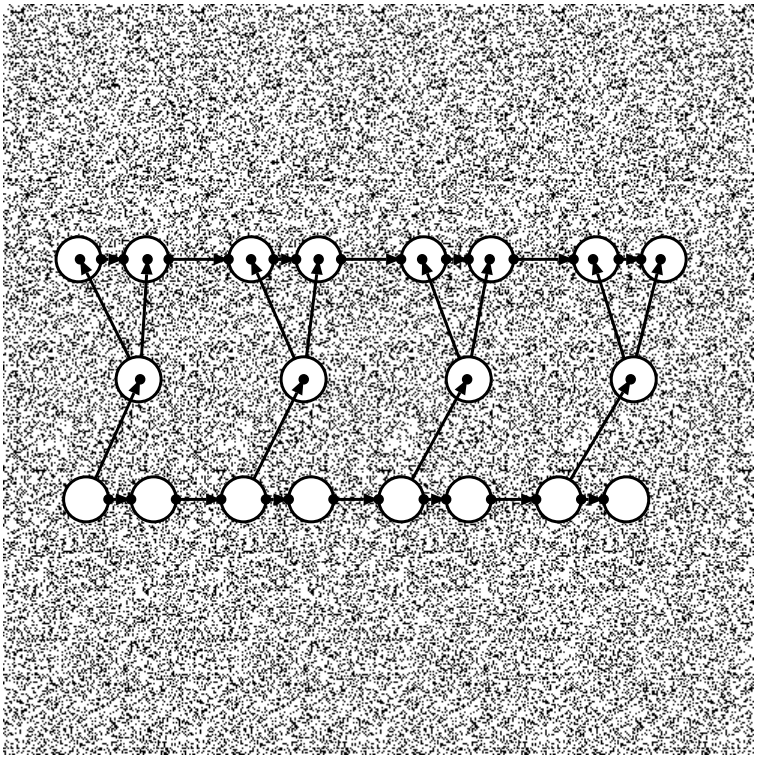
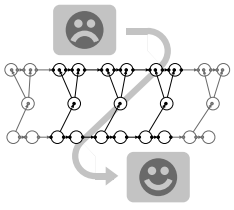
Multidimensional periodic scheduling

Multidimensional Periodic Scheduling

Wim F.J. Verhaegh



Multidimensional Periodic Scheduling



On the cover: a so-called magic eye. By staring at the cover, and focusing some 30 cm behind it, a three-dimensional picture appears. Magic eyes are based on repetitive patterns, with different widths of repetition to create different levels. The hidden picture symbolizes video signal processing.

Multidimensional Periodic Scheduling

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op woensdag 20 december 1995 om 16.00 uur

door

Wilhelmus Franciscus Johannes Verhaegh

geboren te Weert

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. E.H.L. Aarts
en
prof.dr. J.K. Lenstra

Copromotor: dr.ir. J.H.M. Korst

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Verhaegh, Wilhelmus Franciscus Johannes

Multidimensional Periodic Scheduling /
Wilhelmus Franciscus Johannes Verhaegh. -
Eindhoven: Eindhoven University of Technology
Thesis Eindhoven. - With index, ref. - With summary in Dutch
ISBN 90-74445-21-7
Subject headings: scheduling, combinatorial optimization, IC design, high-level synthesis, video signal processing

The work described in this thesis has been carried out at the Philips Research Laboratories in Eindhoven, the Netherlands, as part of the Philips Research programme.

Preface

When I received my mathematical engineering degree, five years ago, the question arose to me whether to get a job or to pursue a Ph.D. degree. This thesis is the result of doing the former without giving up the latter. I have always considered myself lucky to be given the opportunity to do so. To me, working in an industrial research laboratory has always been a very stimulating combination of practice and theory.

Although only my name appears on the cover of this thesis, I am indebted to many people who made this work possible. Therefore, I would like to express my gratitude towards them.

First of all, I would like to thank Emile Aarts for his support and encouragement in the process of doing research and writing this thesis. His tactful way of encouraging, without pushing, helped me to get on very well in my research. Furthermore, our combined pursuit of perfection made our collaboration very pleasant.

Secondly, I would like to thank Jan Karel Lenstra. His extensive knowledge of combinatorial optimization has been of great help filling the gaps in mine.

Next, I would like to express my gratitude to Jan Korst. By carefully reading drafts of this thesis, he pinpointed all inconsistencies and all unmotivated steps, which resulted in many improvements.

I am also grateful to the people that are or have been involved in the Phideo project, in particular to Paul Lippens, who shared an office with me for the past five years, Jef van Meerbergen, and Albert van der Werf. The vivid and open-minded discussions in the Phideo project have always been very stimulating to me. I also owe thanks to Paul van Gorp, for his assistance in tackling the period assignment problem, and Pascal Coumans, for his current contributions to the scheduling approach.

Next, I would like to thank the members of the ‘Algoritmen Club’ and many other colleagues not already mentioned, for the pleasant atmosphere and interesting discussions. I am also grateful to the management of the Philips Research Laboratories for giving me the opportunity to carry out the research described in this thesis. I would like to thank the European Commission for supporting this work in the ESPRIT 2260 project.

Furthermore, I would like to thank my family and friends for their continuing

support and interest in my work.

Finally, I would like to thank my companion in life, Leonore van Dijk. She constantly reminds me of the fact that there are many more pleasant things in life besides doing research.

Eindhoven, October 1995 Wim F.J. Verhaegh

Contents

1

Introduction

This thesis is concerned with the multidimensional periodic scheduling problem. Multidimensional periodic scheduling considers the scheduling of operations that have to be executed repeatedly, with several dimensions of repetition. For instance, consider a bus that repeatedly has to run a certain route. Running the route takes less than an hour, and the bus has to run it ten times a day, with a period of one hour, five times a week, with a period of one day, and each week, with a period of one week. The repeated process of running the route is called a periodic operation, and running a single route is called an execution of the operation. The duration of a run is called the occupation time and the time between the start of two consecutive repetitions in each dimension is called a period. In the example, we have three dimensions of repetition, giving rise to three periods, namely an hour, a day, and a week.

Our interest in the area of multidimensional periodic scheduling originates from the field of digital video signal processing. More precisely, it originates from the problem of designing cost-effective systems that implement video signal processing algorithms.

1.1 Video Signal Processing

Signal processing is concerned with the transformation of input signals into output signals, and plays an important role in areas such as radio, TV, radar, medical dia-

gnosis, and telecommunications. Signal processing is used to obtain a higher sound and picture quality, as well as to realize new features, such as picture-in-picture. Traditionally, signal processing was performed by analog systems, but the introduction of digital systems has strongly stimulated the development of digital signal processing. For instance, digital audio signal processing is applied in a compact disc player. The continuing advances in integrated circuit (IC) technology have allowed a next step in digital signal processing, namely digital video signal processing, opening the way to further improvements in picture quality such as 100 Hz television, and opening the way to many new applications such as video conferencing, multimedia systems, and high-definition television.

A digital video signal can be obtained by sampling a moving picture, which can be regarded to be continuous in time and space. In order to be able to perform digital signal processing, a picture is digitized in time, space, and amplitude. The temporal and spatial digitization is called *sampling*, while amplitude digitization is called *quantization*. Sampling along the temporal axis results in a sequence of still pictures, called *frames*. Each frame consists of a number of *lines*, and each line consists of a number of *pixels*. The exact number of frames per second, lines per frame, and pixels per line depends on the video standard that is used. Consider, for example, a non-interlaced version of the PAL (Phase Alternating Line) standard. There we have 25 frames per second, 625 lines per frame, and 864 pixels per line, resulting in a total of $25 \cdot 625 \cdot 864 = 13.5$ million pixels per second. However, out of the 625 lines per frame only 576 are visible, and out of the 864 pixels per line only 720 are visible.

Each frame is *scanned* in order to map the two-dimensional spatial information into one-dimensional temporal information. The scanning process in PAL is shown in Figure 1.1. In each line, the pixels are scanned from left to right. The lines in a

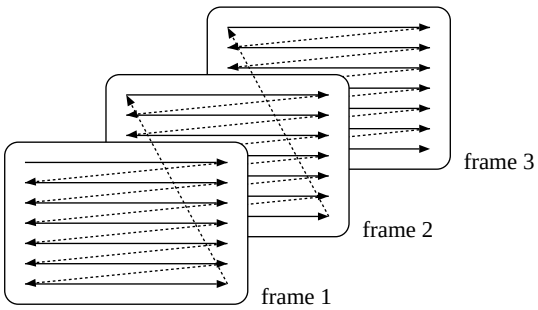


Figure 1.1. The scanning process in the PAL video standard.

frame are scanned from top to bottom. In PAL, the time between two successive frames is $1/25$ of a second, i.e., 40 ms, the time between two successive lines is $1/625 \cdot 40$ ms $= 64$ μs, and the time between two successive pixels is $1/864 \cdot 64$ μs

$\approx$ 74 ns. The scanning process results in a three-dimensional periodic arrival of visible pixels, which is shown in Figure 1.2. The 720 visible pixels in a line arrive

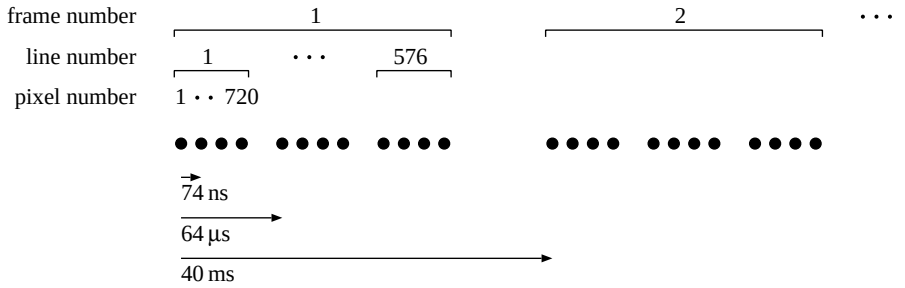


Figure 1.2. The three-dimensional periodic arrival of visible pixels in the PAL video standard.

with a period of 74 ns, which is repeated 576 times with a period of 64 µs, which is again repeated infinitely often with a period of 40 ms. The time between the last pixel of one line and the first pixel of the next line is called the *line blanking*. In this time, the scanner can be moved to the beginning of the next line. Similarly, there is a *frame blanking* between the last line of a frame and the first line of the next frame.

After quantization, the samples can be considered as integers. In this way, a digital video signal can be seen as a stream of integers. Digital video signal processing transforms a given input stream into an output stream. This transformation is described by a so-called *video algorithm*, which specifies the way in which output samples have to be computed from input samples. Because of the repetitive nature of video signals, many video algorithms consist of repetitive executions of operations, resulting in a repetitive production and consumption of data.

The time between the arrival of two successive pixels in video signal processing is very small compared to the time needed to process one pixel. Furthermore, processing of a pixel may require neighboring pixels in other lines, depending on the video algorithm. As a consequence, the processing of successive pixels overlaps in time, and it usually requires parallel executions of the operations.

A second consequence of the high sampling frequency is that only a few operations can share hardware units. Furthermore, these hardware units, called *processing units*, usually perform one specific function, which also limits sharing.

Finally, a characteristic of video signal processing is that memories for storing intermediate data play a significant role in the total cost of an IC. For instance, if processing a pixel involves the neighboring pixels in the previous line, then one needs to store the previous line. The part of the IC area occupied by memories therefore becomes significant, as can be observed from existing ICs for video signal processing. As a result, not only processing units but also memories must be taken

into account during the design of video signal processing systems, and a trade-off between the two types of resources must be made.

Because of the increasing functionality of digital signal processing systems, designing ICs to implement them becomes a highly complex task, resulting in longer design times. However, the design time has to be reduced in order to reduce the time-to-market and the design cost of an IC. Therefore, much effort is spent on developing methodologies for IC design that are supported by computer aided design tools. As an example, we discuss the design methodology Phideo, which has been developed at the Philips Research Laboratories, and which serves as an industrial proving ground for many of the mathematical ideas presented in this thesis.

1.2 The Design Methodology Phideo

The goal of the design methodology Phideo [Lippens et al., 1991] can be stated as follows. Given a video algorithm and timing requirements, find a hardware structure that implements it, in such a way that a minimal IC area is required. The algorithm is represented by a *signal flow graph*, which consists of nodes representing operations and arcs representing data dependencies. The hardware structure consists of processing units (PUs) on which the operations are executed, memories to store the intermediate data, address generators to supply the memories with the correct addresses for reading and writing intermediate data, interconnections consisting of wires and multiplexers for the transportation of data, and a controller to give the correct control signals. This architectural model is schematically shown in Figure 1.3. The hardware designed by Phideo is synchronous, i.e., there is a cent-

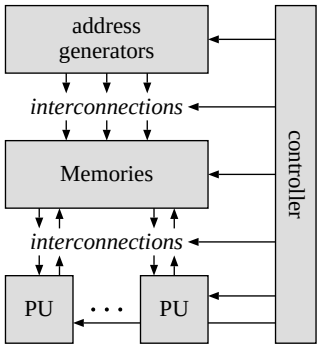


Figure 1.3. The target architecture of Phideo.

ral clock which rules the synchronization between different blocks. The clock frequency is usually a multiple of the sampling frequency.

Since the design problem is too complex to be handled as a whole, the overall design flow of Phideo is decomposed into a number of steps, as shown in Figure

1.4. The decomposition is based on the characteristics of the application domain.

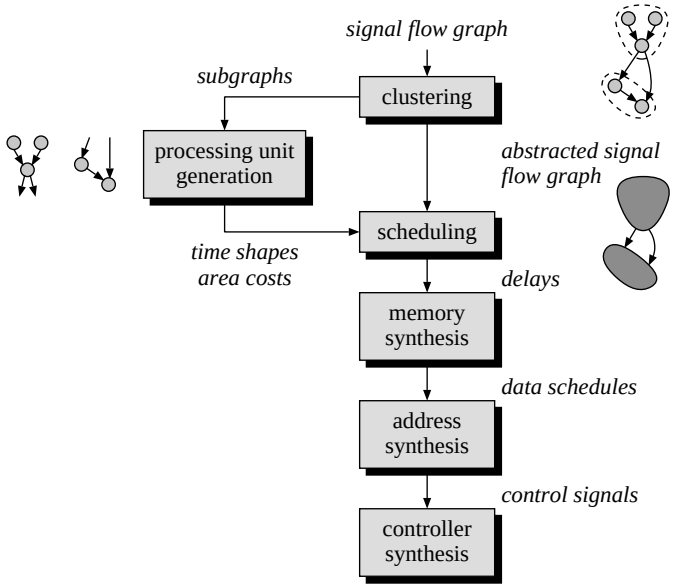


Figure 1.4. The decomposition within Phideo. In the first step clusters in the signal flow graph are defined for which dedicated processing units are generated. Next the clusters are scheduled, a network of memories is constructed, address generators are made, and a controller is synthesized.

Below we discuss some of the steps within Phideo in more detail.

Clustering. The signal flow graph contains many nodes, each corresponding to an elementary operation. Considering all nodes separately during scheduling would be too complex. Furthermore, in the signal flow graph groups of tightly coupled operations can be distinguished, e.g., all elementary operations that make up a filter function. Therefore, in Phideo the first task is to determine which operations are to be grouped together into a cluster. The content of a cluster is identified by a subgraph, for which a dedicated processing unit is generated. Equal clusters can be mapped onto the same processing unit. For the scheduling step each cluster is replaced by an abstract node, so scheduling has to be performed on an abstracted signal flow graph with significantly fewer nodes.

Processing unit generation. For each subgraph defined in the previous step a processing unit has to be synthesized. In order to meet the high clock frequency, retiming, including pipelining, is used [Van der Werf et al., 1991]. By doing this, inputs and outputs of a processing unit may be shifted in time with respect to each other, resulting in a so-called *time shape* of the processing unit. So during schedul-

ing it has to be taken into account that inputs may be required in different clock cycles, and that outputs may be available in different clock cycles. After processing unit generation an area cost is known for each type of processing unit, allowing quantitative trade-offs during scheduling.

Scheduling. The main task during scheduling is to determine for each of the operations in the abstracted signal flow graph a clock cycle in which it has to be started, and to assign the operations to processing units. The operations represent the previously defined clusters. For each of the operations it is specified on which type of processing unit it has to be executed, and for each processing unit the time shape and area costs are specified. Furthermore, timing bounds are given, e.g., on input and output operations. The goal during scheduling is to minimize the total area that is required. Because of the application domain, not only processing units but also memories have a significant contribution to the total area. So both contributions have to be taken into account, and a trade-off has to be made.

A special issue to take care of during scheduling is that operations are multidimensional periodically executed, which places extra demands on constraint checking and cost evaluation.

Memory synthesis. A schedule determines the clock cycles in which the processing units produce output data and when they require new input data, i.e., a schedule determines the required delays of the intermediate data. The data, also called *variables*, are stored in and retrieved from memories. The problem now is to design a configuration of memories and an interconnection network, and to assign the variables to the memories, such that there are no conflicting situations and that the required area is minimal. Conflicts occur when, for instance, two variables have to be stored simultaneously in a memory that has only one input. In that case the conflict has to be resolved, e.g., by assigning the variables to different memories, or by adding hardware for delaying one of the write actions.

Address synthesis. Memory synthesis results in data schedules, which define for each memory the time when data is written into it, and when it is read out again. Next, we have to determine the addresses at which variables are stored, and address generators have to be synthesized to provide these addresses at the correct times. The main problem here is to determine which variables are to be stored at the same memory address, in order to reduce the memory size.

Controller synthesis. Finally, a controller has to be synthesized that provides the correct control signals. For instance, the processing units have to be started at the times determined by scheduling, read-enable and write-enable signals have to be generated for the memories, the control signals for the address generators have to

be generated, and multiplexers have to be controlled in order to route the data to the correct places.

In the past five years, computer-aided design tools have been developed to support the subsequent steps in the design methodology Phideo. Currently, they are being used by several design teams within Philips. In the remainder of this thesis, we restrict ourselves to the scheduling problem. Nevertheless, some of the subproblems in scheduling also occur in other steps within Phideo.

1.3 Informal Statement of the Scheduling Problem

As already mentioned, most video algorithms consist of repetitive executions of operations, resulting in a repetitive production and consumption of data. This can be described by using nested loops and multidimensional arrays. For the moment, we assume that the loop bounds are constants; we return to this assumption in Chapter 8. So, a video algorithm can be viewed as a repetition of executions of operations in several dimensions, each of which corresponds to one loop. A specific execution of an operation can be identified by the corresponding values of the loop iterators. In addition to the repetitive executions, video algorithms contain strict timing requirements that constrain the rates at which input data arrive, and the rates at which output data must be produced.

To handle these characteristics, we introduce an explicit timing model containing operations that are executed periodically. The periods of each operation are given by a *period vector* whose components denote the time between two consecutive iterations in each dimension of repetition. During scheduling, we now have to determine for each operation a period vector and a start time. In addition to this time assignment, we need to assign the operations to processing units, on which they have to be executed.

The constraints during scheduling are threefold. First, we have *timing constraints*, which bound the period vectors and start times of the operations. Some of these bounds may fix the period vectors and start times, e.g., for input and output operations. Secondly, we have *processing unit constraints*, which specify that at most one execution of an operation can occupy a processing unit at a time. Thirdly, we have *precedence constraints*, which are due to data dependencies.

The scheduling objective we consider is to minimize the area occupied by the design, where we have to make a trade-off between processing units and memories. The address generator area can be taken into account in the memory area. The area occupied by the interconnections and the controller is not taken into account. The area of each processing unit is known from the processing unit generation. Hence, the total area occupied by them can be directly obtained from the numbers of allocated processing units. The area occupied by the memories is estimated during

scheduling, based on the required total memory size and memory bandwidth. The memory size is given by the maximum number of variables that are simultaneously alive, while the memory bandwidth is given by the maximum number of simultaneous accesses, i.e., simultaneous productions and consumptions of data by processing units.

So, informally the multidimensional periodic scheduling problem can be stated as follows. Given a signal flow graph and timing requirements, find for each operation a period vector and a start time, and find a processing unit assignment, such that the constraints are satisfied, and such that the total area is minimized.

1.4 Related Work

The problems in computer-aided design of digital ICs have gained much interest in the literature in the past decades. For an overview of these problems and corresponding solution approaches, we refer the reader to McFarland, Parker & Camposano [1990], Borriello & Detjens [1988], Martin & Knight [1993], and Stok [1994].

Since most of the problems raised in the design of digital ICs have a discrete nature, they can be formulated as combinatorial optimization problems. This opens the way to applying the elaborate theory of combinatorial optimization, as can be found in e.g. Papadimitriou & Steiglitz [1982], Schrijver [1986], and Nemhauser & Wolsey [1988]. Furthermore, this allows us to study the computational complexity [Garey & Johnson, 1979] of the design problems.

Scheduling is an important issue in the area of combinatorial optimization. Baker [1974] defines scheduling as the problem of allocating scarce resources to activities over time. Scheduling problems occur in many areas such as IC design, production planning, and computer scheduling, and there exists an extensive body of literature on this subject. For an elaborate introduction to the theory of scheduling we refer to Conway, Maxwell & Miller [1967], Baker [1974], Coffman [1976], French [1982], and Pinedo [1995]. We restrict ourselves to non-preemptive scheduling, which means that executions may not be interrupted. Much of the literature on scheduling concerns the deterministic machine scheduling problem for job shops and flow shops. Nevertheless, several approaches have been presented on scheduling in IC design, such as list scheduling [McFarland, 1986; Haupt, 1989], integer linear programming based scheduling [Gebotys & Elmasry, 1990; Hwang, Lee & Hsu, 1991], and approaches based on domain reduction, also called constraint satisfaction techniques [Paulin & Knight, 1989; Verhaegh, Lippens, Aarts, Korst, Van Meerbergen & Van der Werf, 1995; Timmer & Jess, 1993]. Most of these approaches, however, do not consider periodic operations.

In the area of one-dimensional periodic scheduling, work is done by Korst

[1992], who considers the mapping of video signal processing algorithms onto programmable video signal processors, and by Lee & Messerschmitt [1987], who consider synchronous data flow for digital signal processing. Furthermore, work on one-dimensional periodic scheduling with all operations having the same period is done in the area of pipelined scheduling by Park & Parker [1988] and Verhaegh, Lippens, Aarts, Korst, Van Meerbergen & Van der Werf [1995].

The literature also presents several approaches to the problem of handling multidimensional repetitive executions with multidimensional repetitive productions and consumptions of data, however without periodicity and strict timing requirements. In the area of high-throughput digital signal processing work is done on loop transformations by Goossens, Rabaey, Vandewalle & De Man [1987] and Potkonjak & Rabaey [1994], in which descriptions with loops are modified in order to obtain, for instance, more parallelism and a higher throughput. In their approach the throughput is a result instead of a constraint. Van Swaaij, Franssen, Catthoor & Man [1992] handle the loop transformations by a method based on placement of polytopes. Furthermore, related work is done in the area of systolic array design by Bu, Deprettere & Thiele [1990] and in the area of data flow analysis for parallel program construction by Feautier [1991] and Pugh [1991].

The model in this thesis considers operations that are executed repeatedly, with both multidimensional repetitions and strict periodicity [Verhaegh, Lippens, Aarts, Korst, Van Meerbergen & Van der Werf, 1992]. The executions of the operations are considered as multidimensional repetitions since considering all executions separately is impracticable. The explicit timing in the model, incorporated by the periodicity, has the advantage that it facilitates constraint handling and allows a more adequate cost model. Furthermore, the multidimensional repetitions and the periodicity match very well with the application domain of video signal processing.

1.5 Thesis Outline

The general objective of this thesis is to study the multidimensional periodic scheduling problem, by examining its computational complexity and designing algorithms to find good solutions for it. In the first place, we want to design fast algorithms that can be applied to typical problem instances originating from the design of video signal processing systems. Many choices we make are therefore motivated by the characteristics of the application domain, and special cases of the (sub-)problems we identify are induced by practical situations. Furthermore, the performance of the proposed solution approaches is evaluated by means of several real-life problem instances.

The organization of the remainder of this thesis is as follows. In Chapter 2 we give a model of multidimensional periodic operations and we formulate the multidi-

mensional periodic scheduling problem. In Chapter 3 we discuss the complexity of this problem and two related sub-problems. In Chapter 4 we discuss integer linear programming, which is used to solve the sub-problems. Next, Chapter 5 discusses how to calculate the cost function, and Chapter 6 discusses how to calculate the constraints. Based on that, Chapter 7 presents an algorithm for the multidimensional periodic scheduling problem. In Chapter 8 we extend the problem to the case of parametric signal flow graphs, in which the numbers of repetitions are no longer fixed but depend on parameters, and we present a solution approach for it. Finally, in Chapter 9 we summarize the main results of this thesis.

2

Problem Modeling and Formulation

In this chapter we give a model of multidimensional periodic operations and we formulate the multidimensional periodic scheduling problem. To this end, we first formally define operations and signal flow graphs, in Section 2.1. Next, in Section 2.2 we define schedules which contain the decision variables of the problem. In Section 2.3 we define the timing constraints, the processing unit constraints, and the precedence constraints, and in Section 2.4 we define the objective function. Then, the multidimensional periodic scheduling problem is formulated, in Section 2.5. Finally, Sections 2.6 and 2.7 discuss two special properties of operations that often occur in practice.

2.1 Signal Flow Graphs

Input for scheduling is a signal flow graph representing a video algorithm, and a set of timing requirements. Figure 2.1 shows an example of a part of a video algorithm, in which an operation v is executed in a two-dimensional loop, storing data in a one-dimensional array x. The array is used by an operation u as input, which is executed in a one-dimensional loop.

Before describing a signal flow graph, we first describe the operation types, and the corresponding processing unit types. We assume that the operations in a signal

```
for i0 = 0 to 1
    for i1 = 0 to 2
        x[3i0 + i1] = v(... , ... )
for j0 = 0 to 5
    ... , ... = u(x[j0] )
```

Figure 2.1. Example of a part of a video algorithm, with a two-dimensional operation *v* and a one-dimensional operation *u*. Operation *v* has two inputs and one output; the inputs are not filled in. Operation *u* has one input and two outputs; the latter are not filled in. The data transport between *v* and *u* is described by means of a one-dimensional array *x*.

flow graph must be executed on dedicated processing units, i.e., we assume a one-to-one relation between operation types and processing unit types. Before we define them, we mention that the time unit we maintain throughout this thesis is the clock cycle, and all time points are given by clock cycles $c \in \mathbb{Z}$, where $\mathbb{Z}$ is the set of integer numbers. For an explanation of the symbols in the definition, see the text below.

Definition 2.1 (Operation Type Set). An operation type set is given by a 6-tuple $(T, \tilde{I}, \tilde{O}, r, o, a)$, where

- T is a set of operation types,
- $\tilde{I}(t)$ denotes the set of *operation input ports*, for each $t \in T$,
- $\tilde{O}(t)$ denotes the set of *operation output ports*, for each $t \in T$,
- $r(t, \tilde{p}) \in \mathbb{Z}$ denotes the *relative transfer time*, for each $t \in T$ and $\tilde{p} \in \tilde{P}(t) = \tilde{I}(t) \cup \tilde{O}(t)$,
- $o(t) \in \mathbb{N}$ denotes the *occupation time*, for each $t \in T$, and
- $a(t) \in \mathbb{N}$ denotes the *area cost*, for each $t \in T$.

□

Here, $\mathbb{N}$ is the set of non-negative integers. The definition allows an operation type to have several inputs and several outputs. The time at which input is required or output is available, is expressed by the relative transfer time of an operation port, which is relative to the clock cycle in which an execution of the operation is scheduled. The occupation time denotes the number of clock cycles that an execution of an operation occupies a processing unit without interruption. So, if an execution of an operation of type t is scheduled at time c, then the production or consumption of a variable at operation port $\tilde{p} \in \tilde{P}(t)$ takes place at time $c + r(t, \tilde{p})$, and the processing unit that executes the operation is occupied at time $c, \ldots, c + o(t) \Leftrightarrow 1$.

Figure 2.2 shows an example of an operation of a type t with two operation input ports, $\tilde{I}(t) = \{1,2\}$, and one operation output port, $\tilde{O}(t) = \{3\}$. The relative transfer times are $r(t,1) = 0$, $r(t,2) = 1$, and $r(t,3) = 4$, and the operation is scheduled at time $c = 2$. This implies that the consumption at operation input port 1 takes place

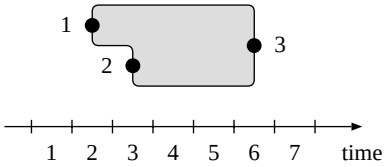


Figure 2.2. An example of an operation of type t with two operation input ports and one operation output port, denoted by black dots. The operation is scheduled at time 2, and the relative transfer times of ports 1, 2, and 3 are 0, 1, and 4, respectively.

at time $2+0=2$, at operation input port 2 at time $2+1=3$, and the production at operation output port 3 takes place at time $2+4=6$. If the occupation time $o(t) = 2$, then this operation occupies a processing unit at times 2 and 3, so the assigned processing unit is again available as of time 4 to start processing new input data. Note that in this way, the processing of successive input data may overlap, which is called *pipelining*. For simplicity, we assume that in further examples the relative transfer times equal zero and the occupation times equal one.

Input and output nodes of a signal flow graph are modeled as operations without operation input ports and operation output ports, respectively.

Now we can define a signal flow graph as follows. For an explanation of the symbols, see the text below.

Definition 2.2 (Signal Flow Graph). A signal flow graph G is given by a 6-tuple $(V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, where

- V is a finite set of multidimensional periodic operations,
- $t(v) \in T$ denotes the operation type, for each $v \in V$,
- $\boldsymbol{I}(v) \in \mathbb{N}_\infty^{\delta(v)}$ denotes the *iterator bound vector*, for each $v \in V$,
- $E \subset O \times I$ is a finite set of directed edges representing data dependencies, where $O = \{(v,\tilde{o}) \mid v \in V \wedge \tilde{o} \in \tilde{O}(t(v))\}$ denotes the set of *output ports*, and $I = \{(v,\tilde{\imath}) \mid v \in V \wedge \tilde{\imath} \in \tilde{I}(t(v))\}$ denotes the set of *input ports*,
- $\boldsymbol{A}(p) \in \mathbb{Z}^{\alpha(p) \times \delta(v)}$ denotes the *index matrix*, for each $p = (v,\tilde{p}) \in P = I \cup O$,
- $\boldsymbol{b}(p) \in \mathbb{Z}^{\alpha(p)}$ denotes the *index offset vector*, for each $p \in P$.

□

Here, $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$. The length $\delta(v)$ of an iterator bound vector $\boldsymbol{I}(v)$ denotes the number of dimensions in which the operation is repeated, i.e., the number of enclosing loops. Each execution of an operation $v \in V$ can be identified by an *iterator vector* $\boldsymbol{i} \in \mathbb{Z}^{\delta(v)}$, for which $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(v)$. So, the iterator i_k in dimension $k = 0,\ldots,\delta \Leftrightarrow 1$ runs from 0 to $I_k(v)$. We assume that only dimension 0 may have an unbounded number of repetitions, denoted by $I_0 = \infty$. The numbers of repetitions in the other dimensions are finite. The set of all possible iterator vectors of operation v is given by $\mathcal{I}(v) = \{\boldsymbol{i} \in \mathbb{Z}^{\delta(v)} \mid \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(v)\}$, and is called the *iterator space*.

Figure 2.3 shows the two-dimensional operation v of Figure 2.1. The operation

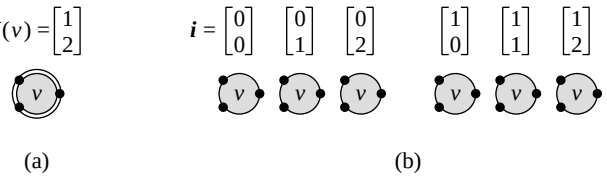


Figure 2.3. (a) An example of a two-dimensional operation, denoted by a double circle, and (b) its executions, denoted by single circles.

has two iterators, i_0 and i_1, with iterator bounds $I_0(v) = 1$ and $I_1(v) = 2$, respectively. This results in six executions of operation v.

Note that if $\delta(v) = 0$ for an operation v, which means that the operation is not enclosed by loops, then the operation has exactly one execution. This execution is identified by the empty vector $[\,]$, i.e., a vector with no entries. Furthermore, we can assume the iterator bounds $I_k(v)$ to be positive, since otherwise there is only one iteration in dimension k.

In a video algorithm, data transport between operations is described by means of multidimensional arrays. In a signal flow graph this is represented by means of output and input ports, and edges between them. At an output port $p = (v, \tilde{o})$ of an operation v, data is produced at each execution $\boldsymbol{i}$ of v, which corresponds to an element of a multidimensional array with dimension $\alpha(p)$. Such an element is indexed by an *index vector* $\boldsymbol{n}(p, \boldsymbol{i}) \in \mathbb{Z}^{\alpha(p)}$, which is given by

$$\boldsymbol{n}(p, \boldsymbol{i}) = \boldsymbol{A}(p)\,\boldsymbol{i} + \boldsymbol{b}(p),$$

given the index matrix $\boldsymbol{A}(p)$ and index offset vector $\boldsymbol{b}(p)$. Note that we assume that the index can be written as a linear expression in the iterator vector, which is a valid assumption in video signal processing. If we now have an input port $q = (u, \tilde{\imath})$ of an operation u, with $(p, q) \in E$, then the array element is consumed at input port q at execution $\boldsymbol{j}$ of operation u, if $\boldsymbol{n}(q, \boldsymbol{j}) = \boldsymbol{n}(p, \boldsymbol{i})$, where similarly $\boldsymbol{n}(q, \boldsymbol{j}) = \boldsymbol{A}(q)\,\boldsymbol{j} + \boldsymbol{b}(q)$. Obviously, we must have that $\alpha(p) = \alpha(q)$ for each $(p, q) \in E$, since both p and q access the same multidimensional array.

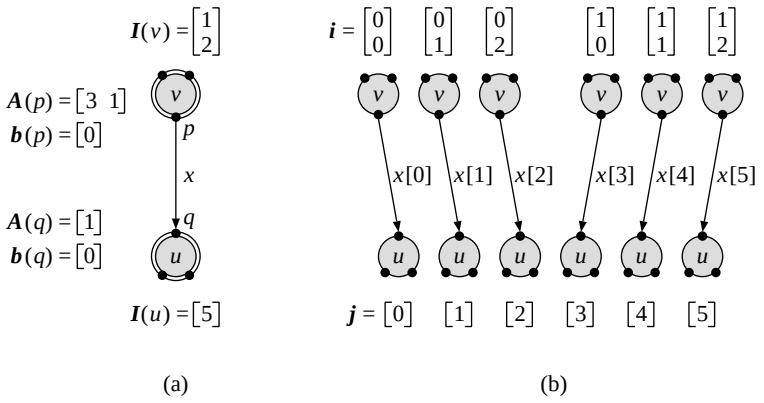


Figure 2.4. (a) An example of data transport between two multidimensional operations via a one-dimensional array x, and (b) the different executions and array elements.

Figure 2.4 shows the data transport between operation v and u of Figure 2.1. The relation between the index vector $\boldsymbol{n}$ and iterator vector $\boldsymbol{i}$ at output port p of operation v is given by

$$\boldsymbol{n}(p,\boldsymbol{i}) = \begin{bmatrix} 3 & 1 \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} = \begin{bmatrix} 3i_0 + i_1 \end{bmatrix}.$$

At input port q of operation u the relation between the index vector $\boldsymbol{n}$ and iterator vector $\boldsymbol{j}$ is given by $\boldsymbol{n}(q,\boldsymbol{j}) = [j_0]$. So, execution $\boldsymbol{j}$ of operation u uses data produced at execution $\boldsymbol{i}$ of operation v if and only if $3i_0 + i_1 = j_0$.

For the production of data we assume single assignments, i.e., each element of an array can be produced at most once. Furthermore, if an operation v is repeated infinitely, i.e., $I_0(v) = \infty$, then we assume that only the first index of the produced and consumed array elements may depend on iterator i_0, with a positive coefficient. This index is often called the *time index* of the array. The assumption is realistic in digital signal processing algorithms, where mostly the infinite repetition and the time index are not described explicitly. The assumption means that for the index matrix $\boldsymbol{A}(p)$ of each port $p = (v, \tilde{p})$ of such an operation v the first column must obey $A_{00}(p) > 0$ and $A_{k0}(p) = 0$ for $k = 1,\ldots,\alpha(p) \Leftrightarrow 1$.

2.2 Schedules

Next, we need the definition of a schedule.

Definition 2.3 (Schedule). Given a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, a schedule σ is given by a time assignment and a processing unit assignment, represented

by a 4-tuple $(\boldsymbol{p}, s, W, h)$, where

- the *time assignment* is given by a period vector $\boldsymbol{p}(v) \in \mathbb{Z}^{\delta(v)}$ and a start time $s(v) \in \mathbb{Z}$, for each operation $v \in V$, and
- the *processing unit assignment* is given by a set W of processing units and a function $h : V \to W$ that assigns each operation to a processing unit that executes it, such that $t(v) = t(h(v))$, for all $v \in V$. Note that we assumed that type function t is defined for both operations and processing units.

The set of all possible schedules is denoted by $\mathcal{S}$. □

The start time $s(v)$ of an operation $v \in V$ is the time of execution $\boldsymbol{i} = \mathbf{0}$. The time $c(v, \boldsymbol{i})$ at which an execution $\boldsymbol{i}$ of operation v is scheduled is given by

$$c(v, \boldsymbol{i}) = \boldsymbol{p}^{\mathrm{T}}(v)\, \boldsymbol{i} + s(v).$$

Figure 2.5 shows two possible time assignments for the operation v of Figure 2.1. For the first of these two time assignments, the time at which execution $\boldsymbol{i}$ of opera-

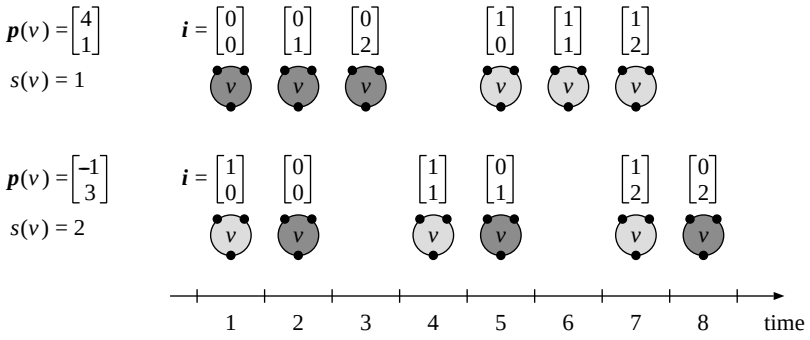


Figure 2.5. Two possible time assignments for a two-dimensional operation.

tion v takes place is given by

$$c(v, \boldsymbol{i}) = \begin{bmatrix} 4 & 1 \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \end{bmatrix} + 1 = 4i_0 + i_1 + 1.$$

The time at which data is produced or consumed at port $p = (v, \tilde{p})$ at execution $\boldsymbol{i}$ of an operation v is given by the time of that execution plus the relative transfer time, i.e.,

$$\begin{aligned} c(p, \boldsymbol{i}) &= \boldsymbol{p}^{\mathrm{T}}(v)\, \boldsymbol{i} + s(v) + r(t(v), \tilde{p}) \\ &= \boldsymbol{p}^{\mathrm{T}}(p)\, \boldsymbol{i} + s(p), \end{aligned}$$

where we define $\boldsymbol{p}(p) = \boldsymbol{p}(v)$ and $s(p) = s(v) + r(t(v), \tilde{p})$. For notational ease, we also define $\boldsymbol{I}(p) = \boldsymbol{I}(v)$, $\mathcal{I}(p) = \mathcal{I}(v)$, and $\delta(p) = \delta(v)$, and we speak of execution $\boldsymbol{i}$ of port p.

2.3 Constraints

Next, we define the constraints that a schedule must satisfy.

Definition 2.4 (Timing Constraints). Given are a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, a schedule $\sigma = (\boldsymbol{p}, s, W, h)$, and for each operation $v \in V$ a lower bound $\underline{\boldsymbol{p}}(v) \in \mathbb{Z}_\infty^{\delta(v)}$ and an upper bound $\overline{\boldsymbol{p}}(v) \in \mathbb{Z}_\infty^{\delta(v)}$ on the period vector $\boldsymbol{p}(v)$, and a lower bound $\underline{s}(v) \in \mathbb{Z}_\infty$ and an upper bound $\overline{s}(v) \in \mathbb{Z}_\infty$ on the start time $s(v)$, where $\mathbb{Z}_\infty = \mathbb{Z} \cup \{-\infty, +\infty\}$. Then the timing constraints specify that

$$\underline{\boldsymbol{p}}(v) \leq \boldsymbol{p}(v) \leq \overline{\boldsymbol{p}}(v) \quad \wedge \quad \underline{s}(v) \leq s(v) \leq \overline{s}(v),$$

for all $v \in V$. □

If an operation v is repeated infinitely, i.e., $I_0(v) = \infty$, then we assume that the period of the corresponding iterator is fixed and positive, i.e., $\underline{p}_0(v) = \overline{p}_0(v) > 0$.

Definition 2.5 (Processing Unit Constraints). Given are a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ and a schedule $\sigma = (\boldsymbol{p}, s, W, h)$. Then for each execution $\boldsymbol{i}$ of an operation $u \in V$, and for each execution $\boldsymbol{j}$ of an operation $v \in V$, with $h(u) = h(v)$ and $(u, \boldsymbol{i}) \neq (v, \boldsymbol{j})$, the processing unit constraints specify that

$$c(u, \boldsymbol{i}) + k \neq c(v, \boldsymbol{j}) + l,$$

for all $k, l \in \{0, \ldots, o - 1\}$, where $o = o(t(u)) = o(t(v))$. □

The processing unit constraints state that at most one execution of an operation can occupy a processing unit at a time. Note that this applies to two executions of two different operations, as well as to two different executions of one operation.

Definition 2.6 (Precedence Constraints). Given are a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ and a schedule $\sigma = (\boldsymbol{p}, s, W, h)$. Then for each execution $\boldsymbol{i}$ of an output port $p \in O$, and for each execution $\boldsymbol{j}$ of an input port $q \in I$, with $(p, q) \in E$, the precedence constraints specify that

$$\boldsymbol{n}(p, \boldsymbol{i}) = \boldsymbol{n}(q, \boldsymbol{j}) \;\Rightarrow\; c(p, \boldsymbol{i}) < c(q, \boldsymbol{j}).$$

□

The precedence constraints state that if at execution $\boldsymbol{i}$ of output port p an array element is produced with the same index as at execution $\boldsymbol{j}$ of a connected input port q, then the production must take place before the consumption.

If we have an output port p and an input port q with $(p, q) \in E$, and both p and q are repeated infinitely, i.e., $I_0(p) = I_0(q) = \infty$, then we have the following situation. At output port p the first index n_0 of the array increases by $A_{00}(p)$ every $p_0(p)$ time units. Similarly, at input port q the first index of the array increases by $A_{00}(q)$ every $p_0(q)$ time units. Since we have an infinite number of repetitions, it is necessary that

the pace of increasing n_0 at the production side and at the consumption side is equal, in order to guarantee that there is no overflow or underflow of data. Therefore, we assume $A_{00}(p)/p_0(p) = A_{00}(q)/p_0(q)$ in this case.

A schedule $\sigma \in \mathcal{S}$ is called feasible if it obeys the timing constraints, the processing unit constraints, and the precedence constraints. The set of all feasible schedules is denoted by $\mathcal{S}'$.

2.4 Objectives

The cost function we consider in this thesis reflects the total area of a design. As already mentioned in Section 1.3, this area is not only determined by the number of processing units of each type, but also by the required total memory size and memory bandwidth. Therefore, we include in the cost function the maximum number of variables that are simultaneously alive and the maximum number of simultaneous memory accesses. To this end, we define a set of *resource types* $T^* = T \cup \{t_v, t_a\}$, where t_v is the resource type corresponding to memory cells, with area cost $a(t_v)$ for each variable, and t_a is the resource type corresponding to memory terminals, with area cost $a(t_a)$ for each access.

In order to avoid multiple countings, we determine the lifetimes and accesses of variables for each cluster of ports belonging to one array, rather than for each data precedence in E. To this end, we first define array clusters.

Definition 2.7 (Array Clusters). Two ports $p, q \in P$ are said to access the same array, denoted by $p \bowtie q$ and pronounced as p joins q, if and only if they are weakly connected, i.e., $p = q$ or there is a path of ports $(p_0, p_1, ..., p_n)$ with $p_0 = p$ and $p_n = q$ and

$$(p_i, p_{i+1}) \in E \quad \vee \quad (p_{i+1}, p_i) \in E$$

for all $i = 0, ..., n \Leftrightarrow 1$. Now, an array cluster is denoted by a set A of ports, which is a subset of P, such that $p \bowtie q$ for all $p, q \in A$, and such that A cannot be extended. The set of all array clusters is denoted by $\mathcal{A}$. □

Figure 2.6 shows an example of a signal flow graph with two array clusters. The first

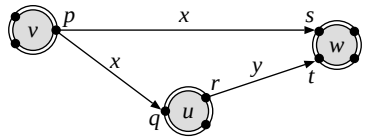


Figure 2.6. An example of a signal flow graph with two array clusters: cluster $\{p, q, s\}$ corresponding to array x and cluster $\{r, t\}$ corresponding to array y.

cluster is $\{p, q, s\}$, corresponding to array x of the corresponding video algorithm,

and the second cluster is $\{r,t\}$, corresponding to array y. For the example of Figure 2.4, we have only one array cluster, $\{p,q\}$.

To determine the number of variables that are simultaneously alive, we first have to determine when a variable is produced and when it is consumed. At execution $\boldsymbol{i}$ of output port p an array element is produced with index vector $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{A}(p)\,\boldsymbol{i}+\boldsymbol{b}(p)$, which takes place at time $c(p,\boldsymbol{i}) = \boldsymbol{p}^{\mathrm{T}}(p)\,\boldsymbol{i}+s(p)$. The array element is consumed at execution $\boldsymbol{j}$ of input port q if $(p,q)\in E$ and $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) = \boldsymbol{A}(q)\,\boldsymbol{j}+\boldsymbol{b}(q)$. This takes place at time $c(q,\boldsymbol{j}) = \boldsymbol{p}^{\mathrm{T}}(q)\,\boldsymbol{j}+s(q)$. Now, assuming that a variable is alive from the first time after its production up to and including the time of its last consumption, the set of array elements of an array cluster $A\in\mathcal{A}$ that are alive at time c is given by

$$\mathcal{L}(A,c) = \{\boldsymbol{n}\in\mathbb{Z}^{\alpha} \mid\ p,q\in A\ \wedge\ (p,q)\in E\ \wedge\ \boldsymbol{i}\in\mathcal{I}(p)\ \wedge\ \boldsymbol{j}\in\mathcal{I}(q)\ \wedge\ \boldsymbol{n}=\boldsymbol{n}(p,\boldsymbol{i})=\boldsymbol{n}(q,\boldsymbol{j})\ \wedge\ c(p,\boldsymbol{i})<c\le c(q,\boldsymbol{j})\}.$$

For an example of lifetimes of variables, see Figure 2.7, which shows the operations

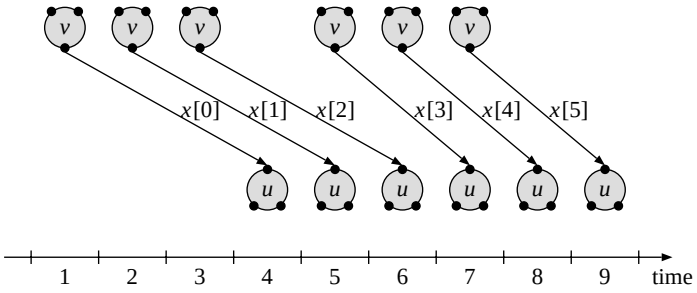


Figure 2.7. An example of productions and consumptions of array elements in time.

v and u of Figure 2.4 for a time assignment $\boldsymbol{p}(v) = [4\ 1]^{\mathrm{T}}$, $s(v)=1$, $\boldsymbol{p}(u)=[1]$, and $s(u)=4$. In the figure, we can see that the element of the array cluster $A=\{p,q\}$ with index vector $[0]$ is produced at time 1 and consumed at time 4, so it is alive at times 2, 3, and 4. The elements that are alive at, for instance, time 4 in this example, are given by $\mathcal{L}(A,4) = \{[0],[1],[2]\}$.

Next, we determine the set of array elements of an array cluster $A\in\mathcal{A}$ that are accessed at time c, i.e., that are produced or consumed at time c. This set is given by

$$\mathcal{B}(A,c) = \{\boldsymbol{n}\in\mathbb{Z}^{\alpha} \mid\ p\in A\ \wedge\ \boldsymbol{i}\in\mathcal{I}(p)\ \wedge\ \boldsymbol{n}=\boldsymbol{n}(p,\boldsymbol{i})\ \wedge\ c(p,\boldsymbol{i})=c\}.$$

Note that multiple accesses of the same array element at the same time are counted only once, since they are collected in a set. For the example of Figure 2.7, we see that the elements that are accessed at, for instance, time 5 are given by $\mathcal{B}(A,5) = \{[1],[3]\}$, since at time 5 element $[1]$ is consumed and element $[3]$ is produced.

Definition 2.8 (Total Area Cost). The total area cost of a schedule $\sigma \in \mathcal{S}'$ is the sum of the processing unit cost, the storage cost, and the access cost, and is given by

$$f(\sigma) = \sum_{w \in W} a(t(w)) \; + \; a(t_{\mathrm{v}}) \max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{L}(A,c)| \; + \; a(t_{\mathrm{a}}) \max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{B}(A,c)|.$$

□

If we take the time assignment of Figure 2.7, and assign operations u and v each to a unique processing unit, then the cost of the corresponding schedule σ is given by $f(\sigma) = a(t(u)) + a(t(v)) + 3a(t_{\mathrm{v}}) + 2a(t_{\mathrm{a}})$. So if all resources have area cost 1, then this schedule has a total area cost of 7.

2.5 Problem Formulation

Using the definitions of the previous sections, we can now define the scheduling problem we consider in this thesis.

Definition 2.9 (Multidimensional Periodic Scheduling (MPS)). Given are a signal flow graph G, a set of resource types T^*, and lower and upper bounds on the period vectors and start times of the operations. Find a schedule σ that obeys the timing constraints, processing unit constraints, and precedence constraints, for which the total area cost $f(\sigma)$ is minimal. □

2.6 Lexicographical Executions

In this section, we discuss a special property that often occurs in practice, and which is used to identify special cases of multidimensional periodic scheduling in the remainder of this thesis.

Often, the periods of repetition of an operation are such that we can identify an innermost loop, with the smallest period, which is completely executed within the next greater period. This next outer loop is then again completely executed within the next greater period, etc. For instance, consider an operation that is repeated for all pixels in a line, and for a number of lines, with the repetition of the pixels being completed within the line period. This characteristic is captured in the following definition.

Definition 2.10 (Lexicographical Execution). An iterator bound vector $\boldsymbol{I} \in \mathbb{N}_\infty^\delta$, a period vector $\boldsymbol{p} \in \mathbb{Z}^\delta$, and an occupation time $o \in \mathbb{N}_+$ are said to give a lexicographical execution, denoted by $\mathrm{lex}(\boldsymbol{I}, \boldsymbol{p}, o)$, if and only if for all vectors $\boldsymbol{i}, \boldsymbol{j} \in \mathbb{Z}^\delta$ with $\boldsymbol{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I}$ holds

$$\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j} \;\Leftrightarrow\; \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} + o \leq \boldsymbol{p}^{\mathrm{T}} \boldsymbol{j}.$$

□

Here, $\mathbb{N}_+$ denotes the set of positive integers, and $<_{\text{lex}}$ stands for lexicographically smaller than, i.e., $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$ if and only if there exists a $k \in \{0,\ldots,\delta \Leftrightarrow 1\}$ such that $i_l = j_l$, for all $l = 0,\ldots,k \Leftrightarrow 1$, and $i_k < j_k$. Sometimes, the occupation time o is omitted, in which case it is assumed to be equal to 1. If $\text{lex}(\boldsymbol{I},\boldsymbol{p},o)$ holds for an operation, then the executions of it take place in order of lexicographically increasing iterator vector $\boldsymbol{i}$.

An example of a lexicographical execution is given by operation v of Figure 2.5 with the first time assignment shown there. The second time assignment in that figure does not give a lexicographical execution. Nevertheless, by permuting the iterators, and by making the periods positive, the second time assignment can be made to give a lexicographical execution.

Checking whether an iterator bound vector, a period vector, and an occupation time give a lexicographical execution can be done in polynomial time by means of the following theorem.

Theorem 2.1. *Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_\infty^\delta$, with $I_k > 0$, for all $k = 0,\ldots,\delta \Leftrightarrow 1$, a period vector $\boldsymbol{p} \in \mathbb{Z}^\delta$, and an occupation time $o \in \mathbb{N}_+$. Then*

$$\text{lex}(\boldsymbol{I},\boldsymbol{p},o)$$

holds if and only if

$$p_0 > p_1 > \cdots > p_{\delta-1} > 0, \tag{2.1}$$

and

$$p_k \geq \sum_{l=k+1}^{\delta-1} p_l I_l + o, \tag{2.2}$$

for all $k = 0,\ldots,\delta \Leftrightarrow 1$.

Proof. To show the necessity of (2.1), we define $\boldsymbol{i},\boldsymbol{j} \in \mathbb{Z}^\delta$ as

$$i_l = \begin{cases} 1 & \text{for } l = k+1 \\ 0 & \text{for } l \neq k+1, \text{ and} \end{cases}$$

$$j_l = \begin{cases} 1 & \text{for } l = k \\ 0 & \text{for } l \neq k. \end{cases}$$

Then $\boldsymbol{0} \leq \boldsymbol{i},\boldsymbol{j} \leq \boldsymbol{I}$ and $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$. So, if $\text{lex}(\boldsymbol{I},\boldsymbol{p},o)$ holds, then we have

$$p_k = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{j} \geq \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + o > \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = p_{k+1}.$$

Furthermore, if we define $\boldsymbol{i} = \boldsymbol{0}$ and $\boldsymbol{j}$ as

$$j_l = \begin{cases} 1 & \text{for } l = \delta \Leftrightarrow 1 \\ 0 & \text{for } l \neq \delta \Leftrightarrow 1, \end{cases}$$

then we derive analogously that $p_{\delta-1} > 0$.

To show the necessity of (2.2), we define $\boldsymbol{i}, \boldsymbol{j} \in \mathbb{Z}^{\delta}$ as

$$i_l = \begin{cases} 0 & \text{for } l = 0,\ldots,k \\ I_l & \text{for } l = k+1,\ldots,\delta \Leftrightarrow 1, \text{ and} \end{cases}$$

$$j_l = \begin{cases} 1 & \text{for } l = k \\ 0 & \text{for } l \neq k. \end{cases}$$

Now again, $\mathbf{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I}$ and $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$. So, if $\text{lex}(\boldsymbol{I}, \boldsymbol{p}, o)$ holds, then we have

$$p_k = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{j} \geq \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + o = \sum_{l=k+1}^{\delta-1} p_l I_l + o.$$

To show the sufficiency of (2.1) and (2.2), we take $\boldsymbol{i}, \boldsymbol{j} \in \mathbb{Z}^{\delta}$ with $\mathbf{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I}$ and $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$. Let k be the first component for which $\boldsymbol{i}$ and $\boldsymbol{j}$ differ, i.e., $i_l = j_l$, for all $l = 0,\ldots,k \Leftrightarrow 1$, and $i_k < j_k$. Then we have

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + o &= \sum_{l=0}^{\delta-1} p_l i_l + o \\ &\leq \sum_{l=0}^{k-1} p_l i_l + p_k i_k + \sum_{l=k+1}^{\delta-1} p_l I_l + o \\ &\leq \sum_{l=0}^{k-1} p_l i_l + p_k (i_k + 1) \\ &\leq \sum_{l=0}^{k-1} p_l j_l + p_k j_k \\ &\leq \sum_{l=0}^{\delta-1} p_l j_l \\ &= \boldsymbol{p}^{\mathrm{T}}\boldsymbol{j}. \end{aligned}$$

Analogously, if $\boldsymbol{i} \geq_{\text{lex}} \boldsymbol{j}$ then $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + o > \boldsymbol{p}^{\mathrm{T}}\boldsymbol{j}$. So, $\text{lex}(\boldsymbol{I}, \boldsymbol{p}, o)$ holds. This completes the proof. □

2.7 Lexicographical Index Orderings

In this section, we discuss a second special property that often occurs in practice, and which is used to identify special cases of multidimensional periodic scheduling. This property, which is called a lexicographical index ordering, is analogous to the property of a lexicographical execution. It implies that a lexicographically larger iterator vector results in a lexicographically larger index vector.

Definition 2.11 (Lexicographical Index Ordering). An iterator bound vector $\boldsymbol{I} \in \mathbb{N}_{\infty}^{\delta}$ and an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$ are said to give a lexicographical index ordering,

denoted by $\mathrm{lio}(\boldsymbol{I},\boldsymbol{A})$, if and only if for all vectors $\boldsymbol{i},\boldsymbol{j} \in \mathbb{Z}^{\delta}$ with $\mathbf{0} \leq \boldsymbol{i},\boldsymbol{j} \leq \boldsymbol{I}$ holds

$$\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j} \;\Leftrightarrow\; \boldsymbol{A}\,\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{A}\,\boldsymbol{j}.$$

□

An example of a lexicographical index ordering is given by the output port p of operation v in Figure 2.4.

Checking whether an iterator bound vector and an index matrix give a lexicographical index ordering can be done in polynomial time by means of the following theorem.

Theorem 2.2. *Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_{\infty}^{\delta}$, with $I_k > 0$ for all $k = 0,\ldots,\delta\Leftrightarrow 1$, and an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha\times\delta}$. Then*

$$\mathrm{lio}(\boldsymbol{I},\boldsymbol{A})$$

holds if and only if

$$\boldsymbol{A}_{\cdot 0} >_{\mathrm{lex}} \boldsymbol{A}_{\cdot 1} >_{\mathrm{lex}} \cdots >_{\mathrm{lex}} \boldsymbol{A}_{\cdot\delta-1} >_{\mathrm{lex}} \mathbf{0}, \tag{2.3}$$

and

$$\boldsymbol{A}_{\cdot k} >_{\mathrm{lex}} \sum_{l=k+1}^{\delta-1} \boldsymbol{A}_{\cdot l}\, I_l \tag{2.4}$$

for all $k = 0,\ldots,\delta\Leftrightarrow 1$. Here, $\boldsymbol{A}_{\cdot k}$ denotes column k of matrix $\boldsymbol{A}$.

Proof. To show the necessity of (2.3), we define $\boldsymbol{i},\boldsymbol{j} \in \mathbb{Z}^{\delta}$ as

$$i_l = \begin{cases} 1 & \text{for } l = k+1 \\ 0 & \text{for } l \neq k+1, \text{ and} \end{cases}$$

$$j_l = \begin{cases} 1 & \text{for } l = k \\ 0 & \text{for } l \neq k. \end{cases}$$

Then $\mathbf{0} \leq \boldsymbol{i},\boldsymbol{j} \leq \boldsymbol{I}$ and $\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j}$. So, if $\mathrm{lio}(\boldsymbol{I},\boldsymbol{A})$ holds, then we have

$$\boldsymbol{A}_{\cdot k} = \boldsymbol{A}\,\boldsymbol{j} >_{\mathrm{lex}} \boldsymbol{A}\,\boldsymbol{i} = \boldsymbol{A}_{\cdot k+1}.$$

Furthermore, if we define $\boldsymbol{i} = \mathbf{0}$ and $\boldsymbol{j}$ as

$$j_l = \begin{cases} 1 & \text{for } l = \delta\Leftrightarrow 1 \\ 0 & \text{for } l \neq \delta\Leftrightarrow 1, \end{cases}$$

then we derive analogously that $\boldsymbol{A}_{\cdot\delta-1} >_{\mathrm{lex}} \mathbf{0}$.

To show the necessity of (2.4), we define $\boldsymbol{i},\boldsymbol{j} \in \mathbb{Z}^{\delta}$ as

$$i_l = \begin{cases} 0 & \text{for } l = 0,\ldots,k \\ I_l & \text{for } l = k+1,\ldots,\delta\Leftrightarrow 1, \text{ and} \end{cases}$$

$$j_l = \begin{cases} 1 & \text{for } l = k \\ 0 & \text{for } l \neq k. \end{cases}$$

Now again, $\mathbf{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I}$ and $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$. So, if $\text{lio}(\boldsymbol{I}, \boldsymbol{A})$ holds, then we have

$$\boldsymbol{A}_{\cdot k} = \boldsymbol{A}\,\boldsymbol{j} >_{\text{lex}} \boldsymbol{A}\,\boldsymbol{i} = \sum_{l=k+1}^{\delta-1} \boldsymbol{A}_{\cdot l}\, I_l.$$

To show the sufficiency of (2.3) and (2.4), we take $\boldsymbol{i}, \boldsymbol{j} \in \mathbb{Z}^{\delta}$ with $\mathbf{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I}$ and $\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$. Let k be the first component for which $\boldsymbol{i}$ and $\boldsymbol{j}$ differ, i.e., $i_l = j_l$ for $l = 0,\ldots,k \Leftrightarrow 1$ and $i_k < j_k$. Then we have

$$\begin{aligned}
\boldsymbol{A}\,\boldsymbol{i} &= \sum_{l=0}^{\delta-1} \boldsymbol{A}_{\cdot l}\, i_l \\
&\leq_{\text{lex}} \sum_{l=0}^{k-1} \boldsymbol{A}_{\cdot l}\, i_l + \boldsymbol{A}_{\cdot k}\, i_k + \sum_{l=k+1}^{\delta-1} \boldsymbol{A}_{\cdot l}\, I_l \\
&<_{\text{lex}} \sum_{l=0}^{k-1} \boldsymbol{A}_{\cdot l}\, i_l + \boldsymbol{A}_{\cdot k}\,(i_k + 1) \\
&\leq_{\text{lex}} \sum_{l=0}^{k-1} \boldsymbol{A}_{\cdot l}\, j_l + \boldsymbol{A}_{\cdot k}\, j_k \\
&\leq_{\text{lex}} \sum_{l=0}^{\delta-1} \boldsymbol{A}_{\cdot l}\, j_l \\
&= \boldsymbol{A}\,\boldsymbol{j}.
\end{aligned}$$

Analogously, if $\boldsymbol{i} \geq_{\text{lex}} \boldsymbol{j}$ then $\boldsymbol{A}\,\boldsymbol{i} \geq_{\text{lex}} \boldsymbol{A}\,\boldsymbol{j}$. So, $\text{lio}(\boldsymbol{I}, \boldsymbol{A})$ holds. This completes the proof. □

3

Complexity Analysis

In this chapter we discuss the computational complexity of the multidimensional periodic scheduling problem. We first consider the complexity of checking processing unit constraints and precedence constraints, in Sections 3.1 and 3.2, respectively, and the various special cases mentioned there. In Section 3.3 we discuss the complexity of the multidimensional periodic scheduling problem in its entirety.

3.1 Processing Unit Constraints

In this section we analyze the computational complexity of checking processing unit constraints. To this end, we use a complementary formulation, and we restrict ourselves to the special case where we have two different operations that are assigned to the same processing unit.

Definition 3.1 (Processing Unit Conflict (PUC)). Given are two operations u and v, with their iterator bound vectors $\boldsymbol{I}(u) \in \mathbb{N}_\infty^{\delta(u)}$, $\boldsymbol{I}(v) \in \mathbb{N}_\infty^{\delta(v)}$, for which $I_k(u), I_k(v) \neq \infty$ for all $k > 0$. Furthermore, their period vectors $\boldsymbol{p}(u) \in \mathbb{Z}^{\delta(u)}$, $\boldsymbol{p}(v) \in \mathbb{Z}^{\delta(v)}$, their start times $s(u), s(v) \in \mathbb{Z}$, and an occupation time $o \in \mathbb{N}_+$ are

given. Determine whether there are vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ and numbers x and y that satisfy

$$\begin{array}{c} \boldsymbol{p}^{\mathrm{T}}(u)\,\boldsymbol{i}+s(u)+x=\boldsymbol{p}^{\mathrm{T}}(v)\,\boldsymbol{j}+s(v)+y \\ \mathbf{0}\le\boldsymbol{i}\le\boldsymbol{I}(u) \\ \mathbf{0}\le\boldsymbol{j}\le\boldsymbol{I}(v) \\ 0\le x\le o\Leftrightarrow 1 \\ 0\le y\le o\Leftrightarrow 1 \\ \boldsymbol{i},\boldsymbol{j},x,y \text{ integer.} \end{array} \tag{3.1}$$

□

3.1.1 The Processing Unit Conflict Problem Reformulated

We reformulate PUC before analyzing its complexity. To this end, we first derive finite upper bounds for the infinite repetitions, if $I_0(u)=\infty$ or $I_0(v)=\infty$.

If $I_0(u)=\infty$ but $I_0(v)\neq\infty$, then we can derive a finite upper bound on i_0 as follows. From (3.1) we derive

$$\begin{aligned} p_0(u)i_0 &= \sum_{k=0}^{\delta(v)-1} p_k(v)j_k+s(v)+y\Leftrightarrow\sum_{l=1}^{\delta(u)-1} p_l(u)i_l\Leftrightarrow s(u)\Leftrightarrow x \\ &\le \sum_{k=0}^{\delta(v)-1} p_k^+(v)j_k+s(v)+y\Leftrightarrow\sum_{l=1}^{\delta(u)-1} p_l^-(u)i_l\Leftrightarrow s(u)\Leftrightarrow x \\ &\le \sum_{k=0}^{\delta(v)-1} p_k^+(v)I_k(v)+s(v)+o\Leftrightarrow 1\Leftrightarrow\sum_{l=1}^{\delta(u)-1} p_l^-(u)I_l(u)\Leftrightarrow s(u), \end{aligned}$$

where $z^+=\max\{z,0\}$ and $z^-=\min\{z,0\}$. Since $p_0(u)>0$ by assumption, i_0 is bounded from above by

$$b(v,u)=\left\lfloor\frac{\sum_{k=0}^{\delta(v)-1}p_k^+(v)I_k(v)+s(v)+o\Leftrightarrow 1\Leftrightarrow\sum_{l=1}^{\delta(u)-1}p_l^-(u)I_l(u)\Leftrightarrow s(u)}{p_0(u)}\right\rfloor, \tag{3.2}$$

and thus we can replace $I_0(u)$ by $b^+(v,u)$. Note that we take $b^+(v,u)$ since $b(v,u)$ may be negative.

If $I_0(u)\neq\infty$ and $I_0(v)=\infty$, then we can replace $I_0(v)$ by a similar bound $b^+(u,v)$.

If both $I_0(u)=\infty$ and $I_0(v)=\infty$, then we can derive finite upper bounds on i_0 and j_0 as follows. Let $p=\mathrm{lcm}(p_0(u),p_0(v))$. If operation u occupies a processing unit at time c, then it also occupies it at time $c+p$. Similarly, if operation v occupies a processing unit at time c, then it also occupies it at time $c+p$. So if a processing unit conflict occurs at time c, then also a conflict occurs at time $c+p$. This means

that if $\boldsymbol{i}, \boldsymbol{j}, x, y$ satisfy (3.1), then $\boldsymbol{i}+\Delta\boldsymbol{i}, \boldsymbol{j}+\Delta\boldsymbol{j}, x, y$ also satisfy (3.1), where

$$\Delta\boldsymbol{i} = \frac{p}{p_0(u)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \Delta\boldsymbol{j} = \frac{p}{p_0(v)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Furthermore, if $\boldsymbol{i}, \boldsymbol{j}, x, y$ satisfy (3.1) and $i_0 \geq p/p_0(u)$ and $j_0 \geq p/p_0(v)$, then $\boldsymbol{i} \Leftrightarrow \Delta\boldsymbol{i}$, $\boldsymbol{j} \Leftrightarrow \Delta\boldsymbol{j}, x, y$ also satisfy (3.1), i.e., there is a solution with a lower value of i_0 and j_0. From this we can conclude that (3.1) has a solution if and only if it has a solution with

$$i_0 < p/p_0(u) \quad \vee \quad j_0 < p/p_0(v). \tag{3.3}$$

Now, if $j_0 < p/p_0(v)$, then we can derive an upper bound $\tilde{b}(v,u)$ on i_0 given by (3.2), where we replace $I_0(v)$ by $p/p_0(v) \Leftrightarrow 1$. So (3.3) implies

$$i_0 < p/p_0(u) \quad \vee \quad i_0 \leq \tilde{b}(v,u),$$

and thus we can replace $I_0(u)$ by $\max\{p/p_0(u) \Leftrightarrow 1\,,\, \tilde{b}(v,u)\}$. In a similar way we can replace $I_0(v)$ by $\max\{p/p_0(v) \Leftrightarrow 1\,,\, \tilde{b}(u,v)\}$, where $\tilde{b}(u,v)$ is given by (3.2), with $I_0(u)$ replaced by $p/p_0(u) \Leftrightarrow 1$.

The next step in the reformulation of PUC is to replace x and y by iterators with period 1 and bound $o \Leftrightarrow 1$. After that, we combine all iterators into one vector, and thus all iterator bounds into one vector $\boldsymbol{I} \in \mathbb{N}^{\delta}$, with $\delta = \delta(u) + \delta(v) + 2$, and all periods into one vector $\boldsymbol{p} \in \mathbb{Z}^{\delta}$. In this way, we can rewrite PUC into the problem to determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} &= s \\ \mathbf{0} \leq \boldsymbol{i} &\leq \boldsymbol{I} \\ \boldsymbol{i} &\text{ integer.} \end{aligned}$$

A final reformulation is to remove iterators with an upper bound or period equal to 0, and to substitute iterators i_k for which $p_k < 0$ by $I_k \Leftrightarrow i'_k$, in order to obtain positive periods. Now we can see that PUC is equivalent to the following problem.

Definition 3.2 (PUC Reformulated). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^{\delta}$, and an integer s. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} &= s \\ \mathbf{0} \leq \boldsymbol{i} &\leq \boldsymbol{I} \\ \boldsymbol{i} &\text{ integer.} \end{aligned} \tag{3.4}$$

□

In the remainder, we assume s to be positive, since otherwise it is trivial that no solution exists.

Note that the reformulation of PUC can be done in polynomial time, and note that the largest number in the reformulation is polynomially bounded by the largest number in the original formulation.

3.1.2 The Complexity of PUC

In this section we discuss the complexity of PUC. For this, we use the subset sum problem [Garey & Johnson, 1979], which is defined as follows.

Definition 3.3 (Subset Sum (SUB)). Given are a finite set A, with for each $a \in A$ a size $s(a) \in \mathbb{N}_+$, and a positive integer B. Determine whether there is a subset $A' \subset A$ for which

$$\sum_{a \in A'} s(a) = B. \tag{3.5}$$

□

SUB is NP-complete, and it can be solved in pseudo-polynomial time [Garey & Johnson, 1979].

Theorem 3.1. *PUC is NP-complete.*

Proof. First, for a given vector $\boldsymbol{i}$ one can check in polynomial time whether it satisfies (3.4), so PUC $\in$ NP. Next, SUB can be reduced to PUC as follows. Let an instance $\mathcal{I}_{\text{sub}}$ of SUB be given with $A = \{a_0, a_1, ..., a_{n-1}\}$. Now we define an instance $\mathcal{I}_{\text{puc}}$ of PUC according to

- $\delta = n$,
- $I_k = 1$, for $k = 0, ..., \delta \Leftrightarrow 1$,
- $p_k = s(a_k)$, for $k = 0, ..., \delta \Leftrightarrow 1$, and
- $s = B$.

If we take the following relation between a solution A' of $\mathcal{I}_{\text{sub}}$ and a solution $\boldsymbol{i}$ of $\mathcal{I}_{\text{puc}}$

$$i_k = 1 \;\Leftrightarrow\; a_k \in A' \;\wedge\; i_k = 0 \;\Leftrightarrow\; a_k \notin A',$$

then

$$\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = \sum_{a \in A'} s(a),$$

and it follows directly that $\mathcal{I}_{\text{sub}}$ has a solution if and only if $\mathcal{I}_{\text{puc}}$ has one. This concludes the proof. □

Theorem 3.2. *PUC can be solved in pseudo-polynomial time.*

Proof. For this proof, we transform an instance $\mathcal{I}_{\text{puc}}$ of PUC into an instance $\mathcal{I}_{\text{sub}}$ of SUB as follows.

- For each $k = 0, ..., \delta \Leftrightarrow 1$ and each $l = 0, ..., I_k \Leftrightarrow 1$ we add an element a_{kl} to A with size $s(a_{kl}) = p_k$.

- $B = s$.

Note that $|A| = \sum_{k=0}^{\delta-1} I_k$, hence this transformation is pseudo-polynomial. Next, we show that $\mathcal{I}_{\text{sub}}$ has a solution if and only if $\mathcal{I}_{\text{puc}}$ has one.

If $\mathcal{I}_{\text{puc}}$ has a solution $\boldsymbol{i}$, then we can make a solution A' of $\mathcal{I}_{\text{sub}}$ by assigning

$$A' = \{a_{kl} \in A \mid 0 \leq l < i_k\},$$

i.e., we put i_k elements a_{ml} with $m = k$ in A'. Then

$$\sum_{a \in A'} s(a) = \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} = s = B,$$

so A' satisfies (3.5).

Next, if A' is a solution of $\mathcal{I}_{\text{sub}}$, then we can make a solution $\boldsymbol{i}$ of $\mathcal{I}_{\text{puc}}$ by assigning

$$i_k = |\{a_{ml} \in A' \mid m = k\}|,$$

for each $k = 0,\ldots,\delta \Leftrightarrow 1$. Then

$$\boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} = \sum_{a \in A'} s(a) = B = s,$$

hence $\boldsymbol{i}$ satisfies (3.4). So, we can conclude that $\mathcal{I}_{\text{puc}}$ has a solution if and only if $\mathcal{I}_{\text{sub}}$ has one, and thus we can use a pseudo-polynomial time algorithm for SUB to solve PUC. □

In PUC, the question is to determine whether there exists a solution $\boldsymbol{i} \in \mathbb{Z}^{\delta}$ satisfying (3.4). Actually finding such a solution can be done by bisecting the domains of the iterators i_k, for $k = 0,\ldots,\delta \Leftrightarrow 1$. For instance, if there exists a solution, and for a certain iterator i_k we have a domain $\{0,\ldots,5 = I_k\}$, then we split the domain into $\{0,\ldots,2\}$ and $\{3,\ldots,5\}$. So we create two new instances of PUC, the first one by replacing I_k by 2, and the second one by replacing $i_k = 3 + i'_k$ with an upper bound $I'_k = 2$ for i'_k.

If the first instance has a solution, which can be verified by means of an algorithm for PUC, then we continue with it. Otherwise, we continue with the second instance. In this way, we can determine a feasible value for an iterator i_k in $\mathcal{O}(\log I_k)$ steps. A complete solution $\boldsymbol{i}$ can then be found by doing this δ times, resulting in $\mathcal{O}(\delta \log I_{\max})$ invocations of an algorithm for PUC, where $I_{\max}$ is the maximal iterator bound. We thus can conclude that finding a solution for PUC is just as hard as determining whether a solution exists.

3.1.3 Special Cases of PUC

In this section we discuss the complexity of four special cases of PUC, induced by practical situations. The first special case of PUC is the case with divisible periods. An example of this case is given by an operation that is repeated for all pixels in

a video line, for all lines in a field, and for a number of fields, and for which the pixel period divides the line period, and the line period divides the field period, as is the case in the example of Figure 1.2. For some well-known problems, such as bin packing, divisibility leads to polynomial-time algorithms [Coffman, Garey & Johnson, 1987].

The special case of PUC with divisible periods is defined as follows.

Definition 3.4 (PUCDP). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^\delta$, sorted in non-increasing order, with $p_{k+1} | p_k$ for all $k = 0,\ldots,\delta\Leftrightarrow 2$, and a positive integer s. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} &= s \\ \boldsymbol{0} \le \boldsymbol{i} &\le \boldsymbol{I} \\ \boldsymbol{i} &\text{ integer.} \end{aligned}$$

□

Theorem 3.3. *PUCDP can be solved in polynomial time.*

Proof. For the proof, we show that if PUCDP has a solution, then the lexicographically maximal solution $\boldsymbol{i}^*$ satisfies

$$i_k^* = \min\{I_k, \lfloor (s \Leftrightarrow \sum_{l=0}^{k-1} p_l i_l^*) \;/\; p_k \rfloor\}, \tag{3.6}$$

for all $k = 0,\ldots,\delta\Leftrightarrow 1$. So, if a solution exists, then there is a lexicographically maximal one, and by using (3.6) it can be computed in polynomial time. Note that in this formula, the elements i_k^* are computed in order of increasing index k, i.e., in order of non-increasing period.

We show that $\boldsymbol{i}^*$ satisfies (3.6) by contradiction. Let $\boldsymbol{i}^*$ be the lexicographically maximal solution of a given instance of PUCDP, and let $k \in \{0,\ldots,\delta\Leftrightarrow 1\}$ be such that

$$i_k^* \neq \min\{I_k, \lfloor (s \Leftrightarrow \sum_{l=0}^{k-1} p_l i_l^*) \;/\; p_k \rfloor\}.$$

Since i_k^* cannot be larger than this minimum, it must be smaller. So,

$$i_k^* \le I_k \Leftrightarrow 1 \;\;\wedge\;\; i_k^* \le \lfloor (s \Leftrightarrow \sum_{l=0}^{k-1} p_l i_l^*) \;/\; p_k \rfloor \Leftrightarrow 1.$$

Then

$$p_k i_k^* \le s \Leftrightarrow \sum_{l=0}^{k-1} p_l i_l^* \Leftrightarrow p_k$$

and thus

$$s \Leftrightarrow \sum_{l=0}^{k} p_l i_l^* \ge p_k.$$

Since $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}^* = s$, this implies that

$$\sum_{l=k+1}^{\delta-1} p_l i_l^* \geq p_k.$$

This in turn implies that there is an $m \in \mathbb{N}$ with $k+1 \leq m < \delta$, and a $j \in \mathbb{N}$ with $0 \leq j \leq i_m^*$, such that

$$\sum_{l=k+1}^{m-1} p_l i_l^* + p_m j \leq p_k \quad \wedge \quad \sum_{l=k+1}^{m-1} p_l i_l^* + p_m (j+1) > p_k. \tag{3.7}$$

Because of the divisibility of the periods we know that

$$\sum_{l=k+1}^{m-1} p_l i_l^* + p_m j = f p_m \quad \wedge \quad p_k = g p_m,$$

for certain $f, g \in \mathbb{N}$. Together with (3.7), this yields

$$f p_m \leq g p_m < (f+1) p_m,$$

which implies that $f = g$, and consequently we obtain

$$\sum_{l=k+1}^{m-1} p_l i_l^* + p_m j = p_k. \tag{3.8}$$

If we now define a vector $\boldsymbol{i}' \in \mathbb{Z}^{\delta}$ as follows

$$i_l' = \begin{cases} i_l^* & \text{for } l = 0,\ldots,k \Leftrightarrow 1 \\ i_k^* + 1 & \text{for } l = k \\ 0 & \text{for } l = k+1,\ldots,m \Leftrightarrow 1 \\ i_m^* \Leftrightarrow j & \text{for } l = m \\ i_l^* & \text{for } l = m+1,\ldots,\delta \Leftrightarrow 1, \end{cases}$$

then (3.8) implies that $\boldsymbol{p}^{\mathrm{T}}(\boldsymbol{i}' \Leftrightarrow \boldsymbol{i}^*) = 0$, so $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}' = s$. Furthermore, $\boldsymbol{0} \leq \boldsymbol{i}' \leq \boldsymbol{I}$. So, we have found a solution that is lexicographically greater than $\boldsymbol{i}^*$, which contradicts the assumptions. □

The second special case of PUC is the case with a lexicographical execution, which is an often occurring property in practice.

Definition 3.5 (PUCL). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^{\delta}$, and a positive integer s. Furthermore, $\mathrm{lex}(\boldsymbol{I}, \boldsymbol{p}, 1)$ holds. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{gathered} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer.} \end{gathered}$$

□

Theorem 3.4. *PUCL can be solved in polynomial time.*

Proof. We again show by contradiction that if PUCL has a solution, then the lexicographically maximal solution $\boldsymbol{i}^*$ satisfies (3.6), for all $k = 0,\ldots,\delta\Leftrightarrow 1$. So, let $\boldsymbol{i}^*$ be the lexicographically maximal solution of a given instance of PUCL, and let $k \in \{0,\ldots,\delta\Leftrightarrow 1\}$ be such that

$$i_k^* \neq \min\{I_k, \lfloor (s \Leftrightarrow \sum_{l=0}^{k-1} p_l i_l^*) \;/\; p_k \rfloor\}.$$

Then we again have

$$i_k^* \leq I_k \Leftrightarrow 1 \;\;\wedge\;\; i_k^* \leq \lfloor (s \Leftrightarrow \sum_{l=0}^{k-1} p_l i_l^*) \;/\; p_k \rfloor \Leftrightarrow 1,$$

and thus

$$\sum_{l=0}^{k} p_l i_l^* + p_k \leq s.$$

So, if we define a vector $\boldsymbol{i}' \in \mathbb{Z}^{\delta}$ as follows

$$i_l' = \begin{cases} i_l^* & \text{for } l = 0,\ldots,k\Leftrightarrow 1 \\ i_k^* + 1 & \text{for } l = k \\ 0 & \text{for } l = k+1,\ldots,\delta\Leftrightarrow 1, \end{cases}$$

then $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}' \leq s$. Furthermore, $\boldsymbol{0} \leq \boldsymbol{i}' \leq \boldsymbol{I}$ and $\boldsymbol{i}^* <_{\text{lex}} \boldsymbol{i}'$, which implies that $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}^* < \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}' \leq s$. This contradicts the fact that $\boldsymbol{i}^*$ is a solution. □

In the reformulation of PUC, the periods and iterator bounds of both operations have been combined into one new period vector and iterator bound vector. Although both operations may have a lexicographical execution, this does not guarantee that the new iterator bound vector and period vector also give a lexicographical execution. Therefore, we identify the following special case.

Definition 3.6 (PUCLL). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^{\delta}$, and a positive integer s. Furthermore, the vector $\boldsymbol{p} \in \mathbb{N}_+^{\delta}$ can be split into two vectors $\boldsymbol{p}' \in \mathbb{N}_+^{\delta'}$ and $\boldsymbol{p}'' \in \mathbb{N}_+^{\delta''}$, with $\delta' + \delta'' = \delta$, and the vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$ can be split accordingly into $\boldsymbol{I}' \in \mathbb{N}_+^{\delta'}$ and $\boldsymbol{I}'' \in \mathbb{N}_+^{\delta''}$, such that $\text{lex}(\boldsymbol{I}', \boldsymbol{p}', 1)$ and $\text{lex}(\boldsymbol{I}'', \boldsymbol{p}'', 1)$ hold. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{gathered} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer}. \end{gathered}$$

□

Theorem 3.5. *PUCLL is NP-complete.*

Proof. Again, the proof is based on a reduction from the subset sum problem. Let

an instance $\mathcal{I}_{\text{sub}}$ of SUB be given with $A = \{a_0,\ldots,a_{n-1}\}$, and let $S = \sum_{a\in A} s(a)$. Without loss of generality, we may assume $B \leq S$, since otherwise it is trivial that no solution exists. Now we define an instance $\mathcal{I}_{\text{pucll}}$ of PUCLL, by defining the vectors $\boldsymbol{p}'$, $\boldsymbol{p}''$, $\boldsymbol{I}'$, and $\boldsymbol{I}''$, and the integer s as

- $\delta' = \delta'' = n$,
- $I'_k = I''_k = 1$, for $k = 0,\ldots,n\Leftrightarrow 1$,
- $p'_k = 2^{n-k}S$, for $k = 0,\ldots,n\Leftrightarrow 1$,
- $p''_k = 2^{n-k}S + s(a_k)$, for $k = 0,\ldots,n\Leftrightarrow 1$, and
- $s = (2^{n+1}\Leftrightarrow 2)S + B$.

Note that this is an instance of PUCLL, since $\text{lex}(\boldsymbol{I}',\boldsymbol{p}',1)$ holds, as follows from

$$\begin{aligned}\sum_{l=k+1}^{\delta'-1} p'_l I'_l + 1 &= \sum_{l=k+1}^{n-1} 2^{n-l}S + 1\\ &= (2^{n-k}\Leftrightarrow 2)S + 1\\ &\leq p'_k,\end{aligned}$$

for all $k = 0,\ldots,\delta'\Leftrightarrow 1$, and $\text{lex}(\boldsymbol{I}'',\boldsymbol{p}'',1)$ holds, as follows from

$$\begin{aligned}\sum_{l=k+1}^{\delta''-1} p''_l I''_l + 1 &= \sum_{l=k+1}^{n-1} 2^{n-l}S + \sum_{l=k+1}^{n-1} s(a_l) + 1\\ &\leq (2^{n-k}\Leftrightarrow 2)S + S + 1\\ &\leq p''_k,\end{aligned}$$

for all $k = 0,\ldots,\delta''\Leftrightarrow 1$.

Next, if we have a solution of $\mathcal{I}_{\text{pucll}}$, which we denote by the vectors $\boldsymbol{i}' \in \mathbb{Z}^n$ and $\boldsymbol{i}'' \in \mathbb{Z}^n$, then we can show that

$$i'_k + i''_k = 1, \tag{3.9}$$

for all $k = 0,\ldots,n\Leftrightarrow 1$. We do this by induction. To this end, we assume that (3.9) holds for all $k = 0,\ldots,m\Leftrightarrow 1$, with $0 \leq m < n$, and we show that it also holds for $k = m$, i.e., we show that $i'_m + i''_m = 1$. Note that the assumption is trivial for $m = 0$.

If $i'_m + i''_m \neq 1$, then either $i'_m = i''_m = 0$ or $i'_m = i''_m = 1$. However, if $i'_m = i''_m = 0$, then we have

$$\begin{aligned}\boldsymbol{p}'^{\,\mathrm{T}}\boldsymbol{i}' + \boldsymbol{p}''^{\,\mathrm{T}}\boldsymbol{i}'' &= \sum_{l=0}^{n-1} 2^{n-l}Si'_l + \sum_{l=0}^{n-1}(2^{n-l}S + s(a_l))i''_l\\ &= \sum_{l=0}^{n-1} 2^{n-l}S(i'_l + i''_l) + \sum_{l=0}^{n-1} s(a_l)i''_l\end{aligned}$$

$$\begin{aligned}
&= \sum_{l=0}^{m-1} 2^{n-l}S + \sum_{l=m+1}^{n-1} 2^{n-l}S(i'_l + i''_l) + \sum_{l=0}^{n-1} s(a_l)i''_l \\
&\leq (2^{n+1} \Leftrightarrow 2^{n-m+1})S + \sum_{l=m+1}^{n-1} 2^{n-l+1}S + S \\
&\leq (2^{n+1} \Leftrightarrow 2^{n-m+1})S + (2^{n-m+1} \Leftrightarrow 4)S + S \\
&= (2^{n+1} \Leftrightarrow 3)S \\
&< s.
\end{aligned}$$

So, $(\boldsymbol{i}', \boldsymbol{i}'')$ is no solution. Furthermore, if $i'_m = i''_m = 1$, then we have

$$\begin{aligned}
\boldsymbol{p}'^{\mathrm{T}}\boldsymbol{i}' + \boldsymbol{p}''^{\mathrm{T}}\boldsymbol{i}'' &= \sum_{l=0}^{n-1} 2^{n-l}S(i'_l + i''_l) + \sum_{l=0}^{n-1} s(a_l)i''_l \\
&= \sum_{l=0}^{m-1} 2^{n-l}S + 2^{n-m+1}S + \sum_{l=m+1}^{n-1} 2^{n-l}S(i'_l + i''_l) + \sum_{l=0}^{n-1} s(a_l)i''_l \\
&\geq (2^{n+1} \Leftrightarrow 2^{n-m+1})S + 2^{n-m+1}S \\
&= 2^{n+1}S \\
&> s.
\end{aligned}$$

So, again $(\boldsymbol{i}', \boldsymbol{i}'')$ is no solution. The only possibility is that $i'_m + i''_m = 1$, and thus we can conclude that $i'_k + i''_k = 1$, for all $k = 0, \ldots, n \Leftrightarrow 1$. Using this, we derive

$$\begin{aligned}
\boldsymbol{p}'^{\mathrm{T}}\boldsymbol{i}' + \boldsymbol{p}''^{\mathrm{T}}\boldsymbol{i}'' &= \sum_{l=0}^{n-1} 2^{n-l}S(i'_l + i''_l) + \sum_{l=0}^{n-1} s(a_l)i''_l \\
&= \sum_{l=0}^{n-1} 2^{n-l}S + \sum_{l=0}^{n-1} s(a_l)i''_l \\
&= (2^{n+1} \Leftrightarrow 2)S + \sum_{l=0}^{n-1} s(a_l)i''_l \\
&= s \Leftrightarrow B + \sum_{l=0}^{n-1} s(a_l)i''_l,
\end{aligned}$$

so $\boldsymbol{p}'^{\mathrm{T}}\boldsymbol{i}' + \boldsymbol{p}''^{\mathrm{T}}\boldsymbol{i}'' = s$ if and only if

$$\sum_{l=0}^{n-1} s(a_l)i''_l = B.$$

Now consider a solution A' of $\mathcal{I}_{\mathrm{sub}}$ and a solution $(\boldsymbol{i}', \boldsymbol{i}'')$ of $\mathcal{I}_{\mathrm{pucll}}$ related according to

$$(i'_k, i''_k) = (0,1) \;\Leftrightarrow\; a_k \in A' \;\wedge\; (i'_k, i''_k) = (1,0) \;\Leftrightarrow\; a_k \notin A'.$$

Then we can see that $\mathcal{I}_{\text{sub}}$ has a solution if and only if $\mathcal{I}_{\text{pucll}}$ has one. So, we can conclude that PUCLL is NP-complete. □

The fourth special case of PUC is the special case with two periods not equal to 1 and one period equal to 1. An example of this case is given by two one-dimensional periodic operations, for instance in one-dimensional periodic scheduling.

Definition 3.7 (PUC2). Given are three iterator bounds $I_0, I_1, I_2 \in \mathbb{N}_+$, two periods $p_0, p_1 \in \mathbb{N}_+$, with $p_0, p_1 \neq 1$, and a positive integer s. Determine whether there are numbers i_0, i_1, i_2 that satisfy

$$\begin{aligned} p_0 i_0 + p_1 i_1 + i_2 &= s \\ 0 \leq i_0 &\leq I_0 \\ 0 \leq i_1 &\leq I_1 \\ 0 \leq i_2 &\leq I_2 \\ i_0, i_1, i_2 &\text{ integer.} \end{aligned} \tag{3.10}$$

□

Theorem 3.6. *PUC2 can be solved in polynomial time.*

Proof. Without loss of generality, we may assume $p_0 \geq p_1$, and consequently (3.10) may be rewritten as

$$\begin{aligned} p_0 i_0 \Leftrightarrow p_1 i_1' = s \Leftrightarrow p_1 I_1 \Leftrightarrow i_2 \\ 0 \leq i_0 \leq I_0 \\ 0 \leq i_1' \leq I_1 \\ 0 \leq i_2 \leq I_2 \\ i_0, i_1', i_2 \text{ integer,} \end{aligned}$$

where we have substituted i_1 by $I_1 \Leftrightarrow i_1'$. Since the period of i_2 equals 1, this problem is equivalent to determining whether there are numbers i_0 and i_1 that satisfy

$$\begin{aligned} p_0 i_0 \Leftrightarrow p_1 i_1 \in [x, y] \\ 0 \leq i_0 \leq I_0 \\ 0 \leq i_1 \leq I_1 \\ i_0, i_1 \text{ integer,} \end{aligned} \tag{3.11}$$

where $x = s \Leftrightarrow p_1 I_1 \Leftrightarrow I_2$ and $y = s \Leftrightarrow p_1 I_1$. The feasible region of (3.11) is shown in Figure 3.1. If we now have two solutions (i_0', i_1') and (i_0'', i_1'') of (3.11), with $i_0' \leq i_0''$ and $i_1' \geq i_1''$, then (i_0', i_1'') is also a solution, since

$$p_0 i_0' \Leftrightarrow p_1 i_1'' \geq p_0 i_0' \Leftrightarrow p_1 i_1' \geq x$$

and

$$p_0 i_0' \Leftrightarrow p_1 i_1'' \leq p_0 i_0'' \Leftrightarrow p_1 i_1'' \leq y.$$

This is also shown in Figure 3.1. It implies that minimizing i_0 and minimizing i_1 can be done independently, and that if (3.11) has a solution, then we can find one

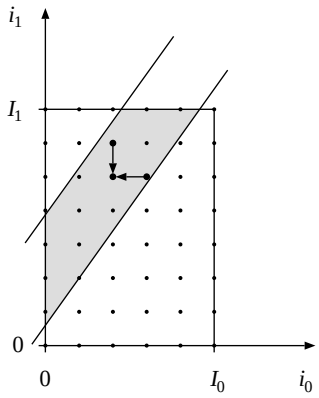


Figure 3.1. The feasible region of (3.11) consists of the integer points in the gray area. The upper diagonal line is given by $p_0 i_0 - p_1 i_1 = x$, and the lower diagonal line is given by $p_0 i_0 - p_1 i_1 = y$. The small arrows show that taking the minimal iterators of two solutions results in a new solution.

by minimizing both i_0 and i_1 subject to

$$\begin{array}{c} p_0 i_0 \Leftrightarrow p_1 i_1 \in [x,y] \\ 0 \leq i_0 \\ 0 \leq i_1 \\ i_0, i_1 \text{ integer.} \end{array} \tag{3.12}$$

In the remainder of the proof, we show that a minimal pair (i_0, i_1) satisfying (3.12) can be found in polynomial time, by means of a recursive procedure. To this end, we make a distinction between three cases: (a) $x \leq 0 \leq y$, (b) $0 < x \leq y$, and (c) $x \leq y < 0$. These cases are illustrated in Figure 3.2.

In case (a), i.e., $x \leq 0 \leq y$, the minimal solution is given by $(i_0, i_1) = (0,0)$, which solves the problem.

In case (b), i.e., $0 < x \leq y$, i_0 has to be at least $\lceil x/p_0 \rceil$. So we can substitute i_0 by $\lceil x/p_0 \rceil + i_0'$ with $i_0' \geq 0$. Then x is replaced by $x \Leftrightarrow p_0 \lceil x/p_0 \rceil \leq 0$, and y is replaced by $y \Leftrightarrow p_0 \lceil x/p_0 \rceil$. Now the instance is reduced to one for which case (a) or (c) applies.

In case (c), i.e., $x \leq y < 0$, we determine $q, r \in \mathbb{N}$ such that

$$p_0 = q p_1 + r \ \ \wedge \ \ 0 \leq r < p_1.$$

Now (3.12) has no solution with $i_1 < q i_0$, since otherwise

$$p_0 i_0 \Leftrightarrow p_1 i_1 > p_0 i_0 \Leftrightarrow p_1 q i_0 = r i_0 \geq 0 > y.$$

Therefore, we can exclude this area from the solution space, which means that a

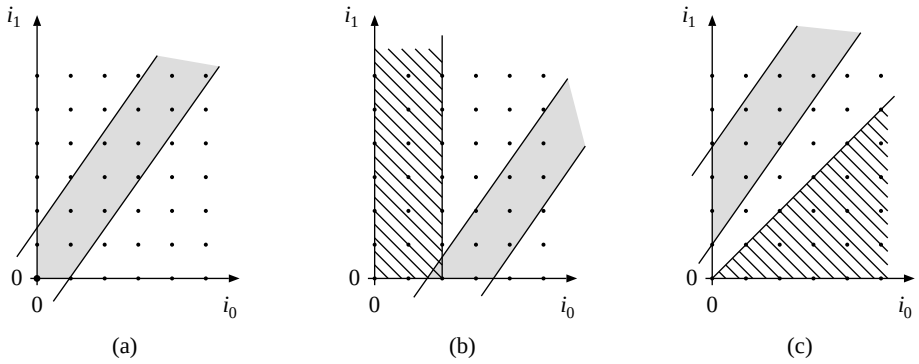


Figure 3.2. The feasible region of (3.12) in three different cases: (a) $x \leq 0 \leq y$, (b) $0 < x \leq y$, and (c) $x \leq y < 0$. In case (a) the origin is the minimal solution. In cases (b) and (c) the shaded areas can be excluded from the iterator space.

solution (i_0, i_1) can be written as

$$(i_0, i_1) = (j_0, qj_0 + j_1),$$

with $j_0, j_1 \geq 0$, and minimizing i_0 and i_1 corresponds to minimizing j_0 and j_1. Hence, we obtain

$$p_0 i_0 \Leftrightarrow p_1 i_1 = p_0 j_0 \Leftrightarrow p_1 (qj_0 + j_1) = rj_0 \Leftrightarrow p_1 j_1,$$

and thus

$$p_0 i_0 \Leftrightarrow p_1 i_1 \in [x, y]$$

if and only if

$$rj_0 \Leftrightarrow p_1 j_1 \in [x, y],$$

which is equivalent to

$$p_1 j_1 \Leftrightarrow rj_0 \in [\Leftrightarrow y, \Leftrightarrow x].$$

So, we have to find a minimal i'_0 and i'_1 such that

$$\begin{gathered} p'_0 i'_0 \Leftrightarrow p'_1 i'_1 \in [x', y'] \\ 0 \leq i'_0 \\ 0 \leq i'_1 \\ i'_0, i'_1 \text{ integer}, \end{gathered}$$

where $p'_0 = p_1$, $p'_1 = r$, $x' = \Leftrightarrow y$, and $y' = \Leftrightarrow x$. In this way we have reduced the instance to an instance with smaller periods.

Following the above procedure, an instance of PUC2 is solved by alternately applying the recipes for cases (b) and (c), until case (a) is achieved or until the smallest period has become zero, in which case the solution is trivial. The number of steps

for this procedure is of the same order as the number of steps that Euclid's algorithm requires for calculating the greatest common divisor of p_0 and p_1. So, PUC2 can be solved in $\mathcal{O}(\log p_0)$ steps, which is polynomially bounded by the size of the instance. □

3.2 Precedence Constraints

In this section we analyze the computational complexity of checking precedence constraints. To this end, we use a complementary formulation.

Definition 3.8 (Precedence Conflict (PC)). Given are an output port p that is connected to an input port q, and their iterator bound vectors $\boldsymbol{I}(p) \in \mathbb{N}_\infty^{\delta(p)}$, $\boldsymbol{I}(q) \in \mathbb{N}_\infty^{\delta(q)}$, for which $I_k(p), I_k(q) \neq \infty$ for all $k > 0$. Furthermore, their period vectors $\boldsymbol{p}(p) \in \mathbb{Z}^{\delta(p)}$, $\boldsymbol{p}(q) \in \mathbb{Z}^{\delta(q)}$, their start times $s(p), s(q) \in \mathbb{Z}$, their index matrices $\boldsymbol{A}(p) \in \mathbb{Z}^{\alpha(p)\times\delta(p)}$, $\boldsymbol{A}(q) \in \mathbb{Z}^{\alpha(q)\times\delta(q)}$, and their index offset vectors $\boldsymbol{b}(p) \in \mathbb{Z}^{\alpha(p)}$, $\boldsymbol{b}(q) \in \mathbb{Z}^{\alpha(q)}$ are given. Determine whether there are vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ that satisfy

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}(p)\,\boldsymbol{i} + s(p) &\geq \boldsymbol{p}^{\mathrm{T}}(q)\,\boldsymbol{j} + s(q) \\ \boldsymbol{A}(p)\,\boldsymbol{i} + \boldsymbol{b}(p) &= \boldsymbol{A}(q)\,\boldsymbol{j} + \boldsymbol{b}(q) \\ \mathbf{0} \leq \boldsymbol{i} &\leq \boldsymbol{I}(p) \\ \mathbf{0} \leq \boldsymbol{j} &\leq \boldsymbol{I}(q) \\ \boldsymbol{i}, \boldsymbol{j} &\text{ integer}. \end{aligned} \tag{3.13}$$

□

3.2.1 The Precedence Conflict Problem Reformulated

We reformulate PC before analyzing its complexity. To this end, we first derive finite upper bounds for the infinite repetitions, if $I_0(p) = \infty$ or $I_0(q) = \infty$, analogously to Section 3.1.1.

If $I_0(p) = \infty$ but $I_0(q) \neq \infty$, then we can derive a finite upper bound on i_0 as follows. From (3.13) we derive

$$\begin{aligned} A_{00}(p)i_0 &= \sum_{k=0}^{\delta(q)-1} A_{0k}(q)\,j_k + b_0(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} A_{0l}(p)\,i_l \Leftrightarrow b_0(p) \\ &\leq \sum_{k=0}^{\delta(q)-1} A_{0k}^{+}(q)\,j_k + b_0(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} A_{0l}^{-}(p)\,i_l \Leftrightarrow b_0(p) \\ &\leq \sum_{k=0}^{\delta(q)-1} A_{0k}^{+}(q)\,I_k(q) + b_0(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} A_{0l}^{-}(p)\,I_l(p) \Leftrightarrow b_0(p). \end{aligned}$$

Since $A_{00}(p) > 0$ by assumption, i_0 is bounded by

$$b(q,p) = \left\lfloor \frac{\sum_{k=0}^{\delta(q)-1} A^+_{0k}(q) I_k(q) + b_0(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} A^-_{0l}(p) I_l(p) \Leftrightarrow b_0(p)}{A_{00}(p)} \right\rfloor, \quad (3.14)$$

and thus we can replace $I_0(p)$ by $b^+(q,p)$.

If $I_0(p) \neq \infty$ and $I_0(q) = \infty$, then we can replace $I_0(q)$ by a similar bound $b^+(p,q)$.

If both $I_0(p) = \infty$ and $I_0(q) = \infty$, then we can derive finite upper bounds on i_0 and j_0 as follows. Let $a = \mathrm{lcm}(A_{00}(p), A_{00}(q))$. Now, if $\boldsymbol{i}$ and $\boldsymbol{j}$ satisfy (3.13), then $\boldsymbol{i} + \Delta\boldsymbol{i}$ and $\boldsymbol{j} + \Delta\boldsymbol{j}$ also satisfy (3.13), where

$$\Delta\boldsymbol{i} = \frac{a}{A_{00}(p)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \Delta\boldsymbol{j} = \frac{a}{A_{00}(q)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

This can be seen as follows. Since by assumption $A_{k0}(p) = 0$, for all $k = 1,\ldots,\alpha(p) \Leftrightarrow 1$, and $A_{k0}(q) = 0$, for all $k = 1,\ldots,\alpha(q) \Leftrightarrow 1$, we see that

$$\boldsymbol{A}(p)\,\Delta\boldsymbol{i} = \begin{bmatrix} a \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \boldsymbol{A}(q)\,\Delta\boldsymbol{j}.$$

Next, since $A_{00}(p)/p_0(p) = A_{00}(q)/p_0(q)$ by assumption, we see that

$$\boldsymbol{p}^{\mathrm{T}}(p)\,\Delta\boldsymbol{i} = \frac{p_0(p)a}{A_{00}(p)} = \frac{p_0(q)a}{A_{00}(q)} = \boldsymbol{p}^{\mathrm{T}}(q)\,\Delta\boldsymbol{j}.$$

So, we can conclude that $\boldsymbol{i} + \Delta\boldsymbol{i}$ and $\boldsymbol{j} + \Delta\boldsymbol{j}$ also satisfy (3.13). Furthermore, if $\boldsymbol{i}$ and $\boldsymbol{j}$ satisfy (3.13) and $i_0 \geq a/A_{00}(p)$ and $j_0 \geq a/A_{00}(q)$, then $\boldsymbol{i} \Leftrightarrow \Delta\boldsymbol{i}$ and $\boldsymbol{j} \Leftrightarrow \Delta\boldsymbol{j}$ also satisfy (3.13), i.e., there is a solution with a lower value of i_0 and j_0. From this we can conclude that (3.13) has a solution if and only if it has a solution with

$$i_0 < a/A_{00}(p) \quad \vee \quad j_0 < a/A_{00}(q). \quad (3.15)$$

Now, if $j_0 < a/A_{00}(q)$, then we can derive an upper bound $\tilde{b}(q,p)$ on i_0 given by (3.14), where we replace $I_0(q)$ by $a/A_{00}(q) \Leftrightarrow 1$. So (3.15) implies

$$i_0 < a/A_{00}(p) \quad \vee \quad i_0 \leq \tilde{b}(q,p),$$

and thus we can replace $I_0(p)$ by $\max\{a/A_{00}(p) \Leftrightarrow 1\,,\, \tilde{b}(q,p)\}$. In a similar way we can replace $I_0(q)$ by $\max\{a/A_{00}(q) \Leftrightarrow 1\,,\, \tilde{b}(p,q)\}$, where $\tilde{b}(p,q)$ is given by (3.14), with $I_0(p)$ replaced by $a/A_{00}(p) \Leftrightarrow 1$.

Next, we combine all iterators into one vector, and thus combine all iterator

bounds into one vector, all periods into one vector, and the two index matrices into one matrix. In this way, we can rewrite PC into the problem to determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} &\geq s \\ \boldsymbol{A}\,\boldsymbol{i} &= \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} &\leq \boldsymbol{I} \\ \boldsymbol{i} &\text{ integer.} \end{aligned}$$

The next reformulation is to remove iterators with an upper bound equal to 0, and to substitute iterators i_k for which column $\boldsymbol{A}_{\cdot k}$ is lexicographically negative by $I_k \Leftrightarrow i'_k$, in order to obtain lexicographically non-negative columns in $\boldsymbol{A}$. Finally, we remove iterators i_k for which column $\boldsymbol{A}_{\cdot k}$ only contains zeros, increasing s by $p_k^+ I_k$. Now we can see that PC is equivalent to the following problem.

Definition 3.9 (PC Reformulated). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{Z}^{\delta}$, an integer s, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$ with lexicographically positive columns, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} &\geq s \\ \boldsymbol{A}\,\boldsymbol{i} &= \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} &\leq \boldsymbol{I} \\ \boldsymbol{i} &\text{ integer.} \end{aligned} \tag{3.16}$$

□

In the remainder, we assume that $\boldsymbol{b}$ is lexicographically positive, since otherwise it is trivial that no solution exists, as is given by the fact that $\boldsymbol{A}\,\boldsymbol{i}$ is a non-negative combination of lexicographically positive columns.

3.2.2 The Complexity of PC

In this section we discuss the complexity of PC. For this, we use the zero-one integer programming problem [Garey & Johnson, 1979], which is defined as follows.

Definition 3.10 (Zero-One Integer Programming (ZOIP)). Given are a matrix $\boldsymbol{M} \in \mathbb{Z}^{m \times n}$, a vector $\boldsymbol{d} \in \mathbb{Z}^{m}$, a vector $\boldsymbol{c} \in \mathbb{Z}^{n}$, and an integer B. Determine whether there is a vector $\boldsymbol{x} \in \{0,1\}^n$ such that $\boldsymbol{M}\,\boldsymbol{x} = \boldsymbol{d}$ and $\boldsymbol{c}^{\mathrm{T}}\boldsymbol{x} \geq B$. □

Without loss of generality, we may assume the columns in $\boldsymbol{M}$ and the vector $\boldsymbol{d}$ to be lexicographically positive. ZOIP is NP-complete in the strong sense [Garey & Johnson, 1979].

Theorem 3.7. *PC is NP-complete in the strong sense.*

Proof. First, for a given vector $\boldsymbol{i}$ one can check in polynomial time whether it satisfies (3.16), so PC $\in$ NP. Next, ZOIP can be reduced to PC. Let an instance $\mathcal{I}_{\text{zoip}}$ of ZOIP be given, then we define an instance $\mathcal{I}_{\text{pc}}$ of PC according to

- $\delta = n$,
- $I_k = 1$, for $k = 0,\ldots,\delta \Leftrightarrow 1$,
- $\boldsymbol{p} = \boldsymbol{c}$,
- $s = B$,
- $\alpha = m$,
- $\boldsymbol{A} = \boldsymbol{M}$, and
- $\boldsymbol{b} = \boldsymbol{d}$.

Now, with the relation $\boldsymbol{x} = \boldsymbol{i}$ we can see that $\mathcal{I}_{\text{zoip}}$ has a solution if and only if $\mathcal{I}_{\text{pc}}$ has one. Since ZOIP is NP-complete in the strong sense, PC is also NP-complete in the strong sense. □

Solving PC is as hard as solving the following optimization variant of it.

Definition 3.11 (Precedence Determination (PD)). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{Z}^\delta$, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$ with lexicographically positive columns, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^\alpha$. Determine the maximum value of $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$, for any $\boldsymbol{i} \in \mathbb{Z}^\delta$ subject to

$$\begin{aligned} \boldsymbol{A}\,\boldsymbol{i} &= \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} &\leq \boldsymbol{I} \\ \boldsymbol{i} &\text{ integer}. \end{aligned}$$

□

The reason that PC is as hard as PD is the fact that $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$ is bounded by $\Leftrightarrow\delta\, p_{\max} I_{\max}$ and $+\delta\, p_{\max} I_{\max}$, where $p_{\max}$ is the maximum of the absolute values of the periods and $I_{\max}$ is the maximum of the iterator bounds. The solution of PD can then be found by bisecting the value range of $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$. For instance, if the value range of $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$ is given by $\{\Leftrightarrow 3,\ldots,14\}$, then we construct a new instance of PC by choosing $s = 5$. If this instance is a yes-instance, then we continue with a value range $\{\Leftrightarrow 3,\ldots,5\}$ for $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$. Otherwise, we continue with a value range $\{6,\ldots,14\}$. In this way, we can determine the solution of PD in $\mathcal{O}(\log(\delta\, p_{\max} I_{\max}))$ steps, resulting in $\mathcal{O}(\log(\delta\, p_{\max} I_{\max}))$ invocations of an algorithm for PC.

By switching from the decision problem PC to the optimization problem PD it is possible to decompose for a given instance the integer linear programming problem into a number of smaller problems.

Similar to PUC, solving PC and PD is as hard as finding a corresponding solution $\boldsymbol{i} \in \mathbb{Z}^\delta$. Such a solution can namely be found by bisecting the domains of the iterators i_k, $k = 0,\ldots,\delta \Leftrightarrow 1$. So, finding a solution $\boldsymbol{i}$ of PC and PD can be found by applying $\mathcal{O}(\delta \log I_{\max})$ times an algorithm for PC and PD, respectively, where $I_{\max}$ is the maximal iterator bound.

3.2.3 Special Cases of PC

In this section we discuss the complexity of five special cases of PC, induced by practical situations. The first special case is the case with a lexicographical index ordering, which is an often occurring property in practice.

Definition 3.12 (PCL). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{Z}^{\delta}$, an integer s, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$ with lexicographically positive columns, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. Furthermore, $\mathrm{lio}(\boldsymbol{I}, \boldsymbol{A})$ holds. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{gathered} \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} \geq s \\ \boldsymbol{A}\,\boldsymbol{i} = \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer.} \end{gathered}$$

□

Theorem 3.8. *PCL can be solved in polynomial time.*

Proof. For the proof, we show that if PCL has a solution, then the lexicographically maximal solution $\boldsymbol{i}^*$ satisfies

$$i_k^* = \min\{I_k, (\boldsymbol{b} \Leftarrow \sum_{l=0}^{k-1} \boldsymbol{A}_{.l} i_l^*) \operatorname{div} \boldsymbol{A}_{.k}\}, \tag{3.17}$$

for all $k = 0, \ldots, \delta \Leftarrow 1$, where we define

$$\boldsymbol{x} \operatorname{div} \boldsymbol{y} = \max\{k \in \mathbb{Z}_{\infty} \mid k\boldsymbol{y} \leq_{\mathrm{lex}} \boldsymbol{x}\},$$

for $\boldsymbol{y} >_{\mathrm{lex}} \boldsymbol{0}$, i.e., the maximal integer number of times a vector $\boldsymbol{y}$ can be subtracted from a vector $\boldsymbol{x}$ that results in a lexicographically non-negative vector. So, if a solution exists, then there is a lexicographically maximal one, and by using (3.17) it can be computed in polynomial time. Similar to PUCL, the elements i_k^* are computed in order of increasing index k, i.e., in order of lexicographically non-increasing column $\boldsymbol{A}_{.k}$.

We show that $\boldsymbol{i}^*$ satisfies (3.17) by contradiction. Let $\boldsymbol{i}^*$ be the lexicographically maximal solution of a given instance of PCL, and let $k \in \{0, \ldots, \delta \Leftarrow 1\}$ be such that

$$i_k^* \neq \min\{I_k, (\boldsymbol{b} \Leftarrow \sum_{l=0}^{k-1} \boldsymbol{A}_{.l} i_l^*) \operatorname{div} \boldsymbol{A}_{.k}\}.$$

Since i_k^* cannot be larger than this minimum, it must be smaller. So,

$$i_k^* \leq I_k \Leftarrow 1 \quad \wedge \quad i_k^* \leq (\boldsymbol{b} \Leftarrow \sum_{l=0}^{k-1} \boldsymbol{A}_{.l}\, i_l^*) \operatorname{div} \boldsymbol{A}_{.k} \Leftarrow 1.$$

Then, using $\boldsymbol{y}(\boldsymbol{x} \operatorname{div} \boldsymbol{y}) \leq_{\text{lex}} \boldsymbol{x}$ for all $\boldsymbol{y} >_{\text{lex}} \boldsymbol{0}$, we have

$$\boldsymbol{A}_{\cdot k}\, i_k^* \leq_{\text{lex}} \boldsymbol{b} \Leftrightarrow \sum_{l=0}^{k-1} \boldsymbol{A}_{\cdot l}\, i_l^* \Leftrightarrow \boldsymbol{A}_{\cdot k}$$

and thus

$$\sum_{l=0}^{k} \boldsymbol{A}_{\cdot l}\, i_l^* + \boldsymbol{A}_{\cdot k} \leq_{\text{lex}} \boldsymbol{b}.$$

So, if we define a vector $\boldsymbol{i}' \in \mathbb{Z}^{\,\delta}$ as follows

$$i'_l = \begin{cases} i_l^* & \text{for } l = 0,\ldots,k \Leftrightarrow 1 \\ i_k^* + 1 & \text{for } l = k \\ 0 & \text{for } l = k+1,\ldots,\delta \Leftrightarrow 1, \end{cases}$$

then $\boldsymbol{A}\,\boldsymbol{i}' \leq_{\text{lex}} \boldsymbol{b}$. Furthermore, $\boldsymbol{0} \leq \boldsymbol{i}' \leq \boldsymbol{I}$ and $\boldsymbol{i}^* <_{\text{lex}} \boldsymbol{i}'$, which implies that $\boldsymbol{A}\,\boldsymbol{i}^* <_{\text{lex}} \boldsymbol{A}\,\boldsymbol{i}' \leq_{\text{lex}} \boldsymbol{b}$. This contradicts the fact that $\boldsymbol{i}^*$ is a solution. □

Analogously to PUCLL, we define the second special case of PC.

Definition 3.13 (PCLL). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{Z}^{\,\delta}$, an integer s, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\,\alpha \times \delta}$ with lexicographically positive columns, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^{\,\alpha}$. Furthermore, the columns of matrix $\boldsymbol{A}$ can be divided among two matrices $\boldsymbol{A}' \in \mathbb{Z}^{\,\alpha \times \delta'}$ and $\boldsymbol{A}'' \in \mathbb{Z}^{\,\alpha \times \delta''}$, with $\delta' + \delta'' = \delta$, and the entries in the vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$ can be divided accordingly among $\boldsymbol{I}' \in \mathbb{N}_+^{\delta'}$ and $\boldsymbol{I}'' \in \mathbb{N}_+^{\delta''}$, such that $\text{lio}(\boldsymbol{I}', \boldsymbol{A}')$ and $\text{lio}(\boldsymbol{I}'', \boldsymbol{A}'')$ hold. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{gathered} \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} \geq s \\ \boldsymbol{A}\,\boldsymbol{i} = \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer}. \end{gathered}$$

□

Theorem 3.9. *PCLL is NP-complete in the strong sense.*

Proof. For this proof, we polynomially transform PC into PCLL. Let an instance $\mathcal{I}_{\text{pc}}$ of PC be given, then we define an instance $\mathcal{I}_{\text{pcll}}$ of PCLL as

$$\boldsymbol{A}_{\text{ll}} = \begin{bmatrix} \mathbf{I} & \mathbf{I} \\ \boldsymbol{A} & \mathbf{O} \end{bmatrix}, \quad \boldsymbol{b}_{\text{ll}} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{b} \end{bmatrix}, \quad \boldsymbol{I}_{\text{ll}} = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{I} \end{bmatrix}, \quad \boldsymbol{p}_{\text{ll}} = \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{0} \end{bmatrix}, \quad s_{\text{ll}} = s.$$

Here, $\mathbf{I}$ is the $\delta \times \delta$ identity matrix, and $\mathbf{O}$ is the $\alpha \times \delta$ zero matrix. Now, for a solution of $\mathcal{I}_{\text{pcll}}$, which we denote by the vectors $\boldsymbol{i}'$ and $\boldsymbol{i}''$, we see that $\boldsymbol{i}' + \boldsymbol{i}'' = \boldsymbol{I}$. Next, we introduce a relation between a solution $\boldsymbol{i}$ of $\mathcal{I}_{\text{pc}}$ and a solution $(\boldsymbol{i}', \boldsymbol{i}'')$ of $\mathcal{I}_{\text{pcll}}$ according to

$$\boldsymbol{i}' = \boldsymbol{i} \;\wedge\; \boldsymbol{i}'' = \boldsymbol{I} \Leftrightarrow \boldsymbol{i}.$$

Then we can see that $\mathcal{I}_{\text{pc}}$ has a solution if and only if $\mathcal{I}_{\text{pcll}}$ has one. So we can conclude that PCLL is NP-complete in the strong sense. □

The third special case of PC is the case with only one index equation, i.e., $\alpha = 1$. Note that this special case also occurs if the original problem can be decomposed into a number of problems with one index equation each.

Definition 3.14 (PC1). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{Z}^\delta$, an integer s, a vector $\boldsymbol{a} \in \mathbb{N}_+^\delta$, and an index offset $b \in \mathbb{N}_+$. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} &\geq s \\ \boldsymbol{a}^{\mathrm{T}}\boldsymbol{i} &= b \\ \mathbf{0} \leq \boldsymbol{i} &\leq \boldsymbol{I} \\ \boldsymbol{i} &\text{ integer.} \end{aligned} \tag{3.18}$$

□

To discuss the complexity of PC1 we use the knapsack problem [Garey & Johnson, 1979], which is defined as follows.

Definition 3.15 (Knapsack (KS)). Given are a finite set U, with for each $u \in U$ a size $s(u) \in \mathbb{N}_+$ and a value $v(u) \in \mathbb{N}_+$, and positive integers B and K. Determine whether there is a subset $U' \subset U$ that satisfies

$$\sum_{u \in U'} s(u) \leq B \quad \wedge \quad \sum_{u \in U'} v(u) \geq K. \tag{3.19}$$

□

KS is NP-complete, and it can be solved in pseudo-polynomial time [Garey & Johnson, 1979].

Theorem 3.10. *PC1 is NP-complete.*

Proof. For the proof, we show that KS can be reduced to PC1. Let an instance $\mathcal{I}_{\text{ks}}$ of KS be given with $U = \{u_0, \ldots, u_{n-1}\}$. Now we define an instance $\mathcal{I}_{\text{pc1}}$ of PC1 according to

- $\delta = n + 1$,
- $I_k = 1$, for $k = 0, \ldots, n \Leftrightarrow 1$, and $I_n = B$,
- $p_k = v(u_k)$, for $k = 0, \ldots, n \Leftrightarrow 1$, and $p_n = 0$,
- $s = K$,
- $a_k = s(u_k)$, for $k = 0, \ldots, n \Leftrightarrow 1$, and $a_n = 1$, and
- $b = B$.

Next, we introduce a relation between a solution U' of $\mathcal{I}_{\text{ks}}$ and a solution $\boldsymbol{i}$ of $\mathcal{I}_{\text{pc1}}$ according to

$$i_k = 1 \;\Leftrightarrow\; u_k \in U' \;\wedge\; i_k = 0 \;\Leftrightarrow\; u_k \notin U',$$

for $k = 0,\ldots,n \Leftrightarrow 1$, and

$$i_n = B \Leftrightarrow \sum_{u \in U'} s(u).$$

Then we have

$$\boldsymbol{a}^{\mathrm{T}}\boldsymbol{i} = \sum_{k=0}^{n-1} a_k i_k + a_n i_n = \sum_{u \in U'} s(u) + B \Leftrightarrow \sum_{u \in U'} s(u) = B = b,$$

and

$$\sum_{u \in U'} s(u) \leq B \;\Leftrightarrow\; i_n \geq 0.$$

Furthermore, we have

$$\sum_{u \in U'} v(u) = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}.$$

So, we can see that $\mathcal{I}_{\text{ks}}$ has a solution if and only if $\mathcal{I}_{\text{pc1}}$ has one. This proves that PC1 is NP-complete. □

Theorem 3.11. *PC1 can be solved in pseudo-polynomial time.*

Proof. For this proof, we transform an instance $\mathcal{I}_{\text{pc1}}$ of PC1 into an instance $\mathcal{I}_{\text{ks}}$ of KS, allowing us to use a pseudo-polynomial time algorithm for KS to solve PC1. The transformation is as follows. First, we define

$$x = \sum_{k=0}^{\delta-1} |p_k| I_k + 1.$$

Consequently, $\Leftrightarrow x < \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} < x$, for all $\boldsymbol{i}$, $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}$. So, without loss of generality, we may assume that $s \geq \Leftrightarrow x$. Next, we define the following.

- For each $k = 0,\ldots,\delta \Leftrightarrow 1$ and each $l = 0,\ldots,I_k \Leftrightarrow 1$ we add an element u_{kl} to U with size $s(u_{kl}) = a_k$ and value $v(u_{kl}) = p_k + 2xa_k$.
- $B = b$.
- $K = s + 2xb$.

Note that $s(u) > 0$, for all $u \in U$, and $K \geq s + 2x > 0$. Furthermore, note that the given transformation can be done in pseudo-polynomial time, and that the largest number in $\mathcal{I}_{\text{pc1}}$ is pseudo-polynomially bounded.

Now, if $\mathcal{I}_{\text{pc1}}$ has a solution $\boldsymbol{i}$, then $\mathcal{I}_{\text{ks}}$ has a solution given by

$$U' = \{u_{kl} \in U \mid 0 \leq l < i_k\},$$

i.e., we put i_k elements u_{ml} with $m = k$ in U'. Then we have

$$\sum_{u \in U'} s(u) = \sum_{k=0}^{\delta-1} a_k i_k = b = B,$$

and

$$\sum_{u \in U'} v(u) = \sum_{k=0}^{\delta-1} (p_k + 2xa_k) i_k \geq s + 2xb = K.$$

So, U' satisfies (3.19).

If $\mathcal{I}_{\mathrm{ks}}$ has a solution U', which can be computed in pseudo-polynomial time, then we construct a solution $\boldsymbol{i}$ of $\mathcal{I}_{\mathrm{pc1}}$ by assigning

$$i_k = |\{u_{ml} \in U' \mid m = k\}|.$$

Then we have

$$\sum_{k=0}^{\delta-1} a_k i_k = \sum_{u \in U'} s(u) \leq B = b,$$

and

$$\begin{aligned} 2x \sum_{k=0}^{\delta-1} a_k i_k &= \sum_{k=0}^{\delta-1} (p_k + 2xa_k) i_k \Leftrightarrow \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} \\ &= \sum_{u \in U'} v(u) \Leftrightarrow \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} \\ &> s + 2xb \Leftrightarrow x \\ &\geq 2x(b \Leftrightarrow 1). \end{aligned}$$

Hence, $b \Leftrightarrow 1 < \boldsymbol{a}^{\mathrm{T}} \boldsymbol{i} \leq b$, i.e., $\boldsymbol{a}^{\mathrm{T}} \boldsymbol{i} = b$. Furthermore, we have

$$\begin{aligned} \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} &= \sum_{k=0}^{\delta-1} (p_k + 2xa_k) i_k \Leftrightarrow \sum_{k=0}^{\delta-1} 2xa_k i_k \\ &= \sum_{u \in U'} v(u) \Leftrightarrow 2xb \\ &\geq s + 2xb \Leftrightarrow 2xb \\ &= s. \end{aligned}$$

So, $\boldsymbol{i}$ satisfies (3.18) and, consequently, we have found a solution of $\mathcal{I}_{\mathrm{pc1}}$ in pseudo-polynomial time. This concludes the proof. □

The following special case of PC is the case with only one index equation, with divisible coefficients. Divisibility of coefficients occurs when, for instance, in a video algorithm a two-dimensional array with indices n_0 and n_1, with $0 \leq n_1 < c$, is substituted by a one-dimensional array with index $n = cn_0 + n_1$.

Definition 3.16 (PC1DC). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{Z}^\delta$, an integer s, a vector $\boldsymbol{a} \in \mathbb{N}_+^\delta$, sorted in non-increasing order, with $a_{k+1} | a_k$ for all $k = 0,\ldots,\delta\Leftrightarrow 2$, and an index offset $b \in \mathbb{N}_+$. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{gathered} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \geq s \\ \boldsymbol{a}^{\mathrm{T}}\boldsymbol{i} = b \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer.} \end{gathered}$$

□

Theorem 3.12. *PC1DC can be solved in polynomial time.*

Proof. In this proof, we determine in polynomial time a vector $\boldsymbol{i}$ for which $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}$ is maximal, subject to

$$\begin{gathered} \boldsymbol{a}^{\mathrm{T}}\boldsymbol{i} = b \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer.} \end{gathered}$$

To this end, we interpret the problem as follows. The number of iterators, δ, is interpreted as a number of block types. Each block type $k = 0,\ldots,\delta\Leftrightarrow 1$ has a size a_k and a profit p_k. Furthermore, of each type k we are given I_k blocks. The question is now to determine for each block type k, a number i_k of blocks to be taken, such that the total size, given by

$$\sum_{k=0}^{\delta-1} a_k i_k,$$

is exactly equal to b, which we call a bag size, and such that the total profit, given by

$$\sum_{k=0}^{\delta-1} p_k i_k,$$

is maximal.

Let the number of different block sizes be given by m, i.e.,

$$m = |\{a_k \mid k = 0,\ldots,\delta\Leftrightarrow 1\}|.$$

Furthermore, let these sizes be denoted by $c_0,\ldots,c_{m-1}$, sorted in decreasing order, i.e.,

$$c_0 > c_1 > \cdots > c_{m-1}.$$

The sizes are divisible, so $c_{j+1} | c_j$, for all $j = 0,\ldots,m\Leftrightarrow 2$.

Next, we re-index the block types, by giving them an index $j \in \{0,\ldots,m\Leftrightarrow 1\}$, denoting that their size equals c_j, and an index l, used to sort the block types in order of non-increasing profit within each group of equal size. So, if we denote the

number of block types with size c_j by n_j, i.e.,

$$n_j = |\{k \mid a_k = c_j\}|,$$

then we re-index the sizes as a_{jl}, for $j = 0,...,m \Leftrightarrow 1$ and $l = 0,...,n_j \Leftrightarrow 1$, and we re-index the profits and numbers of blocks accordingly as p_{jl} and I_{jl}, such that

$$a_{j0} = a_{j1} = \cdots = a_{j,n_j-1} = c_j,$$

and

$$p_{j0} \geq p_{j1} \geq \cdots \geq p_{j,n_j-1},$$

for all $j = 0,...,m \Leftrightarrow 1$.

Now, we can make a distinction between three cases: (a) $c_{m-1} \not| b$, (b) $c_{m-1}|b \wedge m = 1$, and (c) $c_{m-1}|b \wedge m > 1$.

In case (a), the smallest block size c_{m-1} does not divide the bag size b. Since all sizes are multiples of c_{m-1}, this means that no solution exists.

In case (b), we have only one block size, c_0, and it divides the bag size b. This means that we have to take b/c_0 blocks. In order to maximize the total profit, we do this in order of non-increasing profit, i.e., we set

$$i_{0l} = \min\{I_{0l}, b/c_0 \Leftrightarrow \sum_{j=0}^{l-1} i_{0j}\},$$

for $l = 0,...,n_0 \Leftrightarrow 1$. If the total number of blocks, given by $\sum_{l=0}^{n_0-1} I_{0l}$, is at least b/c_0, then this gives a solution with maximal total profit. Otherwise, no solution exists.

In case (c), we have more than one different block size, and the smallest one, c_{m-1}, divides the bag size b. Now, we first determine $q, r \in \mathbb{N}$ such that

$$b = qc_{m-2} + r \quad \wedge \quad 0 \leq r < c_{m-2}.$$

Since the block sizes c_j, for $j < m \Leftrightarrow 1$, are all multiples of c_{m-2}, we know that the part r of the bag has to be filled by blocks of size c_{m-1}. So, to start with, we need to take r/c_{m-1} blocks of size c_{m-1}. Again, we do this in order of non-increasing profit, i.e., we set

$$i_{m-1,l} = \min\{I_{m-1,l}, r/c_{m-1} \Leftrightarrow \sum_{j=0}^{l-1} i_{m-1,j}\},$$

for $l = 0,...,n_{m-1} \Leftrightarrow 1$.

Next, if we use the remaining blocks with size c_{m-1}, then we have to do that in groups of c_{m-2}/c_{m-1} blocks. Furthermore, if we use them, we must do that in order of non-increasing profit, in order to maximize the total profit. Therefore, we line up the remaining blocks of size c_{m-1} in order of non-increasing profit, and then we replace groups of c_{m-2}/c_{m-1} blocks by blocks with size c_{m-2}. For each group of c_{m-2}/c_{m-1} blocks, the new profit p is given by the sum of the profits of the con-

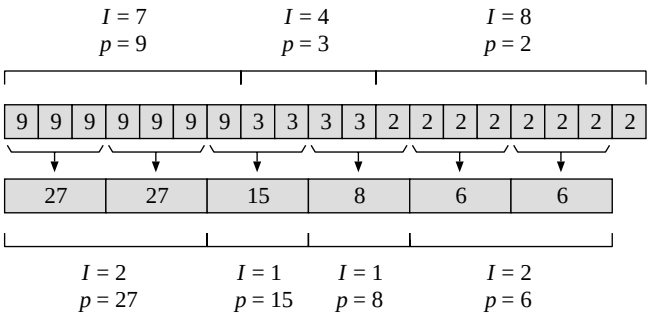


Figure 3.3. Grouping remaining blocks with size c_{m-1} optimally in groups of c_{m-2}/c_{m-1} blocks. In this example, $c_{m-2} = 3c_{m-1}$. One block is wasted. The widths of the boxes denote the sizes of the blocks. The numbers in the boxes denote the profits.

tained blocks. Figure 3.3 shows an example of this grouping. After the grouping, the problem instance is reduced to one with $m \Leftrightarrow 1$ different sizes.

Following the above procedure, an instance of PC1DC is solved by repeatedly applying the recipe for case (c), until case (a) or (b) is achieved. To show that this algorithm has a polynomial time complexity, we first introduce the following. The number of block types with size c_j is given by n_j, $j = 0,\ldots,m \Leftrightarrow 1$. In this subset of types, we identify o_j types l of which exactly one block is given, i.e., $I_{jl} = 1$, and u_j types with no such restriction.

In the grouping phase, remaining blocks with size c_{m-1} are grouped into blocks with size c_{m-2}. The u_{m-1} block types of which an unrestricted number is given, then result in at most u_{m-1} new block types with unrestricted numbers. At the transition from one block type to another, some blocks of one type may be grouped together with blocks of other types. Of each of the u_{m-1} unrestricted block types, however, at most $2(c_{m-2}/c_{m-1} \Leftrightarrow 1)$ blocks are grouped together with blocks of other types. Furthermore, there are o_{m-1} block types of which exactly one block is given, which are also grouped together with blocks of other types. All these combinations of different block types can then be replaced by at most

$$2u_{m-1} + \tfrac{1}{2}o_{m-1}$$

new block types with exactly one block each.

So, in the new subset of n'_{m-2} block types with size c_{m-2}, we can identify o'_{m-2} types of which exactly one block is given, and u'_{m-2} types with no such restriction, where

$$\begin{aligned} u'_{m-2} &\leq u_{m-2} + u_{m-1} \\ o'_{m-2} &\leq o_{m-2} + 2u_{m-1} + \tfrac{1}{2}o_{m-1}. \end{aligned}$$

In general, taking $u'_{m-1} = u_{m-1}$ and $o'_{m-1} = o_{m-1}$, we have

$$\begin{aligned} u'_j &\leq u_j + u'_{j+1} \\ o'_j &\leq o_j + 2u'_{j+1} + \tfrac{1}{2}o'_{j+1}, \end{aligned}$$

for all $j = 0,\ldots,m\Leftrightarrow 2$. If we apply these inequalities recursively, for $j = m\Leftrightarrow 2,\ldots,0$, we obtain

$$\begin{aligned} u'_j &\leq \sum_{t=j}^{m-1} u_t \\ o'_j &\leq \sum_{t=j}^{m-1} o_t + 4\sum_{t=j+1}^{m-1} u_t, \end{aligned}$$

for all $j = 0,\ldots,m\Leftrightarrow 1$. Since the total number of block types is given by δ, we can conclude that $u'_j = \mathcal{O}(\delta)$ and $o'_j = \mathcal{O}(\delta)$, for all $j = 0,\ldots,m\Leftrightarrow 1$.

Now, in each iteration of the recursive procedure, sorting the block types with size c_{m-1} in order of non-increasing profit, and taking blocks of these types to fill up the part r of the bag, takes $\mathcal{O}(\delta\log\delta) + \mathcal{O}(\delta) = \mathcal{O}(\delta\log\delta)$ steps. After that, the grouping takes $\mathcal{O}(\delta)$ steps. So, the total amount of steps in each iteration is $\mathcal{O}(\delta\log\delta)$. The number of iterations is given by m, so the recursive procedure takes $\mathcal{O}(m\delta\log\delta) = \mathcal{O}(\delta^2\log\delta)$ steps, which is polynomially bounded in the size of the instance. □

Example 3.1. For an example of the above procedure, consider an instance of PC1DC given by

$$\boldsymbol{a} = \begin{bmatrix} 12 \\ 4 \\ 2 \\ 2 \end{bmatrix}, \quad \boldsymbol{p} = \begin{bmatrix} 65 \\ 20 \\ 20 \\ 11 \end{bmatrix}, \quad \boldsymbol{I} = \begin{bmatrix} 10 \\ 15 \\ 2 \\ 18 \end{bmatrix}, \quad b = 94.$$

The smallest size is 2, and $94 = 23\cdot 4 + 1\cdot 2$. So, we have to take one block with size 2, which is optimally done by taking a block with profit 20. After that, we group the remaining blocks with size 2 into blocks with size 4. This yields one block with profit 31 and eight blocks with profit 22, as is shown in Figure 3.4. After this, we are left with a problem instance given by

$$\boldsymbol{a}' = \begin{bmatrix} 12 \\ 4 \\ 4 \\ 4 \end{bmatrix}, \quad \boldsymbol{p}' = \begin{bmatrix} 65 \\ 31 \\ 22 \\ 20 \end{bmatrix}, \quad \boldsymbol{I}' = \begin{bmatrix} 10 \\ 1 \\ 8 \\ 15 \end{bmatrix}, \quad b' = 92.$$

Now, $92 = 7\cdot 12 + 2\cdot 4$. So, we have to take two blocks with size 4. This is optimally done by taking one block with profit 31 and one block with profit 22. After that, we

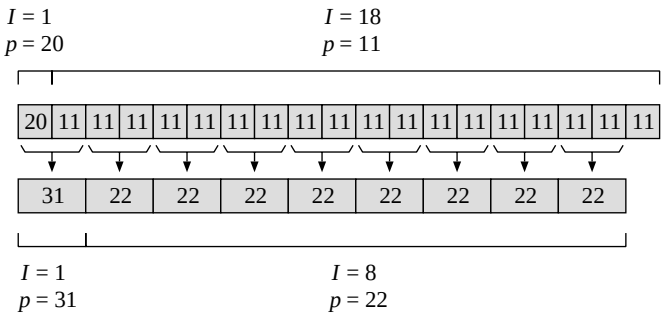


Figure 3.4. Grouping remaining blocks with size 2 optimally in groups of two, giving blocks with size 4. One block is wasted.

group the remaining blocks with size 4 into blocks with size 12. This yields two

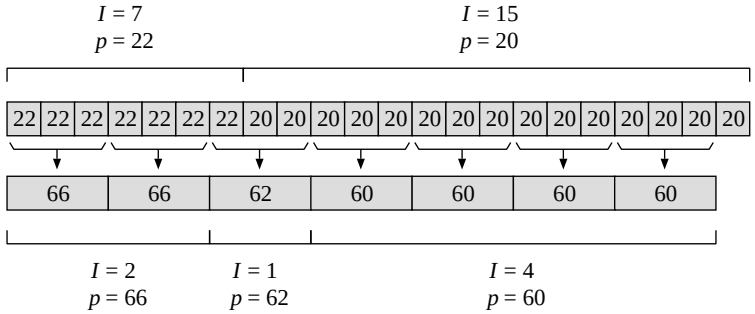


Figure 3.5. Grouping remaining blocks with size 4 optimally in groups of three, giving blocks with size 12. One block is wasted.

blocks with profit 66, one block with profit 62, and four blocks with profit 60, as shown in Figure 3.5. After this, we are left with a problem instance given by

$$\boldsymbol{a}'' = \begin{bmatrix} 12 \\ 12 \\ 12 \\ 12 \end{bmatrix}, \quad \boldsymbol{p}'' = \begin{bmatrix} 66 \\ 65 \\ 62 \\ 60 \end{bmatrix}, \quad \boldsymbol{I}'' = \begin{bmatrix} 2 \\ 10 \\ 1 \\ 4 \end{bmatrix}, \quad b'' = 84.$$

Now, $84 = 7 \cdot 12$. So, we have to take seven blocks. This is optimally done by taking two blocks with profit 66 and five blocks with profit 65.

Now we have solved the instance, yielding a maximal total profit of $1 \cdot 20 + 1 \cdot 31 + 1 \cdot 22 + 2 \cdot 66 + 5 \cdot 65 = 530$. So, there is a precedence conflict if and only if $s \leq 530$. □

From Theorem 3.12 the following corollary is derived.

Corollary 3.1. *Knapsack with divisible item sizes can be solved in polynomial time.* □

This result is obtained by applying the same transformation as described in the proof of Theorem 3.10, to transform instances of knapsack with divisible item sizes into instances of PC1DC. In a similar way we can derive that also a generalization of knapsack, in which of each item u a (bounded) number is given, with divisible item sizes can be solved in polynomial time.

Generalizing PC1DC slightly, we obtain the following special case of PC, in which we have two index equations, i.e., $\alpha = 2$, with divisible coefficients in each of them.

Definition 3.17 (PC2DC). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{Z}^{\delta}$, an integer s, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$ with lexicographically positive columns, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. Furthermore, $\alpha = 2$ and for both rows $k = 0, 1$ we have that

$$A_{kj} | A_{kj'} \quad \vee \quad A_{kj'} | A_{kj},$$

for all $j, j' \in \{0, \ldots, \delta \Leftrightarrow 1\}$. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{gathered} \boldsymbol{p}^{\mathrm{T}} \boldsymbol{i} \geq s \\ \boldsymbol{A}\, \boldsymbol{i} = \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer.} \end{gathered}$$

□

Conjecture 3.1. *PC2DC is NP-complete.* □

3.3 Multidimensional Periodic Scheduling

In this section we discuss the complexity of the multidimensional periodic scheduling problem, as introduced in Section 2.5. In its decision variant, denoted by MPSD, we are given a signal flow graph G, a set of resource types T^*, lower and upper bounds on the period vectors and start times, and an integer K. The question is to find a feasible schedule σ for which the area cost $f(\sigma) \leq K$. To discuss the complexity we also introduce the complementary problem co-MPSD of MPSD, in which the question is to determine whether *no* feasible schedule σ exists with $f(\sigma) \leq K$. The complexity results in this section concern NP-hardness, i.e., the membership of NP is omitted.

Theorem 3.13. *Co-MPSD is NP-hard.*

Proof. For this proof, we reduce PUCLL to co-MPSD. Let an instance $\mathcal{I}_{\text{pucll}}$ of PUCLL be given with iterator bound vectors $\boldsymbol{I}' \in \mathbb{N}_+^{\delta'}$ and $\boldsymbol{I}'' \in \mathbb{N}_+^{\delta''}$, period vectors

$\boldsymbol{p}' \in \mathbb{N}_+^{\delta'}$ and $\boldsymbol{p}'' \in \mathbb{N}_+^{\delta''}$, and positive integer s. We now construct the following instance $\mathcal{I}_{\text{compsd}}$ of co-MPSD. For the operation type set we choose

- $T = \{t\}$, there is only one operation type,
- $\tilde{I}(t) = \tilde{O}(t) = \emptyset$, there are no operation input and output ports,
- the occupation time $o(t) = 1$, and
- the area cost $a(t) = 1$.

For the signal flow graph G we choose

- the set of operations $V = \{u, v\}$,
- operation types $t(u) = t(v) = t$, and
- the iterator bound vectors $\boldsymbol{I}(u) = \boldsymbol{I}'$ and $\boldsymbol{I}(v) = \boldsymbol{I}''$.

In this graph, the set O of output ports and the set I of input ports are both empty. So, the set E of edges is also empty. Next, we fix the period vectors and start times by choosing

- $\underline{\boldsymbol{p}}(u) = \overline{\boldsymbol{p}}(u) = \boldsymbol{p}'$,
- $\underline{\boldsymbol{p}}(v) = \overline{\boldsymbol{p}}(v) = \Leftrightarrow\boldsymbol{p}''$,
- $\underline{s}(u) = \overline{s}(u) = 0$, and
- $\underline{s}(v) = \overline{s}(v) = s$.

Finally, we choose $K = 1$.

Since $\text{lex}(\boldsymbol{I}', \boldsymbol{p}', 1)$ holds, there is no processing unit conflict between two different executions of operation u. So, all executions of u can be performed on one processing unit. Similarly, there is no processing unit conflict between two different executions of v, which implies that all executions of v can also be performed on one processing unit. There remains the question to determine whether u and v can be executed on the same processing unit.

The set of times at which operation u occupies a processing unit is given by

$$\{\boldsymbol{p}'^{\,\mathrm{T}}\boldsymbol{i}' \mid \boldsymbol{i}' \in \mathbb{Z}^{\delta'} \wedge \mathbf{0} \leq \boldsymbol{i}' \leq \boldsymbol{I}'\},$$

and the set of times at which operation v occupies a processing unit is given by

$$\{s \Leftrightarrow \boldsymbol{p}''^{\,\mathrm{T}}\boldsymbol{i}'' \mid \boldsymbol{i}'' \in \mathbb{Z}^{\delta''} \wedge \mathbf{0} \leq \boldsymbol{i}'' \leq \boldsymbol{I}''\}.$$

So, operation u and v occupy a processing unit at the same time if and only if $\boldsymbol{p}'^{\,\mathrm{T}}\boldsymbol{i}' = s \Leftrightarrow \boldsymbol{p}''^{\,\mathrm{T}}\boldsymbol{i}''$ for certain $\boldsymbol{i}'$ and $\boldsymbol{i}''$. This means that u and v must be assigned to different processing units, resulting in a cost $f(\sigma) = 2 > K$, if and only if $\mathcal{I}_{\text{pucll}}$ has a solution. So, $\mathcal{I}_{\text{pucll}}$ is a yes-instance if and only if $\mathcal{I}_{\text{compsd}}$ is a yes-instance. This completes the proof. □

Note that it is not straightforward to show whether or not co-MPSD $\in$ NP. For instance, it is not straightforward to compute the storage and access costs in polynomial time.

The previous complexity theorem is based on the complexity of checking processing unit constraints. If, however, we restrict ourselves to instances of co-MPSD for which the processing unit constraints can be checked in polynomial time, the problem remains NP-hard. This is shown by the following theorem.

Theorem 3.14. *Co-MPSD is NP-hard in the strong sense.*

Proof. For this proof, we reduce PC to co-MPSD. Let an instance $\mathcal{I}_{\text{pc}}$ of PC be given with an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{Z}^\delta$, an integer s, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$, with lexicographically positive columns, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^\alpha$. We now construct the following instance $\mathcal{I}_{\text{compsd}}$ of co-MPSD. For the operation type set we choose

- $T = \{t_1, t_2\}$, there are two operation types,
- $\tilde{I}(t_1) = \emptyset$ and $\tilde{O}(t_1) = \{1\}$, type t_1 only has an operation output port,
- $\tilde{I}(t_2) = \{1\}$ and $\tilde{O}(t_2) = \emptyset$, type t_2 only has an operation input port,
- the relative transfer times $r(t_1, 1) = r(t_2, 1) = 0$,
- the occupation times $o(t_1) = o(t_2) = 0$, and
- the area costs $a(t_1) = a(t_2) = 0$.

The occupation times are zero, which implies that the processing unit constraints are always met. For the signal flow graph G we choose

- the set of operations $V = \{u, v\}$,
- operation types $t(u) = t_1$ and $t(v) = t_2$,
- the iterator bound vectors $\boldsymbol{I}(u) = [\,]$ and $\boldsymbol{I}(v) = \boldsymbol{I}$,
- the set of edges $E = \{(p, q)\}$, with output port $p = (u, 1)$ and input port $q = (v, 1)$,
- the index matrices $\boldsymbol{A}(p) = [\,]$ and $\boldsymbol{A}(q) = \boldsymbol{A}$, and
- the index offset vectors $\boldsymbol{b}(p) = \boldsymbol{b}$ and $\boldsymbol{b}(q) = \boldsymbol{0}$.

Note that we have single productions, since u is executed only once. Next, we fix the period vectors and start times by choosing

- $\underline{\boldsymbol{p}}(u) = \overline{\boldsymbol{p}}(u) = [\,]$,
- $\underline{\boldsymbol{p}}(v) = \overline{\boldsymbol{p}}(v) = \Leftrightarrow \boldsymbol{p}$,
- $\underline{s}(u) = \overline{s}(u) = \Leftrightarrow s$, and
- $\underline{s}(v) = \overline{s}(v) = 0$.

Finally, we choose area costs $a(t_v) = a(t_a) = 0$ and $K = 0$.

If $\mathcal{I}_{pc}$ has a solution $\boldsymbol{i}$, then execution $\boldsymbol{i}$ of input port q uses data produced by execution $[\,]$ of output port p, since

$$\boldsymbol{n}(p,[\,]) = \boldsymbol{A}(p)\,[\,] + \boldsymbol{b}(p) = \boldsymbol{b} = \boldsymbol{A}\,\boldsymbol{i} = \boldsymbol{A}(q)\,\boldsymbol{i} + \boldsymbol{b}(q) = \boldsymbol{n}(q,\boldsymbol{i}).$$

However, we also have

$$c(p,[\,]) = \boldsymbol{p}^{\mathrm{T}}(p)\,[\,] + s(p) = \Leftrightarrow s \geq \Leftrightarrow \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = \boldsymbol{p}^{\mathrm{T}}(q)\,\boldsymbol{i} + s(q) = c(q,\boldsymbol{i}).$$

So, the precedence constraints are not met, and thus no feasible schedule exists.

Next, if $\mathcal{I}_{pc}$ has no solution, then for all vectors $\boldsymbol{i} \in \mathbb{Z}^{\delta}$ with $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}$ holds

$$\boldsymbol{A}\,\boldsymbol{i} = \boldsymbol{b} \;\Rightarrow\; \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} < s,$$

which implies that the precedence constraints are met. Then, a feasible schedule exists, with cost $0 \leq K$. This completes the proof. □

The previous two theorems on the complexity of multidimensional periodic scheduling are based on its sub-problems. If, however, multidimensional periodic scheduling is restricted to instances for which these sub-problems are well solvable, the problem remains NP-hard, as can be seen by the following theorem. For this, we use the strictly periodic single processor scheduling problem [Korst, 1992], which is defined as follows.

Definition 3.18 (Strictly Periodic Single Processor Scheduling (SPSPS)).
Given are a finite set U of operations, and for each operation $u \in U$ a period $q(u) \in \mathbb{N}_+$ and an execution time $e(u) \in \mathbb{N}_+$, with $e(u) \leq q(u)$. Determine whether there is a time assignment $s : U \to \mathbb{Z}$ for which

$$[s(u) + kq(u), s(u) + kq(u) + e(u)) \;\cap\; [s(v) + lq(v), s(v) + lq(v) + e(v)) = \emptyset,$$

for all $u, v \in U$, $u \neq v$, and all $k, l \in \mathbb{Z}$. □

SPSPS is NP-complete in the strong sense [Korst, 1992].

Theorem 3.15. *MPSD is NP-hard in the strong sense.*

Proof. For this proof, we reduce SPSPS to MPSD. Let an instance $\mathcal{I}_{spsps}$ of SPSPS be given, with $U = \{u_0, \ldots, u_{n-1}\}$. We now construct the following instance $\mathcal{I}_{mpsd}$ of MPSD. For the operation type set we choose

- $T = \{t\}$, there is only one operation type,
- $\tilde{I}(t) = \tilde{O}(t) = \emptyset$, there are no operation input and output ports,
- the occupation time $o(t) = 1$, and
- the area cost $a(t) = 1$.

For the signal flow graph G we choose

- the set of operations $V = \{v_0, \ldots, v_{n-1}\}$,

- operation type $t(v) = t$, for all $v \in V$, and
- the iterator bound vectors

$$\boldsymbol{I}(v_k) = \begin{bmatrix} \infty \\ e(u_k) \Leftrightarrow 1 \end{bmatrix},$$

  for all $k = 0,\ldots,n \Leftrightarrow 1$.

In this graph, the set of output ports $O = \emptyset$ and the set of input ports $I = \emptyset$. So, also the set of edges $E = \emptyset$. Next, we fix the period vectors by choosing

$$\underline{\boldsymbol{p}}(v_k) = \overline{\boldsymbol{p}}(v_k) = \begin{bmatrix} q(u_k) \\ 1 \end{bmatrix},$$

for all $k = 0,\ldots,n \Leftrightarrow 1$. Finally, we choose the start time bounds $\underline{s}(v) = \Leftrightarrow\infty$ and $\overline{s}(v) = +\infty$, for all $v \in V$, and we choose $K = 1$. Figure 3.6 shows an example of the correspondence between an operation $u \in U$ and an operation $v \in V$.

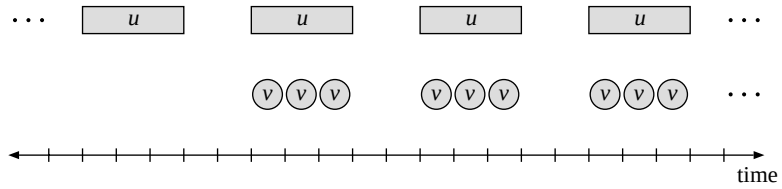


Figure 3.6. The correspondence between an operation $u \in U$ with period 5 and execution time 3, and an operation $v \in V$ with iterator bound vector $[\infty\ 2]^{\mathrm{T}}$, period vector $[5\ 1]^{\mathrm{T}}$, and occupation time 1. The operation u is executed infinitely to the left and to the right, whereas v is only executed infinitely to the right.

Next, we introduce a relation between a time assignment in $\mathcal{I}_{\mathrm{spsps}}$ and a time assignment in $\mathcal{I}_{\mathrm{mpsd}}$ according to

$$s(v_k) = s(u_k),$$

for all $k = 0,\ldots,n \Leftrightarrow 1$. Then, $\mathcal{I}_{\mathrm{spsps}}$ has a solution if and only if all operations in V can be scheduled on one processing unit, i.e., $\mathcal{I}_{\mathrm{mpsd}}$ has a solution with cost $f(\sigma) \leq 1 = K$. This completes the proof. □

4

Integer Linear Programming

In the previous chapter, the processing unit conflict problem and the precedence conflict problem are formulated as integer linear programming (ILP) problems. Since the number of variables in these problems only depends on the number of repetition dimensions of two operations and the number of array dimensions of one multidimensional array, and since these numbers are limited in practice, we opt for solving these ILP problems to optimality. In this chapter, we discuss the method that we use for this.

First, we briefly discuss general ILP solution methods, in Section 4.1. In Section 4.2 we discuss a primal all-integer ILP method that we have chosen to apply to the mentioned problems. Next, in Section 4.3 we extend the method to handle multiple cost functions simultaneously. Finally, in Section 4.4 we discuss a way to improve the efficiency of the method.

4.1 General ILP Solution Methods

Two well-known methods to solve ILP problems are *branch-and-bound* and *cutting plane methods*. Without loss of generality, we may restrict ourselves in this chapter to maximization problems.

Branch-and-bound is a general method to solve combinatorial optimization problems. It is based on iteratively partitioning the set of feasible solutions into

a number of smaller subsets, which is called *branching*. Next, for each generated subset an upper bound on the cost of any of its solutions is computed, which is called *bounding*. If this upper bound is lower than or equal to the cost of the best solution found so far, then the corresponding subset cannot possibly contain a better solution. In that case the subset can be discarded, which is called *pruning*. In the case that we find a feasible solution in a subset with cost equal to the subset's upper bound, then this solution is the optimum of the subset. If in addition the subset is not discarded by pruning, then this solution is also the new best solution found so far.

Commonly, branch-and-bound for an ILP problem is based on its *LP relaxation*, in which the integrality constraints are omitted. An LP relaxation can be solved in polynomial time [Khachiyan, 1979; Karmarkar, 1984], and it results in an upper bound on the cost of the solutions of the corresponding ILP problem, that can be used for bounding. If the obtained solution of the LP-relaxation is an integer solution, then it is an optimal solution of the ILP problem. Otherwise, we select a variable x with a fractional value a in the LP solution, and we branch by splitting the ILP problem into two sub-problems: one with the additional constraint $x \leq \lfloor a \rfloor$ and one with the additional constraint $x \geq \lceil a \rceil$. For more information on branch-and-bound the reader is referred to Lawler & Wood [1966], Beale [1979], Garfinkel [1979], and Papadimitriou & Steiglitz [1982].

Cutting plane methods are also based on LP-relaxations. In a cutting plane algorithm an LP-relaxation is solved a number of times, while each time an extra linear constraint is added, until the obtained solution is integer. In each iteration, the added constraint is such that it excludes the current fractional solution from the solution space of the LP-relaxation, but it does not exclude any integer solution. These constraints are called *cutting planes*. Examples of cutting planes are given by the *Gomory cuts* [Gomory, 1958], which are generally applicable and which can be used in such a way that an integer solution is obtained in a finite number of steps. For more information on cutting plane methods the reader is referred to Schrijver [1986] and Nemhauser & Wolsey [1988].

In order to improve the efficiency of a branch-and-bound algorithm, cutting planes can be added to the LP-relaxations in the solution tree, in order to strengthen the calculated upper bounds. To this end, generally applicable cutting planes such as the Gomory cuts can be added, but also problem-specific cutting planes can be added. In this way, a hybrid algorithm is created, which is called *branch-and-cut*. For more information on (problem specific) cutting planes, also called *valid inequalities*, the reader is again referred to the area of polyhedral combinatorics [Schrijver, 1986; Nemhauser & Wolsey, 1988].

The main disadvantage of the above methods for application to the processing unit conflict problem (PUC) and the precedence conflict problem (PC) is that they are based on LP-relaxations, which yield fractional solutions. In a straightforward

computer implementation this will cause rounding errors, especially since the ILP matrices for PUC and PC are very ill-conditioned. Rounding errors are, however, inadmissible for these problems, since any error in the result may lead to a wrong conclusion about processing unit conflicts or precedence conflicts. This is in contrast to many other ILP problems, in which a fractional solution can easily be rounded to a feasible integer solution. There exists a class of methods, called all-integer ILP methods, that maintain integrality throughout their execution, and consequently do not lead to rounding errors.

Since standard ILP solvers do not support all-integer ILP methods, we choose to use an own implementation of it. An additional advantage of the use of an own implementation is that it is open for extensions and for tailoring towards specific problems. It facilitates the use of multiple cost functions, as shown in Section 4.3, and the extension to parametric ILP, as shown in Section 8.2.

4.2 Primal All-Integer ILP Methods

Shortly after the introduction of integer linear programming algorithms, Gomory [1963] introduced an all-integer method, i.e., if the coefficients in the original matrix are integer, all coefficients remain integer throughout the calculation. Whereas Gomory's method is a dual method, a few years later a primal all-integer ILP method was presented by Young [1968] and Glover [1968].

In this section we discuss the simplified primal algorithm of Glover [1968], and one set of his pivoting choice rules that guarantee convergence in finite time. We first present the notation used, next we discuss the primal simplex method for solving ordinary linear programming problems, and finally we present the all-integer ILP method.

4.2.1 Description of the Problem

Consider the linear programming problem that, for a matrix $\boldsymbol{M} \in \mathbb{R}^{(m+1)\times(n+1)}$, is given by

$$\begin{array}{ll} \text{maximize} & x_0 \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}\,\boldsymbol{t} \\ & x_i \geq 0 \qquad (i \geq 1), \end{array} \tag{4.1}$$

where we maximize over all real vectors

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}, \quad \boldsymbol{t} = \begin{bmatrix} 1 \\ \Leftrightarrow t_1 \\ \vdots \\ \Leftrightarrow t_n \end{bmatrix}.$$

Here, $\mathbb{R}$ is the set of real numbers. The general row equation of $\boldsymbol{x} = \boldsymbol{M}\,\boldsymbol{t}$ is given by

$$x_i = M_{i0} + \sum_{j=1}^{n} M_{ij}(\ominus t_j), \tag{4.2}$$

for $i = 0,\ldots,m$, and we assume that the last n equations have the form

$$x_{m-n+j} = \ominus(\ominus t_j),$$

for $j = 1,\ldots,n$. The variables $x_0,\ldots,x_m$ are the *decision variables* of the linear programming problem, which are expressed linearly in the *non-basic* variables $t_1,\ldots,t_n$. The latter variables also have to be non-negative.

The matrix $\boldsymbol{M}$ is called *primal feasible* if $M_{i0} \geq 0$, for all $i = 1,\ldots,m$, and *dual feasible* if $M_{0j} \geq 0$, for all $j = 1,\ldots,n$. As is well known, an optimal solution of the linear programming problem of (4.1) is given by $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ when $\boldsymbol{M}$ is both primal and dual feasible.

If we require the variables $x_0,\ldots,x_m$ to be integer, we obtain an integer linear programming problem. If $\boldsymbol{M}$ is both primal and dual feasible, and in addition $\boldsymbol{M}_{\cdot 0}$ is all-integer, then $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ is an optimal integer solution.

Example 4.1. Consider the linear programming problem given by

$$\begin{array}{ll} \text{maximize} & 3a + 2b \\ \text{subject to} & 4a + b \leq 13 \\ & 2a + 5b \leq 24 \\ & a, b \geq 0. \end{array}$$

In this case we have five variables

$$\begin{aligned} x_0 &= 3a + 2b \\ x_1 &= 13 \ominus 4a \ominus b \\ x_2 &= 24 \ominus 2a \ominus 5b \\ x_3 &= a \\ x_4 &= b, \end{aligned}$$

of which $x_1,\ldots,x_4$ must be non-negative. The resulting matrix equation is given by

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \left[\begin{array}{c|cc} 0 & \ominus 3 & \ominus 2 \\ \hline 13 & 4 & 1 \\ 24 & 2 & 5 \\ 0 & \ominus 1 & 0 \\ 0 & 0 & \ominus 1 \end{array}\right] \begin{bmatrix} 1 \\ \ominus t_1 \\ \ominus t_2 \end{bmatrix}.$$

Here, the initial non-basic variables are $t_1 = x_3 = a$ and $t_2 = x_4 = b$. The initial matrix $\boldsymbol{M}$ is primal feasible, but not dual feasible. □

4.2.2 The Primal Simplex Algorithm

The primal simplex method for solving the linear programming problem of (4.1) iteratively determines a sequence of representations for $\boldsymbol{x}$,

$$\boldsymbol{x} = \boldsymbol{M}^0\,\boldsymbol{t}^0 = \boldsymbol{M}^1\,\boldsymbol{t}^1 = \ldots = \boldsymbol{M}^k\,\boldsymbol{t}^k,$$

beginning with $\boldsymbol{M}^0 = \boldsymbol{M}$ being primal feasible and $\boldsymbol{t}^0 = \boldsymbol{t}$. Furthermore, we have $M^h_{00} \leq M^{h+1}_{00}$ and $\boldsymbol{M}^h$ primal feasible, for all h. If the optimum is bounded from above, the matrix $\boldsymbol{M}^k$ is also dual feasible.

For simplicity, we let $\boldsymbol{M}$ and $\boldsymbol{t}$ denote any matrix $\boldsymbol{M}^h$ and vector $\boldsymbol{t}^h$, and let $\tilde{\boldsymbol{M}}$ and $\tilde{\boldsymbol{t}}$ denote the corresponding $\boldsymbol{M}^{h+1}$ and $\boldsymbol{t}^{h+1}$. Now, the primal simplex algorithm is given in Figure 4.1.

---

1. If $M_{0j} \geq 0$, for all $j \geq 1$, then $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ is an optimal solution, so stop. Otherwise, select an $s \geq 1$ for which $M_{0s} < 0$.

2. If $M_{is} \leq 0$, for all $i \geq 1$, then the linear programming problem has an unbounded optimum, so stop. Otherwise, compute

$$\theta_s = \min\{M_{i0}/M_{is} \mid i \geq 1 \ \wedge\ M_{is} > 0\}.$$

3. Select a $v \geq 1$ for which $M_{vs} > 0$ and $M_{v0}/M_{vs} = \theta_s$.

4. Determine $\tilde{\boldsymbol{M}}$ by

$$\tilde{\boldsymbol{M}}_{\cdot j} = \begin{cases} -\boldsymbol{M}_{\cdot s}/M_{vs} & \text{if } j = s \\ \boldsymbol{M}_{\cdot j} - \boldsymbol{M}_{\cdot s}(M_{vj}/M_{vs}) & \text{if } j \neq s. \end{cases}$$

5. Let $\tilde{t}_s = x_v$, and $\tilde{t}_j = t_j$, for all $j \neq s$. Designate $\tilde{\boldsymbol{M}}$ and $\tilde{\boldsymbol{t}}$ to be the current matrix $\boldsymbol{M}$ and vector $\boldsymbol{t}$, respectively, and return to Step 1.

Figure 4.1. The primal simplex algorithm.

At this stage, we do not concern ourselves with tie-braking rules in the choices of s and v that guarantee finiteness. In Steps 4 and 5 of the algorithm, the decision variables x_i are expressed linearly in the new non-basic variables $\tilde{t}_j$. This is called *pivoting*, and s and v are called the *pivot column* and *pivot row*, respectively.

Example 4.2. We apply the primal simplex algorithm to Example 4.1. In the first iteration, we select column $s = 1$ and row $v = 1$, resulting in

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \left[\begin{array}{r|rr} 9.75 & 0.75 & \Leftrightarrow 1.25 \\ \hline 0.00 & \Leftrightarrow 1.00 & 0.00 \\ 17.50 & \Leftrightarrow 0.50 & 4.50 \\ 3.25 & 0.25 & 0.25 \\ 0.00 & 0.00 & \Leftrightarrow 1.00 \end{array}\right] \begin{bmatrix} 1 \\ \Leftrightarrow t^1_1 \\ \Leftrightarrow t^1_2 \end{bmatrix}.$$

In the next iteration, we select column $s = 2$ and row $v = 2$, resulting in

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \left[\begin{array}{c|cc} 14.61 & 0.61 & 0.28 \\ \hline 0.00 & \Leftarrow 1.00 & 0.00 \\ 0.00 & 0.00 & \Leftarrow 1.00 \\ 2.28 & 0.28 & \Leftarrow 0.06 \\ 3.89 & \Leftarrow 0.11 & 0.22 \end{array}\right] \begin{bmatrix} 1 \\ \Leftarrow t_1^2 \\ \Leftarrow t_2^2 \end{bmatrix}.$$

Now the matrix is primal and dual feasible, so we have found an optimal solution $a = x_3 = 2.28$, $b = x_4 = 3.89$, with objective value 14.61. □

4.2.3 The Primal All-Integer Algorithm

Next, we assume that the coefficients in the initial matrix $\boldsymbol{M}$ are integer, and that the variables t_j are non-negative integer variables. As Gomory [1963] has shown, (4.2) then implies the cut

$$y = \lfloor M_{i0}/\lambda \rfloor + \sum_{j=1}^{n} \lfloor M_{ij}/\lambda \rfloor (\Leftarrow t_j), \tag{4.3}$$

where $\lambda > 0$ and y is a non-negative integer variable. Thus, $\boldsymbol{x} = \boldsymbol{M}\,\boldsymbol{t}$ may be augmented to include (4.3) without altering the set of feasible integer solutions. Based on this, a basic all-integer algorithm is obtained by replacing Step 3 of the primal simplex algorithm with Step 3′ given in Figure 4.2.

3′. Select as the source equation for (4.3) an equation $i \geq 1$ for which $M_{is} > 0$ and $0 \leq \lfloor M_{i0}/M_{is} \rfloor \leq \theta_s$. Let $\lambda = M_{is}$, and use (4.3) as equation v in Steps 4 and 5.

Figure 4.2. Row selection for a basic all-integer algorithm.

It is not necessary actually to augment $\boldsymbol{x} = \boldsymbol{M}\,\boldsymbol{t}$ with (4.3) if one is interested only in the values of the original variables x_i. Next, if (4.3) as determined by Step 3′ were added to $\boldsymbol{x} = \boldsymbol{M}\,\boldsymbol{t}$, then it would qualify to be selected as equation v in Step 3 of the primal simplex algorithm. Furthermore, its coefficient $M_{vs} = \lfloor M_{is}/M_{is} \rfloor = 1$, which assures that the successive matrix $\boldsymbol{M}$ is all-integer.

Unfortunately, there is no guarantee that the presented basic all-integer algorithm converges to an optimal integer solution in finite time. Therefore, Young [1968] and Glover [1968] extended the algorithm by adding a set of rules to assure finiteness. It is beyond the scope of this thesis to discuss these rules in general and to discuss their finiteness proofs; we restrict ourselves to one specific set of rules, which is due to Glover [1968].

We start by designating a row $r \geq 1$ to be the *reference row*, satisfying $M_{rj} > 0$,

for all $j = 1,\ldots,n$. Such a row can, for instance, be obtained by adding an equation

$$z = T + \sum_{j=1}^{n} (\Leftrightarrow t_j),$$

where T is an upper bound on $\sum_{j=1}^{n} t_j$, and z is a non-negative integer variable. As shown by Glover [1968], one of the properties of row r that is maintained throughout the algorithm is

$$\boldsymbol{M}_{\cdot j} <_{\text{lex}} \mathbf{0} \;\Rightarrow\; M_{rj} > 0.$$

With this reference row, the primal all-integer algorithm is now given in Figure 4.3.

---

1. If $M_{0j} \geq 0$, for all $j \geq 1$, then $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ is an optimal solution, so stop. Otherwise, select an $s \geq 1$ for which $M_{rs} > 0$ and for which

$$\boldsymbol{M}^*_{\cdot s} = \min_{\text{lex}}\{\boldsymbol{M}^*_{\cdot j} \mid j \geq 1 \wedge M_{rj} > 0\},$$

where we define

$$\boldsymbol{M}^*_{\cdot j} = \boldsymbol{M}_{\cdot j}/M_{rj},$$

for all $j \geq 1$ with $M_{rj} > 0$.

2. If $M_{is} \leq 0$, for all $i \geq 1$, then the integer linear programming problem has an unbounded optimum, so stop. Otherwise, compute

$$\theta_s = \min\{M_{i0}/M_{is} \mid i \geq 1 \wedge M_{is} > 0\}.$$

3. Select as the source equation for (4.3) a row $i \geq 1$ for which $M_{is} > 0$ and $0 \leq \lfloor M_{i0}/M_{is} \rfloor \leq \theta_s$. If more than one row is eligible,

- **i.** choose the candidate that maximizes $\tilde{M}_{0\tilde{s}}$, i.e., M_{0s} in the next iteration,
- **ii.** among these, choose the candidate that also maximizes the sum of the negative $\tilde{M}_{0j}$, and
- **iii.** among these, choose the equation with the smallest index.

Let $\lambda = M_{is}$, and use the cut (4.3) as equation v.

4. Determine $\tilde{\boldsymbol{M}}$ by

$$\tilde{\boldsymbol{M}}_{\cdot j} = \begin{cases} -\boldsymbol{M}_{\cdot s} & \text{if } j = s \\ \boldsymbol{M}_{\cdot j} - M_{vj}\boldsymbol{M}_{\cdot s} & \text{if } j \neq s. \end{cases}$$

5. Let $\tilde{t}_s = x_v$, and $\tilde{t}_j = t_j$, for all $j \neq s$. Designate $\tilde{\boldsymbol{M}}$ and $\tilde{\boldsymbol{t}}$ to be the current matrix $\boldsymbol{M}$ and vector $\boldsymbol{t}$, respectively, and return to Step 1.

Figure 4.3. The primal all-integer algorithm.

The rules in the primal all-integer algorithm assure a kind of lexicographic pro-

gress in passing from $\boldsymbol{M}$ to $\tilde{\boldsymbol{M}}$, which is used to prove the algorithm's finiteness. For more details the reader is referred to Glover [1968].

Example 4.3. We apply the primal all-integer algorithm to Example 4.1. We choose row $r = 1$ as reference row. Now in the first iteration, we select column $s = 2$ and row $i = 2$. Then we have $\lambda = 5$, resulting in a cut

$$y = 4 + 0(\Leftrightarrow t_1) + 1(\Leftrightarrow t_2),$$

which means that we have to subtract column 2 four times from column 0, and zero times from column 1. After this, column 2 is multiplied by $\Leftrightarrow 1$, and we get

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \left[\begin{array}{c|cc} 8 & \Leftrightarrow 3 & 2 \\ \hline 9 & 4 & \Leftrightarrow 1 \\ 4 & 2 & \Leftrightarrow 5 \\ 0 & \Leftrightarrow 1 & 0 \\ 4 & 0 & 1 \end{array}\right] \begin{bmatrix} 1 \\ \Leftrightarrow t_1 \\ \Leftrightarrow t_2 \end{bmatrix}.$$

In the next iteration, we select column $s = 1$ and row $i = 1$, resulting in $\lambda = 4$ and a cut

$$y = 2 + 1(\Leftrightarrow t_1) \Leftrightarrow 1(\Leftrightarrow t_2),$$

which means that we have to subtract column 1 two times from column 0, and add it one time to column 2. After multiplying column 1 by $\Leftrightarrow 1$, we then get

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \left[\begin{array}{c|cc} 14 & 3 & \Leftrightarrow 1 \\ \hline 1 & \Leftrightarrow 4 & 3 \\ 0 & \Leftrightarrow 2 & \Leftrightarrow 3 \\ 2 & 1 & \Leftrightarrow 1 \\ 4 & 0 & 1 \end{array}\right] \begin{bmatrix} 1 \\ \Leftrightarrow t_1 \\ \Leftrightarrow t_2 \end{bmatrix}.$$

The decision rules prevent row $i = 2$ being selected, since then the value of M_{0s} in the next iteration would be $\Leftrightarrow 7$, which is smaller than $\Leftrightarrow 1$.

Next, we select column $s = 2$ again and row $i = 1$, resulting in $\lambda = 3$ and a cut

$$y = 0 \Leftrightarrow 2(\Leftrightarrow t_1) + 1(\Leftrightarrow t_2),$$

which means that we have to subtract column 2 zero times from column 0, and add it two times to column 1. After multiplying column 2 by $\Leftrightarrow 1$, we then get

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \left[\begin{array}{c|cc} 14 & 1 & 1 \\ \hline 1 & 2 & \Leftrightarrow 3 \\ 0 & \Leftrightarrow 8 & 3 \\ 2 & \Leftrightarrow 1 & 1 \\ 4 & 2 & \Leftrightarrow 1 \end{array}\right] \begin{bmatrix} 1 \\ \Leftrightarrow t_1 \\ \Leftrightarrow t_2 \end{bmatrix}.$$

Now the matrix is primal and dual feasible, so we have found an optimal integer

solution $a = x_3 = 2$, $b = x_4 = 4$, with objective value 14. □

4.3 Multiple Cost Functions

Sometimes it is desired to maximize multiple cost functions, in a priority-based way. For instance, we want to find a solution $\boldsymbol{x}$ for which x_0 is maximal, and among the optimal solutions we want to find one for which x_1 is maximal. So, we want to find a solution $\boldsymbol{x}$ for which $[x_0\ x_1]^{\mathrm{T}}$ is lexicographically maximal. In this section, we show how to adapt the primal all-integer algorithm such that it finds a lexicographical maximum in a single run, while maintaining the finiteness of the algorithm.

We consider the multi-cost linear programming problem that, for a matrix $\boldsymbol{M} \in \mathbb{R}^{(m+1)\times(n+1)}$ and integer $c \leq m+1$, is given by

$$\begin{array}{rl} \text{maximize}_{\text{lex}} & \boldsymbol{x}_{\langle c\rangle} \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}\,\boldsymbol{t} \\ & x_i \geq 0 \qquad (i \geq c), \end{array}$$

where

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}, \quad \boldsymbol{x}_{\langle c\rangle} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{c-1} \end{bmatrix}, \quad \boldsymbol{t} = \begin{bmatrix} 1 \\ \leftrightarrows t_1 \\ \vdots \\ \leftrightarrows t_n \end{bmatrix}.$$

Again, the last n equations initially have the form

$$x_{m-n+j} = \leftrightarrows(\leftrightarrows t_j),$$

for $j = 1,\ldots,n$. The matrix $\boldsymbol{M}$ is now called primal feasible if $M_{i0} \geq 0$, for all $i = c,\ldots,m$, and dual feasible if $\boldsymbol{M}_{\langle c\rangle j} \geq_{\text{lex}} \boldsymbol{0}$, for all $j = 1,\ldots,n$, where

$$\boldsymbol{M}_{\langle c\rangle j} = \begin{bmatrix} M_{0j} \\ M_{1j} \\ \vdots \\ M_{c-1,j} \end{bmatrix},$$

i.e., the first c entries of column $\boldsymbol{M}_{\cdot j}$. An optimal solution of the multi-cost linear programming problem is given by $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ when $\boldsymbol{M}$ is both primal and dual feasible.

If we require the variables $x_0,\ldots,x_m$ to be integer, we obtain a multi-cost integer linear programming problem. If $\boldsymbol{M}$ is both primal and dual feasible, and in addition $\boldsymbol{M}_{\cdot 0}$ is all-integer, then $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ is an optimal integer solution.

Now, we extend the primal all-integer algorithm to handle multiple cost functions simultaneously. The resulting algorithm is shown in Figure 4.4.

1. If $\boldsymbol{M}_{\langle c\rangle j} \geq_{\text{lex}} \boldsymbol{0}$, for all $j \geq 1$, then $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ is an optimal solution, so stop. Otherwise, select an $s \geq 1$ for which $M_{rs} > 0$ and

$$\boldsymbol{M}^*_{\cdot s} = \min_{\text{lex}}\{\boldsymbol{M}^*_{\cdot j} \mid j \geq 1 \wedge M_{rj} > 0\},$$

where

$$\boldsymbol{M}^*_{\cdot j} = \boldsymbol{M}_{\cdot j}/M_{rj},$$

for all $j \geq 1$ with $M_{rj} > 0$.

2. If $M_{is} \leq 0$, for all $i \geq c$, then the problem has an unbounded optimum, so stop. Otherwise, compute

$$\theta_s = \min\{M_{i0}/M_{is} \mid i \geq c \wedge M_{is} > 0\}.$$

3. Select as the source equation for (4.3) an equation $i \geq c$ for which $M_{is} > 0$ and $0 \leq \lfloor M_{i0}/M_{is} \rfloor \leq \theta_s$. If more than one row is eligible,

- **i.** choose the candidate that lexicographically maximizes $\tilde{\boldsymbol{M}}_{\langle c\rangle \tilde{s}}$, i.e., $\boldsymbol{M}_{\langle c\rangle s}$ in the next iteration,
- **ii.** among these, choose the candidate that also maximizes the sum of the lexicographically negative $\tilde{\boldsymbol{M}}_{\langle c\rangle j}$, and
- **iii.** among these, choose the equation with the smallest index.

Let $\lambda = M_{is}$, and use the cut (4.3) as equation v.

4. Determine $\tilde{\boldsymbol{M}}$ by

$$\tilde{\boldsymbol{M}}_{\cdot j} = \begin{cases} -\boldsymbol{M}_{\cdot s} & \text{if } j = s \\ \boldsymbol{M}_{\cdot j} - M_{vj}\boldsymbol{M}_{\cdot s} & \text{if } j \neq s. \end{cases}$$

5. Let $\tilde{t}_s = x_v$, and $\tilde{t}_j = t_j$, for all $j \neq s$. Designate $\tilde{\boldsymbol{M}}$ and $\tilde{\boldsymbol{t}}$ to be the current matrix $\boldsymbol{M}$ and vector $\boldsymbol{t}$, respectively, and return to Step 1.

Figure 4.4. The multi-cost primal all-integer algorithm.

Theorem 4.1. *The multi-cost primal all-integer algorithm terminates in a finite number of iterations.*

Proof. The difference between the primal all-integer algorithm and the multi-cost primal all-integer algorithm is that instead of comparing cost elements M_{0j}, now cost vectors $\boldsymbol{M}_{\langle c\rangle j}$ are compared lexicographically. Therefore, the finiteness proof of Glover [1968] directly extends to prove the finiteness of the multi-cost primal all-integer algorithm. □

4.4 Pivoting Series

Running the multi-cost primal all-integer algorithm, sometimes a special kind of pivoting series may occur. For instance, consider an initial matrix given by

$$\boldsymbol{M}^0 = \left[\begin{array}{c|cc} 0 & \Leftrightarrow 100 & \Leftrightarrow 101 \\ \hline 10000 & 100 & 101 \\ 100 & 1 & 0 \\ 100 & 0 & 1 \\ 0 & \Leftrightarrow 1 & 0 \\ 0 & 0 & \Leftrightarrow 1 \end{array}\right].$$

This leads to a sequence of matrices given by

$$\boldsymbol{M}^1 = \left[\begin{array}{c|cc} 9999 & \Leftrightarrow 100 & 101 \\ \hline 1 & 100 & \Leftrightarrow 101 \\ 100 & 1 & 0 \\ 1 & 0 & \Leftrightarrow 1 \\ 0 & \Leftrightarrow 1 & 0 \\ 99 & 0 & 1 \end{array}\right], \quad \boldsymbol{M}^2 = \left[\begin{array}{c|cc} 9999 & 100 & \Leftrightarrow 99 \\ \hline 1 & \Leftrightarrow 100 & 99 \\ 100 & \Leftrightarrow 1 & 2 \\ 1 & 0 & \Leftrightarrow 1 \\ 0 & 1 & \Leftrightarrow 2 \\ 99 & 0 & 1 \end{array}\right],$$

$$\boldsymbol{M}^3 = \left[\begin{array}{c|cc} 9999 & \Leftrightarrow 98 & 99 \\ \hline 1 & 98 & \Leftrightarrow 99 \\ 100 & 3 & \Leftrightarrow 2 \\ 1 & \Leftrightarrow 2 & 1 \\ 0 & \Leftrightarrow 3 & 2 \\ 99 & 2 & \Leftrightarrow 1 \end{array}\right], \quad \boldsymbol{M}^4 = \left[\begin{array}{c|cc} 9999 & 98 & \Leftrightarrow 97 \\ \hline 1 & \Leftrightarrow 98 & 97 \\ 100 & \Leftrightarrow 3 & 4 \\ 1 & 2 & \Leftrightarrow 3 \\ 0 & 3 & \Leftrightarrow 4 \\ 99 & \Leftrightarrow 2 & 3 \end{array}\right],$$

$$\vdots$$

$$\boldsymbol{M}^{100} = \left[\begin{array}{c|cc} 9999 & 2 & \Leftrightarrow 1 \\ \hline 1 & \Leftrightarrow 2 & 1 \\ 100 & \Leftrightarrow 99 & 100 \\ 1 & 98 & \Leftrightarrow 99 \\ 0 & 99 & \Leftrightarrow 100 \\ 99 & \Leftrightarrow 98 & 99 \end{array}\right], \quad \boldsymbol{M}^{101} = \left[\begin{array}{c|cc} 10000 & 0 & 1 \\ \hline 0 & 0 & \Leftrightarrow 1 \\ 0 & 101 & \Leftrightarrow 100 \\ 100 & \Leftrightarrow 100 & 99 \\ 100 & \Leftrightarrow 101 & 100 \\ 0 & 100 & \Leftrightarrow 99 \end{array}\right].$$

In this example, the cuts used in iteration 1 until 100 are all of the form

$$y = 0 + 1(\Leftrightarrow t_s) \Leftrightarrow 2(\Leftrightarrow t_q), \tag{4.4}$$

with alternately $s = 1$, $q = 2$ and $s = 2$, $q = 1$. The number of iterations of this kind is determined by the coefficients in the matrix and may be arbitrarily large. In order to reduce the number of iterations, we present a way to execute such a pivoting series in one step.

Let $a, b \geq 1$, $a \neq b$, be such that in a certain iteration the matrix $\boldsymbol{M}$ results in a

pivot column $s = a$ and that the chosen row i results in a cut (4.4) with $q = b$. Then a series of the above kind continues as long as

(i) after $2k$ iterations the pivot column s equals a and the generated cut is (4.4) with $q = b$, and

(ii) after $2k+1$ iterations the pivot column s equals b and the generated cut is (4.4) with $q = a$.

If we denote the largest $k \geq 0$ that satisfies (i) by k_{even}, and the largest $k \geq 0$ that satisfies (ii) by k_{odd}, then the number of iterations of the series is given by

$$n = \min\{2k_{\text{even}} + 2\,,\, 2k_{\text{odd}} + 3\},$$

where we assume $k_{\text{odd}} = \Leftrightarrow 1$ if no $k \geq 0$ satisfies (ii). After $2k \leq n$ iterations in the series, the resulting matrix $\tilde{\boldsymbol{M}}$ is then given by

$$\tilde{\boldsymbol{M}}_{\cdot j} = \begin{cases} \boldsymbol{M}_{\cdot a} + 2k(\boldsymbol{M}_{\cdot a} + \boldsymbol{M}_{\cdot b}) & \text{if } j = a \\ \boldsymbol{M}_{\cdot b} \Leftrightarrow 2k(\boldsymbol{M}_{\cdot a} + \boldsymbol{M}_{\cdot b}) & \text{if } j = b \\ \boldsymbol{M}_{\cdot j} & \text{if } j \neq a, b. \end{cases} \tag{4.5}$$

After $2k+1 \leq n$ iterations in the series, the resulting matrix $\tilde{\boldsymbol{M}}$ is given by

$$\tilde{\boldsymbol{M}}_{\cdot j} = \begin{cases} \Leftrightarrow \boldsymbol{M}_{\cdot a} \Leftrightarrow 2k(\boldsymbol{M}_{\cdot a} + \boldsymbol{M}_{\cdot b}) & \text{if } j = a \\ \Leftrightarrow \boldsymbol{M}_{\cdot b} + (2k+2)(\boldsymbol{M}_{\cdot a} + \boldsymbol{M}_{\cdot b}) & \text{if } j = b \\ \boldsymbol{M}_{\cdot j} & \text{if } j \neq a, b. \end{cases} \tag{4.6}$$

Using (4.5), linear constraints in k can be derived that are sufficient for (i), from which a lower bound on k_{even} can be determined. In a similar way, (4.6) can be used to derive linear constraints in k that are sufficient for (ii), from which a lower bound on k_{odd} can be determined. After calculating the corresponding lower bound on n, we can then execute a series of iterations in one step, using either (4.5) or (4.6).

5

Cost Calculations

The objective we consider in the multidimensional periodic scheduling problem as given in Chapter 2 is to minimize the estimated IC area. In Definition 2.8, the total area cost of a schedule $\sigma \in \mathcal{S}'$ is therefore defined as

$$f(\sigma) = \sum_{w \in W} a(t(w)) \; + \; a(t_{\mathrm{v}}) \max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{L}(A,c)| \; + \; a(t_{\mathrm{a}}) \max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{B}(A,c)|,$$

in which the terms represent the processing unit cost, the storage cost, and the access cost, respectively. In this chapter we discuss these three contributions in more detail.

5.1 Processing Unit Cost

The processing unit cost of a schedule is directly given by the set W of processing units, and can be rewritten as

$$\sum_{t \in T} a(t) \; |W(t)|,$$

where T is the set of operation types, $a(t)$ is the area cost of type t, and $W(t) = \{w \in W \mid t(w) = t\}$ is the set of processing units of type t. Given a time assignment $\tau = (\boldsymbol{p}, s)$, a feasible processing unit assignment can be determined by using the concept of processing unit conflict graphs, given by the following definition.

Definition 5.1 (Processing Unit Conflict Graph). Given are a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ and a time assignment $\tau = (\boldsymbol{p}, s)$. The processing unit conflict graph of processing unit type $t \in T$ is a graph $H(t, \tau)$ with node set $V(t) = \{v \in V \mid t(v) = t\}$, i.e., the set of operations of type t. Furthermore, its edge set $F(t, \tau)$ consists of edges $\{u, v\}$, $u, v \in V(t)$, for which u and v cannot be assigned to the same processing unit, according to the processing unit constraints. So, $\{u, v\} \in F(t, \tau)$ if and only if there is an execution $\boldsymbol{i}$ of u and an execution $\boldsymbol{j}$ of v, with $(u, \boldsymbol{i}) \neq (v, \boldsymbol{j})$ and

$$c(u, \boldsymbol{i}) + x = c(v, \boldsymbol{j}) + y,$$

for certain $x, y \in \{0, \ldots, o(t) \Leftrightarrow 1\}$, where $c(u, \boldsymbol{i})$ is the time at which execution $\boldsymbol{i}$ of operation u is scheduled, and $o(t)$ is the occupation time of operation type t. □

For example, consider a signal flow graph with four operations, u, v, w, and x, that are all of the same type t with occupation time $o(t) = 1$. Furthermore, consider a time assignment τ as depicted in Figure 5.1. The corresponding processing unit conflict graph $H(t, \tau)$ is shown in Figure 5.2.

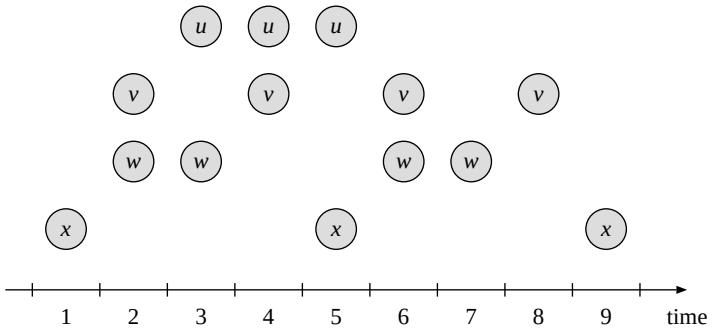


Figure 5.1. A time assignment for a signal flow graph with four operations of the same type t.

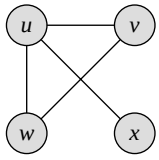


Figure 5.2. The processing unit conflict graph $H(t, \tau)$ corresponding to the example of Figure 5.1.

Finding a minimal processing unit assignment is now equivalent to coloring the graph $H(t, \tau)$ with a minimal number of colors. This means that each node $v \in V(t)$ has to be assigned a color $a(v)$, such that $a(u) \neq a(v)$ for each edge $\{u, v\} \in F(t, \tau)$,

and such that a minimal number of different colors is used. For more information on graph coloring the reader is referred to Harary [1969] and Gibbons [1985].

The processing unit conflict graph of Figure 5.2 can be colored minimally with three colors, so a minimal processing unit assignment for this example uses three processing units of type t.

5.2 Access Cost

The access cost of a schedule is based on the maximum number of simultaneous memory accesses at any time c, which is given by

$$\max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{B}(A,c)|.$$

In this equation, $\mathcal{B}(A,c)$ is the number of accesses at time c of array cluster $A \in \mathcal{A}$, which is given by

$$\mathcal{B}(A,c) = \{\boldsymbol{n} \in \mathbb{Z}^{\alpha} \mid p \in A \wedge \boldsymbol{i} \in \mathcal{I}(p) \wedge \boldsymbol{n} = \boldsymbol{n}(p,\boldsymbol{i}) \wedge c(p,\boldsymbol{i}) = c\},$$

where $\boldsymbol{n}(p,\boldsymbol{i})$ is the index vector of the array element that is accessed at execution $\boldsymbol{i}$ of port p. To determine the access cost efficiently, one must determine the maximum number of simultaneous accesses without explicitly considering the accesses of all array elements, i.e., without enumerating $\mathcal{B}(A,c)$ for all array clusters $A \in \mathcal{A}$.

During scheduling, we consider the memory as one black box with a number of terminals, which are to be connected to the outputs and inputs of the processing units. The productions and consumptions of data at output and input ports $p \in P$ are transported through these terminals. Therefore, we introduce a terminal assignment, which assigns each port $p \in P$ to a memory terminal. We assume that all accesses for a certain port $p \in P$ take place via the same terminal. This assumption is similar to the assumption that all executions of an operation take place on the same processing unit, and is made in order to keep the terminal assignment compact. The terminal assignment is defined as follows.

Definition 5.2 (Terminal Assignment). Given a signal flow graph $G = (V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, a terminal assignment $\mu = (Q,m)$ is given by a set Q of memory terminals and a function $m : P \rightarrow Q$ that assigns each port to a memory terminal. □

The constraints on a terminal assignment are as follows.

Definition 5.3 (Access Constraints). Given are a signal flow graph $G = (V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, a time assignment $\tau = (\boldsymbol{p},s)$, and a terminal assignment $\mu = (Q,m)$. Then for each execution $\boldsymbol{i}$ of a port $p \in P$, and for each execution $\boldsymbol{j}$ of a port $q \in P$, with $m(p) = m(q)$, the access constraints specify that

$$c(p,\boldsymbol{i}) \neq c(q,\boldsymbol{j}),$$

if $p \not\bowtie q$ or $\boldsymbol{n}(p,\boldsymbol{i}) \neq \boldsymbol{n}(q,\boldsymbol{j})$, where $p \bowtie q$ means that p and q access the same array, as given in Definition 2.7. □

So, if two accesses coincide at the same memory terminal, they must refer to the same array cluster and the same array index. Otherwise, two different data would be accessed simultaneously at the same memory terminal. A terminal assignment is called feasible if and only if it obeys the access constraints.

The number $|Q|$ of memory terminals of a feasible terminal assignment is an upper bound on the maximum number of simultaneous accesses. Similar to a processing unit assignment, a feasible terminal assignment for a given time assignment $\tau = (\boldsymbol{p}, s)$ can be determined by using the concept of an access conflict graph, given by the following definition.

Definition 5.4 (Access Conflict Graph). Given are a signal flow graph $G = (V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$ and a time assignment $\tau = (\boldsymbol{p}, s)$. The access conflict graph is a graph $H(t_a,\tau)$ with node set P, i.e., the set of all ports. Furthermore, its edge set $F(t_a,\tau)$ consists of edges $\{p,q\}$, $p,q \in P$, for which p and q cannot be assigned to the same memory terminal, according to the access constraints. So, $\{p,q\} \in F(t_a,\tau)$ if and only if there is an execution $\boldsymbol{i}$ of p and an execution $\boldsymbol{j}$ of q, with

$$c(p,\boldsymbol{i}) = c(q,\boldsymbol{j}),$$

but $p \not\bowtie q$ or $\boldsymbol{n}(p,\boldsymbol{i}) \neq \boldsymbol{n}(q,\boldsymbol{j})$. □

For example, consider the video algorithm and time assignment as given in Figure 5.3. The relative transfer times of the input and output ports are equal to zero. The

for i_0 = 0 to 3 period 1
  $x[i_0] = u(\,)$ start at 1
for j_0 = 0 to 2 period 1
  $= v(x[j_0+1])$ start at 3
for k_0 = 0 to 1 period 1
  $= w(x[k_0])$ start at 2

Figure 5.3. A video algorithm with a given time assignment. A statement 'period 1' at the end of a line denotes that the period of the corresponding iterator is equal to 1. A statement 'start at 1' at the end of a line denotes that the start time of the corresponding operation is equal to 1. In this example, operation u is an input operation, which only produces data and therefore has no arguments. Operations v and w are output operations, which only consume data. Therefore, the left hand sides of the corresponding equations are left empty.

corresponding signal flow graph and the executions of the operations are shown in Figure 5.4. As we can see, port p has an access conflict with port q, since at time 3 port p accesses array element $[2]$, but port q accesses array element $[1]$. Further-

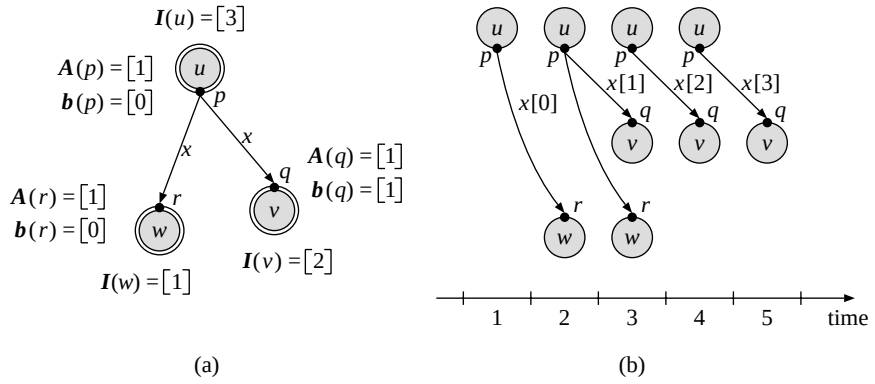


Figure 5.4. (a) The signal flow graph of the video algorithm of Figure 5.3, and (b) the executions of the operations in time.

more, port p has an access conflict with port r. Port q and r do not have an access conflict, since they do not access different data at the same time. The only time at which they both access data is time 3, however, then they access the same element, $[1]$, and hence no access conflict occurs. The corresponding access conflict graph $H(t_a, \tau)$ is shown in Figure 5.5.

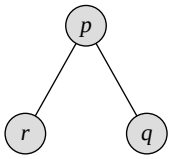


Figure 5.5. The access conflict graph $H(t_a, \tau)$ corresponding to the signal flow graph and time assignment of Figure 5.4.

Similar to a processing unit assignment, finding a minimal terminal assignment is now equivalent to coloring the graph $H(t_a, \tau)$ with a minimal number of colors. The access conflict graph of Figure 5.5 can be colored minimally with two colors, so a minimal terminal assignment for this example uses two memory terminals.

5.3 Storage Cost

The storage cost of a schedule is based on the maximum number of variables that are simultaneously alive at any time c. Given a time assignment $\tau = (\boldsymbol{p}, s)$, this maximum is given by

$$f_{\max}(\tau) = \max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{L}(A, c)|.$$

In this equation we have

$$\mathcal{L}(A,c) = \{\boldsymbol{n} \in \mathbb{Z}^{\alpha} \mid\ p,q \in A \wedge (p,q) \in E \wedge \boldsymbol{i} \in \mathcal{I}(p) \wedge \boldsymbol{j} \in \mathcal{I}(q) \wedge \boldsymbol{n} = \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \wedge c(p,\boldsymbol{i}) < c \leq c(q,\boldsymbol{j})\},$$

which is the number of array elements of array cluster $A \in \mathcal{A}$ that are alive at time c. We can rewrite this equation into

$$\mathcal{L}(A,c) = \{\boldsymbol{n} \in \mathbb{Z}^{\alpha} \mid\ p \in A \cap O \wedge q \in A \cap I \wedge \boldsymbol{i} \in \mathcal{I}(p) \wedge \boldsymbol{j} \in \mathcal{I}(q) \wedge \boldsymbol{n} = \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \wedge c(p,\boldsymbol{i}) < c \leq c(q,\boldsymbol{j})\},$$

under the assumption that $(p,q) \in E$ whenever there is an execution $\boldsymbol{i}$ of output port $p \in A \cap O$ and an execution $\boldsymbol{j}$ of input port $q \in A \cap I$ for which $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})$. This is a valid assumption if the signal flow graph is derived from a video algorithm.

5.3.1 Average Number of Variables

In view of the chosen solution strategy, which is based on linear programming, we approximate the function $f_{\max}$ by a function representing the average number of variables that are simultaneously alive. This is done in order to obtain a function that is linear in the periods and start times of the operations. To determine the average number of variables, we assume that all operations have an infinite number of executions, i.e., $I_0(v) = \infty$, for all $v \in V$, which is usually the case in digital signal processing algorithms.

As discussed in Chapter 2, this assumption implies that the periods $p_0(v)$ are fixed and positive for all operations $v \in V$. Next, only the first index of each produced and consumed array element depends on an infinite iterator, i.e., for the array index matrix $\boldsymbol{A}(p)$ of each port $p \in P$ we have $A_{00}(p) > 0$, and $A_{k0}(p) = 0$, for all $k = 1,\ldots,\alpha(p) \Leftrightarrow 1$. Furthermore, the assumption implies $A_{00}(p)/p_0(p) = A_{00}(q)/p_0(q)$, for all $(p,q) \in E$. If we now define

$$p_{\text{glob}} = \text{lcm}\{p_0(v) \mid v \in V\},$$

then we can see that all executions and all data transports are repeated after p_{glob} time units, i.e., p_{glob} is a *global period*.

During one global period, we have $p_{\text{glob}}/p_0(v)$ iterations in dimension 0 for each operation $v \in V$, i.e., we have $p_{\text{glob}}/p_0(v)$ values for iterator i_0. So, the executions $\boldsymbol{i}$ of operation $v \in V$ of the first g global periods are given by $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(v,g)$, where

$$I_k(v,g) = \begin{cases} g\,p_{\text{glob}}/p_0(v) \Leftrightarrow 1 & \text{if } k = 0 \\ I_k(v) & \text{if } k > 0, \end{cases}$$

and the set of executions of v of the first g global periods is given by $\mathcal{I}(v,g) = \{\boldsymbol{i} \in \mathbb{Z}^{\delta(v)} \mid \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(v,g)\}$. Similarly, for each port $p = (v,\tilde{p}) \in P$, we define the iterator bound vector for the first g global periods by $\boldsymbol{I}(p,g) = \boldsymbol{I}(v,g)$ and we define $\mathcal{I}(p,g) = \mathcal{I}(v,g)$.

The array elements of array cluster $A \in \mathcal{A}$ that are produced at output port $p \in A \cap O$ during its first g global periods are given by

$$\mathcal{N}(p,g) = \{\boldsymbol{n}(p,\boldsymbol{i}) \in \mathbb{Z}^{\alpha(p)} \mid \boldsymbol{i} \in \mathcal{I}(p,g)\}.$$

The array elements of array cluster A that are produced during the first g global periods of all output ports in A are then given by

$$\mathcal{N}(A,g) = \bigcup_{p \in A \cap O} \mathcal{N}(p,g).$$

To determine the average number of variables that are simultaneously alive, we first define the lifetimes of the array elements. To this end, we need the production and the last consumption of each array element. The production of an array element $\boldsymbol{n} \in \mathcal{N}(A,g)$ of array cluster A takes place at time

$$b(A,\boldsymbol{n}) = c(p,\boldsymbol{i}),$$

where $p \in A \cap O$ is the output port and $\boldsymbol{i}$ its execution at which the element is produced, i.e., for which $\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}$. Note that this is a valid definition, since we assumed single assignments. The last consumption of array element $\boldsymbol{n} \in \mathcal{N}(A,g)$ of array cluster A takes place at time

$$e(A,\boldsymbol{n}) = \max\{c(q,\boldsymbol{j}) \mid q \in A \wedge \boldsymbol{j} \in \mathcal{I}(q) \wedge \boldsymbol{n}(q,\boldsymbol{j}) = \boldsymbol{n}\}.$$

Note that if an array element $\boldsymbol{n}$ is not consumed, then by this definition $e(A,\boldsymbol{n}) = b(A,\boldsymbol{n})$. The lifetime of an array element $\boldsymbol{n} \in \mathcal{N}(A,g)$ is now given by $e(A,\boldsymbol{n}) \Leftrightarrow b(A,\boldsymbol{n})$.

Now we can define the average number of variables of the first g global periods of array cluster $A \in \mathcal{A}$ that are simultaneously alive by

$$l_{\text{avg}}(A,g) = \frac{1}{gp_{\text{glob}}} \sum_{\boldsymbol{n} \in \mathcal{N}(A,g)} (e(A,\boldsymbol{n}) \Leftrightarrow b(A,\boldsymbol{n})),$$

and the average number of all variables that are simultaneously alive for a time assignment $\tau = (\boldsymbol{p}, s)$ is now given by

$$f_{\text{avg}}(\tau) = \lim_{g \to \infty} \sum_{A \in \mathcal{A}} l_{\text{avg}}(A,g),$$

which is equivalent to

$$f_{\text{avg}}(\tau) = \lim_{g \to \infty} \frac{1}{gp_{\text{glob}}} \sum_{A \in \mathcal{A}} \sum_{\boldsymbol{n} \in \mathcal{N}(A,g)} (e(A,\boldsymbol{n}) \Leftrightarrow b(A,\boldsymbol{n})).$$

Note that we use a limit in this equation, since there may be a start-up effect. This start-up effect occurs, for instance, when an execution $\boldsymbol{i}$ of an operation v produces data that is consumed by an execution $\boldsymbol{j}$ of an operation u, with $j_0 = i_0 \Leftrightarrow 1$. Then for $i_0 = 0$ there is no corresponding consumption, and thus the corresponding variable

has a zero lifetime. Nevertheless, after a finite number of global periods the average number of variables is the same for each successive global period.

For an example, consider the video algorithm and corresponding signal flow graph of Figure 5.6. The relative transfer times of the input and output ports are

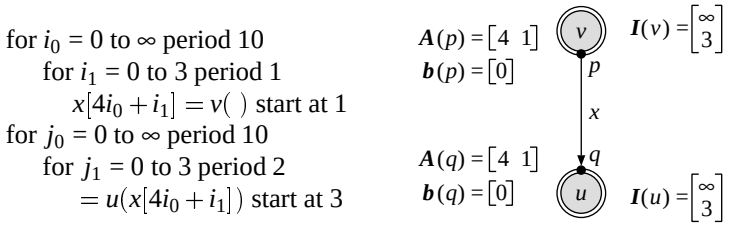


Figure 5.6. An example of a video algorithm and corresponding signal flow graph. The time assignment is given.

equal to zero. In this example, the global period $p_{\text{glob}} = \text{lcm}(10, 10) = 10$. The executions of the first global period are depicted in Figure 5.7. Furthermore, the fig-

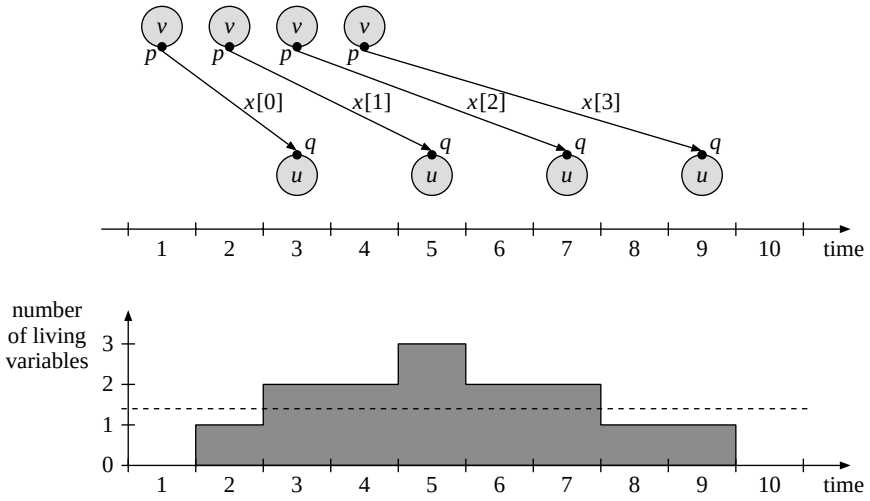


Figure 5.7. The executions of the first global period of the signal flow graph of Figure 5.6, shown at the top of the figure, and the number of variables alive at each time, shown at the bottom. The dashed line denotes the average number of variables that are simultaneously alive, which is equal to 1.4.

ure shows the number of variables that are alive at each time. The array elements that are produced during the first global period are $[0]$, $[1]$, $[2]$, and $[3]$, with lifetimes 2, 3, 4, and 5, respectively. So, the average number of variables of the first global period of array cluster $A = \{p, q\}$ that are simultaneously alive is given by

$l_{\text{avg}}(A,1) = \frac{1}{10}(2+3+4+5) = 1.4$. The average number is also 1.4 in the next global periods, which yields $f_{\text{avg}}(\tau) = 1.4$. The maximum number of variables that are simultaneously alive in this example is $f_{\max}(\tau) = 3$, as can be seen in the figure.

Unfortunately, the function f_{avg} is still not a linear expression in the periods and start times of the operations. Nevertheless, under the condition that each variable is consumed exactly once, we can rewrite it into such a linear expression. This is explained in the next section.

5.3.2 One-to-One Condition

First, let $\mathcal{N}_{\text{prd}}(A,g)$ be the multiset of produced array elements of array cluster $A \in \mathcal{A}$ during the first g global periods, i.e., $\mathcal{N}_{\text{prd}}(A,g) = \mathcal{N}(A,g)$. Next, let $\mathcal{N}_{\text{con}}(A,g)$ be the multiset of consumed array elements of array cluster $A \in \mathcal{A}$ during the first g global periods, i.e.,

$$\mathcal{N}_{\text{con}}(A,g) = \bigcup_{q \in A \cap I} \mathcal{N}(q,g),$$

where $\mathcal{N}(q,g)$ is a multiset defined by

$$\mathcal{N}(q,g) = \{\boldsymbol{n}(q,\boldsymbol{j}) \in \mathbb{Z}^{\alpha(q)} \mid \boldsymbol{j} \in \mathcal{I}(q,g)\}.$$

Now, under the *one-to-one condition*, which states that $\mathcal{N}_{\text{prd}}(A,g) = \mathcal{N}_{\text{con}}(A,g)$, for all $A \in \mathcal{A}$ and $g \in \mathbb{N}$, we can rewrite f_{avg} into a linear expression in the periods and start times of the operations as follows. First, we rewrite

$$\begin{aligned}
f_{\text{avg}}(\tau) &= \lim_{g\to\infty} \frac{1}{g p_{\text{glob}}} \sum_{A\in\mathcal{A}} \sum_{\boldsymbol{n}\in\mathcal{N}(A,g)} (e(A,\boldsymbol{n}) \Leftrightarrow b(A,\boldsymbol{n})) \\
&= \lim_{g\to\infty} \frac{1}{g p_{\text{glob}}} \sum_{A\in\mathcal{A}} g \sum_{\boldsymbol{n}\in\mathcal{N}(A,1)} (e(A,\boldsymbol{n}) \Leftrightarrow b(A,\boldsymbol{n})) \\
&= \frac{1}{p_{\text{glob}}} \sum_{A\in\mathcal{A}} \Big(\sum_{\boldsymbol{n}\in\mathcal{N}_{\text{con}}(A,1)} e(A,\boldsymbol{n}) \Leftrightarrow \sum_{\boldsymbol{n}\in\mathcal{N}_{\text{prd}}(A,1)} b(A,\boldsymbol{n}) \Big) \\
&= \frac{1}{p_{\text{glob}}} \sum_{A\in\mathcal{A}} \Big(\sum_{q\in A\cap I} \sum_{\boldsymbol{j}\in\mathcal{I}(q,1)} c(q,\boldsymbol{j}) \Leftrightarrow \sum_{p\in A\cap O} \sum_{\boldsymbol{i}\in\mathcal{I}(p,1)} c(p,\boldsymbol{i}) \Big) \\
&= \frac{1}{p_{\text{glob}}} \Big(\sum_{q\in I} \sum_{\boldsymbol{j}\in\mathcal{I}(q,1)} c(q,\boldsymbol{j}) \Leftrightarrow \sum_{p\in O} \sum_{\boldsymbol{i}\in\mathcal{I}(p,1)} c(p,\boldsymbol{i}) \Big).
\end{aligned}$$

Next, for a port $p = (v,\tilde{p}) \in P$, the sum of the execution times of the first global period is equal to the number of executions times the average execution time. This is denoted by $C(p)$, which is given by

$$C(p) = \sum_{\boldsymbol{i}\in\mathcal{I}(p,1)} c(p,\boldsymbol{i})$$

$$\begin{aligned} &= \sum_{\boldsymbol{i} \in \mathcal{I}(p,1)} (\boldsymbol{p}^{\mathrm{T}}(p)\, \boldsymbol{i} + s(p)) \\ &= N(p) \cdot (\tfrac{1}{2}\, \boldsymbol{p}^{\mathrm{T}}(p)\, \boldsymbol{I}(p,1) + s(p)) \\ &= N(v) \cdot (\tfrac{1}{2}\, \boldsymbol{p}^{\mathrm{T}}(v)\, \boldsymbol{I}(v,1) + s(v) + r(t(v), \tilde{p})). \end{aligned}$$

Here, $N(p) = N(v)$ is the number of executions of operation v in the first global period, i.e.,

$$N(v) = \prod_{k=0}^{\delta(v)-1} (I_k(v,1) + 1).$$

So, if the one-to-one condition holds, then $f_{\mathrm{avg}}(\tau) = f_{\mathrm{ex}}(\tau)/p_{\mathrm{glob}}$, where

$$f_{\mathrm{ex}}(\tau) = \sum_{q \in I} C(q) \Leftrightarrow \sum_{p \in O} C(p),$$

which is a linear expression in the periods and start times of the operations.

In the example of Figure 5.6 the one-to-one condition holds, so $f_{\mathrm{avg}}(\tau) = f_{\mathrm{ex}}/10 = (C(q) \Leftrightarrow C(p))/10 = (24 \Leftrightarrow 10)/10 = 1.4$, which is equal to the result found earlier.

5.3.3 Stop Operations

Unfortunately, the one-to-one condition does not always hold. For instance, array elements may be used several times by one or more operations, or may not be used at all. In this case, the ends of the variables' lifetimes are not given by the execution times of the input ports, and thus no linear cost function is obtained. To circumvent this, we determine the ends of the variables' lifetimes in a different way.

Since productions and consumptions are executed multidimensional periodically, the sets of moments corresponding to the ends of the variables' lifetimes contain periodicity. Therefore, we represent these moments by a set of multidimensional periodic operations, which we call *stop operations*. A set of stop operations corresponding to an output port p indicates the moments when the produced array elements stop living. At these moments we say that the array elements are *annihilated* by the stop operations. Obviously, the annihilation of an array element must follow its production and its consumptions, which results in precedence constraints between productions and consumptions on the one hand, and annihilations on the other. More formally, stop operations are defined as follows.

Definition 5.5 (Stop Operation Type). A stop operation type is an operation type t_{s} with

- $\tilde{I}(t_{\mathrm{s}}) = \{1\}$, one operation input port,
- the relative transfer time $r(t_{\mathrm{s}}, 1) = 0$,
- $\tilde{O}(t_{\mathrm{s}}) = \emptyset$, no operation output ports,

- the occupation time $o(t_s) = 0$, and
- the area cost $a(t_s) = 0$.

□

We can now define an extension of a signal flow graph with stop operations as follows.

Definition 5.6 (Extended Signal Flow Graph). Given a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, an extended signal flow graph $G' = (V', t, \boldsymbol{I}, E', \boldsymbol{A}, \boldsymbol{b})$ is a graph with

- $V' = V \cup V_s$, where V_s is a finite set of stop operations,
- $t(v) = t_s$, for all $v \in V_s$, and
- $E' = E \cup E_s$, where $E_s \subset P \times S$ is a finite set of *stop edges*, i.e., edges from input and output ports $p \in P$ to *stop ports* $s \in S = \{(v,1) \mid v \in V_s\}$.

The set of stop edges E_s is such that for each stop port $s \in S$ there is exactly one output port $p \in O$ with $(p,s) \in E_s$, and for each $(p,q) \in E$

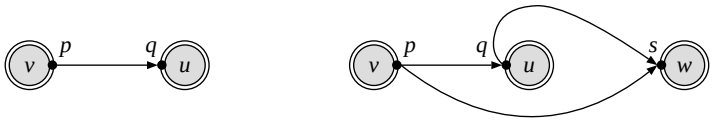
$$(p,s) \in E_s \;\Rightarrow\; (q,s) \in E_s.$$

Next, the stop operations are repeated infinitely, i.e., $I_0(v) = \infty$, for all $v \in V_s$, which implies that the corresponding periods must be fixed and positive. Furthermore, the one-to-one condition must hold between productions and annihilations, i.e., $\mathcal{N}_{\text{prd}}(A,g) = \mathcal{N}_{\text{ann}}(A,g)$, for all $A \in \mathcal{A}$ and $g \in \mathbb{N}$, where $\mathcal{N}_{\text{ann}}(A,g)$ is the multiset defined by

$$\mathcal{N}_{\text{ann}}(A,g) = \bigcup_{s \in A \cap S} \mathcal{N}(s,g).$$

Note that the array clusters A are extended to contain also the stop ports. □

Figure 5.8 shows an example of a signal flow graph and a corresponding extended



(a) (b)

Figure 5.8. (a) An example of an original signal flow graph and (b) a corresponding extended signal flow graph.

signal flow graph. In the figure, a stop operation w is added, with stop port s, and stop edges (p,s) and (q,s) are added.

In the multidimensional periodic scheduling problem now also a time assignment for the stop operations has to be determined, subject to the following additional constraints.

Definition 5.7 (Stop Precedence Constraints). Given are an extended signal flow graph $G' = (V', t, \boldsymbol{I}, E', \boldsymbol{A}, \boldsymbol{b})$ and a time assignment $\tau = (\boldsymbol{p}, s)$. Then for each execution $\boldsymbol{i}$ of an input or output port $p \in P$, and for each execution $\boldsymbol{j}$ of a stop port $s \in S$, with $(p,s) \in E_s$, the stop precedence constraints specify that

$$\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(s,\boldsymbol{j}) \;\Rightarrow\; c(p,\boldsymbol{i}) \leq c(s,\boldsymbol{j}).$$

Note that $c(p,\boldsymbol{i}) = c(s,\boldsymbol{j})$ is also allowed, in contrast to the precedence constraints for data transport. □

With the stop operations, we can determine $f_{\text{avg}}(\tau) = f_{\text{stop}}(\tau)/p_{\text{glob}}$, where

$$f_{\text{stop}}(\tau) = \sum_{s \in S} C(s) \Leftrightarrow \sum_{p \in O} C(p), \tag{5.1}$$

which is again a linear expression in the periods and start times of the operations. Note that we here assumed that for every possible time assignment for the original operations a time assignment for the stop operations can be found such that the annihilations coincide with the ends of the variables' lifetimes. If this is not the case, then $f_{\text{stop}}(\tau)/p_{\text{glob}}$ is an upper bound on $f_{\text{avg}}(\tau)$.

For an example, consider the signal flow graph of Figure 5.9. The relative transfer times of the input and output ports are again equal to zero. In this example, the

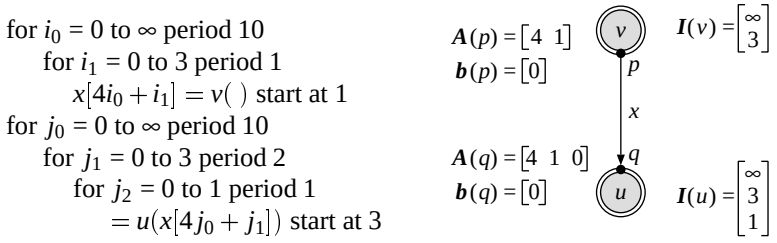


Figure 5.9. An example of a video algorithm and a corresponding signal flow graph in which array elements are consumed twice.

array elements are consumed twice, as we can see in Figure 5.10, which shows the executions of the first global period. Figure 5.11 shows an extended signal flow graph for this example, with a time assignment for the stop operation *w*. The executions of the extended signal flow graph of the first global period are shown in Figure 5.12. The ends of the variables' lifetimes are now given by the executions of the stop port *s*. The average number of variables alive is thus given by $f_{\text{avg}}(\tau) = f_{\text{stop}}(\tau)/p_{\text{glob}} = (C(s) \Leftrightarrow C(p))/p_{\text{glob}} = (28 \Leftrightarrow 10)/10 = 1.8$.

5.3.4 Extended Signal Flow Graph Construction

In the example of Figure 5.12, we have one stop operation, being a copy of the producing operation, that defines the ends of the variables' lifetimes. In general,

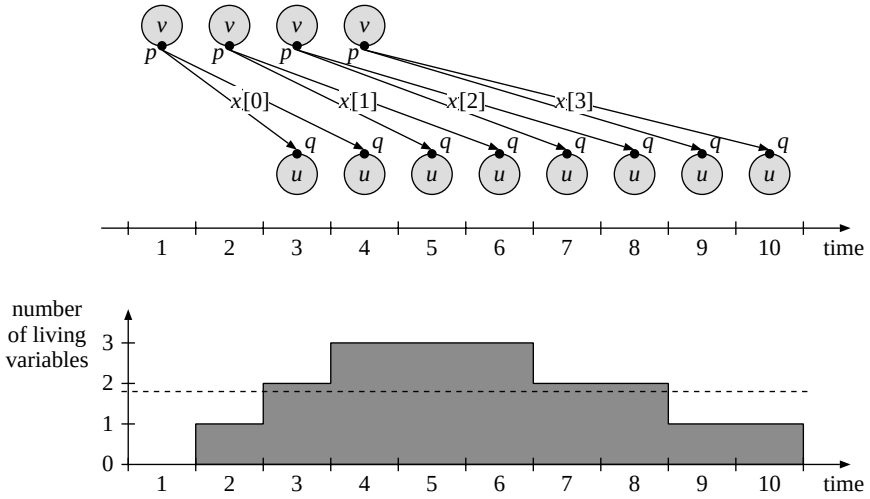


Figure 5.10. The executions of the signal flow graph of Figure 5.9 in the first global period, showing multiple consumptions of array elements. The average number of variables that are simultaneously alive is equal to 1.8.

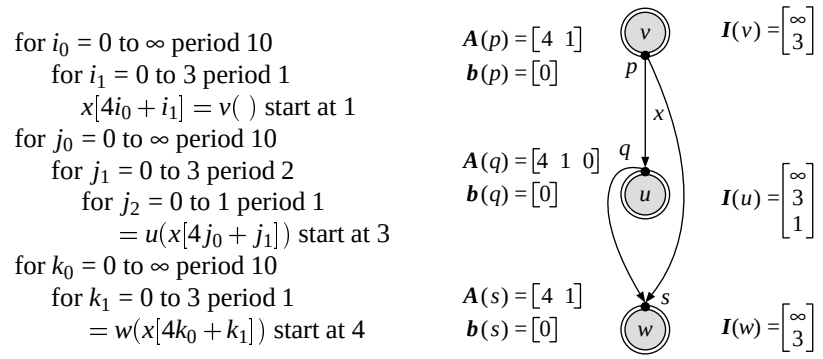


Figure 5.11. An extended video algorithm and signal flow graph corresponding to the example of Figure 5.9.

however, more stop operations may be required. In the worst case, a number of stop operations may be required that is proportional to the number of executions of the producing and consuming operations, yielding an arbitrarily large extended signal flow graph. Therefore, we loosen the requirement that the annihilations coincide with the ends of the variable's lifetimes, resulting in an approximation of the cost function, in order to keep the number of stop operations small.

So, given a signal flow graph, an extended signal flow graph has to be created

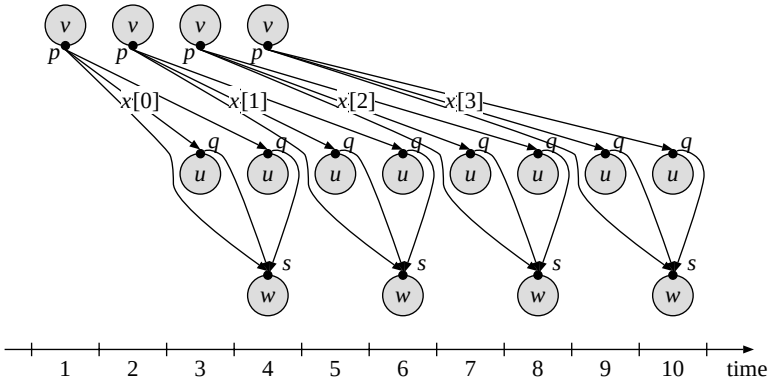


Figure 5.12. The executions of the first global period of the extended signal flow graph of Figure 5.11. The executions of the stop port *s* determine the ends of the variables' lifetimes.

that satisfies the one-to-one condition between productions and annihilations, and which is such that

(i) for every possible time assignment for the original operations a time assignment for the stop operations exists such that they define the ends of the variables' lifetimes as well as possible, and such that

(ii) the number of stop operations is small, in order to keep a tractable problem instance.

Objective (i) is called the *exactness objective*, and objective (ii) is called the *tractability objective*. Unfortunately, the exactness objective and the tractability objective are contradictory in general. Therefore, we present a heuristic approach to construct an extended signal flow graph, which proceeds with the following two steps. First, we construct an initial extended signal flow graph by introducing for each output port a stop operation, such that the one-to-one condition holds. Secondly, we iteratively modify the set of stop operations in order to improve the exactness. To this end, we apply two kinds of transformations, called *domain splitting* and *dimension splitting*. To determine when they should be applied, we use two heuristics that keep the number of stop operations limited.

Initial Extended Signal Flow Graph

In the first step we construct an initial extended signal flow graph by introducing for each output port $p \in P$ a stop operation that is a copy of the production. More formally, for each $p \in P$ we introduce a stop operation w, with stop port s, and

- iterator bound vector $\boldsymbol{I}(w) = \boldsymbol{I}(p)$,

- index matrix $\boldsymbol{A}(s) = \boldsymbol{A}(p)$, and
- index offset vector $\boldsymbol{b}(s) = \boldsymbol{b}(p)$.

Furthermore, we add a stop edge (p,s) to E_s, and for each $q \in I$ with $(p,q) \in E$ we add a stop edge (q,s) to E_s.

Domain Splitting

Given a stop operation, we can split the domain $\{0,...,I_k\}$ of an iterator i_k into two domains $\{0,...,I\}$ and $\{I+1,...,I_k\}$. This results in two new stop operations.

More formally, given a stop operation w with stop port s, a domain split is specified by a number $k \in \{0,...,\delta(w) \Leftrightarrow 1\}$ and a number $I \in \{0,...,I_k(w) \Leftrightarrow 1\}$. This domain split results in two new stop operations w' and w'', with stop ports s' and s'', respectively, and

- $\delta(w') = \delta(w)$,
- $I_l(w') = \begin{cases} I & \text{for } l = k \\ I_l(w) & \text{for } l \neq k, \end{cases}$
- $\boldsymbol{A}(s') = \boldsymbol{A}(s)$,
- $\boldsymbol{b}(s') = \boldsymbol{b}(s)$,
- $\delta(w'') = \delta(w)$,
- $I_l(w'') = \begin{cases} I_k(w) \Leftrightarrow I \Leftrightarrow 1 & \text{for } l = k \\ I_l(w) & \text{for } l \neq k, \end{cases}$
- $\boldsymbol{A}(s'') = \boldsymbol{A}(s)$, and
- $\boldsymbol{b}(s'') = \boldsymbol{b}(s) + (I+1)\boldsymbol{A}_{\cdot k}(s)$.

Note that the domain $\{I+1,...,I_k(w)\}$ of w'' is shifted in order to have lower bound 0 for the corresponding iterator.

For example, consider a stop operation w with corresponding stop port s, given by

$$\delta(w) = 2, \quad \boldsymbol{I}(w) = \begin{bmatrix} \infty \\ 4 \end{bmatrix}, \quad \boldsymbol{A}(s) = \begin{bmatrix} 5 & 1 \end{bmatrix}, \quad \boldsymbol{b}(s) = \begin{bmatrix} 0 \end{bmatrix}.$$

Splitting domain $k = 1$ after execution $I = 2$, i.e., splitting the domain $\{0,...,4\}$ of iterator i_1 into $\{0,...,2\}$ and $\{3,...,4\}$, results in two new stop operations given by

$$\delta(w') = 2, \quad \boldsymbol{I}(w') = \begin{bmatrix} \infty \\ 2 \end{bmatrix}, \quad \boldsymbol{A}(s') = \begin{bmatrix} 5 & 1 \end{bmatrix}, \quad \boldsymbol{b}(s') = \begin{bmatrix} 0 \end{bmatrix},$$

$$\delta(w'') = 2, \ \boldsymbol{I}(w'') = \begin{bmatrix} \infty \\ 1 \end{bmatrix}, \ \boldsymbol{A}(s'') = \begin{bmatrix} 5 & 1 \end{bmatrix}, \ \boldsymbol{b}(s'') = \begin{bmatrix} 0 \end{bmatrix} + 3\begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} 3 \end{bmatrix}.$$

Figure 5.13 shows a graphical representation of this transformation. The figure only shows the executions with $i_0 = 0$.

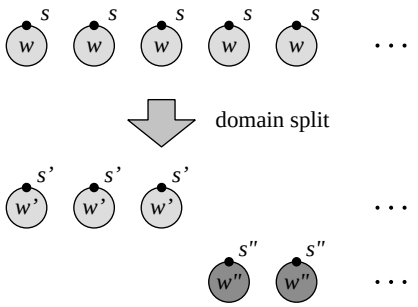


Figure 5.13. An example of domain splitting.

Heuristics for Domain Splitting

By considering a stop operation w with stop port s and an input port q for which $(q,s) \in E_s$, we perform a domain split for iterator i_k if its minimum or maximum value for which a dependency exists is not equal to its lower bound 0 or its upper bound $I_k(w)$, respectively. So, for each $k = 0,...,\delta(w) \Leftrightarrow 1$, we determine

$$\begin{array}{rll} l_k = & \text{minimum} & i_k \\ & \text{subject to} & \boldsymbol{n}(s,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \\ & & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(s) \\ & & \boldsymbol{0} \leq \boldsymbol{j} \leq \boldsymbol{I}(q) \\ & & \boldsymbol{i},\boldsymbol{j} \text{ integer}, \end{array}$$

and we determine

$$\begin{array}{rll} u_k = & \text{maximum} & i_k \\ & \text{subject to} & \boldsymbol{n}(s,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j}) \\ & & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(s) \\ & & \boldsymbol{0} \leq \boldsymbol{j} \leq \boldsymbol{I}(q) \\ & & \boldsymbol{i},\boldsymbol{j} \text{ integer}. \end{array}$$

If $l_k > 0$, then we know that iterator i_k of the stop operation w is not equal to any value in the range $\{0,...,l_k \Leftrightarrow 1\}$, considering the common array elements of input port q and stop port s. Therefore, we split the domain $\{0,...,I_k(w)\}$ of iterator i_k into $\{0,...,l_k \Leftrightarrow 1\}$ and $\{l_k,...,I_k(w)\}$. Similarly, if $u_k < I_k(w)$, then we know that i_k is not equal to any value in the range $\{u_k+1,...,I_k(w)\}$, considering the common array elements of input port q and stop port s. In this case we split the domain $\{0,...,I_k(w)\}$ of iterator i_k into $\{0,...,u_k\}$ and $\{u_k+1,...,I_k(w)\}$. If both $l_k > 0$ and $l_k < I_k(w)$, then two domain splits are performed, resulting in the three domains $\{0,...,l_k \Leftrightarrow 1\}$, $\{l_k,...,u_k\}$, and $\{u_k+1,...,I_k(w)\}$.

Dimension Splitting

Given a stop operation, we can split a dimension k with $I_k + 1$ iterations into two dimensions with $I+1$ and $J+1$ iterations, respectively, with $(I+1)(J+1) = I_k+1$. This means that we substitute an iterator i_k by $(I+1)i'_k + i'_{k+1}$, with $i'_k \in \{0,...,J\}$ and $i'_{k+1} \in \{0,...,I\}$. Note that we assume that $I+1$ divides I_k+1; if not, then we first apply a domain split.

More formally, given a stop operation w with stop port s, a dimension split is specified by a number $k \in \{0,...,\delta(w) \Leftrightarrow 1\}$ and a number $I \in \{1,...,I_k(w) \Leftrightarrow 1\}$ for which $(I+1)|(I_k(w)+1)$. Here we define $(I+1)|\infty$ for any I. The dimension split results in a new stop operation w', with stop port s', and

- $\delta(w') = \delta(w) + 1$,
- $I_l(w') = \begin{cases} I_l(w) & \text{for } l = 0,...,k \Leftrightarrow 1 \\ (I_k(w)+1)/(I+1) \Leftrightarrow 1 & \text{for } l = k \\ I & \text{for } l = k+1 \\ I_{l-1}(w) & \text{for } l = k+2,...,\delta(w') \Leftrightarrow 1, \end{cases}$
- $\boldsymbol{A}_{\cdot l}(s') = \begin{cases} \boldsymbol{A}_{\cdot l}(s) & \text{for } l = 0,...,k \Leftrightarrow 1 \\ (I+1)\boldsymbol{A}_{\cdot k}(s) & \text{for } l = k \\ \boldsymbol{A}_{\cdot (l-1)}(s) & \text{for } l = k+1,...,\delta(w') \Leftrightarrow 1, \text{ and} \end{cases}$
- $\boldsymbol{b}(s') = \boldsymbol{b}(s)$.

Here, we define $(\infty+1)/(I+1) \Leftrightarrow 1 = \infty$, in case $k = 0$.

For example, consider a stop operation w with corresponding stop port s, given by

$$\delta(w) = 2, \quad \boldsymbol{I}(w) = \begin{bmatrix} \infty \\ 5 \end{bmatrix}, \quad \boldsymbol{A}(s) = \begin{bmatrix} 6 & 1 \end{bmatrix}, \quad \boldsymbol{b}(s) = \begin{bmatrix} 0 \end{bmatrix}.$$

Splitting dimension $k = 1$ with $I = 1$, i.e., splitting the dimension *by a factor* 2, results in a new stop operation given by

$$\delta(w') = 3, \quad \boldsymbol{I}(w') = \begin{bmatrix} \infty \\ 2 \\ 1 \end{bmatrix}, \quad \boldsymbol{A}(s') = \begin{bmatrix} 6 & 2 & 1 \end{bmatrix}, \quad \boldsymbol{b}(s') = \begin{bmatrix} 0 \end{bmatrix}.$$

Figure 5.14 shows a graphical representation of this transformation. The figure only shows the executions with $i_0 = 0$.

Heuristics for Dimension Splitting

By considering a stop operation w with stop port s and an input port q for which $(q,s) \in E_s$, we perform a dimension split if the regularity of the annihilation is different to that of the consumption. To this end, we compare the index expressions of stop port s with those of input port q.

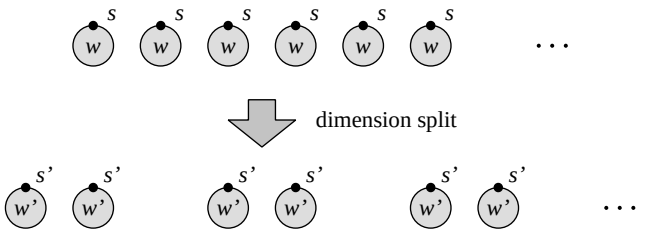


Figure 5.14. An example of dimension splitting.

For example, consider the consuming operation u, with input port q, and the stop operation w, with stop port s, of the algorithm part of Figure 5.15. If we inspect

```
for i0 = 0 to ∞
    for i1 = 0 to 3
        for i2 = 0 to 1
            = u(x[i0][2i1 + i2])
for j0 = 0 to ∞
    for j1 = 0 to 7
        = w(x[j0][j1])
```

Figure 5.15. A part of a video algorithm with consuming operation u and stop operation w, with different regularity.

the index expression $n_1 = 2i_1 + i_2$ at the consumption side, and compare it with $n_1 = j_1$ at the annihilation side, then we observe that if i_1 increases by 1, j_1 should increase by 2. Therefore, it is useful to split the dimension of j_1 by a factor 2, i.e., to substitute j_1 by $2j_1' + j_2'$, with $j_1' \in \{0,...,3\}$ and $j_2' \in \{0,...,1\}$.

In general, we compare the index expressions for each array index n_m, $m = 0,...,\alpha(q) \Leftrightarrow 1$. Given an iterator i_k of the consuming operation u and an iterator j_l of the stop operation w, we compare their factors in the expression for n_m, which are given by $d = A_{mk}(q)$ and $e = A_{ml}(s)$, respectively. Let $f = \mathrm{lcm}(d,e)$, i.e., f is the common regularity of the iterators i_k and j_l in index n_m. In order to obtain this regularity in the annihilation, j_l should be able to make steps of f/e iterations. So, we split dimension l of operation w by a factor f/e. If $f = e$ or if $f/e > I_l(w)$, we do not split the dimension, since then it is void.

Example 5.1. As an example to illustrate all steps in the extended signal flow graph construction, consider the video algorithm of Figure 5.16. First we construct an initial extended signal flow graph, by introducing for each output port a stop operation. In the example, we have only one output port, namely that of operation v, so we obtain one stop operation w. The video algorithm corresponding to the initial extended signal flow graph is given in Figure 5.17. After this, a dimension split by a factor 2

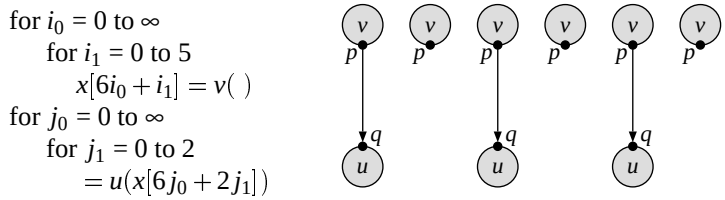


Figure 5.16. An example of a video algorithm to which we apply signal flow graph extension, and the executions in the first global period.

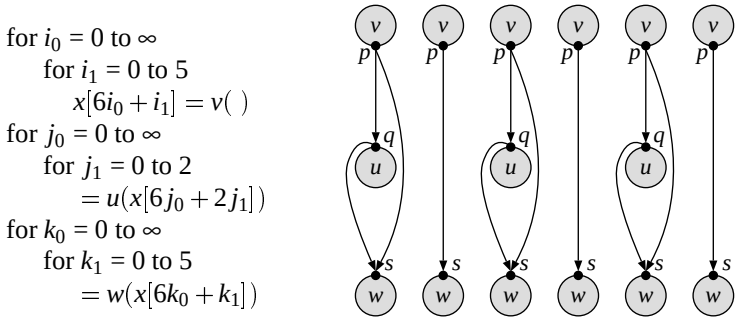


Figure 5.17. The video algorithm corresponding to the initial extended signal flow graph. Operation w is the added stop operation.

is performed on k_1 since at the annihilation side the index n_0 contains the term k_1, but at the consumption side it contains the term $2j_1$. The result is shown in Figure 5.18. Next, a domain split is performed on iterator k_2, since its maximum value is

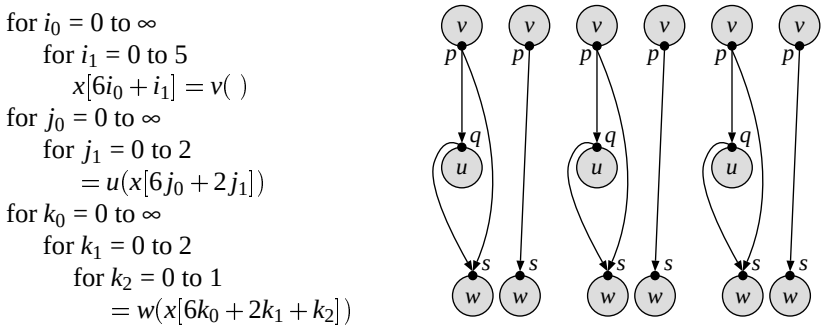


Figure 5.18. The video algorithm after a dimension split.

0, considering the array elements that are consumed by operation u. So, its domain $\{0,1\}$ is split into $\{0\}$ and $\{1\}$, giving rise to two new stop operations w' and w''. The result of this domain split is shown in Figure 5.19. The iterators k_2 of opera-

```
for i0 = 0 to ∞
    for i1 = 0 to 5
        x[6i0 + i1] = v( )
for j0 = 0 to ∞
    for j1 = 0 to 2
        = u(x[6j0 + 2j1])
for k0 = 0 to ∞
    for k1 = 0 to 2
        for k2 = 0 to 0
            = w'(x[6k0 + 2k1 + k2])
for k0 = 0 to ∞
    for k1 = 0 to 2
        for k2 = 0 to 0
            = w''(x[6k0 + 2k1 + k2 + 1])
```

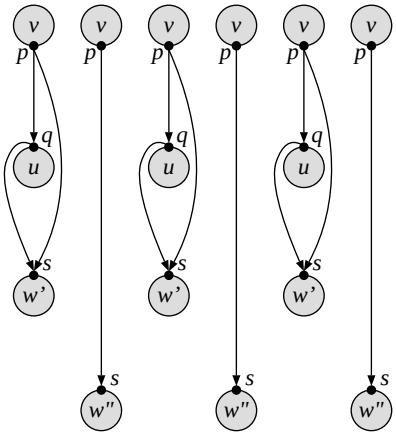


Figure 5.19. The video algorithm after a domain split.

tion w' and w'' can be omitted since their domains contain only one value. Now the heuristics do not apply anymore, so we are finished. As we can see, the stop operation w' annihilates the array elements that are consumed by operation u, and stop operation w'' annihilates the array elements that are not consumed. Operation w' can now be scheduled to coincide with the consumptions, and operation w'' can be scheduled to coincide with the productions. In this way, the stop operations exactly define the ends of the variables' lifetimes.

□

6

Constraint Calculations

In Chapter 2, the constraints of multidimensional periodic scheduling are defined, which are given by the timing constraints, the processing unit constraints, and the precedence constraints, as formulated in Definitions 2.4, 2.5, and 2.6, respectively. Furthermore, in order to determine the access cost, Chapter 5 introduced a terminal assignment, which has to satisfy the access constraints as formulated in Definition 5.3. In this chapter, we discuss how we use the primal all-integer algorithm of Chapter 4 to check the processing unit constraints, the access constraints, and the precedence constraints. The timing constraints, given in Definition 2.4, can be checked straightforwardly, and are therefore not discussed in this chapter.

6.1 Processing Unit Constraints

In this section we discuss how we determine whether or not two operations have a processing unit conflict if they are assigned to the same processing unit, for a given time assignment. So, we are given a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ and a time assignment $\tau = (\boldsymbol{p}, s)$. Furthermore, two operations $u, v \in V$ of the same type $t(u) = t(v) = t$ are given. The question is to determine whether there is an execution $\boldsymbol{i}$ of u and an execution $\boldsymbol{j}$ of v, with $(u, \boldsymbol{i}) \neq (v, \boldsymbol{j})$ and

$$c(u, \boldsymbol{i}) + x = c(v, \boldsymbol{j}) + y,$$

for certain $x, y \in \{0, \ldots, o(t) \Leftrightarrow 1\}$.

We make a distinction between two cases. The first case is $u \neq v$, i.e., we have two different operations, and the second case is $u = v$, i.e., we look for a conflict between two different executions of the same operation.

6.1.1 Two Different Operations

If we have two different operations, the above problem is equivalent to PUC. So, applying the reformulation of Section 3.1.1, we are given an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^\delta$, and a positive integer s. The question is to determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{aligned} &\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ &\mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ &\boldsymbol{i} \text{ integer}. \end{aligned} \tag{6.1}$$

In order to use the primal all-integer algorithm we rewrite this problem into a maximization problem, given by

$$\begin{array}{ll} \text{maximize} & \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \Leftrightarrow s \\ \text{subject to} & \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \leq s \\ & \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ & \boldsymbol{i} \text{ integer}. \end{array} \tag{6.2}$$

This maximum equals zero if and only if (6.1) has a solution. This maximization problem can again be rewritten into the form

$$\begin{array}{lll} \text{maximize} & x_0 & \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}_{\text{puc}}\,\boldsymbol{t} & \\ & x_i \geq 0 & (i \geq 1) \\ & \boldsymbol{x} \text{ integer}, & \end{array} \tag{6.3}$$

by defining

$$\boldsymbol{M}_{\text{puc}} = \left[\begin{array}{c|ccc} \Leftrightarrow s & \Leftrightarrow p_0 & \cdots & \Leftrightarrow p_{\delta-1} \\ \hline s & p_0 & \cdots & p_{\delta-1} \\ \hline 0 & \Leftrightarrow 1 & & \\ \vdots & & \ddots & \\ 0 & & & \Leftrightarrow 1 \\ \hline I_0 & 1 & & \\ \vdots & & \ddots & \\ I_{\delta-1} & & & 1 \\ \hline T & 1 & \cdots & 1 \end{array}\right]. \tag{6.4}$$

Here, $T = \sum_{k=0}^{\delta-1} I_k$ is the sum of the iterator bounds, and the last row of the matrix is used as the reference row for the primal all-integer algorithm. Note that $\boldsymbol{M}_{\text{puc}}$ is primal feasible, as is required for the algorithm. In the rewriting step from (6.2) to

(6.3), we have chosen as initial non-basic variables

$$\boldsymbol{t} = \begin{bmatrix} 1 \\ \Leftrightarrow t_1 \\ \vdots \\ \Leftrightarrow t_n \end{bmatrix} = \begin{bmatrix} 1 \\ \Leftrightarrow i_0 \\ \vdots \\ \Leftrightarrow i_{\delta-1} \end{bmatrix} = \begin{bmatrix} 1 \\ \Leftrightarrow \boldsymbol{i} \end{bmatrix}.$$

Then the objective function is given by

$$x_0 = \Leftrightarrow s + \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i},$$

and the constraints are given by

$$\begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} s \Leftrightarrow \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \\ \boldsymbol{i} \\ \boldsymbol{I} \Leftrightarrow \boldsymbol{i} \\ T \Leftrightarrow \mathbf{1}^{\mathrm{T}}\boldsymbol{i} \end{bmatrix} \geq \mathbf{0},$$

where $\mathbf{1}$ is the all-one vector.

Theorem 6.1. *Starting with a matrix $\boldsymbol{M}_{\mathrm{puc}}$ as given in (6.4), the primal all-integer algorithm solves PUCDP in at most δ iterations.*

Proof. In an instance of PUCDP, the period vector $\boldsymbol{p}$ is sorted in non-increasing order, and $p_{k+1} | p_k$, for all $k = 0,\ldots,\delta \Leftrightarrow 2$. Starting with $\boldsymbol{M}^0 = \boldsymbol{M}_{\mathrm{puc}}$, we show by induction that the matrix $\boldsymbol{M}^h$ in iteration h is of the form

$$\boldsymbol{M}^h = \left[\begin{array}{c|ccccccc} \Leftrightarrow s' & p'_0 & \cdots & p'_{l-1} & \Leftrightarrow p_l & \cdots & \Leftrightarrow p_{\delta-1} \\ \hline s' & \Leftrightarrow p'_0 & \cdots & \Leftrightarrow p'_{l-1} & p_l & \cdots & p_{\delta-1} \\ \hline * & 1 & & & & & \\ \vdots & * & \ddots & & & & \\ * & * & * & 1 & & & \\ 0 & & & & \Leftrightarrow 1 & & \\ \vdots & & & & & \ddots & \\ 0 & & & & & & \Leftrightarrow 1 \\ \hline * & \Leftrightarrow 1 & & & & & \\ \vdots & * & \ddots & & & & \\ * & * & * & \Leftrightarrow 1 & & & \\ I_l & & & & 1 & & \\ \vdots & & & & & \ddots & \\ I_{\delta-1} & & & & & & 1 \\ \hline T' & * & \cdots & * & 1 & \cdots & 1 \end{array}\right], \qquad (6.5)$$

with $l \geq h$. In this matrix, a $*$ stands for any value, and blank entries denote zeros. Furthermore, $p'_k = p_k \vee p'_k = 0$, for all $k = 0,\ldots,l \Leftrightarrow 1$.

Obviously, $\boldsymbol{M}^h$ is of the above form for $h = 0$, since $\boldsymbol{M}^0 = \boldsymbol{M}_{\text{puc}}$. Now suppose $\boldsymbol{M}^h$ is of the above form for $h = a$, then it is also for $h = a+1$, which can be seen as follows.

For matrix $\boldsymbol{M}^a$, the pivot column is column $l+1$, i.e., the column with entry $\Leftrightarrow p_l$ in the cost row. This column results in the lexicographically minimal $\boldsymbol{M}^*_{\cdot j}$, which is due to the fact that $\boldsymbol{p}$ is sorted in non-increasing order and the reference row contains ones in the eligible columns $l+1,\ldots,\delta$.

Next, the pivot row is either row 1, i.e., with pivot element p_l, or row $2+\delta+l$, i.e., with pivot element 1. The last row is not the pivot row since row $2+\delta+l$ is more constraining.

If the pivot row is row 1, then column $l+1$ is subtracted an integer number of times from column 0, and it is added an integer number of times to column $1,\ldots,l$. Because of the divisibility of the periods, this results $M^{a+1}_{ij} = 0$, for row $i = 0,1$ and column $j = 1,\ldots,l$. Furthermore, if we let $k \geq l$ be the maximal index with $p_k = p_l$, then also $M^{a+1}_{ij} = 0$, for $i = 0,1$ and $j = l+2,\ldots,k+1$. Columns $k+2,\ldots,\delta$ remain unchanged. Finally, column $l+1$ is multiplied by $\Leftrightarrow 1$. Now, we can conclude that the matrix $\boldsymbol{M}^{a+1}$ has again the form of (6.5), with $l \geq a+1$.

If the pivot row is row $2+\delta+l$, then column $l+1$ is subtracted I_l times from column 0, and column $l+1$ is multiplied by $\Leftrightarrow 1$. Also in this case $\boldsymbol{M}^{a+1}$ has the form of (6.5), this time with l increased by 1.

From the previous we can conclude that in at most δ iterations the matrix $\boldsymbol{M}$ has only entries $p'_k \geq 0$ in the cost row, and thus has become dual feasible. So, the algorithm terminates with an optimal solution within δ iterations. □

From the proof we can derive that the primal all-integer algorithm determines for instances of PUCDP the same solution as obtained by means of (3.6), in which the iterator values are also determined in order of non-increasing period.

6.1.2 One Operation

In the case of one operation, we look for a conflict between two different executions of it. So, we have the following problem.

Definition 6.1 (Single Operation PU Conflict (SOPUC)). Given are an operation with an iterator bound vector $\boldsymbol{I} \in \mathbb{N}^{\delta}_{\infty}$, for which $I_k \neq \infty$ for $k > 0$. Furthermore, a period vector $\boldsymbol{p} \in \mathbb{Z}^{\delta}$, a start time $s \in \mathbb{Z}$, and an occupation time $o \in \mathbb{N}_+$ are given. Determine whether there are vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ and numbers x and y that

satisfy

$$\begin{array}{c} \boldsymbol{i} \neq \boldsymbol{j} \\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + s + x = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{j} + s + y \\ \boldsymbol{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I} \\ 0 \leq x, y \leq o \Leftrightarrow 1 \\ \boldsymbol{i}, \boldsymbol{j}, x, y \text{ integer.} \end{array} \tag{6.6}$$

□

Before defining an ILP matrix $\boldsymbol{M}$ in order to solve SOPUC with the primal all-integer algorithm, we reformulate SOPUC. To this end, we first derive a finite upper bound for the infinite repetition, if $I_0(v) = \infty$, analogously to Section 3.1.1.

If an instance of SOPUC has a solution, then there is a solution with $i_0 = 0$ or $j_0 = 0$. If $i_0 = 0$, then from (6.6) we derive

$$\begin{aligned} p_0 j_0 &= \sum_{k=1}^{\delta-1} p_k i_k + x \Leftrightarrow \sum_{l=1}^{\delta-1} p_l j_l \Leftrightarrow y \\ &\leq \sum_{k=1}^{\delta-1} p_k^+ i_k + x \Leftrightarrow \sum_{l=1}^{\delta-1} p_l^- j_l \Leftrightarrow y \\ &\leq \sum_{k=1}^{\delta-1} p_k^+ I_k + o \Leftrightarrow 1 \Leftrightarrow \sum_{l=1}^{\delta-1} p_l^- I_l \\ &= \sum_{k=1}^{\delta-1} |p_k| I_k + o \Leftrightarrow 1. \end{aligned}$$

So, j_0 is bounded by

$$b = \left\lfloor \frac{\sum_{k=1}^{\delta-1} |p_k| I_k + o \Leftrightarrow 1}{p_0} \right\rfloor.$$

Similarly, if $j_0 = 0$, we can derive that i_0 is bounded by b. So, if $I_0 = \infty$, we can replace it by b.

Next, we rewrite (6.6) into

$$\begin{array}{c} \boldsymbol{i} \neq \boldsymbol{j} \\ \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{i} \Leftrightarrow \boldsymbol{j}) + x \Leftrightarrow y = 0 \\ \boldsymbol{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I} \\ 0 \leq x, y \leq o \Leftrightarrow 1 \\ \boldsymbol{i}, \boldsymbol{j}, x, y \text{ integer.} \end{array} \tag{6.7}$$

Now, without loss of generality, we may assume that $p_k > 0$, for all $k = 0,\ldots,\delta \Leftrightarrow 1$, and that $\boldsymbol{p}$ is sorted in non-increasing order. Furthermore, we may assume that $I_k > 0$, for all $k = 0,\ldots,\delta \Leftrightarrow 1$. Next, without loss of generality, we may replace $\boldsymbol{i} \neq \boldsymbol{j}$ by

$\boldsymbol{i} <_{\text{lex}} \boldsymbol{j}$, which in turn is equivalent to $\boldsymbol{f}^{\mathrm{T}}\boldsymbol{i} < \boldsymbol{f}^{\mathrm{T}}\boldsymbol{j}$, with $\boldsymbol{f} \in \mathbb{N}_+^\delta$ defined by

$$f_k = \prod_{l=k+1}^{\delta-1} (I_l + 1), \tag{6.8}$$

for $k = 0,\ldots,\delta \Leftrightarrow 1$. Now, (6.7) can be rewritten to

$$\begin{gathered} \boldsymbol{f}^{\mathrm{T}}(\boldsymbol{i} \Leftrightarrow \boldsymbol{j}) \leq \Leftrightarrow 1 \\ \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{i} \Leftrightarrow \boldsymbol{j}) + x \Leftrightarrow y = 0 \\ \boldsymbol{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I} \\ 0 \leq x, y \leq o \Leftrightarrow 1 \\ \boldsymbol{i}, \boldsymbol{j}, x, y \text{ integer}. \end{gathered}$$

A final reformulation is to substitute $\boldsymbol{i} \Leftrightarrow \boldsymbol{j}$ by $\boldsymbol{i}' \Leftrightarrow \boldsymbol{I}$ and $x \Leftrightarrow y$ by $x' \Leftrightarrow o + 1$. Now we can see that SOPUC is equivalent to the following problem.

Definition 6.2 (SOPUC Reformulated). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^\delta$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^\delta$, sorted in non-increasing order, and an occupation time $o \in \mathbb{N}_+$. Determine whether there are a vector $\boldsymbol{i}$ and a number x that satisfy

$$\begin{gathered} \boldsymbol{f}^{\mathrm{T}}\boldsymbol{i} \leq \boldsymbol{f}^{\mathrm{T}}\boldsymbol{I} \Leftrightarrow 1 \\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + x = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{I} + o \Leftrightarrow 1 \\ \boldsymbol{0} \leq \boldsymbol{i} \leq 2\boldsymbol{I} \\ 0 \leq x \leq 2o \Leftrightarrow 2 \\ \boldsymbol{i}, x \text{ integer}, \end{gathered} \tag{6.9}$$

where $\boldsymbol{f} \in \mathbb{N}_+^\delta$ is defined by (6.8). □

In order to use the primal all-integer algorithm to solve SOPUC, we rewrite it into a maximization problem, given by

$$\begin{array}{ll} \text{maximize} & \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \Leftrightarrow \boldsymbol{p}^{\mathrm{T}}\boldsymbol{I} \Leftrightarrow o + 1 \\ \text{subject to} & \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \leq \boldsymbol{p}^{\mathrm{T}}\boldsymbol{I} + o \Leftrightarrow 1 \\ & \boldsymbol{f}^{\mathrm{T}}\boldsymbol{i} \leq \boldsymbol{f}^{\mathrm{T}}\boldsymbol{I} \Leftrightarrow 1 \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq 2\boldsymbol{I} \\ & \boldsymbol{i} \text{ integer}. \end{array}$$

Let z be the optimum of this problem, then by taking $x = \Leftrightarrow z$ we can see that SOPUC has a solution if and only if $\Leftrightarrow z \leq 2o \Leftrightarrow 2$. Now we can rewrite the above maximization problem into the form

$$\begin{array}{lll} \text{maximize} & x_0 & \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}_{\text{sopuc}}\, \boldsymbol{t} & \\ & x_i \geq 0 & (i \geq 1) \\ & \boldsymbol{x} \text{ integer}, & \end{array} \tag{6.10}$$

by defining

$$\boldsymbol{M}_{\text{sopuc}} = \left[\begin{array}{c|ccc} \Leftrightarrow \boldsymbol{p}^{\mathrm{T}}\boldsymbol{I} \Leftrightarrow o+1 & \Leftrightarrow p_0 & \cdots & \Leftrightarrow p_{\delta-1} \\ \hline \boldsymbol{p}^{\mathrm{T}}\boldsymbol{I}+o \Leftrightarrow 1 & p_0 & \cdots & p_{\delta-1} \\ \hline \boldsymbol{f}^{\mathrm{T}}\boldsymbol{I} \Leftrightarrow 1 & f_0 & \cdots & f_{\delta-1} \\ \hline 0 & \Leftrightarrow 1 & & \\ \vdots & & \ddots & \\ 0 & & & \Leftrightarrow 1 \\ \hline 2I_0 & 1 & & \\ \vdots & & \ddots & \\ 2I_{\delta-1} & & & 1 \\ \hline 2T & 1 & \cdots & 1 \end{array}\right]. \tag{6.11}$$

Again, the last row of the matrix is used as the reference row of the primal all-integer algorithm, with $T = \mathbf{1}^{\mathrm{T}}\boldsymbol{I}$ being the sum of the iterator bounds. Note that if $\delta > 0$, then $\boldsymbol{f}^{\mathrm{T}}\boldsymbol{I} \geq 1$, and thus $\boldsymbol{M}_{\text{sopuc}}$ is primal feasible. If $\delta = 0$, then no solution exists.

The next theorem shows that a lexicographical execution is a sufficient condition for an instance of SOPUC to be a no-instance.

Theorem 6.2. *An instance $(\boldsymbol{I}, \boldsymbol{p}, o)$ of SOPUC is a no-instance if*

$$\operatorname{lex}(\boldsymbol{I}, \boldsymbol{p}, o)$$

holds.

Proof. The proof is by contradiction. Let $(\boldsymbol{i}, x)$ satisfy (6.9), then $\boldsymbol{i} <_{\text{lex}} \boldsymbol{I}$ and thus $\operatorname{lex}(\boldsymbol{I}, \boldsymbol{p}, o)$ gives $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + o \leq \boldsymbol{p}^{\mathrm{T}}\boldsymbol{I}$. However, then

$$x = \boldsymbol{p}^{\mathrm{T}}\boldsymbol{I} \Leftrightarrow \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} + o \Leftrightarrow 1 \geq o + o \Leftrightarrow 1 > 2o \Leftrightarrow 2,$$

which contradicts the assumption that $(\boldsymbol{i}, x)$ is a solution. □

In practice, most operations have a lexicographical execution, allowing the above theorem to be applied.

6.2 Access Constraints

In this section we discuss how we determine whether or not two ports have an access conflict if they are assigned to the same memory terminal, for a given time assignment. So, we are given a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ and a time assignment $\tau = (\boldsymbol{p}, s)$. Furthermore, two ports $p, q \in P$ are given. The question is whether there is an execution $\boldsymbol{i}$ of p and an execution $\boldsymbol{j}$ of q, with

$$c(p, \boldsymbol{i}) = c(q, \boldsymbol{j}),$$

but $p \not\bowtie q$ or $\boldsymbol{n}(p, \boldsymbol{i}) \neq \boldsymbol{n}(q, \boldsymbol{j})$.

We make a distinction between two cases. The first case is $p \not\bowtie q$, i.e., p and q belong to different array clusters, and the second case is $p \bowtie q$, i.e., we look for simultaneous accesses of two different elements of the same array cluster.

6.2.1 Two Different Array Clusters

If p and q access different arrays, then there is an access conflict whenever an execution of p and q coincide. In this case, the problem is equivalent to the processing unit conflict problem (PUC) with occupation time 1.

6.2.2 One Array Cluster

In case $p \bowtie q$, the above problem can be formulated as follows.

Definition 6.3 (Access Conflict (AC)). Given are two ports p and q, and their iterator bound vectors $\boldsymbol{I}(p) \in \mathbb{N}_\infty^{\delta(p)}$, $\boldsymbol{I}(q) \in \mathbb{N}_\infty^{\delta(q)}$, for which $I_k(p), I_k(q) \neq \infty$ for all $k > 0$. Furthermore, their period vectors $\boldsymbol{p}(p) \in \mathbb{Z}^{\delta(p)}$, $\boldsymbol{p}(q) \in \mathbb{Z}^{\delta(q)}$, their start times $s(p), s(q) \in \mathbb{Z}$, their index matrices $\boldsymbol{A}(p) \in \mathbb{Z}^{\alpha(p)\times\delta(p)}$, $\boldsymbol{A}(q) \in \mathbb{Z}^{\alpha(q)\times\delta(q)}$, and their index offset vectors $\boldsymbol{b}(p) \in \mathbb{Z}^{\alpha(p)}$, $\boldsymbol{b}(q) \in \mathbb{Z}^{\alpha(q)}$ are given. Determine whether there are vectors $\boldsymbol{i}$ and $\boldsymbol{j}$ that satisfy

$$\begin{aligned} \boldsymbol{A}(p)\,\boldsymbol{i} + \boldsymbol{b}(p) &\neq \boldsymbol{A}(q)\,\boldsymbol{j} + \boldsymbol{b}(q) \\ \boldsymbol{p}^{\mathrm{T}}(p)\,\boldsymbol{i} + s(p) &= \boldsymbol{p}^{\mathrm{T}}(q)\,\boldsymbol{j} + s(q) \\ \mathbf{0} \leq \boldsymbol{i} &\leq \boldsymbol{I}(p) \\ \mathbf{0} \leq \boldsymbol{j} &\leq \boldsymbol{I}(q) \\ \boldsymbol{i}, \boldsymbol{j} &\text{ integer.} \end{aligned} \tag{6.12}$$

□

Before defining an ILP matrix $\boldsymbol{M}$ in order to solve AC with the primal all-integer algorithm, we reformulate the above problem. To this end, we first derive finite upper bounds for the infinite repetitions, if $I_0(p) = \infty$ or $I_0(q) = \infty$, analogously to Section 3.1.1.

If $I_0(p) = \infty$ but $I_0(q) \neq \infty$, then we can derive a finite upper bound on i_0 as follows. From (6.12) we derive

$$\begin{aligned} p_0(p) i_0 &= \sum_{k=0}^{\delta(q)-1} p_k(q)\, j_k + s(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} p_l(p)\, i_l \Leftrightarrow s(p) \\ &\leq \sum_{k=0}^{\delta(q)-1} p_k^+(q)\, j_k + s(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} p_l^-(p)\, i_l \Leftrightarrow s(p) \\ &\leq \sum_{k=0}^{\delta(q)-1} p_k^+(q)\, I_k(q) + s(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} p_l^-(p)\, I_l(p) \Leftrightarrow s(p). \end{aligned}$$

Since $p_0(p) > 0$ by assumption, i_0 is bounded by

$$b(q,p) = \left\lfloor \frac{\sum_{k=0}^{\delta(q)-1} p_k^+(q) I_k(q) + s(q) \Leftrightarrow \sum_{l=1}^{\delta(p)-1} p_l^-(p) I_l(p) \Leftrightarrow s(p)}{p_0(p)} \right\rfloor, \quad (6.13)$$

and thus we can replace $I_0(p)$ by $b^+(q,p)$.

If $I_0(p) \neq \infty$ and $I_0(q) = \infty$, then we can replace $I_0(q)$ by a similar bound $b^+(p,q)$.

If both $I_0(p) = \infty$ and $I_0(q) = \infty$, then we can derive finite upper bounds on i_0 and j_0 as follows. Let $a = \mathrm{lcm}(p_0(p), p_0(q))$. Now, if $\boldsymbol{i}$ and $\boldsymbol{j}$ satisfy (6.12), then $\boldsymbol{i} + \Delta\boldsymbol{i}$ and $\boldsymbol{j} + \Delta\boldsymbol{j}$ also satisfy (6.12), where

$$\Delta\boldsymbol{i} = \frac{a}{p_0(p)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \Delta\boldsymbol{j} = \frac{a}{p_0(q)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

The reason for this is that

$$\boldsymbol{p}^{\mathrm{T}}(p)\, \Delta\boldsymbol{i} = a = \boldsymbol{p}^{\mathrm{T}}(q)\, \Delta\boldsymbol{j},$$

and, based on the assumption $A_{00}(p)/p_0(p) = A_{00}(q)/p_0(q)$,

$$\boldsymbol{A}(p)\, \Delta\boldsymbol{i} = \frac{A_{00}(p)a}{p_0(p)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \frac{A_{00}(q)a}{p_0(q)} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \boldsymbol{A}(q)\, \Delta\boldsymbol{j}.$$

Furthermore, if $\boldsymbol{i}$ and $\boldsymbol{j}$ satisfy (6.12) and $i_0 \geq a/p_0(p)$ and $j_0 \geq a/p_0(q)$, then $\boldsymbol{i} \Leftrightarrow \Delta\boldsymbol{i}$ and $\boldsymbol{j} \Leftrightarrow \Delta\boldsymbol{j}$ also satisfy (6.12), i.e., there is a solution with a lower value of i_0 and j_0. From this we can conclude that (6.12) has a solution if and only if it has a solution with

$$i_0 < a/p_0(p) \ \vee \ j_0 < a/p_0(q). \quad (6.14)$$

Now, if $j_0 < a/p_0(q)$, then we can derive an upper bound $\tilde{b}(q,p)$ on i_0 given by (6.13), where we replace $I_0(q)$ by $a/p_0(q) \Leftrightarrow 1$. So, (6.14) implies

$$i_0 < a/p_0(p) \ \vee \ i_0 \leq \tilde{b}(q,p),$$

and thus we can replace $I_0(p)$ by $\max\{a/p_0(p) \Leftrightarrow 1\,,\, \tilde{b}(q,p)\}$. In a similar way we can replace $I_0(q)$ by $\max\{a/p_0(q) \Leftrightarrow 1\,,\, \tilde{b}(p,q)\}$, where $\tilde{b}(p,q)$ is given by (6.13), with $I_0(p)$ replaced by $a/p_0(p) \Leftrightarrow 1$.

Next, we combine all iterators into one vector, and thus combine all iterator bounds into one vector, all periods into one vector, and the two index matrices into one matrix. In this way, we can rewrite AC into the problem to determine whether

there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{array}{c} \boldsymbol{A}\,\boldsymbol{i} \neq \boldsymbol{b} \\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer}. \end{array}$$

Next, we remove iterators with an upper bound equal to 0, and substitute iterators i_k for which $p_k < 0$ by $I_k \Leftrightarrow i'_k$, in order to obtain non-negative periods. Now we can see that AC is equivalent to the following problem.

Definition 6.4 (AC Reformulated). Given are an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{N}^{\delta}$, an integer s, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. Determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{array}{c} \boldsymbol{A}\,\boldsymbol{i} \neq \boldsymbol{b} \\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer}. \end{array}$$

□

In the remainder, we assume that $s \geq 0$, since otherwise it is trivial that no solution exists.

In order to solve AC, we determine

$$\begin{array}{rll} \underline{\boldsymbol{n}} = \text{ minimum}_{\text{lex}} & \boldsymbol{A}\,\boldsymbol{i} \Leftrightarrow \boldsymbol{b} \\ \text{subject to} & \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ & \boldsymbol{i} \text{ integer}, \end{array} \tag{6.15}$$

and we determine

$$\begin{array}{rll} \overline{\boldsymbol{n}} = \text{ maximum}_{\text{lex}} & \boldsymbol{A}\,\boldsymbol{i} \Leftrightarrow \boldsymbol{b} \\ \text{subject to} & \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ & \boldsymbol{i} \text{ integer}. \end{array} \tag{6.16}$$

Now, if these integer linear programming problems are infeasible, or if $\underline{\boldsymbol{n}} = \overline{\boldsymbol{n}} = \boldsymbol{0}$, then there is no access conflict; otherwise there is. In the remainder of this section, we show how we use the multi-cost primal all-integer algorithm to determine $\overline{\boldsymbol{n}}$. Determining $\underline{\boldsymbol{n}}$ is done analogously.

First, we rewrite (6.16) into the maximization problem given by

$$\begin{array}{rll} \tilde{\boldsymbol{n}} = \text{ maximum}_{\text{lex}} & \begin{bmatrix} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \Leftrightarrow s \\ \boldsymbol{A}\,\boldsymbol{i} \Leftrightarrow \boldsymbol{b} \end{bmatrix} \\ \text{subject to} & \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \leq s \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ & \boldsymbol{i} \text{ integer}. \end{array} \tag{6.17}$$

This problem is always feasible, since $s \geq 0$. Now, (6.16) is feasible if and only if $\tilde{n}_0 = 0$. Furthermore, $\overline{\boldsymbol{n}}$ is then given by

$$\overline{\boldsymbol{n}} = \begin{bmatrix} \tilde{n}_1 \\ \tilde{n}_2 \\ \vdots \\ \tilde{n}_\alpha \end{bmatrix}.$$

Next, we rewrite (6.17) into the form

$$\begin{array}{rl} \text{maximize}_{\text{lex}} & \boldsymbol{x}_{\langle \alpha+1 \rangle} \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}_{\text{ac}}\, \boldsymbol{t} \\ & x_i \geq 0 \qquad (i \geq \alpha+1) \\ & \boldsymbol{x} \text{ integer}, \end{array}$$

by defining

$$\boldsymbol{M}_{\text{ac}} = \left[\begin{array}{c|ccc} \Leftrightarrow s & \Leftrightarrow p_0 & \cdots & \Leftrightarrow p_{\delta-1} \\ \hline \Leftrightarrow b_0 & \Leftrightarrow A_{00} & \cdots & \Leftrightarrow A_{0,\delta-1} \\ \vdots & \vdots & & \vdots \\ \Leftrightarrow b_{\alpha-1} & \Leftrightarrow A_{\alpha-1,0} & \cdots & \Leftrightarrow A_{\alpha-1,\delta-1} \\ \hline s & p_0 & \cdots & p_{\delta-1} \\ \hline 0 & \Leftrightarrow 1 & & \\ \vdots & & \ddots & \\ 0 & & & \Leftrightarrow 1 \\ \hline I_0 & 1 & & \\ \vdots & & \ddots & \\ I_{\delta-1} & & & 1 \\ \hline T & 1 & \cdots & 1 \end{array}\right]. \tag{6.18}$$

Again, $T = \mathbf{1}^{\mathrm{T}}\boldsymbol{I}$ is the sum of the iterator bounds, and the last row is used as the reference row for the multi-cost primal all-integer algorithm. As required for the algorithm, the matrix $\boldsymbol{M}_{\text{ac}}$ is primal feasible.

6.3 Precedence Constraints

In this section we discuss how we determine whether or not precedences are met, for a given time assignment. We do this for the precedence constraints concerning data transport; the stop precedence constraints can be handled in the same way. So, we are given a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ and a time assignment $\tau = (\boldsymbol{p}, s)$. Furthermore, an output port $p \in O$ and a connected input port $q \in I$ are given. The question is whether there is an execution $\boldsymbol{i}$ of p and an execution $\boldsymbol{j}$ of q, with $\boldsymbol{n}(p, \boldsymbol{i}) =$

$\boldsymbol{n}(q,\boldsymbol{j})$ and

$$c(p,\boldsymbol{i}) \geq c(q,\boldsymbol{j}).$$

The above problem is equivalent to PC. Applying the reformulation as discussed in Section 3.2.1, gives us an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_+^{\delta}$, a period vector $\boldsymbol{p} \in \mathbb{Z}^{\delta}$, an integer s, an index matrix $\boldsymbol{A} \in \mathbb{Z}^{\alpha \times \delta}$, with lexicographically positive columns, and an index offset vector $\boldsymbol{b} \in \mathbb{Z}^{\alpha}$. The question is to determine whether there is a vector $\boldsymbol{i}$ that satisfies

$$\begin{array}{c} \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \geq s \\ \boldsymbol{A}\,\boldsymbol{i} = \boldsymbol{b} \\ \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ \boldsymbol{i} \text{ integer}. \end{array} \tag{6.19}$$

In order to obtain an ILP matrix $\boldsymbol{M}$ that is primal feasible, we apply the following steps in order to obtain $\boldsymbol{b} \geq \boldsymbol{0}$. We iteratively check b_l, for $l = 0,\ldots,\alpha \Leftrightarrow 1$. If $b_l \geq 0$, we do nothing. If $b_l < 0$, then we multiply array index l by $\Leftrightarrow 1$, i.e., we multiply b_l by $\Leftrightarrow 1$ and we multiply row $A_{l\cdot}$ by $\Leftrightarrow 1$. If after this multiplication a column $\boldsymbol{A}_{\cdot k}$ has become lexicographically negative, which means that then $A_{lk} < 0$ and $A_{jk} = 0$, for all $j = 0,\ldots,l \Leftrightarrow 1$, then we substitute i_k by $I_k \Leftrightarrow i'_k$. This results in a new vector $\boldsymbol{b}' = \boldsymbol{b} \Leftrightarrow I_k \boldsymbol{A}_{\cdot k}$, of which the entries $b'_j = b_j \geq 0$, for all $j = 0,\ldots,l \Leftrightarrow 1$, and $b'_l \geq b_l \geq 0$. This is done for each column $\boldsymbol{A}_{\cdot k}$ that has become lexicographically negative, after which l is incremented. In this way we eventually obtain an instance of PC with $\boldsymbol{b} \geq \boldsymbol{0}$.

Next, we rewrite (6.19) into a maximization problem, given by

$$\begin{array}{rl} \text{maximize}_{\text{lex}} & \begin{bmatrix} \boldsymbol{A}\,\boldsymbol{i} \Leftrightarrow \boldsymbol{b} \\ \boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} \Leftrightarrow s \end{bmatrix} \\ \text{subject to} & \boldsymbol{A}\,\boldsymbol{i} \leq \boldsymbol{b} \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I} \\ & \boldsymbol{i} \text{ integer}. \end{array} \tag{6.20}$$

Let $\boldsymbol{z}$ be the optimum of this problem, then (6.19) has a solution if and only if $z_l = 0$, for all $l = 0,\ldots,\alpha \Leftrightarrow 1$, and $z_\alpha \geq 0$. If (6.19) has a solution, i.e., the precedence constraints are not met, then s should be increased by at least $z_\alpha + 1$ in order to meet the precedence constraints. So, if $s(q) \Leftrightarrow s(p)$ is increased by at least $z_\alpha + 1$, then the precedence constraints are met. In this way, we can determine for given period vectors the minimal difference $s(q) \Leftrightarrow s(p)$ for which the precedence constraints are met.

The maximization problem (6.20) can be rewritten into the form

$$\begin{array}{rl} \text{maximize}_{\text{lex}} & \boldsymbol{x}_{\langle\alpha+1\rangle} \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}_{\text{pc}}\,\boldsymbol{t} \\ & x_i \geq 0 \qquad (i \geq \alpha+1) \\ & \boldsymbol{x} \text{ integer}, \end{array}$$

by defining

$$\boldsymbol{M}_{\text{pc}} = \left[\begin{array}{c|ccc} \not\Leftrightarrow b_0 & \not\Leftrightarrow A_{00} & \cdots & \not\Leftrightarrow A_{0,\delta-1} \\ \vdots & \vdots & & \vdots \\ \not\Leftrightarrow b_{\alpha-1} & \not\Leftrightarrow A_{\alpha-1,0} & \cdots & \not\Leftrightarrow A_{\alpha-1,\delta-1} \\ \hline \not\Leftrightarrow s & \not\Leftrightarrow p_0 & \cdots & \not\Leftrightarrow p_{\delta-1} \\ \hline\hline b_0 & A_{00} & \cdots & A_{0,\delta-1} \\ \vdots & \vdots & & \vdots \\ b_{\alpha-1} & A_{\alpha-1,0} & \cdots & A_{\alpha-1,\delta-1} \\ \hline 0 & \not\Leftrightarrow 1 & & \\ \vdots & & \ddots & \\ 0 & & & \not\Leftrightarrow 1 \\ \hline I_0 & 1 & & \\ \vdots & & \ddots & \\ I_{\delta-1} & & & 1 \\ \hline T & 1 & \cdots & 1 \end{array}\right]. \tag{6.21}$$

Again, $T = \mathbf{1}^{\mathrm{T}}\boldsymbol{I}$ is the sum of the iterator bounds, and the last row is used as the reference row for the multi-cost primal all-integer algorithm. As required for the algorithm, the matrix $\boldsymbol{M}_{\text{pc}}$ is primal feasible.

6.4 Discussion

In this chapter we have presented initial matrices for the primal all-integer algorithm, by means of which the constraints of multidimensional periodic scheduling can be checked. The strength of this application of the all-integer ILP algorithm is in our view based on two observations.

First, the Gomory cuts that are used in the algorithm are well suited for these kinds of problems. For instance, checking whether two one-dimensional periodic operations with an infinite number of executions coincide somewhere in time can be done by means of a test based on the greatest common divisor of the periods, as is shown by Korst [1992]. Euclid's algorithm to determine the greatest common divisor of two numbers exhibits the following behavior. It repeatedly divides the largest number by the smallest one, after which the algorithm continues with the

remainder and the smallest of the original two numbers. This kind of behavior is also revealed when using the Gomory cuts in the all-integer ILP algorithm. Namely, if in the algorithm a row i is selected with element M_{is} in the pivot column s, and a Gomory cut is derived from this row, then after pivoting the elements M_{ij}, for $j \neq s$, are replaced by their remainders after division by M_{is}.

Secondly, in practice the coefficients in the ILP inequalities are in most cases divisible, or nearly divisible. As shown by Theorem 6.1, the all-integer ILP algorithm benefits from this characteristic. A prerequisite is that the coefficients are handled in a non-increasing order, which is accomplished by making all columns of the initial ILP matrices lexicographically negative and by using a row with entries 1 as the reference row for the algorithm.

7

A Multidimensional Periodic Scheduling Algorithm

In this chapter we present a solution approach for the multidimensional periodic scheduling problem, given by Definition 2.9. Since the problem is too complex to be handled in its entirety, we decompose it into two stages. The decomposition is discussed in Section 7.1, after which the two stages are discussed in Sections 7.2 and 7.3.

7.1 Problem Decomposition

The multidimensional periodic scheduling problem is decomposed into two stages, as shown in Figure 7.1. In the first stage we assign period vectors to all operations and in the second stage we assign start times to the operations and assign the operations to processing units. The period assignment takes place before the start time and processing unit assignment, since the periods are needed to check the processing unit constraints. So, in the first stage a partial solution is constructed, which is completed in the second stage. The constraints and the cost function of the problem that is solved in the second stage are therefore the same as in the original multidimensional periodic scheduling problem. For the problem of the first stage, however, we have to define new constraints and a new cost function.

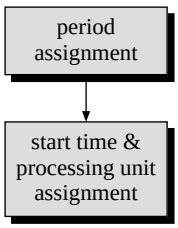


Figure 7.1. The decomposition of the multidimensional periodic scheduling problem into two stages.

In the first stage, we opt for taking the original constraints into account as much as possible. The timing constraints and the processing unit constraints for different executions of the same operation can be taken into account directly, since they only require the periods to be known. Because of the importance of the precedence constraints, we also take them into account during the period assignment stage. To this end, we also determine preliminary start times, which may be altered in the second stage. The inclusion of preliminary start times in the first stage can be done easily, as is shown in Section 7.2. The processing unit constraints for two different operations are not taken into account in the first stage, since otherwise a processing unit assignment needs to be determined.

With respect to cost calculations, we have a similar problem. It is difficult to calculate the total area cost in the first stage. The processing unit cost is difficult to determine since no processing unit assignment is performed. Similarly, it is difficult to determine the access cost. In addition to that, the processing unit cost and access cost do not depend heavily on the periods in practice, since the number of executions of the operations are fixed, and by shifting the start times of the operations in the second stage we may resolve processing unit conflicts, in order to assign operations to the same processing unit.

In the second stage, the periods are given and we determine a start time and a processing unit assignment, for which the total area cost is minimal. In this stage, we possibly deviate from the preliminary start times determined in the first stage, in order to be able to assign more operations to the same processing unit. In practice, we expect that this has only a minor effect on the storage cost, or that the effect is comparable with that of other period assignments. In both cases the optimality of the period assignment is hardly affected.

The problems of the first and second stage are now given by the following definitions.

Definition 7.1 (Period Assignment (PA)). Given are a signal flow graph G, a set of resource types T^*, and lower and upper bounds on the period vectors and start

times of the operations. Find a period assignment $\boldsymbol{p}$, and a start time assignment s, that obey the timing constraints, the precedence constraints, and the processing unit constraints for different executions of the same operation, such that the storage cost $a(t_v)\, f_{\max}(\tau)$ of the resulting time assignment $\tau = (\boldsymbol{p}, s)$ is minimal. □

Definition 7.2 (Fixed Period Multidimensional Periodic Scheduling (FMPS)). Given are a signal flow graph G, a set of resource types T^*, and for each operation a period vector and a lower and upper bound on the start time. Find a start time assignment s and a processing unit assignment (W,h) that obey the timing constraints, the processing unit constraints, and the precedence constraints, such that the total area cost $f(\sigma)$ of the resulting schedule $\sigma = (\boldsymbol{p}, s, W, h)$ is minimal. □

Note that if the period assignment problem has a solution, then there exists a solution of the multidimensional periodic scheduling problem. If however the period assignment problem has no solution, then no solution of the multidimensional periodic scheduling problem exists, irrespective of the number of resources that is used.

7.2 Period Assignment

In this section we present an algorithm for the period assignment problem. Before we do so, we approximate the cost function, and we approximate the processing unit constraints for different executions of the same operation. Next, we present an ILP formulation of the approximate problem, which is solved by means of a branch-and-bound algorithm.

7.2.1 Approximate Cost Function and Constraints

As discussed in Section 5.3, we approximate the cost function $f_{\max}(\tau)$, i.e., the maximum number of variables that are simultaneously alive for a given time assignment, by $f_{\text{stop}}(\tau)/p_{\text{glob}}$, which can be expressed as a linear function in the periods and start times of the operations. To this end, we assume that all operations have an infinite number of executions, and that an extended signal flow graph has been constructed.

The processing unit constraints for different executions of the same operation are substituted by constraints implying lexicographical executions. In practice, this is hardly a restriction. Furthermore, it is in most cases desirable for the synthesis steps of Phideo that follow the scheduling step. Theorem 6.2 shows that a lexicographical execution is a sufficient condition to avoid conflicts. Furthermore, Theorem 2.1 shows that an iterator bound vector $\boldsymbol{I} \in \mathbb{N}_\infty^\delta$, with $I_k > 0$, for all $k = 0,\ldots,\delta \Leftrightarrow 1$, a period vector $\boldsymbol{p} \in \mathbb{N}_+^\delta$, sorted in non-increasing order, and an

occupation time $o \in \mathbb{N}_+$, give a lexicographical execution if and only if

$$p_k \geq \sum_{l=k+1}^{\delta-1} p_l I_l + o,$$

for all $k = 0,\ldots,\delta \Leftrightarrow 1$. We tighten these constraints to

$$p_{\delta-1} \geq o,$$

and

$$p_k \geq p_{k+1}(I_{k+1}+1),$$

for all $k = 0,\ldots,\delta \Leftrightarrow 2$. This is done since then in the case that equality holds, we obtain divisible periods. Now, we replace the processing unit constraints for different executions of the same operation by the following constraints.

Definition 7.3 (Strong Lexicographical Execution Constraints). Given are a signal flow graph $G = (V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, a set of resource types T^*, and a period assignment $\boldsymbol{p}$. Then the strong lexicographical execution constraints specify that for each operation $v \in V$, with occupation time $o(t(v)) > 0$,

$$\mathrm{slex}(v)$$

holds, where $\mathrm{slex}(v)$ is defined to hold if and only if there is a permutation $\boldsymbol{p}'$ of $\boldsymbol{p}(v)$ and a corresponding permutation $\boldsymbol{I}'$ of $\boldsymbol{I}(v)$, such that

$$|p'_{\delta(v)-1}| \geq o(t(v)),$$

and

$$|p'_k| \geq |p'_{k+1}|(I'_{k+1}+1),$$

for all $k = 0,\ldots,\delta(v) \Leftrightarrow 2$. □

Note that, without loss of generality, we may assume $I_k(v) > 0$, for all $v \in V$ and all $k = 0,\ldots,\delta(v) \Leftrightarrow 1$.

With the approximate cost function and constraints, the period assignment problem becomes the following.

Definition 7.4 (Approximate Period Assignment (APA)). Given are an extended signal flow graph $G' = (V',t,\boldsymbol{I},E',\boldsymbol{A},\boldsymbol{b})$, a set of resource types T^*, and lower and upper bounds on the period vectors and start times of the operations. Find a period assignment $\boldsymbol{p}$, and a start time assignment s, that obey the timing constraints, the precedence constraints, the stop precedence constraints, and the strong lexicographical execution constraints, such that $f_{\mathrm{stop}}(\tau)$ as given in (5.1) for the resulting time assignment $\tau = (\boldsymbol{p},s)$ is minimal. □

7.2.2 An ILP Formulation

The approximate period assignment problem can be formulated as an integer linear programming problem with some additional constraints, called P_{apa}, which is given by

$$\begin{array}{lllll}
\text{minimize} & f_{\text{stop}}(\tau) & & & \\
\text{subject to} & \underline{\boldsymbol{p}}(v) \le \boldsymbol{p}(v) \le \overline{\boldsymbol{p}}(v) & (v \in V') & \text{(i)} \\
& \underline{s}(v) \le s(v) \le \overline{s}(v) & (v \in V') & \text{(ii)} \\
& c(p,\boldsymbol{i}) < c(q,\boldsymbol{j}) + w(q) & ((p,\boldsymbol{i},q,\boldsymbol{j}) \in C) & \text{(iii)} \\
& \text{slex}(v) & (v \in V_+) & \text{(iv)} \\
& \boldsymbol{p}(v), s(v) \text{ integer} & (v \in V) & \text{(v)}.
\end{array}$$

In this formulation, $V_+ = \{v \in V \mid o(t(v)) > 0\}$ is the set of operations with positive occupation time,

$$C = \{(p,\boldsymbol{i},q,\boldsymbol{j}) \mid (p,q) \in E' \wedge \boldsymbol{i} \in \mathcal{I}(p) \wedge \boldsymbol{j} \in \mathcal{I}(q) \wedge \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})\},$$

is a set of *communicating port executions*, and

$$w(q) = \begin{cases} 1 & \text{if } q \in S \\ 0 & \text{if } q \in I, \end{cases}$$

is used to make a distinction between original precedences, i.e., $q \in I$, and stop precedences, i.e., $q \in S$. In this way we combine the constraints $c(p,\boldsymbol{i}) < c(q,\boldsymbol{j})$ for $q \in I$, and the constraints $c(p,\boldsymbol{i}) \le c(q,\boldsymbol{j})$ for $q \in S$. As discussed in Section 5.3, the cost function is linear in the periods and start times of the operations. Furthermore, constraints (i), (ii), and (iii) are linear in the periods and start times of the operations. Constraints (iv) and (v) are non-linear, but they can be handled by means of a branch-and-bound approach. In contrast to PUC and PC, a fractional solution of APA can be rounded quite easily towards a feasible integer solution. So, the integrality constraints are less severe and thus no all-integer method is required. Furthermore, branch-and-bound is an elegant technique to handle the constraints (iv).

Before we discuss the branch-and-bound approach for P_{apa}, note that the constraints (iv) and (v) only have to be met for the original operations and not for the stop operations. Furthermore, note that we can omit the integrality constraints on the start times of the operations. Namely, if the period vectors are integer, then the start times will also be integer if we use a technique like the simplex algorithm. This is due to the fact that for fixed, integer periods, we only have constraints of the types

- $s(v) \ge x$,
- $s(v) \le y$, and
- $s(v) \Leftrightarrow s(u) \ge z$,

with x, y, and z integer. Therefore, P_{apa} with fixed, integer periods is equivalent to the dual problem of the minimum-cost maximum-flow problem [Lawler, 1976],

which can be solved using the simplex algorithm.

7.2.3 A Branch-and-Bound Approach

We apply a branch-and-bound method to solve the approximate period assignment problem. To this end, we start with a relaxation of the problem, in which constraints (iv) and (v) of the ILP formulation are omitted. Now, the relaxed problem is a linear programming problem, which we further discuss in Section 7.2.4. In this section, we discuss the branching and bounding rules that are used to obtain a solution that satisfies the strong lexicographical execution constraints and the integrality constraints.

Branching

We distinguish two ways of branching. First we discuss branching in the case where not all of the strong lexicographical execution constraints are met, which we call *lexicographical branching*, and secondly we discuss branching in the case where the integrality constraints are not all met, which we call *integrality branching*.

Lexicographical branching. If at a certain point we have a solution of the relaxed problem, for which not all of the strong lexicographical execution constraints are met, i.e., there is an operation $v \in V_+$ for which $\mathrm{slex}(v)$ does not hold, then we apply a branching rule that determines which dimension will have the smallest period, which one will have the next smallest period, etc. So, the first time we apply such a branch for an operation v, we choose a dimension $k \in \{0,\ldots,\delta(v) \Leftrightarrow 1\}$ for which we impose $|p_k(v)| \geq o(t(v))$. This is done by creating for each k two sub-problems: one with the constraint $p_k(v) \geq o(t(v))$, and one with the constraint $\Leftrightarrow p_k(v) \geq o(t(v))$. The first constraint implies that $p_k(v)$ has to be positive, whereas the second one implies that it has to be negative. In total, this gives $2\delta(v)$ initial branches for operation v.

The next time we apply such a branch for the same operation v, we choose a dimension $k \in \{0,\ldots,\delta(v) \Leftrightarrow 1\}$ that was not previously chosen to be the dimension with the next smallest period. This is done by imposing $|p_k(v)| \geq |p_l(v)|(I_l(v)+1)$, where l denotes the latest chosen dimension used for branching of operation v. Since the sign of period $p_l(v)$ is known from the previous branch, which we denote by $s_l(v) \in \{\Leftrightarrow 1, +1\}$, we can impose the above constraint by introducing again two sub-problems: one with $p_k(v) \geq s_l(v)p_l(v)(I_l(v)+1)$, and one with $\Leftrightarrow p_k(v) \geq s_l(v)p_l(v)(I_l(v)+1)$. In this way, we get $2(\delta(v) \Leftrightarrow n)$ branches, where n is the number of dimensions already used by branches for operation v.

In order to reduce the number of branches, we make an exception for dimension 0 in the above branching rule. This dimension is the dimension with an infinite number of iterations, so it can only be chosen as the dimension with the largest period.

Furthermore, its period has to be positive.

To illustrate the above branching rule, consider an operation v with iterator bound vector

$$\boldsymbol{I}(v) = \begin{bmatrix} \infty \\ 5 \\ 3 \end{bmatrix},$$

and occupation time $o(t(v)) = 2$. This leads to a branching structure as depicted in Figure 7.2. The root of the tree corresponds to the original relaxation. If the

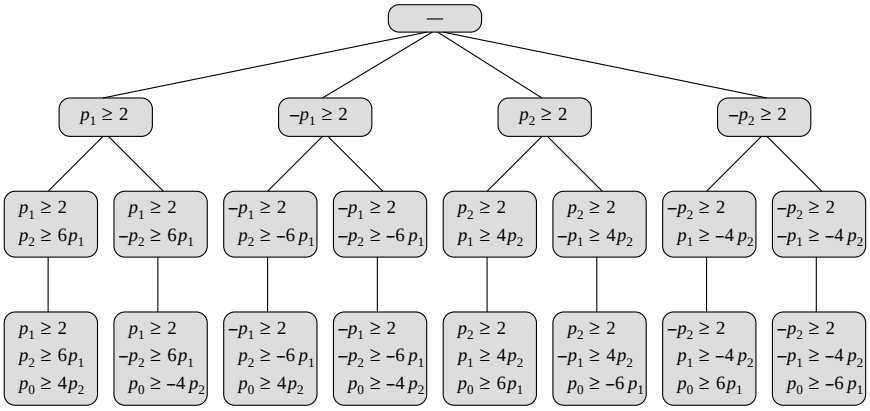


Figure 7.2. The structure of all possible branches to obtain a strong lexicographical execution for an operation with iterator bound vector $[\infty\ 5\ 3]^{\mathrm{T}}$ and occupation time 2. In the nodes the constraints are shown that have been added to the original relaxation.

solution of this relaxation does not satisfy the strong lexicographical execution constraints, then we branch, giving four sub-problems. Each of these sub-problems has an additional linear constraint. The first sub-problem, for instance, has an extra constraint $p_1(v) \geq o(t(v)) = 2$. If the solution of this sub-problem also does not satisfy the strong lexicographical execution constraints, then we branch again, giving two sub-problems. The first one of these two has an additional constraint $p_2(v) \geq p_1(v)(I_1(v)+1) = 6p_1(v)$. If the solution of this sub-problem also does not satisfy the constraints, then we branch again, this time yielding only one sub-problem, with an additional constraint $p_0(v) \geq p_2(v)(I_2(v)+1) = 4p_2(v)$.

The constraints that are added in the above way of branching can be incorporated in the relaxation by adding constraints of the type

$$\boldsymbol{B}(v)\,\boldsymbol{p}(v) \geq \boldsymbol{c}(v),$$

for each operation $v \in V_+$, i.e., we add linear constraints on the period vector of each operation with a positive occupation time.

Integrality branching. The second branching rule is on the integrality constraints. If at a certain point we have a solution of the relaxed problem, for which the integrality constraints are not all met, i.e., there is a period $p_k(v)$ of an operation $v \in V$ that has a non-integer value x, then we create two sub-problems: one with the additional constraint $p_k(v) \leq \lfloor x \rfloor$, and one with the additional constraint $p_k(v) \geq \lceil x \rceil$. These constraints are of the form of the constraints (i) of the ILP formulation, so they can be readily incorporated in the relaxation.

The priority between lexicographical branching and integrality branching is that we prefer the former over the latter, in order to have a faster branch-and-bound approach. This is based on the fact that, in practice, if the strong lexicographical execution constraints are met, then the integrality constraints are also met in most cases. Next, if more than one operation can be selected for lexicographical branching, then we preferably select the operation that was used in the previous branch, if applicable. In this way we first obtain a strong lexicographical execution for this operation, before we turn to branching on another operation. This is based on the observation that, in practice, if an operation is strongly lexicographically executed, then most of its predecessor and successor operations are too. If the operation of the previous branch is already strongly lexicographically executed, we just select an operation with the highest average number of executions per time unit.

Bounding

In order to be able to cut branches from the solution tree, we have to determine for each relaxation a lower bound on the cost of any of the solutions in the corresponding sub-tree. A straightforward lower bound is given by the relaxed cost, i.e, the cost of the relaxed problem, which can be determined by means of the simplex algorithm. We, however, choose to descend slightly deeper in the sub-tree, until the level where a new operation is selected for branching on the strong lexicographical execution constraints, or where a branching on the integrality constraints is applied. We then take the minimum of the relaxed costs of all these sub-relaxations as a lower bound on the cost of the relaxation we started with. Figure 7.3 shows an example of a sub-tree with the relaxed costs and the calculated lower bounds. Note that the relaxed cost are non-decreasing, when descending the tree.

Now, we discard a relaxation and all solutions in the corresponding sub-tree if the lower bound is greater than or equal to the cost of the best solution found so far. In this way, the branch-and-bound technique results in an optimal solution to the approximate period assignment problem. The calculated bounds are also used to determine the order in which the tree of solutions is traversed. At a branch, the sub-trees are considered in order of increasing lower bound, i.e., the most promising sub-tree is traversed first. In this way it is more likely to find a good feasible solution earlier in the course of the algorithm, which yields that possibly more sub-trees can

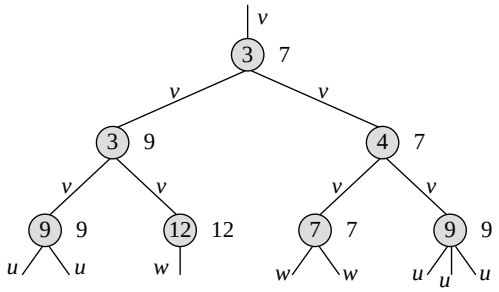


Figure 7.3. An example of the calculation of a lower bound on the cost, based on the relaxed costs of the sub-problems. The numbers in the nodes denote the relaxed costs, and the numbers to the right of the nodes denote the calculated lower bounds. The letters along the edges denote the operations that are used for branching.

be discarded, which in turn decreases the run time.

With a slight modification, we can use the branch-and-bound algorithm as an approximation algorithm, in order to decrease further the run time. To this end, we discard a relaxation and all solutions in the corresponding sub-tree if its lower bound b obeys

$$b \geq \frac{U}{1+e},$$

where U is the cost of the best solution found so far, and e is a positive number. By doing so, the eventual best solution found, which we denote by τ^*, has a cost

$$f_{\text{stop}}(\tau^*) \leq (1+e)\, f_{\text{stop}}(\tau_{\text{opt}}),$$

where τ_{opt} is an optimal solution. So, in this way we obtain a solution that deviates at most a given fraction e from the optimal cost, i.e., e is a maximum relative error. This can be seen as follows.

Let the set of solutions τ that have been discarded during a branch-and-bound run be given by D, and let the set of solutions that have been visited be given by N. Then for each discarded solution $\tau \in D$, we know that

$$f_{\text{stop}}(\tau) \geq \frac{1}{1+e} f_{\text{stop}}(\tau^*),$$

because of the bounding rule. Then we have

$$\begin{aligned} f_{\text{stop}}(\tau_{\text{opt}}) &= \min\{ f_{\text{stop}}(\tau) \mid \tau \in D \cup N\} \\ &= \min\{\min\{ f_{\text{stop}}(\tau) \mid \tau \in D\}, \min\{ f_{\text{stop}}(\tau) \mid \tau \in N\}\} \\ &= \min\{\min\{ f_{\text{stop}}(\tau) \mid \tau \in D\}, f_{\text{stop}}(\tau^*)\} \\ &\geq \min\{\frac{1}{1+e} f_{\text{stop}}(\tau^*), f_{\text{stop}}(\tau^*)\} \end{aligned}$$

$$= \frac{1}{1+e} f_{\text{stop}}(\tau^*),$$

which implies the required result.

Next, we determine in retrospect another relative error, which may be tighter than the specified fraction e since in the course of the algorithm the cost U of the best solution found so far decreases. To this end, we let c^* be the minimum of all lower bounds of the relaxations where the solution tree was cut. If $c^* \geq f_{\text{stop}}(\tau^*)$, then an optimal solution has been visited during the branch-and-bound run, so the actual relative error is zero, i.e., $f_{\text{stop}}(\tau^*) = f_{\text{stop}}(\tau_{\text{opt}})$. If $c^* < f_{\text{stop}}(\tau^*)$, then we use $0 \leq c^* \leq f_{\text{stop}}(\tau_{\text{opt}})$ to derive

$$\frac{f_{\text{stop}}(\tau^*) \Leftrightarrow f_{\text{stop}}(\tau_{\text{opt}})}{f_{\text{stop}}(\tau_{\text{opt}})} \leq \frac{f_{\text{stop}}(\tau^*) \Leftrightarrow c^*}{c^*},$$

i.e., $f_{\text{stop}}(\tau^*)$ is within a fraction

$$e^* = \frac{f_{\text{stop}}(\tau^*) \Leftrightarrow c^*}{c^*} \tag{7.1}$$

from the optimal value. This bound on the relative error may be substantially less than the specified maximum relative error e.

7.2.4 Solving the Relaxation

In this section we discuss how to solve a relaxation of the approximate period assignment problem. This problem can be formulated as a linear programming problem, given by

$$\begin{array}{lll}
\text{minimize} & f_{\text{stop}}(\tau) & \\
\text{subject to} & \underline{\boldsymbol{p}}(v) \leq \boldsymbol{p}(v) \leq \overline{\boldsymbol{p}}(v) & (v \in V') \\
& \underline{s}(v) \leq s(v) \leq \overline{s}(v) & (v \in V') \\
& c(p,\boldsymbol{i}) < c(q,\boldsymbol{j}) + w(q) & ((p,\boldsymbol{i},q,\boldsymbol{j}) \in C) \\
& \boldsymbol{B}(v)\,\boldsymbol{p}(v) \geq \boldsymbol{c}(v) & (v \in V).
\end{array}$$

Since the set C is infinitely large, we cannot directly solve the above problem by means of the simplex method. Therefore, we solve it by means of a constraint-generation technique, which is the dual analog of the column-generation technique [Schrijver, 1986]. This means that we only take a limited subset C' of C into account, and iteratively add tuples $(p,\boldsymbol{i},q,\boldsymbol{j}) \in C$ to it for which the constraints $c(p,\boldsymbol{i}) < c(q,\boldsymbol{j}) + w(q)$ are not met. Since many of the original constraints are redundant, this substantially reduces the number of constraints to be taken into account during the simplex method.

So, we start with $C' = \emptyset$ and solve the relaxation of the approximate period assignment problem with the limited set of constraints. After this, we check for each edge $(p,q) \in E'$ whether there is an execution $\boldsymbol{i}$ of p and an execution $\boldsymbol{j}$ of q for which

$\boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})$ and $c(p,\boldsymbol{i}) \geq c(q,\boldsymbol{j}) + w(q)$. This means that we have to solve for each edge $(p,q) \in E'$ an instance of the precedence conflict problem, which is done as discussed in Section 6.3. If for a certain edge $(p,q) \in E'$ a precedence conflict is found for execution $\boldsymbol{i}$ of p and execution $\boldsymbol{j}$ of q, then we add the tuple $(p,\boldsymbol{i},q,\boldsymbol{j})$ to C', and solve the relaxed problem again. Since we had an optimal solution before adding the extra constraint, we can do this by using the dual simplex method. Then the above procedure of checking and adding precedence constraints is repeated, until an optimum is found for which no precedence constraints are violated.

7.3 Start Time and Processing Unit Assignment

In this section, we present an algorithm for the multidimensional periodic scheduling problem with fixed periods, i.e., we determine a start time and a processing unit assignment.

We opt for a resource and time constrained approach, yielding a feasible schedule for a given set of processing units. So, we assume that the set W of processing units is given, and that we have to assign start times to the operations and to assign the operations to the processing units, such that a feasible schedule is obtained. This assumption is based on the observation that in practice it is easy to determine a near-optimal number of processing units of each type by hand, with respect to the total area cost as given in Definition 2.8.

The algorithm consists of three major steps: an analysis of the precedence constraints, a determination of the initial slack of the operations' start times, and an iterative approach in which start times and processing units are assigned. These steps are discussed in Sections 7.3.1, 7.3.2, and 7.3.3, respectively.

7.3.1 Precedences

The first step in the algorithm is to determine for each pair $(u,v) \in V \times V$ the minimum difference $m(u,v)$ between the start time of operation u and the start time of operation v, such that the precedence constraints are met. To this end, we first determine the minimum difference in start time $m(p,q)$ between an output port $p \in O$ and an input port $q \in I$, $(p,q) \in E$, with period vectors $\boldsymbol{p}(p)$ and $\boldsymbol{p}(q)$, respectively. This is given by

$$m(p,q) = \max\{\boldsymbol{p}^{\mathrm{T}}(p)\,\boldsymbol{i} \Leftrightarrow \boldsymbol{p}^{\mathrm{T}}(q)\,\boldsymbol{j} + 1 \mid \boldsymbol{i} \in \mathcal{I}(p) \ \wedge\ \boldsymbol{j} \in \mathcal{I}(q) \ \wedge\ \boldsymbol{n}(p,\boldsymbol{i}) = \boldsymbol{n}(q,\boldsymbol{j})\},$$

which can be determined as discussed in Section 6.3. Based on this, the minimum difference in start time between an operation u and an operation v is given by

$$\begin{aligned} m(u,v) = \max\{\ & m((u,\tilde{o}),(v,\tilde{\imath})) + r(t(u),\tilde{o}) \Leftrightarrow r(t(v),\tilde{\imath}) \mid \\ & \tilde{o} \in \tilde{O}(t(u)) \ \wedge\ \tilde{\imath} \in \tilde{I}(t(v)) \ \wedge\ ((u,\tilde{o}),(v,\tilde{\imath})) \in E\ \}. \end{aligned} \tag{7.2}$$

With the minimum differences in start times of the operations we can construct the following precedence graph.

Definition 7.5 (Precedence Graph). Given are a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$ and for each operation $v \in V$ a period vector $\boldsymbol{p}(v)$. The corresponding precedence graph $H = (V, A, m)$ is a weighted, directed graph, with node set V and arc set $A = \{(u,v) \in V \times V \mid ((u,\tilde{o}),(v,\tilde{\imath})) \in E\}$, i.e., there is an arc (u,v) in A if and only if there is an edge from an output port of operation u to an input port of operation v. Furthermore, the weight of an edge $(u,v) \in A$ is given by the value of $m(u,v)$ as defined in (7.2). □

Theorem 7.1. *Given are a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, for each operation $v \in V$ a period vector $\boldsymbol{p}(v)$, and the corresponding precedence graph $H = (V, A, m)$. Furthermore, for each operation $v \in V$ a start time $s(v)$ is given. Then, the precedence constraints are met if and only if*

$$s(v) \Leftrightarrow s(u) \geq m(u,v),$$

for all $(u,v) \in A$.

Proof. The proof follows straightforwardly from the definition of the precedence graph and the arc weights. □

In general, the arc weights in a precedence graph H may be negative, and H may contain cycles. However, if H contains a cycle for which the sum of the arc weights is positive, then no feasible schedule exists. For the following, we therefore assume that H contains no cycles of positive weight.

7.3.2 ASAP and ALAP Start Time Assignment

The second step in the algorithm is to determine for each operation an 'as soon as possible' start time assignment and an 'as late as possible' start time assignment, which are defined as follows.

Definition 7.6 (ASAP and ALAP Start Time Assignment). Given are a signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, and for each operation $v \in V$ a period vector $\boldsymbol{p}(v)$. Furthermore, for each operation $v \in V$ a lower bound $\underline{s}(v) \in \mathbb{Z}_\infty$ and an upper bound $\overline{s}(v) \in \mathbb{Z}_\infty$ on the start time $s(v)$ are given. Then, the as soon as possible (ASAP) start time assignment s_{asap} and the as late as possible (ALAP) start time assignment s_{alap} are given by

$$s_{\text{asap}}(v) = \min_{s \in \mathcal{T}} s(v),$$

and

$$s_{\text{alap}}(v) = \max_{s \in \mathcal{T}} s(v),$$

for all $v \in V$, where $\mathcal{T}$ is the set of all feasible start time assignments, i.e., the start time assignments s that, for the given periods, obey the timing constraints and the precedence constraints. □

The as soon as possible start time assignment can be determined by means of the algorithm in Figure 7.4. Since H contains no cycles of positive weight, the updating

---

1. Initialize $s_{\text{asap}}(v) = -\infty$, for all operations $v \in V$.
2. If $s_{\text{asap}}(v) \geq \underline{s}(v)$, for all $v \in V$, then stop.
3. Take an operation $v \in V$ with $s_{\text{asap}}(v) < \underline{s}(v)$, and set $s_{\text{asap}}(v) = \underline{s}(v)$ and $U = \{v\}$. U can be seen as a set of updated operations.
4. If $U = \emptyset$, return to Step 2. Otherwise, take an operation $u \in U$.
5. For all $u' \in V$ with $(u,u') \in A$ and $s_{\text{asap}}(u') - s_{\text{asap}}(u) < m(u,u')$, set $s_{\text{asap}}(u') = s_{\text{asap}}(u) + m(u,u')$ and add u' to U.
6. Remove u from U and return to Step 4.

Figure 7.4. The as soon as possible start time assignment algorithm.

in Steps 4–6 is only repeated a finite number of times. The as late as possible start time assignment can be determined by a similar algorithm.

If after determining the ASAP and ALAP start time assignment, an operation $v \in V$ exists with $s_{\text{asap}}(v) > s_{\text{alap}}(v)$, then no feasible start time assignment exists, so we can stop. If the ASAP start time assignment is finite, i.e., $s_{\text{asap}}(v) \neq \Leftrightarrow\infty$, for all $v \in V$, then it is also a feasible start time assignment. A similar property holds for the ALAP start time assignment.

7.3.3 An Iterative Approach

The third and final step of the algorithm is to determine a start time and a processing unit assignment. We do this by means of an iterative approach, which shows some resemblance with list scheduling [Haupt, 1989]. In this approach, a partial solution is iteratively augmented, by selecting in each iteration an unscheduled operation and assigning it a start time and a processing unit, or reducing its freedom. The partial solutions can be defined as follows.

Definition 7.7 (Partial Schedule). Given are a signal flow graph $G = (V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, for each operation $v \in V$ a period vector $\boldsymbol{p}(v)$, and a set W of processing units. A partial schedule $\tilde{\sigma}$ is given by a 5-tuple $(s_{\text{lo}}, s_{\text{hi}}, F, s, h)$, where

- for each operation $v \in V$ we have a *start time domain* $\{s_{\text{lo}}(v),\ldots,s_{\text{hi}}(v)\}$, which denotes that operation v has to be assigned a start time $s(v) \in \{s_{\text{lo}}(v),\ldots,s_{\text{hi}}(v)\}$, and
- $F \subset V$ is a set of operations that are already scheduled, with fixed start time $s(v) = s_{\text{lo}}(v) = s_{\text{hi}}(v)$, and processing unit $h(v) \in W$ with $t(h(v)) = t(v)$.

Note that s and h are partial functions, with domain F. The set of all possible partial schedules is given by $\tilde{\mathcal{S}}$. □

A partial schedule $\tilde{\sigma}$ is called feasible if and only if s_{lo} and s_{hi} are feasible start time assignments, i.e., $s_{\mathrm{lo}}, s_{\mathrm{hi}} \in \mathcal{T}$, and if the processing unit constraints are met for the operations $v \in F$. The set of all feasible partial schedules is denoted by $\tilde{\mathcal{S}}'$. Note that the processing unit constraints for different executions of the same operation can be checked before scheduling, since the period vectors of the operations are fixed.

A basic algorithm for the multidimensional periodic scheduling problem with fixed periods can now be written as shown in Figure 7.5. Here, we assume that the

1. Initialize the set of scheduled operations $F = \emptyset$, and initialize $s_{\mathrm{lo}}(v) = s_{\mathrm{asap}}(v)$ and $s_{\mathrm{hi}}(v) = s_{\mathrm{alap}}(v)$, for all operations $v \in V$.
2. If $F = V$, then a feasible schedule is found, so stop. Otherwise, take an operation $v \in V \setminus F$.
3. Try to schedule operation v at start time $s_{\mathrm{lo}}(v)$, i.e., try to assign it start time $s(v) = s_{\mathrm{lo}}(v)$ and to assign it to a processing unit $h(v) \in W$, with $t(h(v)) = t(v)$, such that v has no processing unit conflict with operations $u \in F$ that are already assigned to the same processing unit $h(u) = h(v)$.
4. If v can be scheduled at start time $s_{\mathrm{lo}}(v)$, then assign $s(v) = s_{\mathrm{lo}}(v)$ and $s_{\mathrm{hi}}(v) = s_{\mathrm{lo}}(v)$, fix $h(v)$, and add v to F. Otherwise, increase $s_{\mathrm{lo}}(v)$ by 1.
5. If necessary, adjust the start time domains $\{s_{\mathrm{lo}}(u), \ldots, s_{\mathrm{hi}}(u)\}$ of the unscheduled operations $u \in V \setminus F$, such that s_{lo} and s_{hi} are again feasible start time assignments.
6. If now an operation $u \in V \setminus F$ has an empty time domain, i.e., $s_{\mathrm{lo}}(u) > s_{\mathrm{hi}}(u)$, then stop; no schedule is found. Otherwise, return to Step 2.

Figure 7.5. The start time and processing unit assignment algorithm.

ASAP and ALAP start time assignments are both finite.

In the algorithm, there is freedom to choose operation v in Step 2. We use the following priority function to determine which operation is selected. First, we only consider operations $v \in V \setminus F$ for which all preceding operations $u \in V$, i.e., $(u, v) \in A$, with arc weight $m(u, v) > 0$ are already scheduled. Note that we consider the arc weights since the precedence graph may contain cycles. If more than one operation is eligible, we select the operation v for which $s_{\mathrm{hi}}(v) \Leftrightarrow s_{\mathrm{lo}}(v)$ is minimal. Among these, we select the operation v with the lowest $s_{\mathrm{lo}}(v)$.

Next, we slightly alter Step 3 of the algorithm. In this step, the processing unit assignment of previously scheduled operations remains unchanged. However, by reassigning these operations, together with the new operation v, it is more probable that operation v can be scheduled at start time $s_{\mathrm{lo}}(v)$. The reassignment can be done by recoloring the corresponding processing unit conflict graph.

Finally, we alter the increasing of $s_{\text{lo}}(v)$ in Step 4. We determine a possibly larger amount by which it can be increased, in order to improve the efficiency of the algorithm. To this end, we consider the already scheduled operations $u \in F$, of the same type as v, that give a processing unit conflict if v is assigned to the same processing unit and v starts at time $s_{\text{lo}}(v)$. Let the set of these operations u be denoted by F^*. Now, for each of the operations $u \in F^*$, we determine a lower bound $d(u)$ by which $s_{\text{lo}}(v)$ should be increased in order to resolve the processing unit conflict between u and v. Then, $s_{\text{lo}}(v)$ must be increased by at least

$$\min_{u \in F^*} d(u),$$

in order to be able to schedule v at time $s_{\text{lo}}(v)$. The amount $d(u)$, for each $u \in F^*$, is determined as follows.

Let the instance of PUC that corresponds to the processing unit conflict problem between operation u and v be given by $(\boldsymbol{I}, \boldsymbol{p}, s)$, with $\boldsymbol{I} \in \mathbb{N}_+^\delta$, $\boldsymbol{p} \in \mathbb{N}_+^\delta$, and $s \in \mathbb{N}$. Furthermore, let $\boldsymbol{i} \in \mathbb{Z}^\delta$ be a solution of it, i.e., $\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}$ and $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s$. Now the algorithm in Figure 7.6 determines an interval $\{l,\ldots,r\}$, with $l \leq s \leq r$, such that $(\boldsymbol{I}, \boldsymbol{p}, s')$ is a yes-instance of PUC, for all $s' \in \{l,\ldots,r\}$.

---

1. Set $l = s$ and $r = s$, and set $D = \{0,\ldots,\delta-1\}$. Let $\mathcal{I}$ be the set of vectors $\boldsymbol{i}' \in \mathbb{Z}^\delta$, for which

$$i'_k = \begin{cases} i_k & \text{if } k \in D \\ 0,\ldots,I_k & \text{if } k \notin D, \end{cases}$$

then

$$\{\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}' \mid \boldsymbol{i}' \in \mathcal{I}\} = \{l,\ldots,r\}.$$

This property is maintained throughout the algorithm.

2. Select a $k \in D$ for which $p_k \leq r - l + 1$. If no such k exists, then stop.

3. Now from each solution $\boldsymbol{i}' \in \mathcal{I}$, with $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}' \in \{l,\ldots,r\}$, we can create new solutions by decreasing or increasing i'_k, thereby decreasing or increasing $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}'$ by multiples of p_k. The element i'_k can be decreased by at most i_k, and increased by at most $I_k - i_k$. So, in this way we can obtain solutions $\boldsymbol{i}''$ with $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i}'' \in \{l + zp_k,\ldots,r + zp_k\}$, for $z = -i_k,\ldots,I_k - i_k$. Since $p_k \leq r - l + 1$, these intervals overlap, so we may subtract $p_k i_k$ from l and add $p_k(I_k - i_k)$ to r. Next, remove k from D, and return to Step 2.

Figure 7.6. The processing unit conflict interval algorithm.

Now, in order to resolve the conflict, s must be increased by at least $r \Leftrightarrow s + 1$, or decreased by at least $s \Leftrightarrow l + 1$. So, $d(u)$ is given by $r \Leftrightarrow s + 1$ or by $s \Leftrightarrow l + 1$, depending on whether the sign of v's start time in s is positive or negative in the reformulation of PUC, described in Section 3.1.1.

7.4 Numerical Results

In this section we present some numerical results of the solution approach discussed in the previous sections. The presented run times are achieved by a C++ implementation on an HP9000/735 system, running under HP-UX/09.05. The examples we discuss are typical for video signal processing.

Example 7.1. Consider the video algorithm of Figure 7.7. Here, the periods of the

for i_0 = 0 to ∞ period 16
  for i_1 = 0 to 3 period 2
    $x[i_0][i_1+4] = in(\,)$ start in $\{0,\ldots,16\}$
for j_0 = 0 to ∞ period 16
  for j_1 = 0 to 3 period in $\{-4,\ldots,4\}$
    $x[j_0][j_1] = pr(x[j_0][j_1+4])$ start in $\{0,\ldots,16\}$
for k_0 = 0 to ∞ period 16
  for k_1 = 0 to 7 period 1
    $= out(x[k_0][k_1])$ start in $\{0,\ldots,16\}$

Figure 7.7. The video algorithm of Example 7.1, where in each global period of 16 time units four samples are taken as input and are processed, after which the four newly created samples and the original ones are sent to the output. A statement 'period in $\{-4,\ldots,4\}$' at the end of a line denotes that the period of the corresponding iterator has a lower bound -4 and an upper bound 4. A statement 'start in $\{0,\ldots,16\}$' at the end of a line denotes that the start time of the corresponding operation has a lower bound 0 and an upper bound 16.

input and output operations are fixed. Only $p_1(pr)$ is to be determined. Furthermore, the start times of the operations have to be determined. The operations are of different types, so the processing unit assignment is trivial. The relative transfer times of all operation ports are equal to zero, and the occupation times of all operation types are equal to one.

For this example, the signal flow graph extension takes approximately 0.1 second, resulting in two stop operations. In the period assignment stage, an initial simplex tableau is created consisting of 18 rows and 9 columns. During the solution step of the first relaxation, 15 rows are added in order to obey the precedence constraints. The relaxation appears to obey also the lexicographical execution constraints and the integrality constraints, so no branching is applied. The resulting optimal solution is given by $p_1(pr) = 1$, $s(in) = 0$, $s(pr) = 4$, and $s(out) = 5$, which yield an average of $34/16$ variables that are simultaneously alive. The run time for the period assignment is approximately 0.1 second.

In the second stage, the start times are assigned to the same values as the preliminary start times of the first stage. The resulting schedule is shown in Figure 7.8. The run time for the second stage is less than 0.1 second. □

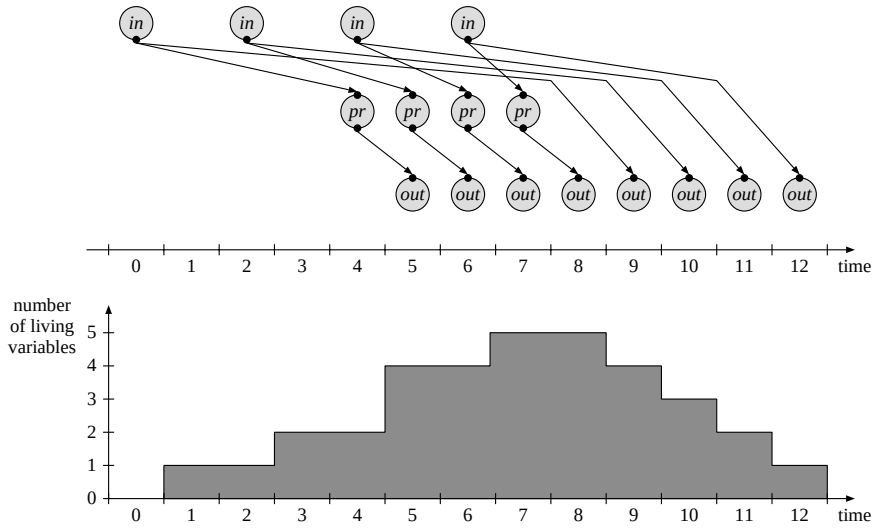


Figure 7.8. The schedule obtained for Example 7.1, shown at the top of the figure, and the resulting number of living variables at each time, shown at the bottom. Only the first global period of the algorithm is shown.

Example 7.2. Consider the matrix summation algorithm of Figure 7.9. Again, the

for i_1 = 0 to 3 period in $\{1,\ldots,4\}$
  for i_2 = 0 to 3 period in $\{1,\ldots,4\}$
    $x[i_1][i_2] = in(\,)$ start in $\{0,\ldots,32\}$
for j_1 = 0 to 3 period in $\{1,\ldots,4\}$
  $y[j_1][0] = x[j_1][0]$ start in $\{0,\ldots,32\}$
for k_1 = 0 to 3 period in $\{1,\ldots,4\}$
  for k_2 = 0 to 2 period in $\{1,\ldots,4\}$
    $y[k_1][k_2+1] = y[k_1][k_2] + x[k_1][k_2+1]$ start in $\{0,\ldots,32\}$
$z[0] = y[0][3]$ start in $\{0,\ldots,32\}$
for l_1 = 0 to 2 period in $\{1,\ldots,4\}$
  $z[l_1+1] = z[l_1] + y[l_1+1]$ start in $\{0,\ldots,32\}$
$= out(z[3]\,)$ start in $\{0,\ldots,32\}$

Figure 7.9. The video algorithm of Example 7.2, in which a 4×4 matrix is taken as an input and where the elements are summed. The video algorithm is repeated every 16 time units; the infinite repetition is not indicated.

relative transfer times of all operation ports are equal to zero, and the occupation times of all operation types are equal to one. For this instance, the signal flow graph extension takes approximately 2.1 seconds, resulting in five stop operations. In the period assignment stage, an initial simplex tableau is created of 46 rows and 23

columns. During the period assignment stage, at most 25 rows have been added to obey the precedence constraints. Again no branching is applied. The optimal solution has an average of $45/16$ variables simultaneously alive. The run time is approximately 0.6 second.

In the second stage, the start times are assigned to the same values as the preliminary start times of the first stage. The resulting schedule is shown in Figure 7.10. As we can see in the figure, the additions can be assigned to the same pro-

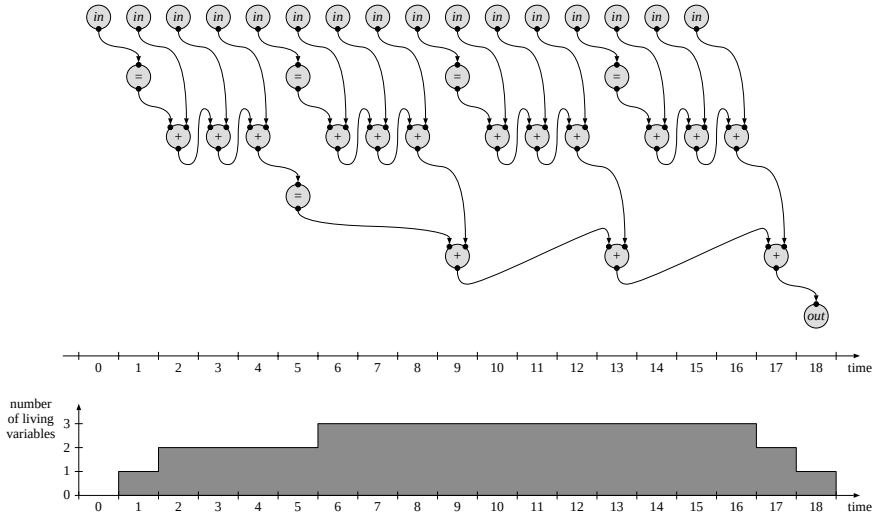


Figure 7.10. The schedule obtained for Example 7.2, shown at the top of the figure, and the resulting number of living variables at each time, shown at the bottom. Only the first global period of the algorithm is shown.

cessing unit. The assignment operations $(=)$ are assigned to different processing units, since their hardware only consists of a wire, with negligible cost. The run time for the second stage is approximately 0.1 second. □

Example 7.3. Next, we evaluate the effect of the maximum relative error on the quality of the period assignment and the run time for three signal flow graphs. Some characteristics of these three instances are shown in Table 7.1. For these three instances we applied the period assignment algorithm, for different values of the maximum relative error $e_{\max}$. The periods in the instances CRD and LPC were restricted to positive values. The periods in the instance BPR had no restriction on their signs. The results are shown in Table 7.2. As we can see, the calculated bound on the relative error is often significantly less than the specified bound, and by specifying a larger maximum error we can reduce the run time. Furthermore, we can see that all runs result in optimal solutions. □

Table 7.1. Characteristic numbers of three video algorithms. Here, $|V|$ denotes the number of operations, $|V_s|$ denotes the number of stop operations in the extended signal flow graph, t_{ext} denotes the run time in seconds for the signal flow graph extension, and r and c denote the number of rows and columns, respectively, of the initial simplex tableau for period assignment.

instance	$\|V\|$	$\|V_s\|$	t_{ext}	r	c
CRD	6	8	0.9	66	33
LPC	8	18	1.6	110	55
BPR	7	22	4.5	132	66

Table 7.2. Characteristics of the period assignment runs for the instances CRD, LPC, and BPR, for different values of the specified maximum relative error e_{max}. Shown are the calculated error e_{cal}, as given in (7.1), the actual deviation e_{act} from the optimum, the maximum number of rows r_{max} in the simplex tableau during the period assignment run, the difference with the initial number of rows r, and the run time t_{per} in seconds for the period assignment.

instance	e_{max}	e_{cal}	e_{act}	r_{max}	$r_{max} - r$	t_{per}
CRD	10^6	596	0	109	43	1.7
	100	36	0	109	43	2.3
	10	0	0	109	43	2.3
LPC	10^6	0.57	0	190	80	8.6
	0.1	0	0	190	80	10.5
BPR	10^6	0.39	0	256	124	31.1
	0.1	0.006	0	273	141	104.0
	0.001	0	0	273	141	108.9

Example 7.4. Consider the discrete cosine transformation (DCT) of Lippens et al. [1994]. Here, we have 35 operations. The extended signal flow graph contains 47 stop operations, and is constructed in approximately 7.0 seconds. The initial simplex tableau for period assignment has 362 rows and 181 columns. During period assignment, the used simplex tableau has at most 737 rows, which is 375 more than initially. The maximum depth in the solution tree that has been visited is depth 16. An optimum solution of about 66 variables alive simultaneously is obtained in approximately 1 hour and 21 minutes. The main part of this run time is taken by solving the LP-relaxations, which is done by means of a straightforward implementation of the simplex algorithm. Using a more sophisticated implementation instead, the run time may be reduced substantially. Furthermore, we mention that in the example we only specified the global period and the periods of the input and output operations. The other periods were left unbounded. In most practical situations,

however, the periods are more restricted, e.g., their signs are known, which significantly reduces the solution tree.

In the second stage, the start times are assigned and the operations are assigned to processing units. The run time for this is approximately 6.9 seconds. After this, an average of about 147 variables are simultaneously alive; the maximum is 156 variables. The reason for the increase is that operations had to be postponed in order to resolve processing unit conflicts. The global period in this instance is 32 time units. Of the first global period, the first execution takes place at time 1, and the last one at time 183. This means that executions of six different iterations of the outermost loop overlap in time, making it impracticable to solve the scheduling problem by hand. Figure 7.11 shows the executions of the operations in the time window from 0 to 31.

□

7.5 Discussion

In this chapter we have presented a solution approach for the multidimensional periodic scheduling problem. The algorithms we developed can be used to find a feasible schedule in an acceptable run time. Furthermore, they can be used to complete a partial schedule, given by the user. In the latter case, the algorithm is also used to check the part of the schedule that has been given.

The presented techniques are used in the design methodology Phideo in an iterative and interactive way. To this end, the obtained schedules are reported to the user in a textual and graphical way, allowing the user to evaluate the results and to provide feedback for the scheduling steps. If no schedule is obtained, diagnostic information is reported to the user, e.g., a bottleneck resource is reported. Interaction within Phideo proceeds in several ways.

First, arcs may be added to the precedence graph, to impose constraints on the differences in the start times of the operations. They can be incorporated straightforwardly in the second stage of the decomposition, and they are also easily incorporated in the period assignment stage, since they are linear constraints.

Secondly, the processing unit assignment may be restricted by preassigning the processing units for several operations, or by imposing a constraint that, for instance, two operations should be assigned to the same processing unit. These kinds of constraints can be taken into account during the coloring of the processing unit conflict graphs.

Thirdly, the lower and upper bounds on the periods and start times may be further restricted. For instance, one can take the lower bounds of the start times in the second stage of the decomposition equal to the preliminary start times of the first stage, in order to obtain a schedule with a lower storage cost.

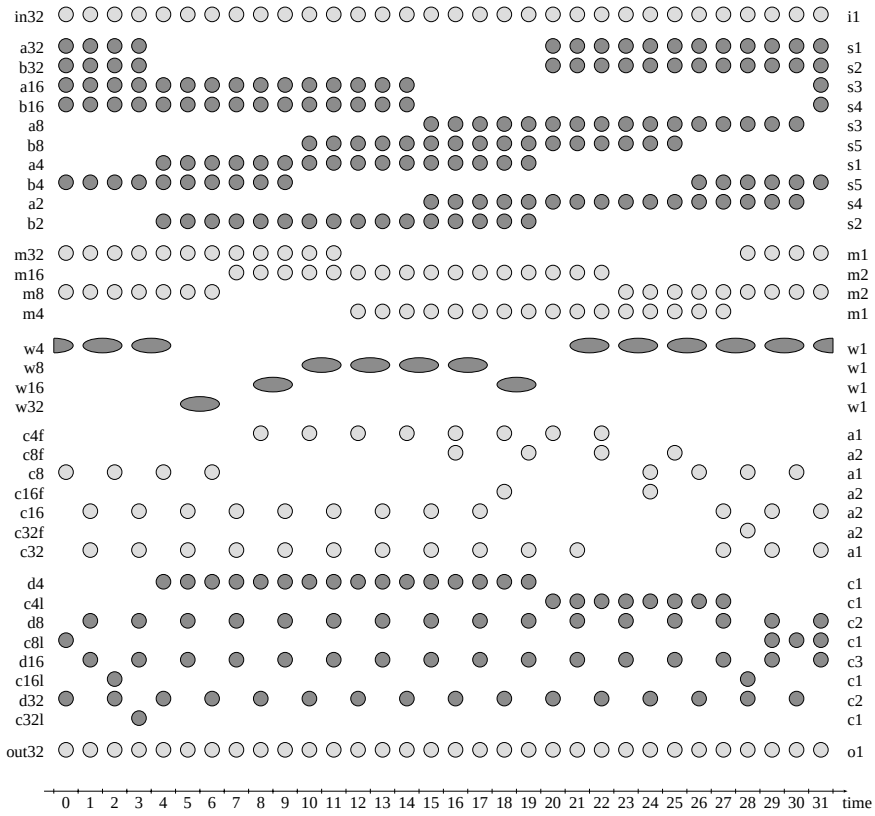


Figure 7.11. The executions of the operations of the DCT instance in the time window from 0 to 31. At the left hand side the operation names are shown. At the right hand side the processing units to which they are assigned are shown. The occupation times of all operation types are equal to one, except for the operation type of operations w4, w8, w16, and w32, whose occupation time is equal to two.

The above ways of interaction can be used to guide the algorithm towards a feasible solution if no solution is found, or to improve a solution with respect to, for instance, storage cost. Furthermore, they can be used to find solutions that give better results in the subsequent steps of Phideo. In practice, the scheduling algorithm together with the ways of interaction are satisfactory to find good solutions in a reasonable amount of time. Nevertheless, development of a more sophisticated scheduling algorithm, in order to reduce the amount of interaction, remains a topic of further research.

8

Parametric Multidimensional Periodic Scheduling

This chapter discusses the extension of multidimensional periodic scheduling to the case of parametric signal flow graphs. This means that the numbers of repetitions of the operations are no longer constant, but they depend on parameters. For instance, consider a video algorithm where the number of pixels on a video line is a parameter, whose value can vary between 512 and 1024. Now, we have to determine a *parametric schedule*, and it has to be feasible for any of these values. For the IC design, this means that a processor has to be designed that can execute the video algorithm for any of the values of the parameter. When the processor is started, for instance, by switching on the TV, the actual value of the parameter is loaded into the processor, and the processor executes the algorithm for that number of pixels on a line. When one switches to another channel, a new value of the parameter is loaded, equal to the number of pixels per line corresponding to that channel, and the processor is restarted to execute the video algorithm for the new number of pixels per line.

The use of parameters for the numbers of repetitions introduces parametric periods and start times. For instance, if the number of pixels on a video line is parametric, then the period between two successive lines is likely to be also parametric.

Furthermore, the index expressions of multidimensional arrays also become parametric. These changes in the model are discussed in Section 8.1. Next, in Section 8.2 we present a primal all-integer ILP algorithm that incorporates parameters. In Section 8.3, we discuss how to handle the constraint calculations in the case of parameters. Finally, in Section 8.4 we present an algorithm to determine a start time and processing unit assignment for parametric signal flow graphs. The period assignment problem in the case of parameters is not addressed.

8.1 Problem Modeling Revisited

Input for scheduling in the case of parameters is a parametric signal flow graph, representing a parametric video algorithm. Figure 8.1 shows an example of such an algorithm, in which a matrix transposition is performed for a matrix of parametric size.

```
parameter π0 in {2,...,8}
parameter π1 in {2,...,8}
for i0 = 0 to π0 − 1 period π1
    for i1 = 0 to π1 − 1 period 1
        x[i0][i1] = in( )
for j0 = 0 to π1 − 1 period π0
    for j1 = 0 to π0 − 1 period 1
        = out(x[j1][j0])
```

Figure 8.1. Example of a parametric video algorithm, in which a $\pi_0 \times \pi_1$ matrix is transposed, i.e., it is read in row by row, and written out column by column. In the first two lines of the algorithm, the parameters π_0 and π_1 are declared and their ranges are specified.

8.1.1 Parameters

Before defining parametric signal flow graphs, we first formally define parameters.

Definition 8.1 (Parameter Set). A parameter set is given by a 3-tuple $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$, where

- $\boldsymbol{\pi}$ is a ρ-dimensional vector of parameters,
- $\underline{\boldsymbol{\pi}} \in \mathbb{Z}^\rho$ is a vector of parameter lower bounds, and
- $\overline{\boldsymbol{\pi}} \in \mathbb{Z}^\rho$ is a vector of parameter upper bounds,

with $\underline{\boldsymbol{\pi}} \leq \overline{\boldsymbol{\pi}}$. The set of all possible *parameter settings* is given by

$$\Pi = \{\boldsymbol{x} \in \mathbb{Z}^\rho \mid \underline{\boldsymbol{\pi}} \leq \boldsymbol{x} \leq \overline{\boldsymbol{\pi}}\},$$

which means that the parameter vector $\boldsymbol{\pi}$ can assume any vector from Π, i.e., each parameter π_i can assume any value in $\{\underline{\pi}_i,\ldots,\overline{\pi}_i\}$. □

Next, we need the definition of parametric expressions.

Definition 8.2 (Parametric Expressions). Given a parameter set $(\boldsymbol{\pi},\underline{\boldsymbol{\pi}},\overline{\boldsymbol{\pi}})$, the set $\mathcal{E}$ of parametric expressions, or expressions for short, is the set of integer polynomials in $\boldsymbol{\pi}$, i.e., $\mathcal{E}=\mathbb{Z}[\boldsymbol{\pi}]$. On this set we define a partial ordering $<_{\text{all}}$ by

$$\varepsilon <_{\text{all}} \varphi \;\Leftrightarrow\; \forall_{\boldsymbol{\pi}\in\Pi}:\ \varepsilon(\boldsymbol{\pi}) < \varphi(\boldsymbol{\pi}),$$

for all $\varepsilon,\varphi\in\mathcal{E}$. The relations $>_{\text{all}}$, $\leq_{\text{all}}$, $\geq_{\text{all}}$, $=_{\text{all}}$, and $\neq_{\text{all}}$ are defined analogously. □

Note that if $\varepsilon >_{\text{all}} \varphi$ does not hold, then $\varepsilon \leq_{\text{all}} \varphi$ does not necessarily have to hold, since it may occur that $\varepsilon(\boldsymbol{\pi})$ is smaller than $\varphi(\boldsymbol{\pi})$ for some parameter settings $\boldsymbol{\pi}\in\Pi$, but that for other settings $\varepsilon(\boldsymbol{\pi})$ is larger than $\varphi(\boldsymbol{\pi})$.

If $\boldsymbol{\pi}=[\pi_0\ \pi_1]^{\mathrm{T}}$, then an example of a parametric expression $\varepsilon\in\mathcal{E}$ is given by $\varepsilon(\boldsymbol{\pi})=5\pi_0^2\pi_1+2\pi_0+\pi_1 \Leftrightarrow 10$. If we define $\underline{\boldsymbol{\pi}}=[10\ \ 1]^{\mathrm{T}}$ and $\overline{\boldsymbol{\pi}}=[50\ \ 20]^{\mathrm{T}}$, and $\varphi(\boldsymbol{\pi})=4\pi_0^2\pi_1 \Leftrightarrow 80$, then $\varepsilon >_{\text{all}} \varphi$, since $\varepsilon(\boldsymbol{\pi}) \Leftrightarrow \varphi(\boldsymbol{\pi})=\pi_0^2\pi_1+2\pi_0+\pi_1 \Leftrightarrow 90 \geq 10^2\cdot 1+2\cdot 10+1 \Leftrightarrow 90=31$, for all $\boldsymbol{\pi}\in\Pi$.

With a given set $\mathcal{E}$ of parametric expressions we associate a set $\mathcal{P}=\{\varepsilon\in\mathcal{E}\mid\varepsilon\geq_{\text{all}}0\}$ of expressions that are greater than or equal to zero for all parameter settings. For later use, we also define $\mathcal{P}_\infty=\mathcal{P}\cup\{\infty\}$ and $\mathcal{E}_\infty=\mathcal{E}\cup\{\Leftrightarrow\infty,+\infty\}$.

8.1.2 Parametric Signal Flow Graphs

We now define a parametric signal flow graph as follows.

Definition 8.3 (Parametric Signal Flow Graph). Given a parameter set $(\boldsymbol{\pi},\underline{\boldsymbol{\pi}},\overline{\boldsymbol{\pi}})$, a parametric signal flow graph G is given by a 6-tuple $(V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, where

- V is a finite set of multidimensional periodic operations,
- $t(v)\in T$ denotes the operation type, for each $v\in V$,
- $\boldsymbol{I}(v)\in\mathcal{P}_\infty^{\delta(v)}$ denotes the iterator bound vector, for each $v\in V$,
- $E\subset O\times I$ is a finite set of directed edges representing data dependencies, where $O=\{(v,\tilde{o})\mid v\in V\wedge\tilde{o}\in\tilde{O}(t(v))\}$ denotes the set of output ports, and $I=\{(v,\tilde{\imath})\mid v\in V\wedge\tilde{\imath}\in\tilde{I}(t(v))\}$ denotes the set of input ports,
- $\boldsymbol{A}(p)\in\mathcal{E}^{\alpha(p)\times\delta(v)}$ denotes the index matrix, for each $p=(v,\tilde{p})\in P=I\cup O$,
- $\boldsymbol{b}(p)\in\mathcal{E}^{\alpha(p)}$ denotes the index offset vector, for each $p\in P$.

□

The difference with a non-parametric signal flow graph is in the domains of the iterator bound vectors, the index matrices, and the index offset vectors.

For a parametric signal flow graph, the iterator space $\mathcal{I}(v)$ of an operation $v \in V$ is also parametric. For each parameter setting $\boldsymbol{\pi} \in \Pi$, it is given by

$$\mathcal{I}(v)(\boldsymbol{\pi}) = \{\boldsymbol{i} \in \mathbb{Z}^{\delta(v)} \mid \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(v)(\boldsymbol{\pi})\}.$$

For example, the operation *in* of the parametric video algorithm of Figure 8.1 has an iterator bound vector given by

$$\boldsymbol{I}(in)(\boldsymbol{\pi}) = \begin{bmatrix} \pi_0 \Leftrightarrow 1 \\ \pi_1 \Leftrightarrow 1 \end{bmatrix},$$

which results in an iterator space

$$\mathcal{I}(in)(\boldsymbol{\pi}) = \{\boldsymbol{i} \in \mathbb{Z}^2 \mid 0 \leq i_0 \leq \pi_0 \Leftrightarrow 1 \ \wedge \ 0 \leq i_1 \leq \pi_1 \Leftrightarrow 1\}.$$

Next, the index vector $\boldsymbol{n}(p,\boldsymbol{i})$ of a multidimensional array that is accessed at execution $\boldsymbol{i}$ of a port p is also parametric, given by

$$\boldsymbol{n}(p,\boldsymbol{i})(\boldsymbol{\pi}) = \boldsymbol{A}(p)(\boldsymbol{\pi})\,\boldsymbol{i} + \boldsymbol{b}(p)(\boldsymbol{\pi}),$$

for all $\boldsymbol{\pi} \in \Pi$. For example, consider a port p with index matrix and index offset vector given by

$$\boldsymbol{A}(p)(\boldsymbol{\pi}) = \begin{bmatrix} \pi_0\pi_1 & \pi_0 \Leftrightarrow \pi_1 \end{bmatrix}, \quad \boldsymbol{b}(p)(\boldsymbol{\pi}) = \begin{bmatrix} 2\pi_0 + 5 \end{bmatrix},$$

then

$$\begin{aligned} \boldsymbol{n}(p,\boldsymbol{i})(\boldsymbol{\pi}) &= \begin{bmatrix} \pi_0\pi_1 & \pi_0 \Leftrightarrow \pi_1 \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \end{bmatrix} + \begin{bmatrix} 2\pi_0 + 5 \end{bmatrix} \\ &= \begin{bmatrix} \pi_0\pi_1 i_0 + (\pi_0 \Leftrightarrow \pi_1) i_1 + 2\pi_0 + 5 \end{bmatrix}. \end{aligned}$$

Note that the index vector is linear in the iterator vector, although the coefficients are polynomials in the parameters.

For a parametric signal flow graph we have the same assumptions as for a non-parametric signal flow graph. So, for the iterator bound vectors we again assume that only the first entry may be infinite. Next, for the production of data we again assume single assignments, i.e., for each parameter setting, each element of an array can be produced at most once. Furthermore, if an operation v is repeated infinitely, i.e., $I_0(v) = \infty$, then we again assume $A_{00}(p) >_{\text{all}} 0$, and $A_{k0} =_{\text{all}} 0$, for all $k = 1,\ldots,\alpha(p) \Leftrightarrow 1$.

8.1.3 Parametric Schedules

Next, we need the definition of a parametric schedule.

Definition 8.4 (Parametric Schedule). Given a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$ and a parametric signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, a parametric schedule σ is given by a parametric time assignment and a processing unit assignment, represented by a 4-tuple $(\boldsymbol{p}, s, W, h)$, where

- the parametric time assignment is given by a period vector $\boldsymbol{p}(v) \in \mathcal{E}^{\delta(v)}$ and a start time $s(v) \in \mathcal{E}$, for each operation $v \in V$, and
- the processing unit assignment is given by a set W of processing units and a function $h : V \to W$ that assigns each operation to a processing unit that executes it, such that $t(v) = t(h(v))$, for all $v \in V$.

The set of all possible parametric schedules is denoted by $\mathcal{S}$. □

A parametric schedule results in a parametric time $c(v, \boldsymbol{i})$ at which an execution $\boldsymbol{i}$ of operation v is scheduled, given by

$$c(v, \boldsymbol{i})(\boldsymbol{\pi}) = \boldsymbol{p}^{\mathrm{T}}(v)(\boldsymbol{\pi})\, \boldsymbol{i} + s(v)(\boldsymbol{\pi}),$$

for all $\boldsymbol{\pi} \in \Pi$. For example, if we assign $s(in) = 1$ in the video algorithm of Figure 8.1, then

$$c(in, \boldsymbol{i})(\boldsymbol{\pi}) = \begin{bmatrix} \pi_1 & 1 \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \end{bmatrix} + 1 = \pi_1 i_0 + i_1 + 1.$$

Again, the time is linear in the iterators, although the coefficients are polynomials in the parameters.

8.1.4 Parametric Constraints

Next, we define the constraints that a parametric schedule must satisfy. They are similar to the non-parametric constraints, except that now the latter must be satisfied for all possible parameter settings.

Definition 8.5 (Parametric Timing Constraints). Given are a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$, a parametric signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, a parametric schedule $\sigma = (\boldsymbol{p}, s, W, h)$, and for each operation $v \in V$ a parametric lower bound $\underline{\boldsymbol{p}}(v) \in \mathcal{E}_\infty^{\delta(v)}$ and a parametric upper bound $\overline{\boldsymbol{p}}(v) \in \mathcal{E}_\infty^{\delta(v)}$ on the period vector $\boldsymbol{p}(v)$, and a parametric lower bound $\underline{s}(v) \in \mathcal{E}_\infty$ and a parametric upper bound $\overline{s}(v) \in \mathcal{E}_\infty$ on the start time $s(v)$. Then the parametric timing constraints specify that

$$\underline{\boldsymbol{p}}(v) \leq_{\text{all}} \boldsymbol{p}(v) \leq_{\text{all}} \overline{\boldsymbol{p}}(v) \quad \wedge \quad \underline{s}(v) \leq_{\text{all}} s(v) \leq_{\text{all}} \overline{s}(v),$$

for all $v \in V$. □

Again, if an operation v is repeated infinitely, i.e., $I_0(v) = \infty$, then we assume that the period of the corresponding iterator is fixed and positive, i.e., $\underline{p}_0(v) =_{\text{all}} \overline{p}_0(v) >_{\text{all}} 0$. Note that the period may still depend on the parameters.

Definition 8.6 (Parametric Processing Unit Constraints). Given are a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$, a parametric signal flow graph $G = (V, t, \boldsymbol{I}, E, \boldsymbol{A}, \boldsymbol{b})$, and a parametric schedule $\sigma = (\boldsymbol{p}, s, W, h)$. Then for each parameter setting $\boldsymbol{\pi} \in \Pi$, for each execution $\boldsymbol{i} \in \mathcal{I}(u)(\boldsymbol{\pi})$ of an operation $u \in V$, and for each execution

$\boldsymbol{j} \in \mathcal{I}(v)(\boldsymbol{\pi})$ of an operation $v \in V$, with $h(u) = h(v)$ and $(u,\boldsymbol{i}) \neq (v,\boldsymbol{j})$, the parametric processing unit constraints specify that

$$c(u,\boldsymbol{i})(\boldsymbol{\pi}) + k \neq c(v,\boldsymbol{j})(\boldsymbol{\pi}) + l,$$

for all $k,l \in \{0,\ldots,o \Leftrightarrow 1\}$, where $o = o(t(u)) = o(t(v))$. □

Definition 8.7 (Parametric Precedence Constraints). Given are a parameter set $(\boldsymbol{\pi},\underline{\boldsymbol{\pi}},\overline{\boldsymbol{\pi}})$, a parametric signal flow graph $G = (V,t,\boldsymbol{I},E,\boldsymbol{A},\boldsymbol{b})$, and a parametric schedule $\sigma = (\boldsymbol{p},s,W,h)$. Then for each parameter setting $\boldsymbol{\pi} \in \Pi$, for each execution $\boldsymbol{i} \in \mathcal{I}(p)(\boldsymbol{\pi})$ of an output port $p \in O$, and for each execution $\boldsymbol{j} \in \mathcal{I}(q)(\boldsymbol{\pi})$ of an input port $q \in I$, with $(p,q) \in E$, the parametric precedence constraints specify that

$$\boldsymbol{n}(p,\boldsymbol{i})(\boldsymbol{\pi}) = \boldsymbol{n}(q,\boldsymbol{j})(\boldsymbol{\pi}) \;\Rightarrow\; c(p,\boldsymbol{i})(\boldsymbol{\pi}) < c(q,\boldsymbol{j})(\boldsymbol{\pi}).$$

□

Again, if we have an output port p and an input port q, with $(p,q) \in E$, and both p and q are infinitely repeated, i.e., $I_0(p) = I_0(q) = \infty$, then we assume that the pace of increasing n_0 at the production side and at the consumption side is equal, in order to guarantee that there is no overflow or underflow of data. So, we assume that for all parameter settings $\boldsymbol{\pi} \in \Pi$, $A_{00}(p)(\boldsymbol{\pi})/p_0(p)(\boldsymbol{\pi}) = A_{00}(q)(\boldsymbol{\pi})/p_0(q)(\boldsymbol{\pi})$, in this case, i.e., we assume $A_{00}(p)p_0(q) =_{\text{all}} A_{00}(q)p_0(p)$.

A parametric schedule $\sigma \in \mathcal{S}$ is called feasible if it obeys the parametric timing constraints, the parametric processing unit constraints, and the parametric precedence constraints. The set of all feasible parametric schedules is denoted by $\mathcal{S}'$.

An example of a parametric time assignment for the video algorithm of Figure 8.1, that satisfies the parametric precedence constraints, is given by $s(in) = 1$ and $s(out) = \pi_0\pi_1 \Leftrightarrow \pi_0 \Leftrightarrow \pi_1 + 3$; see Figure 8.2. The most constraining data dependence

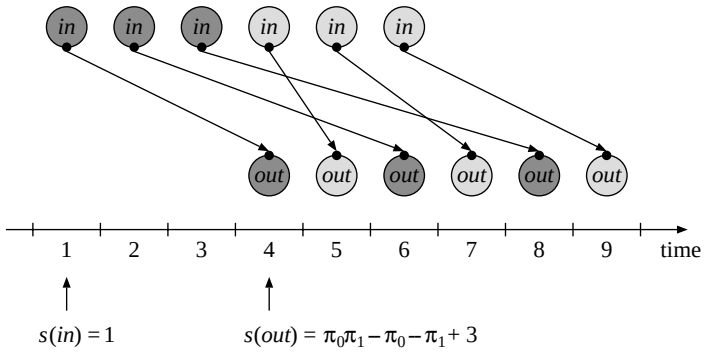


Figure 8.2. A feasible parametric time assignment for the transposition example of Figure 8.1, here shown for $\pi_0 = 2$ and $\pi_1 = 3$.

is the one concerning $\boldsymbol{i} = [\pi_0 \Leftrightarrow 1 \;\; 0]^{\mathrm{T}}$ and $\boldsymbol{j} = [0 \;\; \pi_0 \Leftrightarrow 1]^{\mathrm{T}}$, which specifies that

$$s(in) + \pi_1(\pi_0 \Leftrightarrow 1) < s(out) + 1(\pi_0 \Leftrightarrow 1),$$

which is equivalent to

$$s(out) \Leftrightarrow s(in) > \pi_0\pi_1 \Leftrightarrow \pi_0 \Leftrightarrow \pi_1 + 1.$$

8.1.5 Parametric Objectives

Next, we revisit the total area cost of a schedule, which consists of a processing unit cost, a storage cost, and an access cost. The processing unit cost of a parametric schedule is determined similarly to that of a non-parametric schedule. The storage cost and access cost are determined differently, since the set $\mathcal{L}(A,c)$, denoting the array elements of array cluster $A \in \mathcal{A}$ that are alive at time c, and the set $\mathcal{B}(A,c)$, denoting the array elements of array cluster A that are accessed at time c, now depend on the parameters. For each parameter setting $\boldsymbol{\pi} \in \Pi$, they are given by

$$\mathcal{L}(A,c)(\boldsymbol{\pi}) = \{\boldsymbol{n} \in \mathbb{Z}^{\alpha} \mid p,q \in A \;\wedge\; (p,q) \in E \;\wedge\; \boldsymbol{i} \in \mathcal{I}(p)(\boldsymbol{\pi}) \;\wedge\; \boldsymbol{j} \in \mathcal{I}(q)(\boldsymbol{\pi}) \;\wedge\; \boldsymbol{n} = \boldsymbol{n}(p,\boldsymbol{i})(\boldsymbol{\pi}) = \boldsymbol{n}(q,\boldsymbol{j})(\boldsymbol{\pi}) \;\wedge\; c(p,\boldsymbol{i})(\boldsymbol{\pi}) < c \leq c(q,\boldsymbol{j})(\boldsymbol{\pi})\},$$

and

$$\mathcal{B}(A,c)(\boldsymbol{\pi}) = \{\boldsymbol{n} \in \mathbb{Z}^{\alpha} \mid p \in A \;\wedge\; \boldsymbol{i} \in \mathcal{I}(p)(\boldsymbol{\pi}) \;\wedge\; \boldsymbol{n} = \boldsymbol{n}(p,\boldsymbol{i})(\boldsymbol{\pi}) \;\wedge\; c(p,\boldsymbol{i})(\boldsymbol{\pi}) = c\}.$$

Now we can define the total area cost of a parametric schedule, which is a straightforward extension of the non-parametric total area cost, as given in Definition 2.8.

Definition 8.8 (Parametric Total Area Cost). The parametric total area cost of a parametric schedule $\sigma \in \mathcal{S}'$ is the sum of the processing unit cost, the storage cost, and the access cost, and is given by

$$\begin{aligned} f(\sigma) \;\; = \;\; & \sum_{w \in W} a(t(w)) \\ & + a(t_{\mathrm{v}}) \max_{\boldsymbol{\pi} \in \Pi} \max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{L}(A,c)(\boldsymbol{\pi})| \\ & + a(t_{\mathrm{a}}) \max_{\boldsymbol{\pi} \in \Pi} \max_{c \in \mathbb{Z}} \sum_{A \in \mathcal{A}} |\mathcal{B}(A,c)(\boldsymbol{\pi})|. \end{aligned}$$

□

8.1.6 Parametric Problem Formulation

The parametric multidimensional periodic scheduling problem can now be defined as follows.

Definition 8.9 (Parametric Multidimensional Periodic Scheduling (PMPS)). Given are a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$, a parametric signal flow graph G, a set of resource types T^*, and parametric lower and upper bounds on the period vectors

and start times of the operations. Find a parametric schedule σ that obeys the parametric timing constraints, the parametric processing unit constraints, and the parametric precedence constraints, for which the parametric total area cost $f(\sigma)$ is minimal. □

Theorem 8.1. *The decision variant of PMPS is NP-hard in the strong sense.*
Proof. PMPS is a generalization of MPS. □

8.1.7 Parametric Lexicographical Executions

Next, we give a definition of lexicographical executions in the case of parameters.

Definition 8.10 (Parametric Lexicographical Execution). An iterator bound vector $\boldsymbol{I} \in \mathcal{P}_\infty^\delta$, a period vector $\boldsymbol{p} \in \mathcal{E}^\delta$, and an occupation time $o \in \mathbb{N}_+$ are said to give a parametric lexicographical execution, denoted by $\mathrm{plex}(\boldsymbol{I}, \boldsymbol{p}, o)$, if and only if for all parameter settings $\boldsymbol{\pi} \in \Pi$ and for all vectors $\boldsymbol{i}, \boldsymbol{j} \in \mathbb{Z}^\delta$ with $\mathbf{0} \leq \boldsymbol{i}, \boldsymbol{j} \leq \boldsymbol{I}(\boldsymbol{\pi})$ holds

$$\boldsymbol{i} <_{\mathrm{lex}} \boldsymbol{j} \;\Leftrightarrow\; \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\, \boldsymbol{i} + o \leq \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\, \boldsymbol{j}.$$

□

Checking whether an iterator bound vector, a period vector, and an occupation time give a parametric lexicographical execution, can be done by means of the following theorem.

Theorem 8.2. *Given are an iterator bound vector $\boldsymbol{I} \in \mathcal{P}_\infty^\delta$, with $I_k >_{\mathrm{all}} 0$, for all $k = 0,\ldots,\delta \Leftrightarrow 1$, a period vector $\boldsymbol{p} \in \mathcal{E}^\delta$, and an occupation time $o \in \mathbb{N}_+$. Then*

$$\mathrm{plex}(\boldsymbol{I}, \boldsymbol{p}, o)$$

holds if and only if

$$p_0 >_{\mathrm{all}} p_1 >_{\mathrm{all}} \cdots >_{\mathrm{all}} p_{\delta-1} >_{\mathrm{all}} 0,$$

and

$$p_k \geq_{\mathrm{all}} \sum_{l=k+1}^{\delta-1} p_l I_l + o,$$

for all $k = 0,\ldots,\delta \Leftrightarrow 1$.
Proof. The proof is analogous to the proof of Theorem 2.1. □

8.2 Parametric ILP

In this section we present a primal all-integer ILP algorithm that incorporates parameters. We do this for problems with one cost function only; the extension to multiple cost functions is analogous to that of Section 4.3. In this section, we first describe the parametric ILP problem. Secondly, we describe how to determine bounds on parametric expressions and how to perform divisions on parametric expressions.

Finally, we describe a parametric primal all-integer ILP algorithm, based on the latter two methods.

8.2.1 Description of the Problem

The parametric ILP problem we consider is the following.

Definition 8.11 (Parametric ILP). Given are a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$ and a matrix $\boldsymbol{M} \in \mathcal{E}^{(m+1)\times(n+1)}$. Find a parametric expression $z \in \mathcal{E}$, such that for all parameter settings $\boldsymbol{\pi} \in \Pi$,

$$\begin{array}{rll} z(\boldsymbol{\pi}) = \text{maximum} & x_0 & \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}(\boldsymbol{\pi})\,\boldsymbol{t} & \\ & x_i \geq 0 & (i \geq 1) \\ & \boldsymbol{x} \text{ integer}, & \end{array}$$

where for each $\boldsymbol{\pi} \in \Pi$ we maximize over all integer vectors

$$\boldsymbol{x} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_m \end{bmatrix}, \quad \boldsymbol{t} = \begin{bmatrix} 1 \\ \Leftrightarrow t_1 \\ \vdots \\ \Leftrightarrow t_n \end{bmatrix}.$$

Again, the general row equation of $\boldsymbol{x} = \boldsymbol{M}(\boldsymbol{\pi})\,\boldsymbol{t}$ is given by

$$x_i = M_{i0}(\boldsymbol{\pi}) + \sum_{j=1}^{n} M_{ij}(\boldsymbol{\pi})\ (\Leftrightarrow t_j),$$

for $i = 0,\ldots,m$, and the last n equations initially have the form

$$x_{m-n+j} = \Leftrightarrow(\Leftrightarrow t_j),$$

for $j = 1,\ldots,n$. □

For example, consider a parametric ILP problem given by

$$\begin{array}{ll} \text{maximize} & \pi_1 a \\ \text{subject to} & a \leq \pi_0 \\ & a \leq 2\pi_1 \\ & a \geq 0 \\ & a \text{ integer}, \end{array} \tag{8.1}$$

with parameter lower bounds $\underline{\boldsymbol{\pi}} = [5\ \ 12]^{\mathrm{T}}$ and upper bounds $\overline{\boldsymbol{\pi}} = [10\ \ 16]^{\mathrm{T}}$, i.e., $5 \leq \pi_0 \leq 10$ and $12 \leq \pi_1 \leq 16$. In this example, we have four variables

$$\begin{aligned} x_0 &= \pi_1 a \\ x_1 &= \pi_0 \Leftrightarrow a \\ x_2 &= 2\pi_1 \Leftrightarrow a \\ x_3 &= a, \end{aligned}$$

of which x_1, x_2, and x_3 must be non-negative. Initially, the non-basic variable is $t_1 = x_3 = a$. The optimum solution of this parametric ILP problem is given by $a = \pi_0$, resulting in

$$z(\boldsymbol{\pi}) = \pi_1 \pi_0,$$

since the constraint $a \leq \pi_0$ is tighter than the constraint $a \leq 2\pi_1$, for the given parameter bounds.

Since there are instances of the parametric ILP problem for which no solution $z \in \mathcal{E}$ exists, we also define a relaxed problem, resulting in an expression that bounds the optimum from above. If for an instance of the parametric processing unit conflict problem the parametric ILP problem does not have a solution, then we may solve the relaxed parametric ILP problem, from which we possibly can conclude that there is no processing unit conflict.

Definition 8.12 (Relaxed Parametric ILP). Given are a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$ and a matrix $\boldsymbol{M} \in \mathcal{E}^{(m+1)\times(n+1)}$, of the same form as in the parametric ILP problem. Find a parametric expression $z \in \mathcal{E}$, such that for all parameter settings $\boldsymbol{\pi} \in \Pi$,

$$\begin{array}{rll} z(\boldsymbol{\pi}) \geq \text{maximum} & x_0 & \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}(\boldsymbol{\pi})\, \boldsymbol{t} & \\ & x_i \geq 0 & (i \geq 1) \\ & \boldsymbol{x} \text{ integer}, & \end{array}$$

and such that there is no other parametric expression $z' \in \mathcal{E}$ satisfying these constraints, for which $z' \leq_{\text{all}} z$ but not $z' =_{\text{all}} z$. □

For example, consider the instance given in (8.1), but with parameter lower bounds $\underline{\boldsymbol{\pi}} = [5\ 6]^{\mathrm{T}}$ and upper bounds $\overline{\boldsymbol{\pi}} = [20\ 16]^{\mathrm{T}}$, i.e., $5 \leq \pi_0 \leq 20$ and $6 \leq \pi_1 \leq 16$. Now, for some parameter settings the constraint $a \leq \pi_0$ is tighter than the constraint $a \leq 2\pi_1$, but for other parameter settings it is the other way around. One solution of the relaxed parametric ILP problem is now given by $a = \pi_0$, resulting in

$$z(\boldsymbol{\pi}) = \pi_1 \pi_0,$$

and a second solution is given by $a = 2\pi_1$, resulting in

$$z(\boldsymbol{\pi}) = 2\pi_1^2.$$

Here, the relaxations consist of omitting the constraint $a \leq 2\pi_1$ for the first solution, and omitting the constraint $a \leq \pi_0$ for the second solution. As we can see, there is no unique solution for this instance of the relaxed parametric ILP problem. Note that if an instance of the parametric ILP problem has a solution z, then all solutions z' of the corresponding instance of the relaxed problem satisfy $z' =_{\text{all}} z$.

8.2.2 Bounds on Parametric Expressions

Before presenting a parametric ILP algorithm, we need a routine to compare parametric expressions. To this end, we present in this section two theorems that can be used to determine lower and upper bounds on parametric expressions. The first theorem gives straightforward bounds, which can be computed very fast. If these bounds are tight enough to determine, for instance, whether an expression $\varepsilon >_{\text{all}} 0$, then no tighter bounds have to be computed. If not, the second theorem can be applied to give tighter bounds, at the expense of some more computing time.

Theorem 8.3. *Given are a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$ with $\underline{\boldsymbol{\pi}} \geq \mathbf{0}$, and a parametric expression $\varepsilon \in \mathcal{E}$ that can be written as*

$$\varepsilon(\boldsymbol{\pi}) = \sum_{t \in T} c_t \prod_{p \in P_t} \pi_p,$$

where T is a set of terms, with for each term $t \in T$ an integer coefficient $c_t \neq 0$ and a multiset P_t of parameter indices. Then a lower and an upper bound on ε are given by

$$\min_{\boldsymbol{\pi} \in \Pi} \varepsilon(\boldsymbol{\pi}) \geq \sum_{t \in T_+} c_t \prod_{p \in P_t} \underline{\pi}_p + \sum_{t \in T_-} c_t \prod_{p \in P_t} \overline{\pi}_p, \tag{8.2}$$

and

$$\max_{\boldsymbol{\pi} \in \Pi} \varepsilon(\boldsymbol{\pi}) \leq \sum_{t \in T_+} c_t \prod_{p \in P_t} \overline{\pi}_p + \sum_{t \in T_-} c_t \prod_{p \in P_t} \underline{\pi}_p, \tag{8.3}$$

where $T_+ = \{t \in T \mid c_t > 0\}$ and $T_- = \{t \in T \mid c_t < 0\}$. If, furthermore, $P_{t_+} \cap P_{t_-} = \emptyset$, for all $t_+ \in T_+$ and $t_- \in T_-$, then equality holds in (8.2) and (8.3).

Proof. Since $\boldsymbol{\pi} \geq \underline{\boldsymbol{\pi}} \geq \mathbf{0}$, we know that

$$\prod_{p \in P} \pi_p \geq 0,$$

for any multiset P of parameter indices. Therefore, a lower bound on a term

$$c_t \prod_{p \in P_t} \pi_p$$

is obtained by filling in the lower bounds of the parameters if $c_t > 0$, and by filling in the upper bounds if $c_t < 0$. In a similar way, an upper bound on each of the terms can be determined.

If, furthermore, $P_{t_+} \cap P_{t_-} = \emptyset$, for all $t_+ \in T_+$ and $t_- \in T_-$, then the minimum value of the expression $\varepsilon(\boldsymbol{\pi})$ is obtained by taking $\pi_p = \underline{\pi}_p$ if there is a $t \in T_+$ with $p \in P_t$, and taking $\pi_p = \overline{\pi}_p$ if there is a $t \in T_-$ with $p \in P_t$. The maximum of $\varepsilon(\boldsymbol{\pi})$ is determined analogously. □

The condition $\underline{\boldsymbol{\pi}} \geq \mathbf{0}$ in the above theorem can be obtained by substituting the parameters π_p with $\underline{\pi}_p < 0$ by $\underline{\pi}_p + \pi'_p$. This results in a new parameter π'_p with lower

bound 0 and upper bound $\overline{\pi}_p \Leftrightarrow \underline{\pi}_p$.

The next theorem can be used to derive tighter bounds, in the case that the condition $P_{t_+} \cap P_{t_-} = \emptyset$ does not hold for all $t_+ \in T_+$ and $t_- \in T_-$.

Theorem 8.4. *Given are a parameter set* $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$*, and a parametric expression* $\varepsilon \in \mathcal{E}$ *that can be written as*

$$\varepsilon(\boldsymbol{\pi}) = q(\boldsymbol{\pi})d(\boldsymbol{\pi}) + r(\boldsymbol{\pi}),$$

with $q, d, r \in \mathcal{E}$*. If the expression* d *is bounded by two other expressions* $l, u \in \mathcal{E}$*, i.e.,* $l \leq_{\text{all}} d \leq_{\text{all}} u$*, then*

$$\min_{\boldsymbol{\pi}\in\Pi} \varepsilon(\boldsymbol{\pi}) \geq \min\{\min_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})l(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\ ,\ \min_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})u(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\}, \tag{8.4}$$

and

$$\max_{\boldsymbol{\pi}\in\Pi} \varepsilon(\boldsymbol{\pi}) \leq \max\{\max_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})l(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\ ,\ \max_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})u(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\}. \tag{8.5}$$

Proof. If we define $\Pi_+ = \{\boldsymbol{\pi} \in \Pi \mid q(\boldsymbol{\pi}) \geq 0\}$ and $\Pi_- = \{\boldsymbol{\pi} \in \Pi \mid q(\boldsymbol{\pi}) \leq 0\}$, then we can derive that

$$\begin{aligned}
\min_{\boldsymbol{\pi}\in\Pi} \varepsilon(\boldsymbol{\pi}) &= \min_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})d(\boldsymbol{\pi}) + r(\boldsymbol{\pi})) \\
&= \min\{\min_{\boldsymbol{\pi}\in\Pi_+}(q(\boldsymbol{\pi})d(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\ ,\ \min_{\boldsymbol{\pi}\in\Pi_-}(q(\boldsymbol{\pi})d(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\} \\
&\geq \min\{\min_{\boldsymbol{\pi}\in\Pi_+}(q(\boldsymbol{\pi})l(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\ ,\ \min_{\boldsymbol{\pi}\in\Pi_-}(q(\boldsymbol{\pi})u(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\} \\
&\geq \min\{\min_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})l(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\ ,\ \min_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})u(\boldsymbol{\pi}) + r(\boldsymbol{\pi}))\}.
\end{aligned}$$

The proof of (8.5) is analogous. □

A special case of the above theorem can be applied by writing an expression $\varepsilon \in \mathcal{E}$ as

$$\varepsilon(\boldsymbol{\pi}) = q(\boldsymbol{\pi})\pi_p + r(\boldsymbol{\pi}),$$

with $q, r \in \mathcal{E}$. For this special case we have $d(\boldsymbol{\pi}) = \pi_p$, with bounds $\underline{\pi}_p$ and $\overline{\pi}_p$, resulting in

$$\min_{\boldsymbol{\pi}\in\Pi} \varepsilon(\boldsymbol{\pi}) \geq \min\{\min_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})\underline{\pi}_p + r(\boldsymbol{\pi}))\ ,\ \min_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})\overline{\pi}_p + r(\boldsymbol{\pi}))\},$$

and

$$\max_{\boldsymbol{\pi}\in\Pi} \varepsilon(\boldsymbol{\pi}) \leq \max\{\max_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})\underline{\pi}_p + r(\boldsymbol{\pi}))\ ,\ \max_{\boldsymbol{\pi}\in\Pi}(q(\boldsymbol{\pi})\overline{\pi}_p + r(\boldsymbol{\pi}))\}.$$

To show that this can give tighter bounds, consider the following example with parameters $\boldsymbol{\pi} = [\pi_0\ \ \pi_1]^{\mathrm{T}}$ and bounds $\underline{\boldsymbol{\pi}} = [5\ \ 3]^{\mathrm{T}}$ and $\overline{\boldsymbol{\pi}} = [10\ \ 20]^{\mathrm{T}}$, and expression

$$\varepsilon(\boldsymbol{\pi}) = \pi_0\pi_1 \Leftrightarrow 4\pi_1 + 12.$$

Applying Theorem 8.3 then gives a lower bound

$$\min_{\boldsymbol{\pi}\in\Pi} \varepsilon(\boldsymbol{\pi}) \geq 5 \cdot 3 \Leftrightarrow 4 \cdot 20 + 12 = \Leftrightarrow 53,$$

whereas writing

$$\varepsilon(\boldsymbol{\pi}) = (\pi_0 \Leftrightarrow 4)\pi_1 + 12,$$

and applying Theorem 8.4, gives a lower bound

$$\begin{aligned} \min_{\boldsymbol{\pi}\in\Pi} \varepsilon(\boldsymbol{\pi}) &\geq \min\{\min_{\boldsymbol{\pi}\in\Pi}((\pi_0 \Leftrightarrow 4) \cdot 3 + 12)\ ,\ \min_{\boldsymbol{\pi}\in\Pi}((\pi_0 \Leftrightarrow 4) \cdot 20 + 12)\} \\ &= \min\{15, 32\} \\ &= 15, \end{aligned}$$

which is not only a better lower bound, but also the minimum value of ε.

Although the above two theorems generally provide rough bounds on the expressions, they are good enough to be used for solving the parametric ILP problem instances we encounter in practice. Furthermore, they can be computed very fast, which is a prerequisite for use in a parametric ILP algorithm.

8.2.3 Division of Parametric Expressions

In addition to comparing expressions, a parametric ILP algorithm requires a division routine for parametric expressions. This means that we are given two expressions $\varepsilon, d \in \mathcal{E}$, with the divisor $d >_{\text{all}} 0$, and we want to determine two other expressions $q, r \in \mathcal{E}$, with

$$\varepsilon(\boldsymbol{\pi}) = q(\boldsymbol{\pi})d(\boldsymbol{\pi}) + r(\boldsymbol{\pi}), \tag{8.6}$$

such that the remainder r satisfies $0 \leq_{\text{all}} r <_{\text{all}} d$.

The method we apply is the following. First, we perform a long division of the polynomial ε by the polynomial d. In order to obtain integer coefficients, we round fractional coefficients to the closest integer value. In this way, we obtain initial expressions $q, r \in \mathcal{E}$ that satisfy (8.6), but for which not necessarily $0 \leq_{\text{all}} r <_{\text{all}} d$ holds. Next, we add to q the smallest possible integer k, and subtract kd from r, such that we obtain $r <_{\text{all}} d$. This value of k can be determined by means of a binary search, starting with an interval determined by the bounds on the expressions r and d. If after this second step $r \geq_{\text{all}} 0$, then a solution is found. Otherwise, we try the above procedure with rounding the coefficients up, or with a rounding towards zero. If a solution (q, r) is found, we say that ε can be divided by d, and we denote its quotient q by $\lfloor \varepsilon / d \rfloor$. Note that if no solution is found, this may be due to the fact that no $q, r \in \mathcal{E}$ exist that satisfy (8.6).

Similarly to the determination of bounds, the above procedure for division is quite straightforward. Nevertheless, in practice it is highly applicable to find a quotient and a remainder, if they exist.

8.2.4 The Parametric Primal All-Integer ILP Algorithm

Based on the methods of the previous two sections, we now present a parametric primal all-integer ILP algorithm, which is used to solve the sub-problems in parametric multidimensional periodic scheduling. In order to keep the algorithm simple, we do not use a reference row as in Glover's primal all-integer algorithm, which was used to assure finite convergence. We assume that the initial matrix $\boldsymbol{M}$ is primal feasible, i.e., that $M_{i0} \geq_{\text{all}} 0$, for all $i \geq 1$. The algorithm is shown in Figure 8.3. Note that the algorithm may abort at several places, in which case no solution is

---

1. If $M_{0j} \geq_{\text{all}} 0$, for all $j \geq 1$, then $\boldsymbol{x} = \boldsymbol{M}_{\cdot 0}$ is an optimal solution, resulting in a solution $z = M_{00}$ for the parametric ILP problem, so stop. Otherwise, select an $s \geq 1$ such that $M_{0s} \leq_{\text{all}} 0$ but not $M_{0s} =_{\text{all}} 0$, if such an s exists. If no such s exists, then abort.

2. If $M_{is} \leq_{\text{all}} 0$, for all $i \geq 1$, then the problem has an unbounded optimum, so stop. If there is a row $i \geq 1$ for which neither $M_{is} \leq_{\text{all}} 0$ nor $M_{is} >_{\text{all}} 0$, then abort. Otherwise, determine a row $i \geq 1$ with $M_{is} >_{\text{all}} 0$, for which M_{i0} can be divided by M_{is} and

$$M_{js} \lfloor M_{i0}/M_{is} \rfloor \leq_{\text{all}} M_{j0},$$

for all $j \geq 1$ with $M_{js} >_{\text{all}} 0$.

3. Divide the elements M_{ij}, $j = 0,\ldots,n$, by M_{is}, in order to determine the cut

$$y = \lfloor M_{i0}/M_{is} \rfloor + \sum_{j=1}^{n} \lfloor M_{ij}/M_{is} \rfloor (-t_j).$$

If one of the elements cannot be divided, then abort. Otherwise, use this cut as equation v in the following steps.

4. Determine $\tilde{\boldsymbol{M}}$ by

$$\tilde{\boldsymbol{M}}_{\cdot j} = \begin{cases} -\boldsymbol{M}_{\cdot s} & \text{if } j = s \\ \boldsymbol{M}_{\cdot j} - M_{vj}\boldsymbol{M}_{\cdot s} & \text{if } j \neq s. \end{cases}$$

5. Let $\tilde{t}_s = x_v$, and $\tilde{t}_j = t_j$, for all $j \neq s$. Designate $\tilde{\boldsymbol{M}}$ and $\tilde{\boldsymbol{t}}$ to be the current matrix $\boldsymbol{M}$ and vector $\boldsymbol{t}$, respectively, and return to Step 1.

Figure 8.3. The parametric primal all-integer algorithm.

found. What to do in this case is discussed after the following example.

Example 8.1. Consider a parametric ILP problem given by

$$\begin{array}{ll} \text{maximize} & 3\pi_1 a + b + \pi_0\pi_2 \\ \text{subject to} & 2\pi_1 a + 2b \leq 2\pi_0\pi_2 + 7 \\ & 0 \leq a \leq \pi_0 \\ & 0 \leq b \leq \pi_0\pi_2 \\ & a, b \text{ integer}, \end{array}$$

with parameter lower bounds $\underline{\boldsymbol{\pi}} = [4\ 2\ 3]^{\mathrm{T}}$ and upper bounds $\overline{\boldsymbol{\pi}} = [14\ 3\ 5]^{\mathrm{T}}$. In this example, we have six variables

$$\begin{aligned} x_0 &= 3\pi_1 a + b + \pi_0\pi_2 \\ x_1 &= 2\pi_0\pi_2 + 7 \Leftrightarrow 2\pi_1 a \Leftrightarrow 2b \\ x_2 &= \pi_0 \Leftrightarrow a \\ x_3 &= \pi_0\pi_2 \Leftrightarrow b \\ x_4 &= a \\ x_5 &= b, \end{aligned}$$

of which $x_1,\ldots,x_5$ must be non-negative. Initially, the non-basic variables are $t_1 = x_4 = a$ and $t_2 = x_5 = b$. The initial matrix $\boldsymbol{M}$ is given by

$$\left[\begin{array}{r|rr} \pi_0\pi_2 & \Leftrightarrow 3\pi_1 & \Leftrightarrow 1 \\ \hline 2\pi_0\pi_2 + 7 & 2\pi_1 & 2 \\ \pi_0 & 1 & 0 \\ \pi_0\pi_2 & 0 & 1 \\ 0 & \Leftrightarrow 1 & 0 \\ 0 & 0 & \Leftrightarrow 1 \end{array}\right].$$

In the first iteration, we select column $s = 1$, since $\Leftrightarrow 3\pi_1 <_{\text{all}} 0$. This gives two candidate pivot rows, namely $i = 1$ and $i = 2$. Row $i = 2$ is chosen, since π_0 can be divided by 1, and

$$2\pi_1 \cdot \pi_0 \leq_{\text{all}} 2\pi_0\pi_2 + 7,$$

as can be seen from the fact that

$$2\pi_0\pi_2 + 7 \Leftrightarrow 2\pi_0\pi_1 = 2\pi_0(\pi_2 \Leftrightarrow \pi_1) + 7 \geq 0 + 7 \geq 0.$$

The cut that is determined from row $i = 1$ is then given by

$$y = \pi_0 + 1(\Leftrightarrow t_1) + 0(\Leftrightarrow t_2),$$

which results in the next matrix to be

$$\left[\begin{array}{r|rr} \pi_0\pi_2+3\pi_0\pi_1 & 3\pi_1 & \Leftrightarrow 1 \\ \hline 2\pi_0\pi_2 \Leftrightarrow 2\pi_0\pi_1+7 & \Leftrightarrow 2\pi_1 & 2 \\ 0 & \Leftrightarrow 1 & 0 \\ \pi_0\pi_2 & 0 & 1 \\ \pi_0 & 1 & 0 \\ 0 & 0 & \Leftrightarrow 1 \end{array}\right].$$

In the second iteration, we select column $s = 2$, since $\Leftrightarrow 1 <_{\text{all}} 0$. This gives again two candidate pivot rows, namely $i = 1$ and $i = 3$. Row $i = 1$ is chosen, since $2\pi_0\pi_2 \Leftrightarrow 2\pi_0\pi_1 + 7$ can be divided by 2, giving quotient $\pi_0\pi_2 \Leftrightarrow \pi_0\pi_1 + 3$, and since

$$1 \cdot (\pi_0\pi_2 \Leftrightarrow \pi_0\pi_1 + 3) \leq_{\text{all}} \pi_0\pi_2.$$

The cut that is determined from row $i = 1$ is then given by

$$y = \pi_0\pi_2 \Leftrightarrow \pi_0\pi_1 + 3 \Leftrightarrow \pi_1(\Leftrightarrow t_1) + 1(\Leftrightarrow t_2),$$

which results in the next matrix to be

$$\left[\begin{array}{r|rr} 2\pi_0\pi_2+2\pi_0\pi_1+3 & 2\pi_1 & 1 \\ \hline 1 & 0 & \Leftrightarrow 2 \\ 0 & \Leftrightarrow 1 & 0 \\ \pi_0\pi_1 \Leftrightarrow 3 & \pi_1 & \Leftrightarrow 1 \\ \pi_0 & 1 & 0 \\ \pi_0\pi_2 \Leftrightarrow \pi_0\pi_1 + 3 & \Leftrightarrow \pi_1 & 1 \end{array}\right].$$

In the third iteration, the algorithm stops with an optimal solution, given by $z = 2\pi_0\pi_2 + 2\pi_0\pi_1 + 3$. □

As we can see in the parametric primal all-integer algorithm, there are several occasions where the algorithm may abort. In those cases, no solution of the parametric ILP problem is found. This may be due to the fact that no solution exists, or due to the fact that the methods for bounding and division of parametric expressions are not effective enough. In practice, the reason is mostly the former of the two. In the case that the algorithm aborts, we may apply some of the following relaxation rules, in order to continue with the algorithm and to find a solution of the relaxed parametric ILP problem.

- If in Step 2 there is a row $i \geq 1$ for which neither $M_{is} \leq_{\text{all}} 0$ nor $M_{is} >_{\text{all}} 0$, then omit that row.
- If in Step 2 there is a row $i \geq 1$ with $M_{is} >_{\text{all}} 0$ and for which a cut can be computed in Step 3, but for which there is another row $j \geq 1$ with $M_{js} >_{\text{all}} 0$ that does not satisfy

$$M_{js}\lfloor M_{i0}/M_{is}\rfloor \leq_{\text{all}} M_{j0},$$

then omit that row j in order to be able to use row i as the pivot row.

- If in Step 3 no cut can be determined because the elements M_{ij} in row i can not all be divided by M_{is}, then try to divide all the elements M_{ks} in column s by a common factor, such that the divisions can be performed.

The first two relaxation rules concern omitting of rows in the matrix. This means that the corresponding constraints are ignored. The third relaxation rule concerns a change in the integrality constraints. If column s is divided by a common factor f, then the non-basic variable t_s is substituted by t'_s/f, with t'_s being a new non-basic variable. Constraining t'_s to integer values then corresponds to constraining t_s to integer multiples of $1/f$, instead of integer multiples of 1.

8.3 Constraint Calculations Revisited

Based on the parametric primal all-integer algorithm of the previous section, the constraint calculations can be done in a similar way as in Chapter 6, using the same initial ILP matrices. In the following sections, we revisit each of the constraint calculations, showing how the parametric ILP results are interpreted and where relaxations can be applied. The problem formulations we use for this are the same as in Chapter 6, except that we now consider parametric versions of them.

8.3.1 Processing Unit Constraints

Again, we make a distinction between two cases, i.e., the case with two different operations and the case with one operation.

Two Different Operations

In the case of two different operations, we solve an instance of the parametric version of the processing unit conflict problem, by solving an instance of the relaxed parametric ILP problem, cf. (6.3), given by

$$\begin{array}{rll} z(\boldsymbol{\pi}) \geq \text{maximum} & x_0 & \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}_{\text{puc}}(\boldsymbol{\pi})\, \boldsymbol{t} & \\ & x_i \geq 0 & (i \geq 1) \\ & \boldsymbol{x} \text{ integer}. & \end{array}$$

The initial matrix $\boldsymbol{M}_{\text{puc}}(\boldsymbol{\pi})$ is the same as in (6.4), except that we omit the last row of it, i.e., the reference row. This is done since it is redundant, and it is not used in the parametric algorithm.

If the parametric primal all-integer algorithm finds a solution z, and this solution satisfies $z <_{\text{all}} 0$, then there is no processing unit conflict. Otherwise, i.e., if the algorithm aborts or if it results in a solution z for which $z <_{\text{all}} 0$ does not hold, then we assume there is a conflict.

One Operation

In the case of one operation, we solve an instance of the parametric version of SOPUC by solving an instance of the parametric ILP problem, without relaxations, cf. (6.10), given by

$$\begin{aligned} z(\boldsymbol{\pi}) = \text{maximize} \quad & x_0 \\ \text{subject to} \quad & \boldsymbol{x} = \boldsymbol{M}_{\text{sopuc}}(\boldsymbol{\pi})\, \boldsymbol{t} \\ & x_i \geq 0 \qquad (i \geq 1) \\ & \boldsymbol{x} \text{ integer}. \end{aligned}$$

The initial matrix $\boldsymbol{M}_{\text{sopuc}}(\boldsymbol{\pi})$ is the same as in (6.11), except that we again omit the last row. If the parametric primal all-integer algorithm finds a solution z, and this solution satisfies $\Leftrightarrow z >_{\text{all}} 2o \Leftrightarrow 2$, where o is the occupation time, then there is no conflict. Otherwise, we assume there is a conflict.

The next theorem gives a sufficient condition for an instance of the parametric version of SOPUC to be a no-instance.

Theorem 8.5. *An instance $(\boldsymbol{I}, \boldsymbol{p}, o)$ of the parametric version of SOPUC is a no-instance if*

$$\text{plex}(\boldsymbol{I}, \boldsymbol{p}, o)$$

holds.

Proof. The proof is analogous to the proof of Theorem 6.2. □

8.3.2 Access Constraints

We restrict ourselves to the case of two ports that access one array; for two different arrays the problem is equivalent to the processing unit conflict problem with two different operations, as mentioned in Section 6.2.1.

The parametric version of the access conflict problem is solved as follows. In a similar way as in (6.15) and (6.16), we determine vectors $\underline{\boldsymbol{n}} \in \mathcal{E}^{\alpha}$ and $\overline{\boldsymbol{n}} \in \mathcal{E}^{\alpha}$, for which

$$\begin{aligned} \underline{\boldsymbol{n}}(\boldsymbol{\pi}) \leq \ \text{minimum}_{\text{lex}} \quad & \boldsymbol{A}(\boldsymbol{\pi})\, \boldsymbol{i} \Leftrightarrow \boldsymbol{b}(\boldsymbol{\pi}) \\ \text{subject to} \quad & \boldsymbol{p}^{\text{T}}(\boldsymbol{\pi})\, \boldsymbol{i} = s(\boldsymbol{\pi}) \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(\boldsymbol{\pi}) \\ & \boldsymbol{i} \text{ integer}, \end{aligned}$$

and

$$\begin{aligned} \overline{\boldsymbol{n}}(\boldsymbol{\pi}) \geq \text{maximum}_{\text{lex}} \quad & \boldsymbol{A}(\boldsymbol{\pi})\, \boldsymbol{i} \Leftrightarrow \boldsymbol{b}(\boldsymbol{\pi}) \\ \text{subject to} \quad & \boldsymbol{p}^{\text{T}}(\boldsymbol{\pi})\, \boldsymbol{i} = s(\boldsymbol{\pi}) \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(\boldsymbol{\pi}) \\ & \boldsymbol{i} \text{ integer}, \end{aligned} \tag{8.7}$$

for all parameter settings $\boldsymbol{\pi} \in \Pi$. Again, if these parametric ILP problems are infeasible, or if $\underline{\boldsymbol{n}} =_{\text{all}} \boldsymbol{0}$ and $\overline{\boldsymbol{n}} =_{\text{all}} \boldsymbol{0}$, then there is no access conflict; otherwise we assume

there is. The vector $\overline{\boldsymbol{n}}$ is determined as follows; determining $\underline{\boldsymbol{n}}$ is done analogously.

First, we determine a vector $\tilde{\boldsymbol{n}} \in \mathcal{E}^{\alpha+1}$, such that

$$\begin{array}{rl} \tilde{\boldsymbol{n}}(\boldsymbol{\pi}) = \text{maximum}_{\text{lex}} & \begin{bmatrix} \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} \Leftrightarrow s(\boldsymbol{\pi}) \\ \boldsymbol{A}(\boldsymbol{\pi})\,\boldsymbol{i} \Leftrightarrow \boldsymbol{b}(\boldsymbol{\pi}) \end{bmatrix} \\ \text{subject to} & \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} \leq s(\boldsymbol{\pi}) \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(\boldsymbol{\pi}) \\ & \boldsymbol{i} \text{ integer}, \end{array} \tag{8.8}$$

for all parameter settings $\boldsymbol{\pi} \in \Pi$, cf. (6.17). If $\tilde{n}_0 <_{\text{all}} 0$, then (8.7) is infeasible. Otherwise, we take

$$\overline{\boldsymbol{n}} = \begin{bmatrix} \tilde{n}_1 \\ \tilde{n}_2 \\ \vdots \\ \tilde{n}_\alpha \end{bmatrix}.$$

To determine $\tilde{\boldsymbol{n}}$, we rewrite (8.8) into an instance of the parametric ILP problem with multiple costs, given by

$$\begin{array}{rll} \tilde{\boldsymbol{n}}(\boldsymbol{\pi}) = \text{maximum}_{\text{lex}} & \boldsymbol{x}_{\langle \alpha+1 \rangle} & \\ \text{subject to} & \boldsymbol{x} = \boldsymbol{M}_{\text{ac}}(\boldsymbol{\pi})\,\boldsymbol{t} & \\ & x_i \geq 0 & (i \geq \alpha+1) \\ & \boldsymbol{x} \text{ integer}. & \end{array}$$

The initial matrix $\boldsymbol{M}_{\text{ac}}(\boldsymbol{\pi})$ is the same as in (6.18), except that we again omit the last row.

Although (8.8) describes an instance of the parametric ILP problem, we can apply some of the relaxations mentioned in Section 8.2.4. They are applicable as long as we do not omit the constraint

$$\boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} \leq s(\boldsymbol{\pi}),$$

since this constraint is used together with the first cost function, $\tilde{n}_0$, in order to realize the equality $\boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} = s(\boldsymbol{\pi})$. By applying the relaxations, we obtain a relaxed solution as formulated in (8.7).

8.3.3 Precedence Constraints

The last constraint calculation we revisit is that of the precedence constraints. For such a constraint, we want to find an expression $d \in \mathcal{E}$ for which

$$\begin{array}{rl} d(\boldsymbol{\pi}) \geq \text{maximum} & \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} \Leftrightarrow s(\boldsymbol{\pi}) \\ \text{subject to} & \boldsymbol{A}(\boldsymbol{\pi})\,\boldsymbol{i} = \boldsymbol{b}(\boldsymbol{\pi}) \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(\boldsymbol{\pi}) \\ & \boldsymbol{i} \text{ integer}, \end{array} \tag{8.9}$$

for all parameter settings $\boldsymbol{\pi} \in \Pi$. If this problem is infeasible, or if it has a solution d that satisfies $d <_{\text{all}} 0$, then the precedence constraint is met. Otherwise, the precedence is met if s is increased by $d+1$, i.e., if the difference in start time between the corresponding output and input port is increased by $d+1$. If equality occurs in (8.9), this is the minimum amount by which the difference in start time must be increased in order to meet the constraint. If inequality occurs in (8.9), it is a safe value.

In order to determine d, we solve an instance of the parametric ILP problem, cf. (6.20), given by

$$\begin{aligned} \boldsymbol{z}(\boldsymbol{\pi}) = \text{maximum}_{\text{lex}} \quad & \begin{bmatrix} \boldsymbol{A}(\boldsymbol{\pi})\,\boldsymbol{i} \leqslant \boldsymbol{b}(\boldsymbol{\pi}) \\ \boldsymbol{p}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} \leqslant s(\boldsymbol{\pi}) \end{bmatrix} \\ \text{subject to} \quad & \boldsymbol{A}(\boldsymbol{\pi})\,\boldsymbol{i} \leq \boldsymbol{b}(\boldsymbol{\pi}) \\ & \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{I}(\boldsymbol{\pi}) \\ & \boldsymbol{i} \text{ integer}. \end{aligned} \tag{8.10}$$

If there is an l, $l = 0,\ldots,\alpha \leqslant 1$, for which $z_l <_{\text{all}} 0$, then (8.9) is infeasible. Otherwise, a solution of (8.9) is given by $d = z_\alpha$. Next, the parametric ILP problem (8.10) is rewritten into the form

$$\begin{aligned} \text{maximize}_{\text{lex}} \quad & \boldsymbol{x}_{\langle \alpha+1 \rangle} \\ \text{subject to} \quad & \boldsymbol{x} = \boldsymbol{M}_{\text{pc}}(\boldsymbol{\pi})\,\boldsymbol{t} \\ & x_i \geq 0 \qquad (i \geq \alpha+1) \\ & \boldsymbol{x} \text{ integer}, \end{aligned}$$

where the initial matrix $\boldsymbol{M}_{\text{pc}}(\boldsymbol{\pi})$ is the same as in (6.21), except that again the last row is omitted.

Again, (8.10) describes an instance of the parametric ILP problem. The relaxations mentioned in Section 8.2.4 are nevertheless applicable, as long as we take care that whenever a constraint

$$\boldsymbol{A}_{l\cdot}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} \leq b_l(\boldsymbol{\pi})$$

is omitted, for an $l = 0,\ldots,\alpha \leqslant 1$, then also cost function l must be omitted, i.e., the cost function

$$x_l = \boldsymbol{A}_{l\cdot}^{\mathrm{T}}(\boldsymbol{\pi})\,\boldsymbol{i} \leqslant b_l(\boldsymbol{\pi}).$$

8.4 Parametric Start Time and Processing Unit Assignment

In this section, we present an algorithm for the parametric multidimensional periodic scheduling problem with given periods. Although the periods are given, they may still depend on the parameters. We assume however that each period is either positive for all parameter settings, or negative for all settings. This is a valid as-

sumption in video signal processing, since repetitions are done either forward or backwards in time.

In the following sections, we revisit the three major steps of the start time and processing unit assignment algorithm of Section 7.3, i.e., the analysis of the precedences, the ASAP and ALAP start time assignment, and the iterative approach.

8.4.1 Parametric Precedences

The first step of the algorithm is to determine for each pair $(u,v) \in V \times V$ a safe lower bound $m(u,v) \in \mathcal{E}$ on the difference between the start time of operation u and the start time of operation v, such that the precedence constraints are met. To this end, we first determine a safe lower bound $m(p,q) \in \mathcal{E}$ on the difference between the start time of an output port $p \in O$ and the start time of an input port $q \in I$, with $(p,q) \in E$. Such a lower bound $m(p,q)$ must satisfy

$$\begin{aligned} m(p,q)(\boldsymbol{\pi}) \geq \max\{\ & \boldsymbol{p}^{\mathrm{T}}(p)(\boldsymbol{\pi})\,\boldsymbol{i} \Leftrightarrow \boldsymbol{p}^{\mathrm{T}}(q)(\boldsymbol{\pi})\,\boldsymbol{j} + 1 \mid \\ & \boldsymbol{i} \in \mathcal{I}(p)(\boldsymbol{\pi}) \wedge \boldsymbol{j} \in \mathcal{I}(q)(\boldsymbol{\pi}) \wedge \boldsymbol{n}(p,\boldsymbol{i})(\boldsymbol{\pi}) = \boldsymbol{n}(q,\boldsymbol{j})(\boldsymbol{\pi})\}, \end{aligned}$$

for all parameter settings $\boldsymbol{\pi} \in \Pi$, and it can be determined as discussed in Section 8.3.3. Based on this, a safe lower bound on the difference in start time between an operation u and an operation v is given by an $m(u,v) \in \mathcal{E}$ for which

$$\begin{aligned} m(u,v)(\boldsymbol{\pi}) \geq \max\{\ & m((u,\tilde{o}),(v,\tilde{\imath}))(\boldsymbol{\pi}) + r(t(u),\tilde{o}) \Leftrightarrow r(t(v),\tilde{\imath}) \mid \\ & \tilde{o} \in \tilde{O}(t(u)) \wedge \tilde{\imath} \in \tilde{I}(t(v)) \wedge ((u,\tilde{o}),(v,\tilde{\imath})) \in E\ \}, \end{aligned} \tag{8.11}$$

for all $\boldsymbol{\pi} \in \Pi$. With these expressions $m(u,v)$, $u,v \in V \times V$, we can again determine a precedence graph $H = (V,A,m)$ as in Section 7.3.1.

In order to determine $m(u,v)$ in (8.11), we need a method to determine the maximum of two parametric expressions, i.e., to determine for two expressions $a,b \in \mathcal{E}$ an expression $c \in \mathcal{E}$ for which

$$c(\boldsymbol{\pi}) \geq \max\{a(\boldsymbol{\pi}), b(\boldsymbol{\pi})\},$$

for all $\boldsymbol{\pi} \in \Pi$. Such a c is called a *parametric maximum*. If $a \geq_{\text{all}} b$ or $b \geq_{\text{all}} a$, then trivially a parametric maximum is given by $c = a$ or $c = b$, respectively. Otherwise, we first rewrite $\max\{a(\boldsymbol{\pi}), b(\boldsymbol{\pi})\} = a(\boldsymbol{\pi}) + \max\{0, b(\boldsymbol{\pi}) \Leftrightarrow a(\boldsymbol{\pi})\}$, and next apply the following theorem.

Theorem 8.6. *Given are a parameter set $(\boldsymbol{\pi}, \underline{\boldsymbol{\pi}}, \overline{\boldsymbol{\pi}})$ with $\underline{\boldsymbol{\pi}} \geq \mathbf{0}$, and a parametric expression $\varepsilon \in \mathcal{E}$ that can be written as*

$$\varepsilon(\boldsymbol{\pi}) = \sum_{t \in T} c_t \prod_{p \in P_t} \pi_p,$$

where T is a set of terms, with for each term $t \in T$ an integer coefficient $c_t \neq 0$ and

a multiset P_t of parameter indices. Then an expression $\varphi \in \mathcal{E}$, given by

$$\varphi(\boldsymbol{\pi}) = \sum_{t \in T} c_t^+ \prod_{p \in P_t} \pi_p,$$

satisfies

$$\varphi(\boldsymbol{\pi}) \geq \max\{0, \varepsilon(\boldsymbol{\pi})\},$$

for all $\boldsymbol{\pi} \in \Pi$.

Proof. Straightforward. □

In the following, we denote the parametric maximum of two expressions a and b, determined by means of the above rules, by $\operatorname{pmax}\{a, b\}$. In a similar way we can define a parametric minimum $\operatorname{pmin}\{a, b\}$.

8.4.2 Parametric ASAP and ALAP Start Time Assignment

The second step in the algorithm is to determine for each operation a parametric as soon as possible start time assignment and a parametric as late as possible start time assignment. These are determined in a similar way as the ASAP and ALAP start time assignment in Section 7.3.2. The resulting algorithm for the parametric as soon as possible start time assignment is given in Figure 8.4. The parametric as

1. Initialize $s_{\text{asap}}(v) = -\infty$, for all operations $v \in V$.
2. If $s_{\text{asap}}(v) \geq_{\text{all}} \underline{s}(v)$, for all $v \in V$, then stop.
3. Take an operation $v \in V$ for which $s_{\text{asap}}(v) \geq_{\text{all}} \underline{s}(v)$ does not hold, and set $s_{\text{asap}}(v) = \operatorname{pmax}\{s_{\text{asap}}(v), \underline{s}(v)\}$ and $U = \{v\}$. U can be seen as a set of updated operations.
4. If $U = \emptyset$, return to Step 2. Otherwise, take an operation $u \in U$.
5. For all $u' \in V$ with $(u, u') \in A$ for which $s_{\text{asap}}(u') - s_{\text{asap}}(u) \geq_{\text{all}} m(u, u')$ does not hold, set $s_{\text{asap}}(u') = \operatorname{pmax}\{s_{\text{asap}}(u'), s_{\text{asap}}(u) + m(u, u')\}$ and add u' to U.
6. Remove u from U and return to Step 4.

Figure 8.4. The parametric as soon as possible start time assignment algorithm.

late as possible start time assignment can be determined by a similar algorithm.

If after determining the parametric ASAP and ALAP start time assignment an operation $v \in V$ exists for which $s_{\text{asap}}(v) \leq_{\text{all}} s_{\text{alap}}(v)$ does not hold, then we stop, without having found a feasible start time assignment. Note, however, that this does not necessarily mean that no feasible parametric start time assignment exists.

8.4.3 An Iterative Approach

The third and final step of the algorithm is to determine a parametric start time assignment and a processing unit assignment, by means of an iterative approach. Again, a partial solution is iteratively augmented, by selecting in each iteration an

unscheduled operation and assigning it a start time and a processing unit, or reducing its start time domain. The partial solutions in the case of parameters are defined similarly to the partial solutions in Section 7.3.3.

A basic algorithm for the parametric multidimensional periodic scheduling problem with given periods can now be written as in Figure 8.5.

---

1. Initialize the set of scheduled operations $F = \emptyset$, and initialize $s_{\mathrm{lo}}(v) = s_{\mathrm{asap}}(v)$ and $s_{\mathrm{hi}}(v) = s_{\mathrm{alap}}(v)$, for all operations $v \in V$.

2. If $F = V$, then a feasible schedule is found, so stop. Otherwise, take an operation $v \in V \setminus F$.

3. Try to schedule operation v at start time $s_{\mathrm{lo}}(v)$, i.e., try to assign it start time $s(v) = s_{\mathrm{lo}}(v)$ and to assign it to a processing unit $h(v) \in W$, with $t(h(v)) = t(v)$, such that v has no processing unit conflict with operations $u \in F$ that are already assigned to the same processing unit $h(u) = h(v)$.

4. If v can be scheduled at start time $s_{\mathrm{lo}}(v)$, then assign $s(v) = s_{\mathrm{lo}}(v)$ and $s_{\mathrm{hi}}(v) = s_{\mathrm{lo}}(v)$, fix $h(v)$, and add v to F. Otherwise, increase $s_{\mathrm{lo}}(v)$.

5. If necessary, adjust the start time domains $\{s_{\mathrm{lo}}(u),\ldots,s_{\mathrm{hi}}(u)\}$ of the unscheduled operations $u \in V \setminus F$, such that s_{lo} and s_{hi} are again feasible start time assignments.

6. If now an operation $u \in V \setminus F$ has possibly an empty time domain, i.e., $s_{\mathrm{lo}}(u) \leq_{\mathrm{all}} s_{\mathrm{hi}}(u)$ does not hold, then stop; no schedule is found. Otherwise, return to Step 2.

Figure 8.5. The parametric start time and processing unit assignment algorithm.

One step in the algorithm remains to be elaborated, namely the increasing of $s_{\mathrm{lo}}(v)$ in Step 4. This is done in a similar way as in Section 7.3.3, based on the resulting processing unit conflicts when v is started at time $s_{\mathrm{lo}}(v)$.

Let again F^* be the set of operations $u \in F$ that give a processing unit conflict if v is assigned to the same processing unit and v starts at time $s_{\mathrm{lo}}(v)$. For each of the operations $u \in F^*$, we determine an amount $d(u) \in \mathcal{P}$ to increase $s_{\mathrm{lo}}(v)$ with, in order to resolve the conflict. Since a parametric minimum of these expressions $d(u)$ may be an expression that is negative for some parameter settings, we increase $s_{\mathrm{lo}}(v)$ by the expression $d(u)$ that has the smallest lower bound.

The amount $d(u)$, for each $u \in F^*$, is again determined by means of a solution of PUC. So, let the instance of the parametric version of PUC that corresponds to the processing unit conflict problem between operation u and v be given by $(\boldsymbol{I}, \boldsymbol{p}, s)$, with $\boldsymbol{I} \in \mathcal{P}^\delta$, $\boldsymbol{p} \in \mathcal{P}^\delta$, and $s \in \mathcal{P}$. Furthermore, let $\boldsymbol{i} \in \mathcal{E}^\delta$ be a solution of it, i.e., $\mathbf{0} \leq_{\mathrm{all}} \boldsymbol{i} \leq_{\mathrm{all}} \boldsymbol{I}$ and $\boldsymbol{p}^{\mathrm{T}}\boldsymbol{i} = s$. Now the algorithm in Figure 8.6 determines an interval $\{l,\ldots,r\}$, $l, r \in \mathcal{E}$, around s, that also results in a conflict, cf. the algorithm in Figure 7.6.

Now, the conflict may be resolved by increasing s by $r \Leftrightarrow s+1$, or by decreasing

1. Set $l = s$ and $r = s$, and set $D = \{0,\ldots,\delta-1\}$.
2. Select a $k \in D$ for which $p_k >_{\text{all}} r - l + 1$ does not hold. If no such k exists, then stop.
3. Subtract $p_k i_k$ from l and add $p_k(I_k - i_k)$ to r. Next, remove k from D, and return to Step 2.

Figure 8.6. The parametric processing unit conflict interval algorithm.

s by $s \Leftrightarrow l+1$. So, $d(u)$ is given by $r \Leftrightarrow s+1$ or by $s \Leftrightarrow l+1$, depending on whether the sign of v's start time in s is positive or negative in the PUC instance.

The above algorithm is the same as the processing unit conflict interval algorithm of Section 7.3.3, except that integers are replaced by parametric expressions. This, however, may result in a shift of the start time that is not necessarily the minimum shift, as we can see in the following example. Consider an instance of the parametric version of PUC, given by

$$\boldsymbol{I} = \begin{bmatrix} 2 \end{bmatrix}, \quad \boldsymbol{p} = \begin{bmatrix} \pi_0 \end{bmatrix}, \quad s = 0,$$

where the parameter $\pi_0 \in \{1,2\}$. A solution of this instance is given by $\boldsymbol{i} = [0]$. Applying the above algorithm then results in a parametric processing unit conflict interval

$$\{l,\ldots,r\} = \{0,\ldots,2\pi_0\}.$$

So, in order to resolve the conflict, s is increased by $2\pi_0 + 1$, which is at most equal to 5. However, increasing s by 3 already resolves the conflict, as we can see from the fact that

$$\{p_0 i_0 \mid 0 \le i_0 \le 2\} = \begin{cases} \{0,1,2\} & \text{if } \pi_0 = 1 \\ \{0,2,4\} & \text{if } \pi_0 = 2. \end{cases}$$

So, the amount to increase the start time with, which is $2\pi_0 + 1 \ge_{\text{all}} 3$ in the example, may be more than is strictly necessary. Nevertheless, the algorithm gives satisfactory results in practice, where the ranges of the parameters are generally larger.

8.5 Numerical Results

Example 8.2. Consider the transposition example of Figure 8.1. The constraints on the start times of the operations are given by $s(in) \ge_{\text{all}} 1$ and $s(out) \le_{\text{all}} \pi_0\pi_1$. The resulting schedule is given by $s(in) = 1$ and $s(out) = \pi_0\pi_1 \Leftrightarrow \pi_0 \Leftrightarrow \pi_1 + 3$, which is depicted in Figure 8.2. The run time for this instance is about 0.13 second, of which 0.09 second is for the calculation of the precedences and the ASAP and ALAP schedules, and 0.04 second for the iterative algorithm. □

Example 8.3. Consider the parametric matrix summation algorithm of Figure 8.7,

cf. Example 7.2. The algorithm is repeated every $\pi_0\pi_1$ time units, and the con-

```
parameter π0 in {3,...,10}
parameter π1 in {3,...,10}
for i1 = 0 to π0 − 1 period π1
    for i2 = 0 to π1 − 1 period 1
        x[i1][i2] = in( )
for j1 = 0 to π0 − 1 period π1
    y[j1][0] = x[j1][0]
for k1 = 0 to π0 − 1 period π1
    for k2 = 0 to π1 − 2 period 1
        y[k1][k2 + 1] = y[k1][k2] + x[k1][k2 + 1]
z[0] = y[0][π1 − 1]
for l1 = 0 to π0 − 2 period π1
    z[l1 + 1] = z[l1] + y[l1 + 1]
 = out(z[π0 − 1])
```

Figure 8.7. The video algorithm of Example 8.3, in which a $\pi_0 \times \pi_1$ matrix is taken as an input and where the elements are summed. The video algorithm is repeated every $\pi_0\pi_1$ time units; the infinite repetition is not indicated.

straints on the start times are given by $0 \leq_{\text{all}} s(v) \leq_{\text{all}} 2\pi_0\pi_1$, for all operations v.

For this instance, the run time is about 0.82 second, of which 0.72 second is for the calculation of the precedences and the ASAP and ALAP schedules, and 0.10 second for the iterative algorithm. The resulting schedule is given in Figure 8.8. All additions have been assigned to the same processing unit.

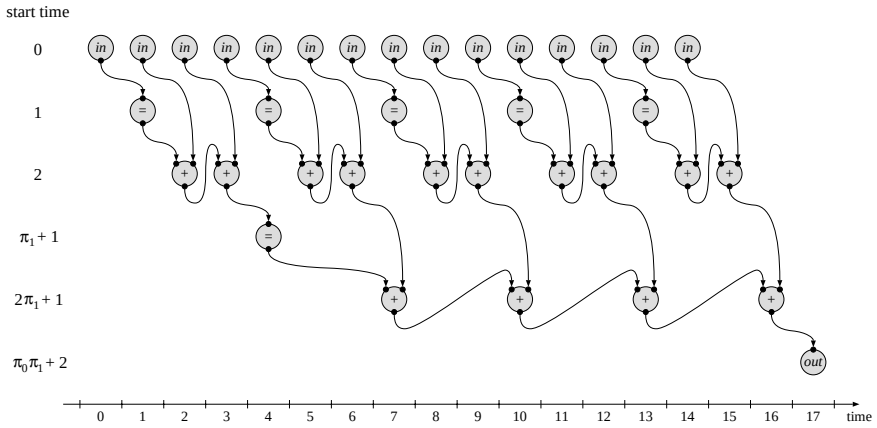


Figure 8.8. The schedule obtained for Example 8.3, here drawn for $\pi_0 = 5$ and $\pi_1 = 3$. Only the first global period is shown. At the left hand side the start times of the operations are given.

□

Example 8.4. Consider the video algorithm of Figure 8.9. In each period of 15

parameter π_0 in $\{4,\ldots,9\}$
for i_1 = 0 to $\pi_0 - 1$ period 1
    $x[i_1] = in(\,)$
$a[0] = x[0]$
for j_1 = 0 to $\pi_0 - 2$ period 1
    $a[j_1+1] = a[j_1] + x[j_1+1]$
$= out(a[\pi_0 - 1])$
for k_1 = 0 to $14 - \pi_0$ period 1
    $y[k_1] = in(\,)$
$b[0] = y[0]$
for l_1 = 0 to $13 - \pi_0$ period 1
    $b[l_1+1] = b[l_1] + y[l_1+1]$
$= out(b[14 - \pi_0])$

Figure 8.9. The video algorithm of Example 8.4, in which two rows are taken in, of length π_0 and $15 - \pi_0$, respectively. Next, the sum of both rows is determined. The video algorithm is repeated every 15 time units; the infinite repetition is not indicated.

time units, a row of π_0 elements and a row of $15 \Leftrightarrow \pi_0$ elements are taken in, with $\pi_0 \in \{4,\ldots,9\}$. Next, the sums of both rows are determined, and these two results are sent out. In this instance, we have taken only one processing unit for the input operations, one processing unit for the additions, and one processing unit for the output operations, which means that the processing unit constraints may be restrictive. The constraints on the start times are given by $0 \leq_{\text{all}} s(v) \leq_{\text{all}} 20$, for all operations v.

For this instance, the run time is about 0.56 second, of which 0.39 second is for the calculation of the precedences and the ASAP and ALAP schedules, and 0.17 second for the iterative algorithm. The resulting schedule is given in Figure 8.10. All input operations have been assigned to the same processing unit. The same thing holds for the additions, and for the two output operations.

□

8.6 Discussion

In this chapter we extended the multidimensional periodic scheduling problem to include parameters, and we adapted the original solution approach to handle the extended problem. To this end, we developed a parametric primal all-integer ILP algorithm, and applied it to the constraint calculations in the case of parameters. Finally, we presented an algorithm to determine a start time and processing unit assignment for parametric signal flow graphs. The period assignment problem in the

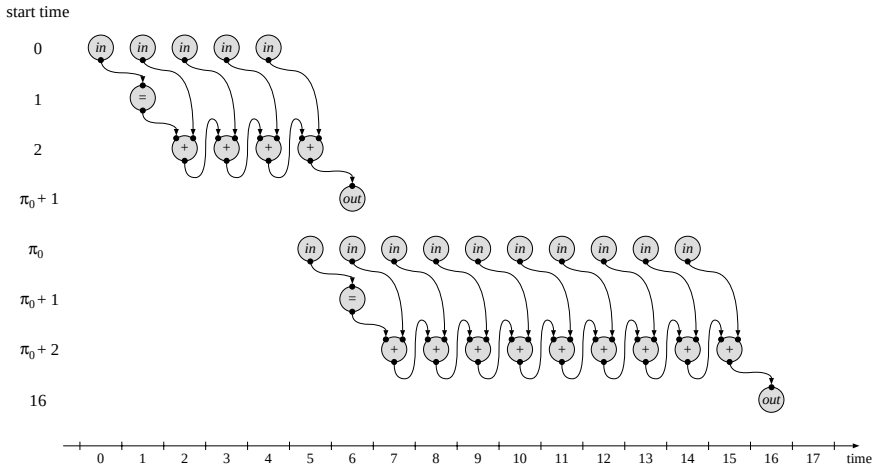


Figure 8.10. The schedule obtained for Example 8.4, here drawn for $\pi_0 = 5$. Only the first global period is shown. At the left hand side the start times of the operations are given. In this example, the second *in* operation has been postponed, in order to be assigned to the same processing unit as the first *in* operation.

case of parameters has not been addressed.

The extended solution approach is based on relatively simple methods for bounding and division of parametric expressions. In practice, these methods appear to give satisfactory results. Also the methods for the constraint calculations in the case of parameters give satisfactory results. The algorithm to find a start time and processing unit assignment can be used in a similar interactive way as the algorithm for non-parametric video algorithms, although the interaction is slightly more important. In practice, the scheduling algorithm is highly applicable to find good solutions in a reasonable amount of time.

9

Conclusion

In this thesis, we have considered the multidimensional periodic scheduling problem, inspired by the problem of designing cost-effective hardware systems that implement video signal processing algorithms. In this scheduling problem, operations are repeatedly executed, with several dimensions of repetition. The question is to find a time assignment, consisting of periods and start times, and a processing unit assignment, that obey the timing constraints, processing unit constraints, and precedence constraints, and that minimize a weighted sum of the processing unit cost, the storage cost, and the access cost.

The multidimensional periodic scheduling problem has been shown to be NP-hard in the strong sense. This is caused amongst others by the NP-completeness of the feasibility checks, which is due to the fact that the number of executions in one global period is generally not bounded by a polynomial in the size of an instance. Next, the two major sub-problems of multidimensional periodic scheduling, which are concerned with the checking of the processing unit constraints and the precedence constraints, have been shown to be NP-complete and NP-complete in the strong sense, respectively. We have identified various special cases of these sub-problems, induced by practical situations. Several of these special cases have been shown to be solvable in polynomial time.

Next, we have discussed a primal all-integer ILP algorithm to solve the sub-problems. The main advantage of an all-integer algorithm is the absence of round-

ing errors, which lead to unreliable results. The chosen ILP algorithm is applied in such a way that it benefits from the special properties of the well-solvable special cases, resulting in a provably polynomial number of steps for these cases. This results in a generally applicable approach, which runs in polynomial time for the well-solvable special cases. The run time of the all-integer algorithm for the subproblems is small in practice, which is a prerequisite when it is to be invoked a large number of times in one run of a scheduling algorithm.

The applied cost function is a weighted sum of the processing unit cost, the storage cost, and the access cost. The processing unit cost and the access cost can be determined by coloring a conflict graph, which may be done either optimally or approximately. The latter may be preferred when it is invoked a large number of times in one run of a scheduling algorithm. The storage cost has been approximated by a function that can be expressed as a linear function of the periods and start times of the operations. To this end, a signal flow graph is extended by stop operations, which denote the moments in time when variables stop living.

Next, we have presented a decomposition strategy for the multidimensional periodic scheduling problem. In a first stage, periods are assigned to the operations such that the approximate storage cost is minimized. In a second stage, the operations are assigned start times and processing units. The period assignment problem of the first stage has been formulated as an ILP problem with some additional constraints. A branch-and-bound approach is presented to find solutions that satisfy the non-linear constraints, and a constraint-generation technique is presented to solve the relaxations. The problem of the second stage concerns a start time and processing unit assignment, and it is handled by an iterative approach. After determination of the precedence constraints and the initial slack of the operations, the slack of the operations is iteratively reduced, until they are scheduled at a fixed time and are assigned to processing units. The iterative approach is a fast heuristic, which is highly applicable when used interactively.

The final subject we have considered is the extension of the problem to include parameters, allowing the design of more flexible hardware systems. In this case, the numbers of repetitions, period bounds, start time bounds, and array index matrices and offset vectors, are no longer constants, but are polynomials in parameters, with a specified range for each parameter. The problem then is to find a parametric time assignment and a processing unit assignment, such that the originally formulated constraints are met for all parameter values in a given domain. Restricting the problem to instances with given periods, we have presented an approach to determine a parametric start time and a processing unit assignment, based on the original approach. To this end, we have extended the methods discussed for the case without parameters to incorporate parameters. The resulting heuristic is found to be highly applicable in practice, when used in an interactive way.

Bibliography

BAKER, K.R. [1974], *Introduction to Sequencing and Scheduling*, John Wiley & Sons, New York.

BEALE, E.M.L. [1979], Branch and bound methods for mathematical programming systems, *Annals of Discrete Mathematics* **5**, 201–219.

BORRIELLO, G., AND E. DETJENS [1988], High-level synthesis: Current status and future directions, *Proceedings of the 25th Design Automation Conference*, 477–482.

BU, J., E. DEPRETTERE, AND L. THIELE [1990], Systolic array implementation of nested loop programs, *Proceedings International Conference on Application Specific Array Processing*, 31–42.

COFFMAN, JR., E.G. (ed.) [1976], *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York.

COFFMAN, JR., E.G., M.R. GAREY, AND D.S. JOHNSON [1987], Bin packing with divisible item sizes, *Journal of Complexity* **3**, 406–428.

CONWAY, R.W., W.L. MAXWELL, AND L.W. MILLER [1967], *Theory of Scheduling*, Addison-Wesley, Reading.

FEAUTIER, P. [1991], Dataflow analysis of array and scalar references, *International Journal of Parallel Programming* **20**, 23–53.

FRENCH, S. [1982], *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Ellis Horwood, Chichester.

GAREY, M.R., AND D.S. JOHNSON [1979], *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York.

GARFINKEL, R.S. [1979], Branch and bound methods for integer programming, in: N. Christofides, A. Mingozzi, P. Toth, and M. Sandi (eds.), *Combinatorial Optimization*, John Wiley & Sons, New York, 1–20.

GEBOTYS, C.H., AND M.I. ELMASRY [1990], A global optimization approach for architectural synthesis, *Proceedings of the International Conference on Computer Aided Design*, 258–261.

GIBBONS, A. [1985], *Algorithmic Graph Theory*, Cambridge University Press, Cambridge.

GLOVER, F. [1968], A new foundation for a simplified primal integer programming

algorithm, *Operations Research* **16**, 727–740.

GOMORY, R.E. [1958], Outline of an algorithm for integer solutions to linear programs, *Bulletin of the American Mathematical Society* **64**, 275–278.

GOMORY, R.E. [1963], An all-integer integer programming algorithm, in: J.F. Muth and G.I. Thompson (eds.), *Industrial Scheduling*, Prentice-Hall, Englewood Cliffs.

GOOSSENS, G., J. RABAEY, J. VANDEWALLE, AND H. DE MAN [1987], An efficient microcode-compiler for custom DSP-processors, *Proceedings of the International Conference on Computer-Aided Design*, 24–27.

HARARY, F. [1969], *Graph Theory*, Addison-Wesley, Reading.

HAUPT, R. [1989], A survey of priority-rule based scheduling, *OR Spektrum* **11**, 3–16.

HWANG, C.T., J.H. LEE, AND Y.C. HSU [1991], A formal approach to the scheduling problem in high level synthesis, *IEEE Transactions on Computer-Aided Design* **10**, 464–475.

KARMARKAR, N. [1984], A new polynomial-time algorithm for linear programming, *Combinatorica* **4**, 373–395.

KHACHIYAN, L.G. [1979], A polynomial algorithm in linear programming, *Doklady Akademiia Nauk SSSR* **244**, 1093–1096. Translated into English in *Soviet Mathematics Doklady* **20**, 191–194.

KORST, J.H.M. [1992], *Periodic Multiprocessor Scheduling*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, the Netherlands.

LAWLER, E.L. [1976], *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

LAWLER, E.L., AND D.E. WOOD [1966], Branch-and-bound methods: A survey, *Operations Research* **14**, 699–719.

LEE, E.A., AND D.G. MESSERSCHMITT [1987], Static scheduling of synchronous data flow programs for digital signal processing, *IEEE Transactions on Computers* **C-36**, 24–35.

LIPPENS, P.E.R., J.L. VAN MEERBERGEN, W.F.J. VERHAEGH, D.M. GRANT, AND A. VAN DER WERF [1994], Design of a 30 MHz, 32/16/8-points DCT processor with Phideo, in: J. Rabaey, P. Chau, and J. Eldon (eds.), *VLSI Signal Processing VII*, IEEE Press, New York, 24–32.

LIPPENS, P.E.R., J.L. VAN MEERBERGEN, A. VAN DER WERF, W.F.J. VERHAEGH, B.T. MCSWEENEY, J.A. HUISKEN, AND O.P. MCARDLE [1991], PHIDEO: A silicon compiler for high speed algorithms, *Proceedings of the European Conference on Design Automation*, 436–441.

MARTIN, R.S., AND J.P. KNIGHT [1993], Operations research in the high-level synthesis of integrated circuits, *Computers Operations Research* **20**, 845–856.

MCFARLAND, M.C. [1986], Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions, *Proceedings 25th Design Automation Conference*, 474–480.

MCFARLAND, M.C., A.C. PARKER, AND R. CAMPOSANO [1990], The high-level synthesis of digital systems, *Proceedings of the IEEE* **78**, 301–318.

NEMHAUSER, G.L., AND L.A. WOLSEY [1988], *Integer and Combinatorial Optimization*, John Wiley & Sons, New York.

PAPADIMITRIOU, C.H., AND K. STEIGLITZ [1982], *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs.

PARK, N., AND A.C. PARKER [1988], Sehwa: A software package for synthesis of pipelines from behavioral specifications, *IEEE Transactions on Computer-Aided Design* **7**, 356–370.

PAULIN, P.G., AND J.P. KNIGHT [1989], Force-directed scheduling for the behavioral synthesis of ASICs, *IEEE Transactions on Computer-Aided Design* **8**, 661–679.

PINEDO, M. [1995], *Scheduling: Theory, Algorithms, and Systems*, Prentice-Hall, Englewood Cliffs.

POTKONJAK, M., AND J. RABAEY [1994], A scheduling and resource allocation algorithm for hierarchical signal flow graphs, *Proceedings of the Design Automation Conference*, 7–12.

PUGH, W. [1991], The omega test: A fast and practical integer programming algorithm for dependence analysis, *Proceedings Supercomputing*, 18–22.

SCHRIJVER, A. [1986], *Linear and Integer Programming*, John Wiley & Sons, Chichester.

STOK, L. [1991], *Architectural Synthesis and Optimization of Digital Systems*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven, the Netherlands.

STOK, L. [1994], Data path synthesis, *Integration, the VLSI Journal* **18**, 1–71.

SWAAIJ, M.F.X.B. VAN, F.H.M. FRANSSEN, F.V.M. CATTHOOR, AND H.J. DE MAN [1992], Modeling data flow and control flow for high level memory management, *Proceedings of the European Conference on Design Automation*, 8–13.

TIMMER, A.H., AND J.A.G. JESS [1993], Execution interval analysis under resource constraints, *Proceedings of the International Conference on Computer-Aided Design*, 454–459.

VERHAEGH, W.F.J. [1990], Scheduling problems in video signal processing, Master's thesis, Eindhoven University of Technology, Eindhoven, the Netherlands.

VERHAEGH, W.F.J., E.H.L. AARTS, J.H.M. KORST, AND P.E.R. LIPPENS [1991], Improved force-directed scheduling, *Proceedings of the European*

Conference on Design Automation, 430–435.

VERHAEGH, W.F.J., P.E.R. LIPPENS, E.H.L. AARTS, J.H.M. KORST, J.L. VAN MEERBERGEN, AND A. VAN DER WERF [1992], Modelling periodicity by PHIDEO streams, talk presented at the *Sixth International Workshop on High Level Synthesis*.

VERHAEGH, W.F.J., P.E.R. LIPPENS, E.H.L. AARTS, J.H.M. KORST, J.L. VAN MEERBERGEN, AND A. VAN DER WERF [1995], Improved force-directed scheduling in high-throughput digital signal processing, *IEEE Transactions on Computer Aided Design* **14**, 945–960.

VERHAEGH, W.F.J., P.E.R. LIPPENS, E.H.L. AARTS, J.H.M. KORST, A. VAN DER WERF, AND J.L. VAN MEERBERGEN [1992], Efficiency improvements for force-directed scheduling, *Proceedings of the International Conference on Computer-Aided Design*, 286–291.

WERF, A. VAN DER, B.T. MCSWEENEY, J.L. VAN MEERBERGEN, P.E.R. LIPPENS, AND W.F.J. VERHAEGH [1991], Hierarchical retiming including pipelining, *Proceedings VLSI*, 11.2.1–11.2.10.

YOUNG, R.D. [1968], A simplified primal (all-integer) integer programming algorithm, *Operations Research* **16**, 750–782.

Symbol Index

The numbers refer to the pages of first occurrence.

General Symbols

Operations

Signal Flow Graphs

E	set of data edges	13
$\boldsymbol{A}$	index matrix	13
$\boldsymbol{b}$	index offset vector	13
$\boldsymbol{i}$	iterator vector	14
$\mathcal{I}$	iterator space	14
O	set of output ports	13
I	set of input ports	13
$P = I \cup O$	set of all ports	13
$\boldsymbol{n}(p, \boldsymbol{i})$	index vector at execution $\boldsymbol{i}$ of port p	14
V_s	set of stop operations	79
E_s	set of stop edges	79
S	set of stop ports	79

Schedules

$\sigma = (\boldsymbol{p}, s, W, h)$	schedule	16
$\boldsymbol{p}$	period vector	16
s	start time	16
W	set of processing units	16
h	processing unit assignment function	16
$c(v, \boldsymbol{i})$	time of execution $\boldsymbol{i}$ of operation v	16
$c(p, \boldsymbol{i})$	time of execution $\boldsymbol{i}$ of port p	16
s_asap	as soon as possible (ASAP) start time	114
s_alap	as late as possible (ALAP) start time	114

Array Clusters

$\bowtie$	join	18
$\mathcal{A}$	set of array clusters	18
$\mathcal{L}(A, c)$	set of living variables of array cluster A at time c	19
$\mathcal{B}(A, c)$	set of accessed variables of array cluster A at time c	19

Cost Functions

f	total area cost	20
f_max	storage cost: maximum number of variables	73
f_avg	storage cost: average number of variables	75
f_ex	storage cost based on execution times	78
f_stop	storage cost based on stop operations	80
p_glob	global period	74

Lexicographical Properties

Parameters

Author Index

Subject Index

Samenvatting

In dit proefschrift behandelen we het *meerdimensionaal periodiek planningsprobleem*. In dit planningsprobleem worden operaties herhaaldelijk uitgevoerd, met verscheidene dimensies van herhaling. Beschouw bijvoorbeeld een bus die een bepaalde route moet rijden. Het rijden van de route duurt minder dan een uur, en de bus rijdt hem tien keer per dag, met een periode van een uur, vijf keer per week, met een periode van een dag, en elke week, met een periode van een week. Het herhaalde proces van het rijden van de route noemen we een periodieke operatie, en het rijden van een enkele route noemen we een executie van de operatie. De duur van één rit noemen we de bezettingstijd, en de tijd tussen twee opeenvolgende herhalingen in elke dimensie noemen we een periode. In het voorbeeld van de bus hebben we drie dimensies van herhaling, wat resulteert in drie periodes, namelijk een uur, een dag en een week.

Onze interesse in het meerdimensionaal periodiek planningsprobleem vindt zijn oorsprong in de videosignaalbewerking. Om iets preciezer te zijn, het is gebaseerd op het probleem van het ontwerpen van kosteneffectieve geïntegreerde schakelingen die algoritmen voor videosignaalbewerking implementeren. Gegeven zo'n video-algoritme, met meerdimensionaal periodieke operaties, is de vraag nu om een tijdstoekenning te vinden, bestaande uit periodes en starttijden, en een toekenning te vinden van de operaties aan rekeneenheden. Deze toekenningen moeten voldoen aan een aantal beperkingen. Dit zijn de tijdsbeperkingen, de beperking dat rekeneenheden maximaal één operatie tegelijk uitvoeren en de beperking dat tussenresultaten berekend zijn voordat ze gebruikt worden. Verder eisen we van de toekenningen dat de totale oppervlakte van de geïntegreerde schakeling minimaal is. Deze oppervlakte is gegeven als een gewogen som van de benodigde rekeneenheden, de geheugengrootte en het aantal geheugenpoorten.

Van het meerdimensionaal periodiek planningsprobleem hebben we aangetoond dat het NP-lastig is in de sterke zin. De oorzaak hiervan ligt onder andere in de NP-compleetheid van het controleren van de beperkingen, wat op zich weer te wijten is aan het feit dat het aantal executies binnen één globale periode in het algemeen niet begrensd is door een polynoom in de grootte van een instantie. Vervolgens hebben we aangetoond dat de twee belangrijkste deelproblemen van meerdimensionaal periodiek plannen, dat zijn het controleren van de beperkingen van

rekeneenheden en het controleren van de precedentiebeperkingen, NP-compleet, respectievelijk, NP-compleet in de sterke zin zijn. We hebben verscheidene speciale gevallen van deze deelproblemen geïdentificeerd, ingegeven door praktijksituaties, waarvan er sommige zijn op te lossen in polynomiale tijd.

Vervolgens hebben we een geheeltallig lineair programmeringsalgoritme besproken dat gebruik maakt van een tableau met louter gehele getallen. Dit algoritme wordt toegepast om de bovengenoemde deelproblemen op te lossen. Doordat het algoritme gebruik maakt van een geheeltallig tableau, kan voorkomen worden dat er afrondfouten ontstaan, die kunnen leiden tot foutieve resultaten. Het gekozen algoritme is zodanig toegepast dat het profiteert van de speciale eigenschappen van de goed oplosbare speciale gevallen, wat resulteert in een bewijsbaar polynomiaal aantal stappen voor deze gevallen. Dit resulteert in een algemeen toepasbare aanpak, die de goed oplosbare speciale gevallen oplost in een polynomiale tijd. De rekentijd van het algoritme voor de genoemde deelproblemen is klein in de praktijk, wat een vereiste is als het veelvuldig aangeroepen moet worden door een planningsalgoritme.

De kostenfunctie is gegeven als een gewogen som van het aantal rekeneenheden, de geheugengrootte en het aantal geheugenpoorten. De kosten voor rekeneenheden en geheugenpoorten kunnen bepaald worden door het kleuren van een conflictengraaf, hetgeen optimaal of bij benadering kan gebeuren. Het laatste kan de voorkeur hebben indien het een groot aantal keren moet gebeuren in een planningsalgoritme. De benodigde geheugengrootte wordt bij benadering bepaald door een functie die lineair uitgedrukt kan worden in de periodes en starttijden van de operaties. Hiervoor worden eerst zogenaamde stopoperaties geïntroduceerd, die de momenten aanduiden waarop tussenresultaten niet meer nodig zijn.

Vervolgens hebben we een decompositiestrategie gegeven voor het meerdimensionaal periodiek planningsprobleem. In de eerste stap worden periodes toegekend aan de operaties, zodanig dat de benodigde geheugengrootte minimaal is. In de tweede stap worden starttijden toegekend aan de operaties, en worden de operaties aan rekeneenheden toegekend. Het periodetoekenningsprobleem van de eerste stap hebben we gemodelleerd als een geheeltallig lineair programmeringsprobleem met enkele additionele beperkingen. We hebben een aanpak gepresenteerd die gebaseerd is op een ,,branch-and-bound"-techniek voor de niet-lineaire beperkingen, en een ,,constraint-generation"-techniek om de relaxaties op te lossen. Het probleem in de tweede stap betreft het toekennen van starttijden en rekeneenheden, en hebben we aangepakt door middel van een iteratieve methode. Na het bepalen van de precedenties en de initiële vrijheid van alle operaties, wordt de vrijheid van elke operatie iteratief verminderd, tot die uiteindelijk op een tijdstip vastgezet wordt en aan een rekeneenheid wordt toegekend. De iteratieve aanpak is snel, wat een interactief gebruik ervan toestaat. Gebaseerd op grafische terugkoppeling is

een gebruiker in staat om interactie te plegen, om zodoende een goede planning te verkrijgen.

Het laatste onderwerp dat we hebben bestudeerd is de uitbreiding van het probleem met parameters, om zodoende meer flexibiliteit in de resulterende schakelingen te verkrijgen. In dit geval zijn de aantallen herhalingen van de operaties niet langer constant, maar zijn ze gegeven door polynomen in parameters, met een gegeven domein voor elke parameter. Het probleem is nu om een parametrische tijdstoekenning te vinden en een toekenning te vinden van operaties aan rekeneenheden, zodanig dat aan de originele beperkingen wordt voldaan voor alle parameterwaarden in een gegeven domein. We hebben ons beperkt tot het geval waarbij periodes gegeven zijn, en voor dit geval hebben we een aanpak gepresenteerd voor het bepalen van een parametrische starttijdtoekenning en een toekenning van de rekeneenheden, gebaseerd op de aanpak voor het geval zonder parameters. Dit hebben we gedaan door de bestaande methoden uit te breiden zodanig dat die parameters aankunnen. Het resulterende algoritme is goed bruikbaar in de praktijk, gebruik makend van interactie.

Curriculum Vitae

Wim Verhaegh was born on January 14, 1967, in Weert, the Netherlands. From 1985 to 1990 he studied mathematics at the Eindhoven University of Technology. He graduated with honors in April 1990, on the subject of scheduling in the design of video signal processing systems. His Master's thesis was written under the supervision of E.H.L. Aarts and J.H.M. Korst.

Since his graduation, he has been with the Philips Research Laboratories in Eindhoven. As a member of the Phideo project, he has been working on high-level synthesis of DSP systems for video applications, with the emphasis on scheduling problems and techniques. In this work, he combines his interests in combinatorial optimization, complexity theory, and the design and analysis of algorithms, with his interests in digital electronic circuits. Wim is a member of the 'Algoritmen Club', and an executive committee member of the 'Vereniging voor Wiskundig Ingenieurs Eindhoven'.