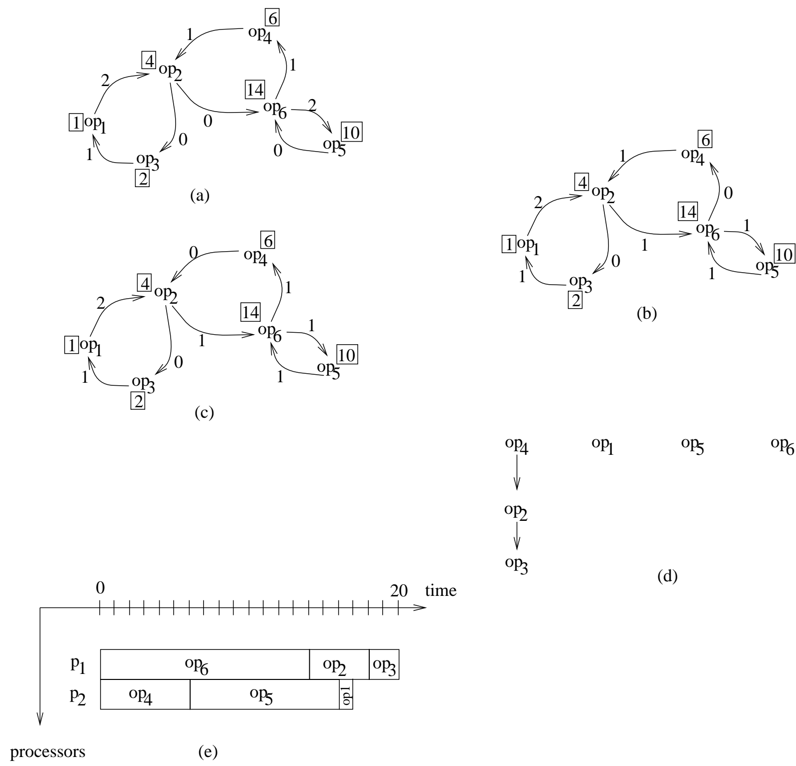


Figure 6: (a): Initial dependence graph G
(b) and (c): First step of Algorithm CDR
(d): Corresponding acyclic graph
(e): A corresponding list scheduling allocation: $\lambda = 20$

5 Minimizing the number of edges of the acyclic graph

Our purpose in this section is to find a retimed graph with the minimum number of null weight edges among all retimed graphs whose longest path has the best possible length Φ_{opt}. Removing edges of non null weight will give an acyclic graph that matches both objectives stated at the end of Section 3.3.

Consider step 1 of Algorithm CDR in which we use the retiming algorithm of Leiserson and Saxe [9] (Algorithm 2 with or without the improved algorithm for step 3). This retiming algorithm does minimize the length Φ of the longest path of null weight into a dependence graph, but is does not necessarily minimize the number of null weight edges. See again our key example, Figure 6 (c), for which $\Phi = 14$. We can apply yet another retiming to obtain the graph of figure 7 (a). The length of the longest path of null weight is still $\Phi = 14$, but the total number of null weight edges is smaller. This implies that the corresponding acyclic graph G_a (see Figure 7 (b)) contains fewer edges than the acyclic graph of Figure 6 (d) and therefore, is likely to induce a smaller initiation interval[3] (that is the case in our example: we find an initiation interval equal to 19 (see Figure 7 (c))[4].

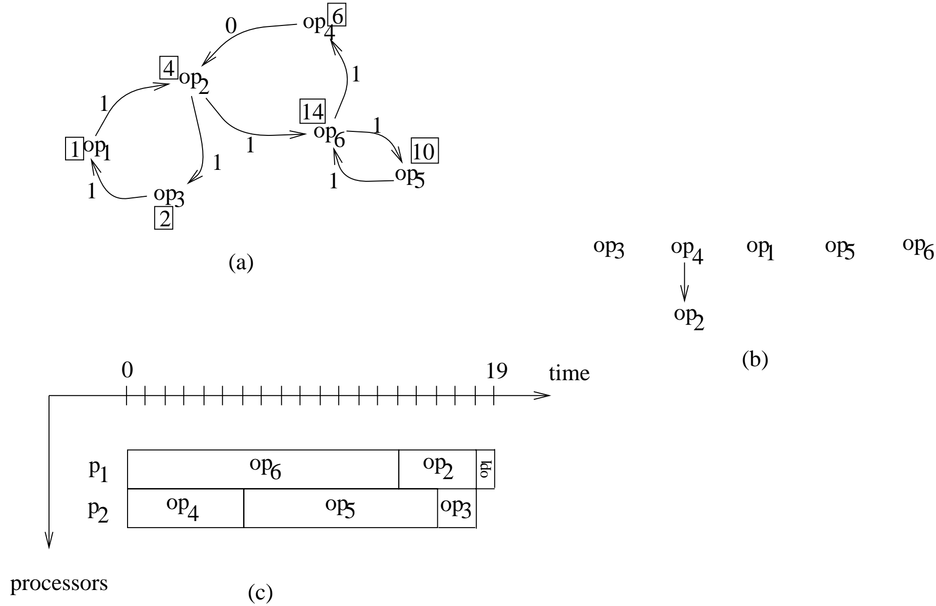


Figure 7: (a): the final retimed graph
$q(op_2) = q(op_5) = 0,\ q(op_1) = q(op_3) = q(op_4) = q(op_6) = 1$
(b): The corresponding acyclic graph
(c): A corresponding list scheduling allocation: $\lambda = 19$

Recall that a retiming q such that $\Phi(G_q) = \Phi_{opt}$ is any integral solution to the following system (see formulation of Theorem 1):

$$\begin{cases} q(v) - q(u) + d(e) \geq 0 \text{ for every edge } u \xrightarrow{e} v \in E \\ q(v) - q(u) + D(u,v) \geq 1 \text{ for all vertices } u, v \in V \text{ such that } \Delta(u,v) > \Phi_{opt} \end{cases} \quad (6)$$

[3] List scheduling a graph which is a subset of another graph will not always produce a smaller execution time. But intuition shows that it will in most practical cases (the fewer constraints, the more freedom).

[4] It turns out that $\lambda = 19$ is the best possible integer initiation interval with $p = 2$ processors: the sum of all operation delays is 37, and $\lceil \frac{37}{2} \rceil = 19$.

Among these retimings, we want to select one particular retiming q for which the number of null weight edges in G_q is minimized. This can be done as follows:

Lemma 4 *Let $G = (V, E, \delta, d)$ be a synchronous circuit. A retiming q such that $\Phi(G_q) = \Phi_{opt}$ and such that the number of null weight edges in G_q is minimized can be found in polynomial time by solving the following integer linear program:*

$$\begin{cases} \min \sum_{e \in E} v(e) \\ 0 \leq v(e) \leq 1 \\ q(v) - q(u) + d(e) + v(e) \geq 1 \text{ for every edge } u \xrightarrow{e} v \in E \\ q(v) - q(u) + D(u,v) \geq 1 \text{ for all vertices } u, v \in V \text{ such that } \Delta(u,v) > \Phi_{opt} \end{cases} \quad (7)$$

Proof Consider an optimal integer solution (q, v) to system 7. q defines a retiming for G with $\Phi(G_q) = \Phi_{opt}$ since system 6 is satisfied: indeed $q(v) - q(u) + d(e) + v(e) \geq 1$ and $v(e) \leq 1$ implies $q(v) - q(u) + d(e) \geq 0$.

Note that each $v(e)$ is constrained by only one equation: $q(v) - q(u) + d(e) + v(e) \geq 1$. There are two cases:

- The edge e in G_q has null weight, i.e. $q(v) - q(u) + d(e) = 0$. Then, $v(e) = 1$ is the only possibility.
- The edge e in G_q has a positive weight, i.e. $q(v) - q(u) + d(e) \geq 1$ (recall that q and d are integers). In this case, the minimal value for v is 0.

Therefore, given a retiming q, $\sum_{e \in E} v(e)$ is minimal when it is equal to the number of null weight edges in G_q.

Now, it remains to show that such an optimal integer solution can be found in polynomial time. For that, we write System 7 in matrix form as $\min\{cx \mid Ax \leq b\}$:

$$\min\{(0,1)\begin{pmatrix} q \\ v \end{pmatrix} \mid \begin{pmatrix} 0 & -I_d \\ 0 & I_d \\ C & I_d \\ C' & 0 \end{pmatrix} \begin{pmatrix} q \\ v \end{pmatrix} \leq \begin{pmatrix} 0 \\ 1 \\ d-1 \\ D-1 \end{pmatrix}\}$$

where C is the transpose of the $|V| \times |E|$-incidence matrix of G, C' is the transpose of the incidence matrix of the graph G' whose edges are the pairs (u, v) such that $\Delta(u, v) > \Phi_{opt}$ and I_d is the $|E| \times |E|$ identity matrix.

Note that if M is a totally unimodular matrix, then so are ${}^tM$ and $\begin{pmatrix} M & \begin{matrix} Id \\ 0 \end{matrix} \end{pmatrix}$. The matrix $\begin{pmatrix} C \\ C' \end{pmatrix}$ is also the transpose of an incidence matrix (the incidence matrix of $G \cup G'$), thus it is totally unimodular (see [10, page 274, example 2]). Then, $\begin{pmatrix} C & I_d \\ C' & 0 \end{pmatrix}$ is totally unimodular, $\begin{pmatrix} 0 & I_d \\ C & I_d \\ C' & 0 \end{pmatrix}$ is totally unimodular, and finally A is also totally unimodular.

Therefore, solving the ILP Problem 7 is not NP-complete: System 7 considered as an LP problem has an integral optimum solution (Corollary 19.1a in [10]) and such an integral solution can be

found in polynomial time (Theorem 16.2 in [10]). ■

Let us summarize how this refinement can be incorporated into our software pipelining heuristic: first, we compute Φ_{opt} the minimum achievable clock period for G, then we solve System 7 and we obtain a retiming q. We define G_a as the acyclic graph whose edges have null weight in G_q: the longest path in G_a is minimized and the number of edges in G_a is minimized. Finally, we schedule G_a as in Algorithm CDR. We call this heuristic the modified CDR (or simply mCDR).

Remark: Solving System 7 can be expensive although polynomial. An optimization that permits to reduce the complexity is to pre-compute the strongly connected components G_i of G and to solve the problem separately for each component G_i. Then, a retiming that minimizes the number of null weight edges in G_q is built by adding suitable constants to each retiming q_i so that all edges that link different components have positive weights. Future work will try to find a pure graph-theoretic approach to the resolution of System 7, so that the practical complexity of our software pipelining heuristic is decreased.

6 Load balancing

We have restricted so far initiation intervals to *integer* values. As mentioned in Section 2.1, searching for *rational* initiation intervals might give better results, but at the price of an increase in complexity: searching for $\lambda = \frac{p}{q}$ can be achieved by unrolling the original loop nest by a factor of q, thereby processing an extended dependence graph with many more vertices and edges.

In this section, we propose a simple heuristic to alleviate potential load imbalance between processors, and for which there is no need to unroll the graph.

Remember the principle of the four previously described heuristics (GS, mGS, CDR and mCDR). First, an acyclic graph G_a is built from G. Then, G_a is scheduled by a list scheduling technique. This defines the schedule σ_a inside each slice of length λ (the initiation interval). Finally, slices are concatenated, a slice being initiated just after the completion of the previous one.

The main weakness of this principle is that slices do not overlap. Since the schedule in each slice has been defined by an As-Soon-As-Possible (ASAP) list scheduling, what usually happens is that many processors are idle during the last time steps of the slice. The idea to remedy this problem is to try to fill these "holes" in the schedule with the tasks of the next slice. For that, instead of scheduling the next slice with the same schedule σ_a, we schedule it with an As-Late-As-Possible (ALAP) so that "holes" may appear in the *first* time steps of the slice. Then, between two successive slices, processors are permuted so that the computational load is (nearly) equally balanced when concatenating both slices.

Let us formulate this more precisely. Define a retiming q for G, for example, the retiming that minimizes the period of G_q and that minimizes the number of edges of G_q with no registers. Delete from G_q all edges that have at least one register (i.e. whose weight is positive): this defines G_a. Then, define for G_a an As-Soon-As-Possible (ASAP) list scheduling σ_{as} and an As-Late-As-Possible (ALAP) σ_{al}. Denote by λ_s and λ_l the respective makespans of σ_{as} and σ_{al}. By construction, $\sigma_s(op_i, k) = \sigma_{as}(op_i) + \lambda_s(q(op_i) + k)$ and $\sigma_l(op_i, k) = \sigma_{al}(op_i) + \lambda_l(q(op_i) + k)$ are both valid cyclic schedules for G. Note that both define the same slices, only the organization inside slices may be different.

To define the final schedule σ, computations of slice $2K$ (even slices) for σ_{as} will be scheduled in slice K of σ and organized with σ_{al}, whereas computations of slice $2K+1$ (odd slices) for σ_l will be scheduled in slice K and organized with σ_l. However, they will be delayed so that dependences between operations in slice $2K$ and in slice $2K+1$ are respected. In other words, we try to determine

a schedule of the following form:

$$\begin{cases} \sigma(op_i, k) = \sigma_{as}(op_i) + \lambda(q(op_i) + k) \text{ if } q(op) + k = 2K \\ \sigma(op_i, k) = D + \sigma_{al}(op_i) + \lambda(q(op_i) + k) \text{ if } q(op) + k = 2K + 1 \end{cases} \tag{8}$$

where $0 \leq D \leq \lambda_s$, $\lambda = D + \lambda_l$ and D is minimized. D is the time step at which the first operations that correspond to odd slices of σ_l are initiated. By this construction, all dependences are respected except possibly some dependences between operations of slice $2K$ and operations of slice $2K + 1$: D has to be chosen sufficiently large so that these remaining dependences are satisfied.

Let $C_{r,s}$ (resp. $C_{r,l}$) be the set of operations that are allocated to processor r in schedule σ_s (resp. in schedule σ_l). Denote by $E_{r,t}$ the set of edges $e = (op_i, op_j)$ such that $op_i \in C_{r,s}$, $op_j \in C_{t,l}$ and $q(op_j) - q(op_i) + d(e) = 1$. $E_{r,t}$ induces a set of constraints on D that can be formulated as follows. Let $\rho(C_r, C_t)$ be the minimum value which meets the following inequality:

$$\rho(C_r, C_t) + \sigma_{al}(op_j) \geq \sigma_{as}(op_i) + \delta(op_i) \text{ if } e = (op_i, op_j) \in E_{r,t}$$

Intuitively $\rho(C_r, C_t)$ is the minimum distance between tasks in slice $2K$ assigned to processor r and tasks in slice $2K + 1$ assigned to processor t, which are linked by a dependence. Dependence constraints are now expressed as:

$$D \geq \rho(C_r, C_t) \text{ for } 1 \leq r, t \leq p \tag{9}$$

Next we have to concatenate two successive slices so that all processors receive the same amount of work. The idea is to make a permutation between processors: clusters for slice $2K + 1$ will not be recomputed but target processors will be interchanged to take into account possible load imbalance from slice $2K$. Let $t_r = \max_{op \in C_{r,s}}(\sigma_{as}(op) + \delta(op))$ and $u_t = \min_{op \in C_{t,l}}(\sigma_{al}(op))$. Let π be a permutation of $\{1, \ldots, p\}$ such that $C_{r,s}$ and $C_{\pi(r),l}$ will be allocated to the same processor. Resource constraints are expressed as follows:

$$D + u_{\pi(r)} \geq t_r \text{ for } 1 \leq r \leq p \tag{10}$$

Clearly, the optimal permutation π that minimizes D is defined as:

$$\begin{cases} u_{\pi(1)} \leq \ldots \leq u_{\pi(p)} \\ t_1 \leq \ldots \leq t_p \end{cases}$$

We obtain the final heuristic:

1. Use Algorithm mCDR to compute the acyclic graph G_a. Compute an ASAP list scheduling σ_{as} and an ALAP list scheduling σ_{al}.

2. Compute the minimum distance D from equations 9 and 10.

3. The final schedule is expressed according to equation 8.

As both σ_{as} and σ_{al} are list schedulings, we can prove the same guarantee bound for this heuristic as for Algorithms CDR and mCDR since $\lambda \leq \lambda_s + \lambda_l$ (and each slice computes two iterations instead of one). The possible gain is that rather than executing slices without overlap, we have tried to interleave them as tightly as possible.

Example Consider our key example again. Figure 7(c) shows a possible allocation of an instance of G_a provided by an ASAP list scheduling. Figure 8 shows an allocation provided by an ALAP list scheduling and Figure 9 the concatenation of these two instances. The initiation interval λ that we obtain is equal to 37 for two instances. i.e. $\lambda = 18.5$, which is better [5] than the initiation interval obtained with Algorithm mCDR (Figure 7(c)).

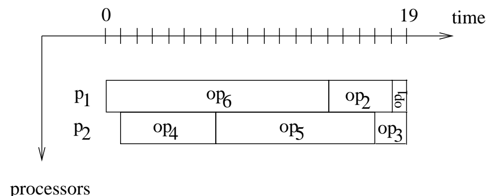


Figure 8: ALAP scheduling

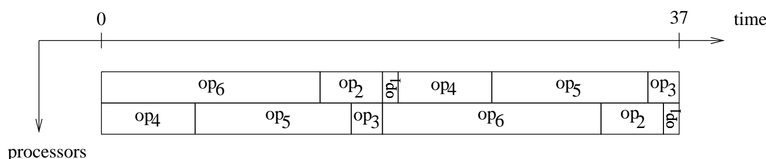


Figure 9: Concatenation of two instances

7 Conclusion

In this paper, we have presented a new heuristic for the *SP* problem. We have built upon results of Gasperoni and Schwiegelshohn, and we have made clear the link between software pipelining and retiming.

In the single resource class case, our new heuristic is guaranteed, with a better bound than that of [6]. Unfortunately, we cannot extend the guarantee to the many resource classes case, because list scheduling itself is not guaranteed in this case.

We point out that our CDR heuristic has a low complexity. As for mCDR, further work will be aimed at deriving an algorithmic implementation that will not require the use of Integer Linear Programming (even though the particular instance of ILP invoked in mCDR is polynomial).

Finally, note that all edge-cutting heuristics lead to cyclic schedulings where slices do not overlap (by construction). Our final load-balancing technique is a first step to overcome this limitation. It would be very interesting to derive methods (more sophisticated than loop unrolling) to synthesize resource-constrained schedulings where slices can overlap.

[5] This cannot be improved further: the two processors are always busy, as $\sum_i \delta(op_i) = 37 = 2\lambda$.

References

[1] A. Aiken and A. Nicolau. Perfect pipelining: A new loop parallelization technique. In *ESOP 88*, volume 300 of *Lectures Notes in Computer Science*, pages 221–235. Springer Verlag, 1988.

[2] C.Hanen and A.Munier. Cyclic scheduling on parallel processors: an overview. Technical Report 822, Laboratoire de Recherche en Informatique, Universite de Paris Sud, Centre d'Orsay, 1993.

[3] E. G. Coffman. *Computer and job-shop scheduling theory.* John Wiley, 1976.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, editors. *Introduction to Algorithms.* The MIT Press, 1990.

[5] Paul Feautrier. Fine-grain scheduling under resource constraints. In *Languages and Compilers or Parallel Computing*, number 892 in Lectures Notes in Computer Science, pages 1–15. Springer Verlag, 1994.

[6] F.Gasperoni and U.Schwiegelshohn. Generating close to optimum loop schedules on parallel processors. *Parallel Processing Letters*, 4(4):391–403, 1994.

[7] R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained, rate-optimal software pipelining. In *COMPAR 94-VAPP VI*, volume 854 of *Lectures Notes in Computer Science.* Springer Verlag, 1994.

[8] J.Wang, C.Einsenbeis, M.Jourdan, and B.Su. Decomposed software pipelining. *International Journal of Parallel Programming*, 22(3):351–373, 1994.

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

[10] Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.

Contents

1 Introduction **1**

2 Known results on the *SP* problem **3**
2.1 Problem formulation 3
2.2 Related work 4

3 Going from cyclic scheduling to acyclic scheduling **5**
3.1 Some properties of cyclic scheduling 5
3.2 The heuristic of Gasperoni and Schwiegelshohn 7
3.2.1 Optimal schedule for unlimited resources 7
3.2.2 Algorithm GS for p resources 8
3.2.3 Performances of Algorithm GS 9
3.3 Cutting edges by retiming 9

4 Minimizing the longest path of the acyclic graph **11**
4.1 Retiming algorithms 11
4.2 A new scheduling algorithm: Algorithm CDR 13
4.2.1 Correctness of Algorithm CDR 14
4.2.2 Performances of Algorithm CDR 14
4.2.3 Link between λ_∞ and Φ_{opt} 15

5 Minimizing the number of edges of the acyclic graph **18**

6 Load balancing **20**

7 Conclusion **22**