UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

3

BASIC DEFINITIONS AND LOOP TRANSFORMATIONS

3.1 INTRODUCTION

This chapter presents some basic concepts used in this work, as well as the loop transformations used by the methodologies proposed in the following chapters.

First of all, we show the way to represent loops and the architecture in which the loop must be executed. A loop is represented by means of a *data dependence graph* (DDG). A DDG is a directed graph in which nodes represent instructions from the loop body, and edges represent data dependences between instruction pairs. Initially, only one iteration of the loop is represented in the DDG. Section 3.2 shows how a loop is represented.

An architecture is a set of *functional units* (FUs). Different types of architectures can be defined depending on the execution model of instructions. In this work, we consider three types of architectures: sequential circuits obtained by *high-level synthesis* tools, *superscalar processors* and *very long instruction word processors*.

Let us consider a set of FUs and a set of instructions which use such a set of FUs. On one hand, the set of FUs is represented as an array . Each element from the array states the number of available FUs of a given type. On the other hand, the way to execute the instructions is represented by a matrix, called the *execution pattern* of each instruction. The number of columns of the matrix is the number of different FUs used by the instruction. The number of rows is the number of

cycles required to execute the instruction. The element (i, j) of the matrix states the number of functional units of type j required to execute the instruction at cycle i. Section 3.3 presents in detail the description of the architecture.

The topology of the DDG of a loop and the architecture in which the loop is executed impose two lower bounds on loop execution.

- The number of FUs in the architecture clearly impose a lower bound. For example, if only one multiplier is available, no more than one multiplication can be executed per cycle.
- Furthermore, dependences in the DDG may also impose a lower bound. Let us assume that a given instruction u requires as data input a result calculated by itself in the previous iteration. With this assumption, no more than one instance of u can be executed at the same time. Therefore, each iteration of the loop will take at least one cycle.

Section 3.4 shows how to analytically compute such lower bounds.

The objective of this work is to propose methodologies to efficiently pipeline a loop under resources and/or timing constraints. Initially, a DDG represents a single iteration of the loop; that is, all the instructions belong to the same iteration. A pipelined schedule of a loop contains instructions belonging to different iterations. In Section 3.5 we propose *dependence retiming*, a loop transformation to obtain a DDG which contains instructions from different iterations. Therefore, if the transformed DDG is scheduled it follows that the loop is pipelined. *Dependence retiming* has been previously used by other authors to reduce the number of buffers (registers) required to execute a loop [LRS83, Lei83, LS83, LS91].

Sometimes is possible to execute more than one iteration of the loop at the same time. For example, a loop without dependences can be executed completely in parallel. All the instructions from all the iterations can be simultaneously executed. We present in Section 3.6 the *loop unrolling* transformation. *Loop unrolling* produces a DDG containing more than one iteration of the loop. *Loop unrolling* combined with *dependence retiming* may achieve a DDG such that an optimal pipelined execution of the loop may be found by a simple scheduling algorithm. *Dependence retiming* and *loop unrolling* are the core of the methodologies proposed in the following chapters.

3.2 REPRESENTATION OF A LOOP

Our scope will be limited to single nested loops whose body is a basic block, i.e. containing neither conditional sentences nor other loop statements.

We will assume that the number of iterations of the loop, K, is known before the execution of the loop (either statically at compile time or or dynamically at run time). Henceforth, the ith execution of the loop body will be denoted by *iteration* i. Without loss of generality, we will assume i ranges between 0 and $K - 1$. If u is an instruction of the loop, u_j will denote the execution of instruction u at iteration j.

The instructions of a loop may be executed in a different order without changing the semantics of the loop, depending on the data dependences existing among the instructions. Data dependences in a loop fall into two categories [PW86] [BC90]:

- *Local data dependences* between instructions from the same iteration (u_i and v_i). These kinds of data dependences are also called *intra-loop dependences* (ILDs).
- *Global data dependences* between instructions from different iterations (u_i and v_j, with $i < j$). These kinds of data dependences are also called *loop-carried dependences* (LCDs).

A loop will be represented by a labelled directed graph called *pipelining graph* (or π-graph). A π-graph is a 4-tuple $\pi = G(V, E, \lambda, \delta)$, where:

- V is the set of vertices. Each vertex $u \in V$ represents an instruction of the loop body. The number of instructions of the loop body will be denoted by $|V|$.
- E is the set of edges. Each edge $e = (u, v) \in E$ represents a data dependence between instructions u and v.
- λ and δ are two mappings, $\lambda : V \mapsto \mathbb{N}$ and $\delta : E \mapsto \mathbb{N}$, representing the *iteration index* (λ) and the number of iterations traversed by the dependence (δ), also called *dependence distance* or *dependence weight*.

A node $u_{\lambda(u)}$ represents the execution of instruction $u_{i+\lambda(u)}$ at the ith iteration of the loop. Therefore, λ states an iteration skew with respect to other instructions. For example, if u_0 and v_1 are instructions from the same π-graph, u_i and v_{i+1} will be executed during the i*th* iteration of the loop. Note that this representation implicitly pipelines the loop.

On the other hand, for a given dependence e, $\delta(e)$ is the number of iterations traversed by the dependence. If e is an ILD then $\delta(e) = 0$. If e represents an LCD between u_i and v_j then $\delta(e) = j - i$.

A dependence (u, v) will be depicted as $u_{\lambda(u)} \xrightarrow{\delta(u,v)} v_{\lambda(v)}$. We will use $u \xrightarrow{\delta(u,v)} v$ to denote a dependence in which $\lambda(u) = \lambda(v)$. Thus, An ILD (u, v) will be depicted as $u \xrightarrow{0} v$ or simply $u \longrightarrow v$.

Definition 3.1 : Initial π-graph

A π-graph $\pi = G(V, E, \lambda, \delta)$ *is an* initial π-graph *if,* $\forall u \in V,\ \lambda(u) = 0$.

Let us show an example of an *initial* π-graph. Figure 3.1 shows a loop and its representation in an initial π-graph. Dependence (A, B) is an LCD with distance one, stating that the result computed by instruction A at iteration i is consumed by instruction B at iteration $i+1$. On the other hand, dependence (A, C) is an ILD, stating that the result computed by instruction A from an iteration is consumed by instruction C in the same iteration. Note that the index of all instructions is the same, indicating that all of them belong to the same iteration of the loop.

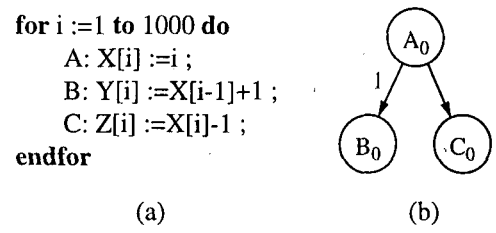

Figure 3.1 Representation of a loop by means of a π-graph

Dependences studied in this work are uniform across iterations. This means that, if a dependence $u_i \xrightarrow{d} v_j$ exists for a given loop, then the dependence $u_{i+x} \xrightarrow{d} v_{j+x}$ also exists (assuming that the instruction indices are in the range $[0, K-1]$). Non-uniform dependences [Ban89] are beyond the scope of this work.

Lemma 3.1 *$u \xrightarrow{d} v$ and $u_i \rightarrow v_{i+d}$ represent the same uniform dependence.*

Proof: An edge $u \xrightarrow{d} v$ represents a data dependence between instructions $u_{j+\lambda(u)}$ and $v_{j+\lambda(v)+\delta(u,v)}$. Since $\lambda(u) = \lambda(v)$ (otherwise the dependence could not be represented as $u \xrightarrow{d} v$), we have that the dependence exists between instructions $u_{j+\lambda(u)}$ and $v_{j+\lambda(u)+\delta(u,v)}$. The lemma is proved by taking $i = j + \lambda(u)$. □

Definition 3.2 : Equivalent π-graphs

Two π-graphs, $\pi = G(V, E, \lambda, \delta)$ and $\pi' = G(V, E, \lambda', \delta')$, are equivalent *if, $\forall (u, v) \in E$ the following equation holds:*

$$\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v) \tag{3.1}$$

Theorem 3.1 *Two equivalent π-graphs represent the same loop.*

Proof: It is sufficient to prove that $u_{\lambda(u)} \xrightarrow{\delta(u,v)} v_{\lambda(v)}$ and $u_{\lambda'(u)} \xrightarrow{\delta'(u,v)} v_{\lambda'(v)}$ represent the same dependence if $\lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v)$.

Let us assume that $\lambda'(u) = \lambda(u) + i$. With this assumption, we have:

$$\begin{aligned} \lambda(v) - \lambda(u) + \delta(u, v) &= \lambda'(v) - \lambda(u) - i + \delta'(u, v) \\ \lambda(v) + \delta(u, v) &= \lambda'(v) - i + \delta'(u, v) \\ \lambda(v) + \delta(u, v) + i &= \lambda'(v) + \delta'(u, v) \end{aligned} \tag{3.2}$$

Since $u_{\lambda(u)} \xrightarrow{\delta(u,v)} v_{\lambda(v)}$ represents a uniform dependence, $u_{\lambda(u)+i} \xrightarrow{\delta(u,v)} v_{\lambda(v)+i}$ also exists $\forall i \in \mathbb{N}$. Such a dependence represents a uniform dependence between $u_{\lambda(u)+i}$ and $v_{\lambda(v)+\delta(u,v)+i}$. By substituting $\lambda(u)+i$ with $\lambda'(u)$ and $\lambda(v)+\delta(u,v)+i$ with $\lambda'(v)+\delta'(u,v)$, we see that the same dependence is represented by $u_{\lambda'(u)} \rightarrow v_{\lambda'(v)+\delta'(u,v)}$ and $u_{\lambda'(u)} \xrightarrow{\delta'(u,v)} v_{\lambda'(v)}$.

Since dependences in both π-graphs represent the same data dependences between nodes, and given that the topology of the loop has not been changed, we conclude that both π-graphs represent the same loop. □

Definition 3.2 states that a dependence can be represented by different labellings, λ and δ, as far as Equation (3.1) is fulfilled. This is the key feature used in our proposal, which transforms a π-graph by changing the labellings λ and δ, and seeks a new labellings which impose less constraints to the scheduling process. Figure 3.2 shows an example of three equivalent π-graphs.

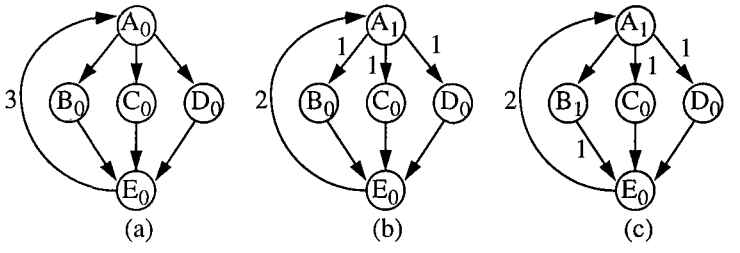

Figure 3.2 Equivalent π-graphs

3.3 REPRESENTATION OF THE ARCHITECTURE

3.3.1 Representation of resources

An architecture is defined by a set of resources and a set of instructions. The *functional units* (FUs) required to execute single arithmetic and logic operations (additions, multiplications, ...) are called resources. The memory address units (required to execute *load* and *store* instructions) and the read/write register ports (when the registers visible from the machine language are organized in banks) are also called resources, as well as, in general, any other unit required to execute an instruction. FUs may be pipelined.

The set of resources of the architecture, $\mathcal{R}$, is represented as an array. The number of elements in the array is the number of different types of resources of the architecture. The element $\mathcal{R}[i]$ indicates the number of resources of type i available in the architecture. Each resource of type i may perform several single operations. For example, an ALU may add, subtract or compare two numbers.

3.3.2 Representation of instructions

Nodes in the π-graph represent instructions from the loop body. Defining how a given instruction uses the resources of the architecture is the responsibility of the designer. An instruction may perform several single operations, may need several resources to be executed and may take several cycles in executing to completion.

Definition 3.3 L_u**: Result latency of instruction** u

The result latency *of instruction* u, L_u, *is the number of cycles required by* u *to produce a result.*

Definition 3.4 IL_u**: Issue latency of instruction** u

The issue latency *of instruction* u, IL_u, *is defined as the minimum number of cycles required between the issuing of two instructions of type* u *to the same FUs.*

The terms *result latency* and *issue latency* have been taken from [Joh90]. For simplicity in the notation, we will henceforth use the terms *latency* and *result latency* in the same sense. Since a node represents an instruction, we will also use the terms *latency* of an instruction and *latency* of a node in the same way.

In order to completely define an instruction, an *execution pattern* must be described by the designer for each instruction. The execution pattern defines how and when the resources of the architecture are used in the execution of the instruction. The execution of any instruction will be statically led by its execution pattern, and therefore an instruction will always be executed in the same way. The execution pattern of u is represented as a matrix of L_u rows and R_u columns, R_u being the *number of different types of resources used by* u. The element (i, j) of the matrix states the number of resources of type i required at cycle j (cycles are referred to the starting of the execution of the instruction) to execute u. The architecture must have at least this number of resources of type i to execute u, otherwise, it would be impossible to execute the loop. Let us illustrate the above-mentioned definitions with an example.

3.3.3 Example of representation of instructions

The objective is to represent instruction *axpy* $R1, R2, R3$ ($R1 = R1 * R2 + R3$), together with the required architecture. Let us assume that the architecture has:

- A microprogrammed control and a bank of registers with two read/write ports. The result latency for each one of the register ports, labelled as Reg_{port}, is one.
- A fully pipelined float point multiplier with latency two, labelled as FU_{mul}, which issues one operation per cycle.
- An FU able to perform float point additions in one cycle, labelled as FU_{add}.

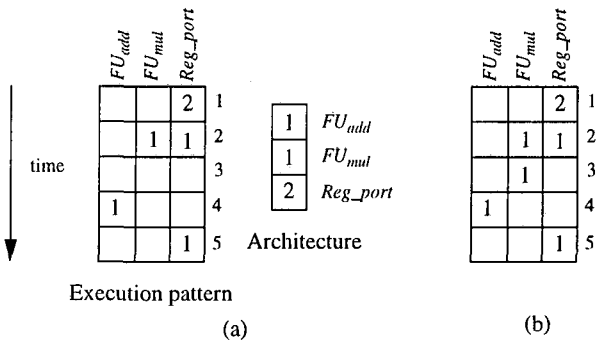

Figure 3.3 Description of an architecture
(a) Execution pattern of *axpy* and architecture to execute it
(b) Execution pattern of *axpy* when the multiplier is not pipelined

Figure 3.3(a) shows the execution pattern of instruction *axpy* and the minimum architecture required to execute it. For the sake of clarity, a "0" in the matrix has been represented as an empty cell. Instruction *axpy* executes as follows:

1. The values of $R1$ and $R2$ are read from the bank of registers at first cycle (two register ports are required).
2. The multiplication starts at second cycle. The value of $R3$ is also loaded from the bank of registers at second cycle (only one register port is required).
3. The sum starts at fourth cycle, since the multiplier requires two cycles to compute the result of $R1 * R2$.
4. $R1$ is written at the fifth cycle (one register port is only required).

Instruction *axpy* has a result latency $L_{axpy} = 5$ and uses 3 different types of resources: *register ports, FP-multipliers and FP-adders* . The load of $R3$ could be delayed until the third cycle without increasing the latency of *axpy*, given that the value of $R3$ is not used by the FP-adder until the fourth cycle. Choosing the cycle in which $R3$ will be loaded is a designer's decision. Also note that, in spite of the FP-multiplier not being used at the third cycle, the FP-adder must wait until the fourth cycle before starting the sum. If the FP-multiplier were not pipelined, cycle 3 would have an "1", as Figure 3.3(b) shows.

When any instruction does not use some of the resources in a given cycle, they may be used by other instructions. In general, the execution of instructions can be overlapped. This overlapping improves the resource utilization done by the loop execution, and therefore the execution throughput. This is one of the goals of this work.

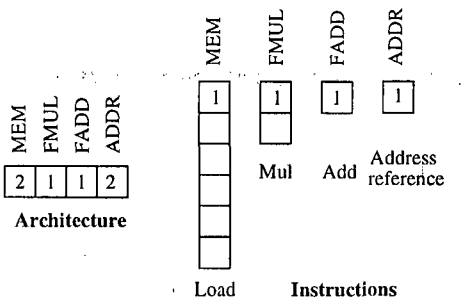

Figure 3.4 Subset of the architecture of the Cydra 5 Computer

3.3.4 Example of architecture

The representation described in Sections 3.3.1 and 3.3.2 enable us to model architectures oriented to the high-level synthesis of circuits (henceforth, high-level synthesis systems), superscalar processors and VLIW processors. We will use the *inner product* example from Figures 1.4 and 1.5 to illustrate how the instructions and resources are represented. We will use the description of a subset of the Cydra 5 *Directed Dataflow*^{TM} architecture [Rau88, RYYT89, SM88] to show an architecture able to execute the loop. For the sake of future comparisons, we will make the same assumptions as in [DHB89]:

- A memory load has a six-cycle latency. There are two memory reference units (MEM).
- A floating point addition has a one-cycle latency. There is one FU (FADD) to execute it.
- A floating point multiplication has a two-cycle latency. There is one FU (FMUL) to execute it.
- An address addition has a one-cycle latency. There are two address adders (ADDR).
- The memory reference FUs and the floating point multiplier are fully pipelined. Therefore, each one of the six FUs may issue an operation per cycle.

Figure 3.4 presents the description of the subset of the architecture of Cydra 5 able to execute the inner product of Figure 1.5 by using the previous assumptions. Figure 3.5(a) shows the π-graph representing the compiled *inner product* from Figure 1.5. Figure 3.5(b) shows a possible schedule of an iteration of the loop by using the architecture of Cydra 5 described in Figure 3.4. The cycles at which an instruction uses a resource are marked as "X". Bold rectangles denote the cycles used by each instruction in its execution.

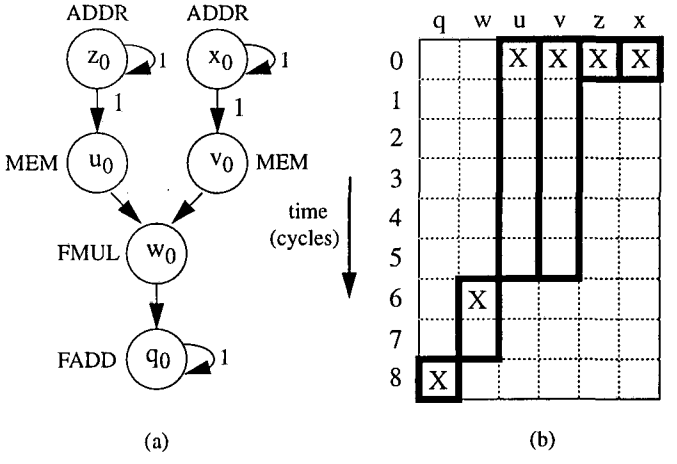

Figure 3.5 Execution of *compiled* inner product
(a) π-graph representing the *inner product*
(b) A possible schedule in the Cydra 5 architecture

3.4 BOUNDS ON LOOP EXECUTION

Definition 3.5 *II*: **Initiation interval of a loop schedule**

The initiation interval *of a loop schedule, II, is the average number of cycles elapsed between the initiation of two consecutive iterations of the loop.*

The number of resources of the architecture limits the minimum initiation interval (*MII*) achievable by any schedule of a loop. This lower bound is denoted as resource-constrained MII ($ResMII(\pi)$). In the same way, the cycles produced by the dependences in the loop also limit the *MII*. This bound is denoted as recurrence-constrained MII ($ResMII(\pi)$) [RST92].

3.4.1 Resource-constrained MII

Definition 3.6 $ResMII_{R_i}$: **Lower bound on the II due to resource** R_i

The minimum number of cycles required to execute an iteration of the loop by taking into account the resource i, $ResMII_i$*, can be computed by dividing the number of cycles used by the resource i in the execution of an iteration by the number of available resources of such a type,* $\mathcal{R}[i]$*. Let* EP_u *be the matrix representing the execution pattern of u.* $ResMII_i$ *is:*

$$ResMII_i = \frac{\sum_{u \in V} \sum_{j=1}^{L_u} EP_u[j,i]}{\mathcal{R}[i]} \tag{3.3}$$

The resource i which determines the maximum $ResMII_i$ is the one that determines $ResMII(\pi)$.

Definition 3.7 $ResMII(\pi)$: **Resource-constrained MII**

The minimum initiation interval (*MII*) of the loop π executed with the set of resources $\mathcal{R}$, $ResMII(\pi)$, *is defined as:*

$$ResMII(\pi) = \max_{i \in \mathcal{R}} ResMII_i \tag{3.4}$$

Theorem 3.2 $II \geq ResMII(\pi)$ *for any schedule of* π.

Proof:

The proof of the lemma is trivial by using the two previous definitions. If a schedule exists with an II so that $II < ResMII(\pi)$, then a resource i exists for which $II < ResMII_i$. But this contradicts the definition of $ResMII_i$. □

The previous definitions assume that an instruction is always executed in the same type of FU, and therefore it always executes in the same number of cycles. We will use this approach in this work.

However, a particular operation may be executed on multiple FUs, in which case it is said to have *multiple alternatives*, with a different execution pattern corresponding to each one. Furthermore, these FUs might not be equivalent. For instance, a floating-point multiply might be executed on two FUs, of which only one is capable of executing divide operations. The exact *ResMII* can be computed by performing a bin-packing of the execution patterns for all the instructions. Bin-packing is a problem which is of exponential complexity. Complex execution patterns (containing several resources) and *multiple alternatives* make it even more complex, and it is impractical, in general, to compute the *ResMII* exactly. Instead, an approximate value must be computed. Accordingly, the *ResMII* may be computed by first sorting the instructions in the loop body in increasing order of the number of alternatives. As each instruction is taken in order from the list, the number of times it uses each resource is added to the usage count for that resource. For each instruction, the alternative which yields the lowest partial *ResMII* is selected, i.e. the usage count of the most heavily used resource at that point. When all instructions have been considered, the usage count for the most heavily used resource constitutes the $ResMII(\pi)$ [Rau94].

3.4.2 Recurrence-constrained MII

Definition 3.8 : Recurrence

A set of dependences $R = \{(u_1, u_2), (u_2, u_3), \ldots, (u_n, u_1)\}$ *is called a* cycle *or a* recurrence.

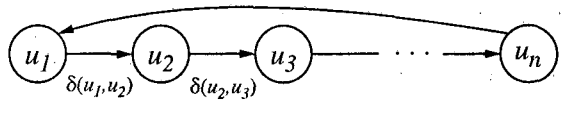

Figure 3.6 Recurrence in a loop

Definition 3.9 δ_R: Weight of a recurrence R

The weight *of a recurrence* R, δ_R, *is defined as:*

$$\delta_R = \sum_{(u,v)\in R} \delta(u,v)$$

Definition 3.10 $\mathcal{L}_R$: Latency of a recurrence R

The latency *of a recurrence* R, $\mathcal{L}_R$, *is defined as:*

$$\mathcal{L}_R = \sum_{(u,v)\in R} L_u$$

Theorem 3.3 *The minimum number of cycles elapsed between the initiation of two consecutive iterations imposed by a recurrence R, $RecMII_R$, is [RG81]:*

$$RecMII_R = \frac{\mathcal{L}_R}{\delta_R} \tag{3.5}$$

Proof:

Let us consider a recurrence R, as shown in Figure 3.6. For any instruction u so that $\exists(u, v) \in R$, u_i must be scheduled at least $\mathcal{L}_R$ cycles before $u_{i+\delta_R}$. Therefore, the minimum number of cycles elapsed between the execution of u_i and u_{i+1} is, on average[1], $\frac{\mathcal{L}_R}{\delta_R}$. □

Definition 3.11 $RecMII(\pi)$**: Recurrence-constrained MII**

In a loop without recurrences, $RecMII(\pi) = 0$. If a loop has more than one recurrence, the one with the highest $RecMII_R$ will determine $RecMII(\pi)$. Therefore, the minimum initiation interval due to the recurrences is:

$$RecMII(\pi) = \begin{cases} 0 & \text{if the loop has no recurrences} \\ \max\limits_{R \subseteq E} RecMII_R & \text{if the loop has recurrences} \end{cases} \tag{3.6}$$

Complexity of computing $RecMII(\pi)$

In the worst case, the number of recurrences of a π-graph grows exponentially with $|E|$. However, finding $RecMII(\pi)$ can be done in polynomial time by using *Karp's algorithm* [Kar78] to find the *minimum mean-weight cycle* in a graph[2]. The algorithm is as follows:

1. Decompose π into a set of strongly connected components $\{\pi_i\}$. This can be done in linear time $O(V + E)$[Meh84].
2. For each $\pi_i = G(V_i, E_i, \lambda_i, \delta_i)$, calculate $W(\pi_i)$ as follows. Let $n = |V_i|$, and let us take a vertex $s \in V_i$. Since π_i is a strongly connected component of π, all the other vertices are reachable from s. Let $d_k(s, v)$ be the weight of the shortest path from s to v consisting exactly of k edges. Let $l_k(s, v)$ be the sum of the latencies of the instructions represented by nodes included in this path[3]. The weight of the minimum mean-weight cycle of π_i can be computed as:

$$W(\pi_i) = \min_{v \in V} \max_{0 \le k \le n-1} \frac{d_n(s, v) - d_k(s, v)}{l_n(s, v) - l_k(s, v)}$$

   This can be done in $O(VE)$ time[Kar78].

[1] Note that $\delta_R > 0$, otherwise a dependence $u_i \rightarrow u_i$ would exist, and finding a schedule would be impossible.

[2] *Karp's algorithm* has been modified here. Since the original algorithm assumes each node has a latency equal to one, the mean weight of a recurrence R is defined as $\frac{\delta_R}{|R|}$. Here, the mean weight of a recurrence R is defined as $\frac{\delta_R}{\mathcal{L}_R}$.

[3] The latency of the node v is not added in this sum.

3. *RecMII*(π) is:

$$RecMII(\pi) = \max_{\pi_i} \frac{1}{W(\pi_i)}$$

The problem of finding the *minimum mean-weight cycle* is a variation of the *shortest path* problem. Therefore, the problem of finding the *minimum mean-weight cycle* can be solved in polynomial time by using (with slight modifications) any algorithm which solves the *shortest paths* problem (algorithms of *Dijkstra's* [Dij59], *Bellman-Ford's* [Law76] or *Gabow-Tarjan's* [GT89], among others).

3.4.3 Minimum initiation interval and throughput

Definition 3.12 *MII*(π) **: Minimum initiation interval of** π

An absolute lower bound on the initiation interval achievable in the execution of a loop, the Minimum Initiation Interval *(MII), is the maximum between ResMII*(π) *and RecMII*(π).

$$MII(\pi) = \max(RecMII(\pi), ResMII(\pi)) \tag{3.7}$$

Theorem 3.4 *If* π *and* π' *are equivalent* π*-graphs, then* $MII(\pi) = MII(\pi')$.

Proof:

It is sufficient to prove that, for any recurrence R, $\delta_R = \delta'_R$.

By using Equation (3.1), we have:

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

hence

$$\delta_R = \delta'_R + \sum_{(u,v)\in R} \lambda(u) - \sum_{(u,v)\in R} \lambda(v) + \sum_{(u,v)\in R} \lambda'(v) - \sum_{(u,v)\in R} \lambda'(u)$$

Since R is a recurrence

$$\sum_{(u,v)\in R} \lambda(u) - \sum_{(u,v)\in R} \lambda(v) = \sum_{(u,v)\in R} \lambda'(v) - \sum_{(u,v)\in R} \lambda'(u) = 0$$

and therefore $\delta_R = \delta'_R$. □

Therefore, we conclude that the *minimum initiation interval* (*MII*) is a property of the loop, and not of its representation. For this reason, the terms *MII* of a loop and *MII* of a π-graph will be used indiscriminately.

When each iteration of π is executed in $MII(\pi)$ cycles (on average), the average number of instructions executed per cycle is maximum. With this assumption, the loop is executed with *maximum throughput*.

Definition 3.13 *MaxTh*(π) **: Maximum execution throughput of** π

An upper bound for the execution throughput *of* π, *MaxTh*(π), *is the average number of iterations executed per cycle when the loop executes with the minimum initiation interval.*

$$MaxTh(\pi) = \frac{1}{MII(\pi)} \tag{3.8}$$

3.5 DEPENDENCE RETIMING

In high-level synthesis, a transformation called *retiming* [LS91] is used to move buffers across nodes in a graph. It is oriented to reduce the number of buffers required to execute a loop, and it has been widely used in real-time signal processing applications (digital signal processing, filters, etc.). *Retiming* has also been widely used in systolic arrays [LRS83, Lei83, LS83]. Researchers in compilers for parallel architectures also use similar transformations, generically called *software pipelining* [Lam88].

3.5.1 Dependence retiming transformation

In order to transform a given π-graph $\pi = G(V, E, \lambda, \delta)$ into another equivalent one $\pi' = G(V, E, \lambda', \delta')$, we define the transformation *dependence retiming*, which has similar effects to the above cited *retiming*. *Dependence retiming* transforms a dependence (u, v) with $\delta(u, v) = i$ into another dependence with $\delta'(u, v) = i + 1$. This is done by performing the following steps:

- $\lambda'(u) = \lambda(u) + 1$
- $\forall (u, w) \in E,\ \delta'(u, w) = \delta(u, w) + 1$
- $\forall (w, u) \in E,\ \delta'(w, u) = \delta(w, u) - 1$

Dependence retiming must be applied to edges so that no dependence with negative distance is produced in the transformed π-graph. It produces an equivalent π-graph, since Equation (3.1) is fulfilled for each edge in the π-graph. We will use *dependence retiming* to transform a π-graph into an equivalent one executable in fewer cycles. We now give an example.

Example

Figure 3.7(a) shows an *initial* π*-graph* representing a loop and a possible way of executing the loop. For the sake of simplicity, we will assume here that all instructions can be executed in a single cycle. With this assumption, LCD (A, A) is always honored by the sequential execution of the iterations of the loop. Therefore, the scheduling of each iteration must only take into account ILDs (A, B) and (A, C).

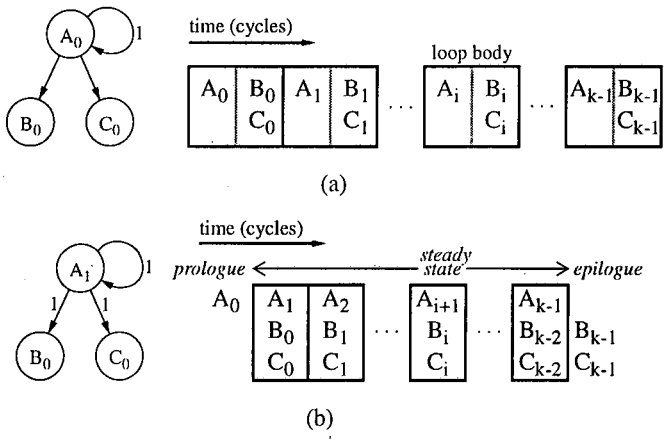

Figure 3.7 Equivalent π-graphs and their schedules
(a) Initial π-graph of a loop and its schedule
(b) Equivalent π-graph and its schedule

Figure 3.7(b) shows an equivalent π-graph after retiming dependence (A, B) (note that the distance of (A, C) has also been increased). ILDs have been transformed into LCDs, and no dependence influences the scheduling of each iteration. An iteration of the retimed π-graph contains instructions belonging to different iterations in the initial loop. Therefore, *dependence retiming* allows the loop to be pipelined. The execution of the loop now consists of three parts: *prologue*, *steady state*, and *epilogue* [Lam88]. The *prologue* and *epilogue* must be included to initiate and conclude the loop execution properly.

Since no ILDs exist in the π-graph from Figure 3.7(b), instructions A_1, B_0 and C_0 (and, in general, instructions A_{i+1}, B_i and C_i) can be executed at the same time if sufficient resources are available. Therefore, the loop represented by the retimed π-graph may be executed in half the time[4] required by the loop represented by the initial π-graph.

3.6 LOOP UNROLLING

3.6.1 Loop unrolling transformation

Loop unrolling [DH79] is a well-known loop transformation. *Unrolling* m times an initial π-graph π generates another *initial π-graph* π^m in which each instruction and each dependence in π appears m times. *Unrolling* the initial π-graph m times produces an initial π-graph representing the loop unrolled m times.

For each vertex u representing instructions u_i $(i = 0..K-1)$ in π, π^m has m vertices (labelled as $u^0, u^1, \ldots, u^{m-1}$), so that each u^j represents all the instructions u_i such that $i \bmod m = j$.

[4] We ignore the length of the *prologue* and the *epilogue* since, in general, their execution time scarcely influences in the total execution time of the loop.

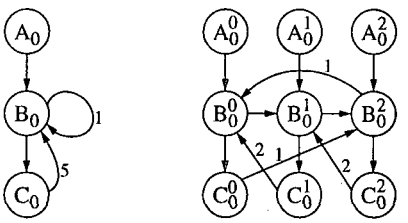

Figure 3.8 Unrolling a π-graph 3 times

```
function unroll (π, m);
    V^m := ∅; E^m := ∅; {set of vertices (V^m) and edges (E^m)}
    for each u ∈ V do
        for i := 0 to m − 1 do
            V^m := V^m ∪ {u^i};
            λ^m(u^i) := 0;
    for each (u, v) ∈ E do
        for i := 0 to m − 1 do
            E^m := E^m ∪ {(u^i, v^((i+δ(u,v)) mod m))};
            δ^m(u^i, v^((i+δ(u,v)) mod m)) := ⌊(i+δ(u,v))/m⌋;
    return π^m;
```

Algorithm 3.1 Algorithm to unroll a π-graph m times

Likewise, for each dependence $u \xrightarrow{d} v$ in π, m dependences $u^j \xrightarrow{\lfloor (j+d)/m \rfloor} v^{(j+d) \bmod m}$ $(j = 0..K-1)$ are created in π^m.

Figure 3.8 shows an example of π-graph unrolling. The labelled vertex u_i^j of π^m represents instruction u^j at iteration i. Consequently, the same instruction is represented by instruction $u_{i \cdot m+j}$ in π.

Algorithm 3.1 shows how to create a π-graph representing a loop unrolled m times. The proposed algorithm executes in $O(V + E)$ time.

3.6.2 Π-graphs with integer MII

In general, a π-graph representing multiple instances of the loop body is required to obtain a schedule in *MII* cycles [JC92b]. Such a π-graph will be called a *multiple-instanced π-graph*.

Let us show an example. The π-graph in Figure 3.9(a) has $MII(\pi) = \frac{2}{3}$ (we assume that the latency of all instructions is one). This means that, at best, three iterations will be issued every two cycles. Since the initiation interval represents a number of cycles, it must be an integer. Thus,

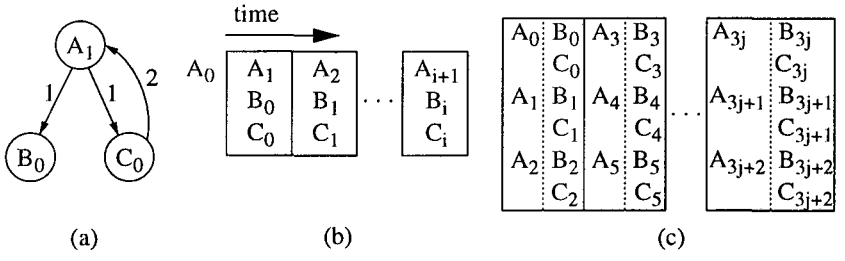

Figure 3.9 Scheduling a π-graph and a multiple-instanced π-graph
(a) π-graph π
(b) Schedule of π
(c) Schedule of π^3

the shortest II for any schedule of π is $II = 1$ (see Figure 3.9(b)). A multiple-instanced π-graph representing three instances of the loop body (π^3) may produce a schedule with the minimum initiation interval, as shown in Figure 3.9(c).

For the sake of simplicity, and without loss of generality, we will use the terms π-graph and *multiple-instanced π-graph* to mean the same.

Theorem 3.5 $MII(\pi^m) = m \cdot MII(\pi)$

Proof:

Given that π and π^m represent the same loop, the execution time with minimum initiation interval must be the same. The loop represented by π must be executed K times, while π^m must be executed only $\frac{K}{m}$ times. Therefore:

$$\frac{K}{m} \cdot MII(\pi^m) = K \cdot MII(\pi) \Rightarrow MII(\pi^m) = m \cdot MII(\pi)$$

□

MII is always a rational number. A π-graph with integer minimum initiation interval can be obtained as follows:

1. Calculate the MII of the initial π-graph. Let $MII(\pi) = \frac{a}{b}$.
2. Calculate $m = \frac{b}{\gcd(a,b)}$
3. Generate $\pi^m = unroll\ (\pi, m)$. The MII of π^m is $MII(\pi^m) = \frac{a}{\gcd(a,b)}$

3.7 SUMMARY AND CONCLUSIONS

This chapter shows the way to represent a loop by using a data dependence labeled graph, called π-graph. A loop can be represented by different but equivalent π-graphs. Equations are provided to check when two π-graphs are equivalent. A transformation, *dependence retiming*, is proposed to transform a π-graph into an equivalent one in linear time.

This chapter also describes the way to represent an architecture. Lower bounds on the initiation interval achievable in the execution of a loop, and upper bounds on the instruction parallelism and the execution throughput are derived. Algorithms to compute such bounds in polynomial time are also presented.

This chapter establishes lower bounds in the execution time of a loop by calculating the lower bound in the execution time of an iteration. This is key factor in this work, since we will show in Chapter 6 the way to find a schedule which executes the loop with the minimum execution time.

Finally, this chapter presents *dependence retiming* and *loop unrolling*, the two loop transformations which are the core of this work.