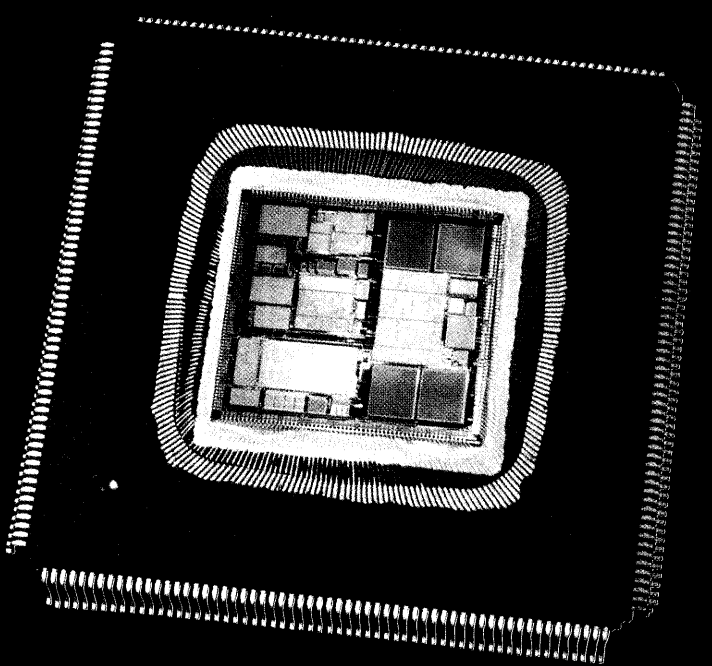
Processing unit design

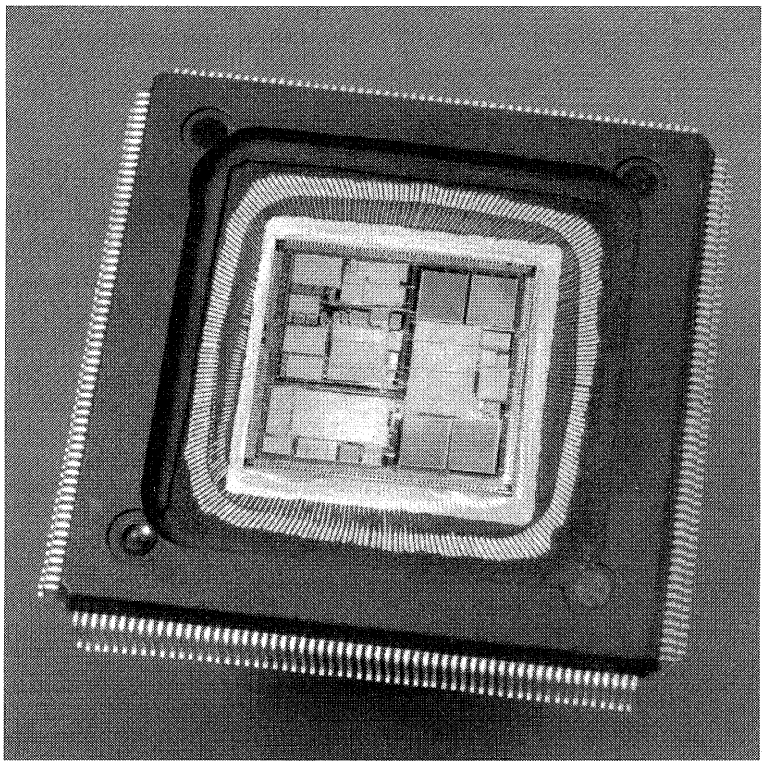
Processing Unit Design

Albert van der Werf

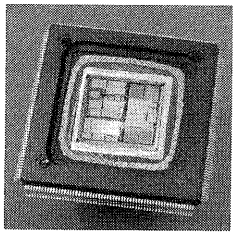


Processing Unit Design

Albert van der Werf



Processing Unit Design



On the cover: I.McIC, a single-chip MPEG2 video encoder. This IC was designed in the period of 1994 to 1996 under the project leadership of the author. To make the circuit run at the required clock frequency, retiming techniques were applied as described in this thesis.

Processing Unit Design

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. M. Rem, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op woensdag 26 augustus 1998 om 16.00 uur

door

Albert van der Werf

geboren te Leeuwarden

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. E.H.L. Aarts
en
prof.dr.ir. Th. Krol

Copromotor: dr.ir. W.F.J. Verhaegh

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

van der Werf, Albert

Processing Unit Design/
Albert van der Werf. -
Eindhoven: Eindhoven University of Technology
Thesis Eindhoven. - With ref. - With summary in Dutch and English
ISBN 90-74445-41-1
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.

aan Marijn

Preface

In the summer of 1989, I became employed by Philips Research with the ambition to design state-of-the-art ICs. I had to wait for such work for five years and first became a member of a team that researched methods for the design of ICs for digital signal processing in video applications. The approach that this team set up resulted in the design methodology Phideo. My task was to develop a method for the design of the basic processing units in video ASICs.

The retiming algorithm, which had just been reported in the literature [Leiserson, Rose & Saxe, 1983], became the starting point of my work. The effect of the algorithm can be viewed as a shift of registers throughout a network of functional operations. The challenge was to make the retiming algorithm available in a practical form to IC designers. This involved writing software as well as improving the retiming algorithm in order to make run times acceptable. The software implementation of the newly developed algorithm, called RetLab (an anagram of Albert), is currently being utilized by a large IC-design community within Philips, even for ICs in other applications than digital video systems. In the early nineties a retiming tool called Optima, which is based on RetLab, was even marketed outside Philips, albeit temporarily [Sondervan, 1993; Sondervan, 1994].

I found another challenge in developing algorithms for which the retiming technique is combined with resource sharing methods. A major part of this thesis is devoted to research on this subject. Unfortunately, it was much more difficult to make these algorithms easily accessible to the IC design community. Nevertheless, many valuable results were obtained, which give insight into algorithm development, especially in the application of simulated annealing. Moreover, its usefulness was clearly demonstrated in the design of a research vehicle [Lippens, Van Meerbergen, Verhaegh, Grant & Van der Werf, 1994].

At the end of 1994, I stopped with this work and devoted myself to the design of I.McIC, an IC for real-time MPEG2 encoding of digital video [Van der Werf, Brüls, Kleihorst, Waterlander, Verstraelen & Friedrich, 1997]. My involvement in design methods, especially the method I had developed myself, continued after this shift. Instead of being a developer, I became a user of my own design method. Although many designers within the Philips IC-design community were already enthusiastic users, it was very satisfactory that I could benefit from my own previous work as

well.

The shift in work did leave me with an unsatisfied feeling however. Although the success of the work was there, all the theory that had been built up during the years could not be easily and consistently presented. Many students had helped me with the research and had written reports in different styles. Furthermore, I had presented papers at conferences and workshops, in each case covering only a small part of the work. The task that was left was to convert a number of papers and to transform material from students into a thesis. This task comprised more than just stapling a number of papers together, which is reflected in the time it took me to write this thesis.

That I actually have written this thesis, I owe to Emile Aarts, who continued to encourage me to pursue a doctorate. I would like to thank him for that and also for the many philosophical discussions that we had and which I enjoyed.

I also want to express my gratitude to the former students whose results were undoubtedly crucial for the work presented here. I list their names here in alphabetical order: Babette de Fluiter, Eugene Heijnen, Richard Lukkassen, Marcel Peek, Christian Toma, and Arjen Vestjens.

Next I would like to thank Jef Van Meerbergen, Paul Lippens, and Wim Verhaegh, who were my fellow team members in the Phideo project. The discussions we had were very stimulating and often turned into a brainstorm producing enough ideas for a lifetime full of research. I am greatly indebted to Wim, who very carefully read drafts of this thesis.

I am very grateful to the people of Philips Electronic Design & Test, especially Rob Gerritsen, who has now left that organization. He was convinced of the strength of retiming from the very beginning of my work and showed great entrepreneurship in taking over the development of RetLab from me.

The work described in this thesis was carried out from 1989 through 1994 at the Philips Research Laboratories as part of the Philips Research programme. The research was supported by the European Commission in the ESPRIT 2260 project. I would like to thank the management of Philips Research, especially Engel Roza, for giving me the opportunity to use the research for this thesis.

Finally, I give my special thanks to my loving wife Anke for her understanding and support in numerous ways that I could never have done without.

Veghel, May 1998 Albert van der Werf.

Contents

1 Introduction **1**
- 1.1 Signal flow graphs and processing units . . . 1
- 1.2 Multiplexing, timefolding, and retiming . . . 3
- 1.3 Informal statement of the PU design problem . . . 5
- 1.4 Towards a solution approach . . . 11
- 1.5 Context and related work . . . 12
- 1.6 Organization of the thesis . . . 15

2 Formal Model **17**
- 2.1 Signal flow graphs . . . 17
- 2.2 Timing model . . . 19
  - 2.2.1 The timing model of operations and signal edges . . . 19
  - 2.2.2 Paths in SFGs and their timing model . . . 20
- 2.3 Mapping SFGs onto a target SFG . . . 21
  - 2.3.1 Folding factor . . . 21
  - 2.3.2 Allocation . . . 22
  - 2.3.3 Assignment . . . 22
  - 2.3.4 Retiming . . . 23
- 2.4 False paths and loops in PUs . . . 24
- 2.5 Mapping a target SFG onto a PU . . . 26
- 2.6 Processing unit design problem . . . 28
- 2.7 Special cases of PUDP . . . 28
  - 2.7.1 Multiplexing problem . . . 29
  - 2.7.2 Timefolding problem . . . 29
  - 2.7.3 Retiming problem . . . 30

3 Complexity Analysis and Decomposition **33**
- 3.1 Complexity of the processing unit design problem . . . 33
- 3.2 Decomposition of the processing unit design problem . . . 37
  - 3.2.1 Subproblem order . . . 37
  - 3.2.2 Formal subproblem definitions . . . 40
- 3.3 Complexity of subproblems . . . 47

3.3.1 Complexity of the operator allocation problem 48
3.3.2 Complexity of the operator assignment problem 48
3.3.3 Complexity of the operator duplication problem 52
3.3.4 Complexity of the generalized retiming problem 54
3.4 Overview of the complexity of PUDP and its subproblems 63

4 Basic Solution Methods **65**
4.1 Local search 65
4.1.1 Iterative improvement 66
4.1.2 Simulated annealing 66
4.2 Network flow 68
4.2.1 Network flow problems 68
4.2.2 Shortest path algorithms 70
4.2.3 Maximum-flow algorithms 72
4.2.4 Minimum-cost flow algorithms 74
4.2.5 Maximum-weight bipartite matching problem 79

5 Operator Assignment **81**
5.1 Local search 82
5.1.1 Solution space, cost, and neighborhood structure 82
5.1.2 Implementation 83
5.1.3 Experimental results 86
5.2 Large SFGs 86
5.2.1 First experimental results 88
5.2.2 Analysis 88
5.2.3 Neighborhood reductions 89
5.2.4 Experimental results 92

6 Operator Duplication **95**
6.1 Reformulating the operator duplication problem 96
6.2 Solution space, cost, and neighborhood structure 97
6.3 Implementation of local search 99
6.4 Experimental results 102

7 Timing Analysis **107**
7.1 Basic timing properties 108
7.2 Redundancy 110
7.3 Implementation of speed constraint derivation 116
7.3.1 Constraint generation 117
7.3.2 Delay and weight labeling 117

7.3.3 Efficient labeling 121
7.4 Experimental results 123

8 Retiming **127**
8.1 Solution space 128
8.1.1 Timing constraint satisfaction 129
8.1.2 Phase constraint satisfaction 133
8.2 Cost 137
8.3 Neighborhood structure 138
8.4 Simulated annealing 142
8.5 Experimental results 144

9 Conclusions **147**

Bibliography **151**

Author index **157**

Summary **159**

Samenvatting **161**

Curriculum Vitae **163**

1

Introduction

In this thesis we are concerned with the design of area-efficient *Processing Units* (PUs) that can execute one or more functions, specified as *Signal Flow Graphs* (SFGs), in one or more clock cycles and at a high clock frequency. The functions are part of a signal processing algorithm specifying the behaviour of a digital electronic system. The PUs are applied in video signal processors as part of an *Integrated Circuit* (IC) in silicon.

In this chapter we introduce the concepts of SFGs and PUs and we describe a number of techniques for designing PUs. The organization of this introductory chapter is as follows. SFGs and PUs are introduced in Section 1.1. Three basic techniques to design PUs, namely *multiplexing*, *timefolding*, and *retiming*, are discussed in Section 1.2. Section 1.3 discusses the relation between SFGs and PUs, and informally introduces the PU design problem. The solution strategy for the PU design problem that is described in this thesis is presented in Section 1.4. The context and related work of PU design is discussed in Section 1.5.

1.1 Signal flow graphs and processing units

A signal flow graph represents a function that consists of *operations* of which the *input* and *output terminals* are connected by *edges*; see Figure 1.1*a*. Every operation in an SFG is executed repeatedly, for an indefinite period of time; see Fig-

ure 1.1*b*. An output terminal produces a *signal* on an edge, which is consumed at an input terminal. A signal may be used in a subsequent execution of the SFG. In that case, the edge represents a number of succeeding signal values concurrently. The *weight* on an edge refers to the number of values of previous iterations.

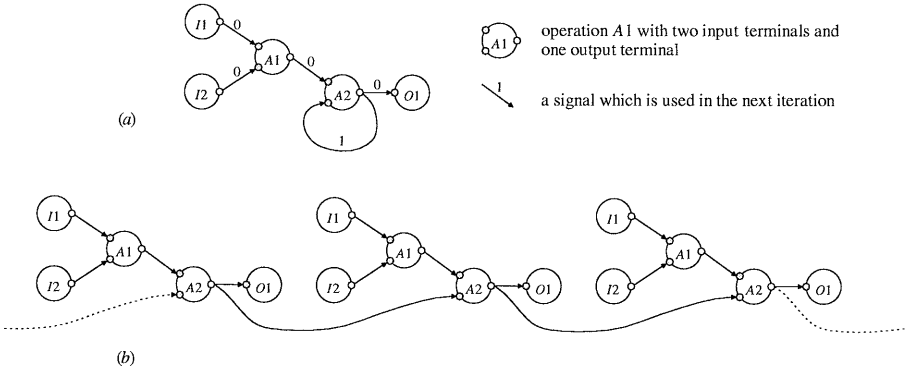


Figure 1.1. An example of a signal flow graph (*a*) with two input operations, $I1$ and $I2$, and one output operation $O1$. The operations $A1$ and $A2$ represent additions. An indefinite repetition of executions of the SFG is shown in (*b*).

We consider PUs that consist of *operators*, *registers*, and *multiplexers* connected by point-to-point connections called *wires*; see Figure 1.2. A processing unit can execute one or more functions, specified as SFGs, in one or more clock cycles.

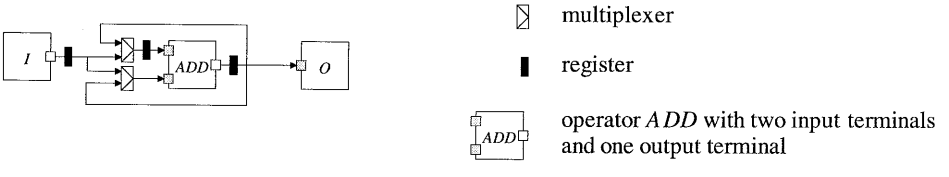


Figure 1.2. An example of a processing unit. It has one input operator, I, one output operator, O, and one adder, ADD. Furthermore, it has two multiplexers and three registers.

In this thesis we are concerned with the synchronous processing of digital signals, where a clock signal synchronizes the flow of data. A clock signal is a sequence of clock cycles that are repeated at a certain frequency. The transfer of data from one clock cycle to the next is performed by registers. At the end of a clock

cycle, which is marked by a rising edge of the clock signal, data at the input of a register is stored and available at its output during the next clock cycle.

The time it takes within a clock cycle for an operator to return a value at an output terminal after a change of value at one of its input terminals is called the *asynchronous delay*. A sequence of operators and multiplexers, connected by wires, comprises a path. The length of a path represents the total asynchronous delay of the operators and multiplexers on the path. The maximum clock frequency at which a PU can operate is determined by the so-called critical path, which is the longest path between any two subsequent registers and which must have a delay smaller than the clock period. Within the delay of the critical path all the signals of a PU will settle to a certain value. *Asynchronous feedback loops*, which are loops that contain no registers, are not allowed.

We speak of a pipelined execution of functions when more than one execution of a function is performed in parallel in the same clock cycle, but the executions are in different stages. In that case it takes a number of clock cycles before the result of a function's execution is available at the output operators. The number of clock cycles it takes to get a result is called the *synchronous delay* or the *latency*.

In a PU, data is routed from output terminals of operators to input terminals of operators by a switching network containing registers and multiplexers. A multiplexer has two input terminals, one output terminal, and a select input. The select input indicates from which input terminal the data must be routed to the output terminal.

Here, we assume that a global controller is present outside a PU that generates the select signals for the multiplexers. Furthermore, we assume that a clock signal is present for the registers. The clock terminals of registers and the select terminals of multiplexers are omitted in figures for the sake of clarity.

1.2 Multiplexing, timefolding, and retiming

Below we consider three basic techniques that are jointly used in the design of processing units: multiplexing, timefolding, and retiming.

Multiplexing is a technique to construct a *multi-functional* PU that can perform more than one function. The goal with multiplexing is to save on hardware costs by reusing operators for the execution of more than one operation of different SFGs.

Depending on the function that is executed, the signals produced by operators must be routed to certain input terminals of other operators. To this end, a switching network between the operators is present that contains multiplexers. Although a multi-functional PU requires less area for operators than two or more single-functional PUs, it uses additional area for the switching network. Moreover, its

multiplexers introduce additional asynchronous delay in signal paths and therefore may decrease the maximum clock frequency at which a function can be executed.

The multiplexing technique is explained with the following example. Given are the SFGs of a butterfly and a 5-tap filter, as shown in Figure 1.3. For more detailed descriptions we refer to Rabiner & Gold [1975], Oppenheim & Schafer [1975], and Roberts & Mullis [1987]. A PU that can perform either a butterfly or a 5-tap filter is shown in Figure 1.4.

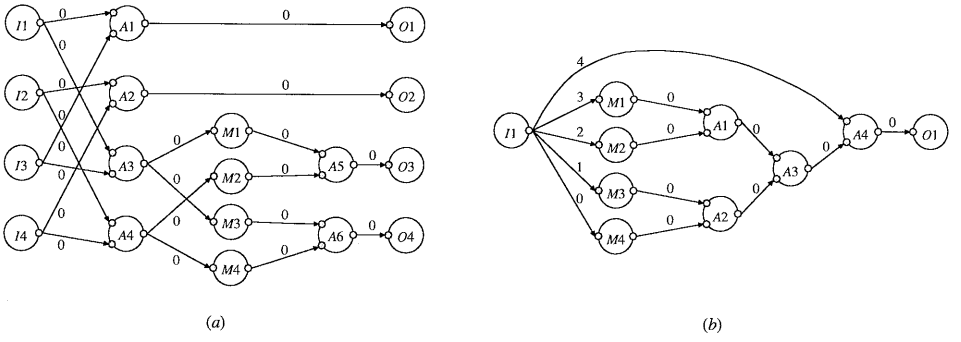


Figure 1.3. Two functions, namely a butterfly (*a*) and a 5-tap filter (*b*), specified as SFGs.

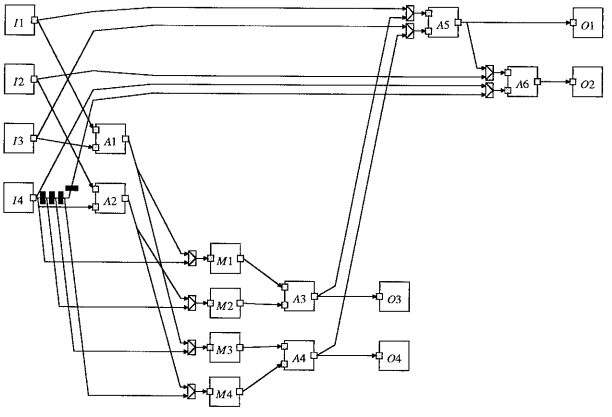


Figure 1.4. A PU that can either execute a butterfly or a 5-tap filter.

Timefolding is a technique to construct a *multi-cycle* PU that can execute a function in more than one clock cycle. The goal with timefolding is to save on hardware

costs by reusing operators for the execution of more than one operation of the same SFG. In the case of timefolding, the executions of operations are spread over more than one clock cycle.

Timefolding may delay operations for a number of clock cycles and consequently signals may need to be delayed one or more clock cycles. To this end, the switching network contains registers to store signals for one or more clock cycles. Furthermore, it contains multiplexers to route signals to their destinations. A multi-cycle PU requires less area for operators than a single-cycle PU, but additional area is used by the switching network. The multiplexers in the switching network may increase the longest path, but the registers in it may decrease it; therefore the effect on the maximum clock frequency at which the function can be executed is hard to predict.

An example of a multi-cycle PU is shown in Figure 1.2. Every two clock cycles it can execute a function according to the SFG shown in Figure 1.1*a*. The time at which operations are executed is shown in Figure 1.5*a*. This can also be compactly represented in the SFG by (modified) weights on its edges; see Figure 1.5*b*. The multiplexer inputs that are selected are indicated in Figure 1.6.

Retiming is a technique to construct a PU that executes a function at a required clock frequency. The technique is called retiming, because the position of registers in a PU is determined by a change of the times at which operations are executed. Since the placement of the registers determines the longest path between two registers in the PU, retiming affects the maximum clock frequency at which functions can be executed. Retiming may result in a change in the production and consumption times of output and input signals, respectively. Consequently, executions of functions may become overlapping in time, resulting in a pipelined implementation. Examples of PUs that are designed using retiming are shown in Figure 1.7.

The three design techniques *multiplexing*, *timefolding*, and *retiming* can be combined. A PU that can perform either a butterfly or a 5-tap filter every two clock cycles is shown in Figure 1.8.

1.3 Informal statement of the PU design problem

The relationship between SFGs and a PU that can execute them can be defined in two steps. Firstly the SFGs are mapped onto a new target SFG, and secondly this SFG is implemented by a PU.

The first mapping, from one or more source SFGs to a target SFG, is defined by three sets of variables. These are the number of operations *allocated* in the target SFG, each of which corresponds to an operator in the PU, the operator to

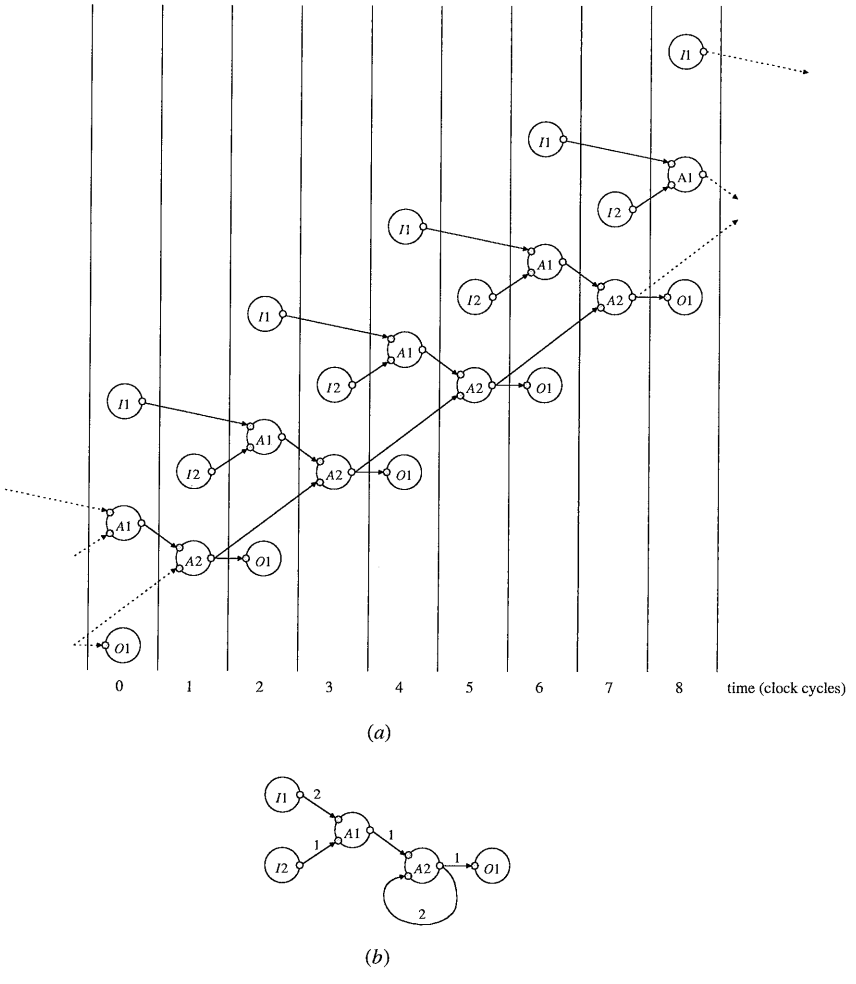


Figure 1.5. (*a*) The time at which the operations of the SFG of Figure 1.1*a* are executed. Note that the SFG is executed in a timefolded and pipelined way. (*b*) The SFG of Figure 1.1*a* with its weights modified according to the times at which its operations are executed as shown in (*a*).

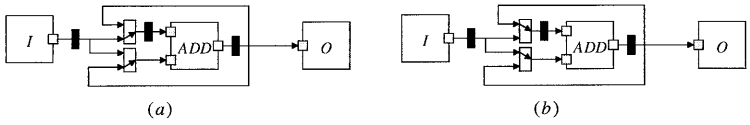


Figure 1.6. A multi-cycle PU that executes the SFG of Figure 1.1. The multiplexers indicate the input that is selected. The situation of (*a*) corresponds to the clock cycles in which operations $I2$ and $A2$ are executed. The situation of (*b*) corresponds to the clock cycles in which operations $I1$, $A1$, and $O1$ are executed.

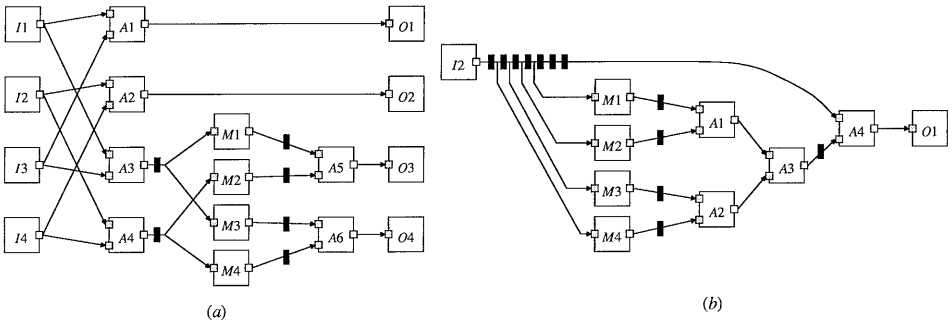


Figure 1.7. PUs designed using the retiming technique for the butterfly (*a*) and the 5-tap filter (*b*), which are specified by the SFGs shown in Figure 1.3. We assumed that on a single path one multiplication or two additions can be performed in one clock cycle and that the delays of the input and output operators can be neglected.

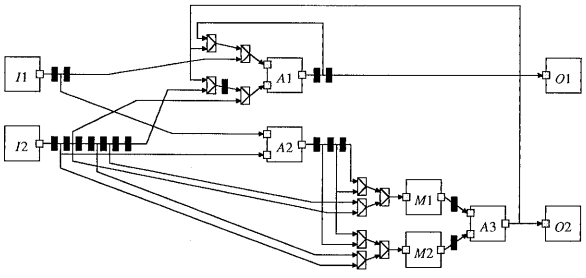


Figure 1.8. A PU that every two clock cycles can perform either a butterfly or a 5-tap filter, which are specified by the SFGs shown in Figure 1.3. We assume that on a path one multiplication or two additions can be performed in one clock cycle and that the delays of the input and output operators can be neglected.

which each operation is *assigned*, and the time at which it is executed. The latter is indicated relative to an initial execution time and is called *retiming*. To design a PU we have to determine these variables; therefore we call them *decision variables*.

The variables must satisfy the following design constraints. There must be sufficient operators available to execute the operations. The time at which an operation is executed must be such that no path is longer than the clock period. Furthermore, two operations of one SFG must be executed at different clock cycles when they are mapped onto the same operator. No asynchronous feedback loops may be present in the PU designed.

The three sets of decision variables define a target SFG in which the edges are determined by the assignment of operations to operators. Every edge in the target SFG has a corresponding edge in one of the source SFGs. The weight of the edge in the target SFG is determined by the execution times of the producing and consuming operations and the original weight of the corresponding edge in one of the SFGs. The effect of retiming is that delaying the consuming operation increases the weight, while delaying the producing operation decreases it.

We take as an example the design of the PU that is shown in Figure 1.8, which every two clock cycles can execute either a butterfly or a 5-tap filter, which are specified by the SFGs shown in Figure 1.3. The allocation consists of two input operators, two output operators, two multipliers, and three adders. Table 1.1 gives the assignment. The retiming of the operations and its effect on the weights of the edges in the source SFG is shown in Figure 1.9. The target SFG constructed according to the allocation, assignment, and retiming is shown in Figure 1.10.

The second part of the mapping of SFGs onto a PU is the mapping of the

Table 1.1. The assignment of operations to operators for the example of mapping the SFGs of the butterfly and the 5-tap filter as shown in Figure 1.3 onto the PU of Figure 1.8.

		Operations																									
		Butterfly																5-tap filter									
		I1	*I2*	*I3*	*I4*	*O1*	*O2*	*O3*	*O4*	*A1*	*A2*	*A3*	*A4*	*A5*	*A6*	*M1*	*M2*	*M3*	*M4*	*I1*	*O1*	*A1*	*A2*	*A3*	*A4*	*M1*	*M2*	*M3*	*M4*
Operators	*I1*	•	•																										
	I2			•	•															•									
	O1					•	•														•								
	O2							•	•																				
	A1									•	•													•	•				
	A2											•	•																
	A3													•	•							•	•						
	M1															•		•								•		•	
	M2																•		•								•		•

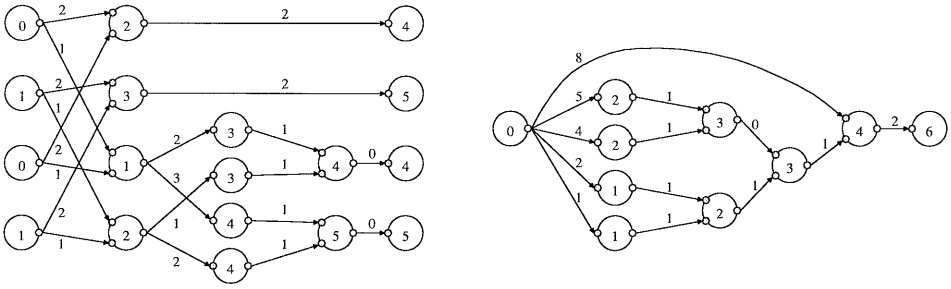


Figure 1.9. The SFGs of the butterfly (*a*) and the 5-tap filter (*b*). The number in an operation denotes its retiming with respect to the SFGs shown in Figure 1.3. The weights on the edges are adjusted according to the retiming.

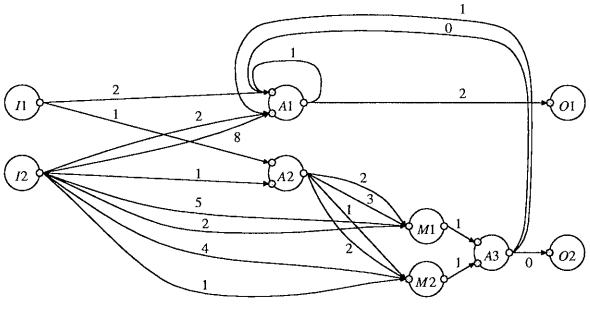


Figure 1.10. A target SFG constructed in the design process for a multi-functional multi-cycle PU that can execute either the butterfly or the 5-tap filter. The weights correspond to the SFGs shown in Figure 1.9 and the assignment is according to Table 1.1.

target SFG onto the PU. There, the sets of decision variables are associated with the *register allocation*, indicating for every terminal the number of registers that are placed there, and the *multiplexer allocation*, indicating for every input terminal the number of multiplexers placed there. To implement an edge, the total number of registers allocated to the corresponding input and output terminals should be at least the weight of the edge. Furthermore, the number of multiplexers allocated at an input terminal should be equal to one less than the number of edges connected to it with both different sources and with different weights. An example of a register and multiplexer allocation is shown in Figure 1.11. Note that we limit ourselves to the sharing of registers by edges in the target SFG that are incident to the same output or input terminal.

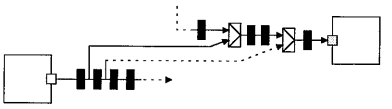


Figure 1.11. An example of a register and multiplexer allocation. At the output terminal four registers are allocated. At the input terminal four registers and two multiplexers are allocated. The connection indicated by a solid line between the input and output terminals corresponds with an edge with a weight equal to four.

The second part of the mapping can be considered as a sophisticated cost evaluation, i.e., main decisions are taken in the first part, but their costs are determined in the second part.

The objective of PU design

Our objective is to develop a method for designing a PU that occupies minimal silicon area and runs at the required clock frequency, using a combination of multiplexing, timefolding, and retiming techniques. We use an estimate of the area given by a weighted sum of the number of operators, multiplexers, registers, and wires.

The effect of each of the techniques, and certainly that of a combination of them, on the area of the PU and the clock frequency at which it can execute the SFGs cannot well be predicted in general. Since the area is a minimization goal, and the clock frequency is a constraint, the retiming technique is the most important one to obtain at least a feasible PU.

1.4 Towards a solution approach

In this section we discuss an approach to developing a solution method for the efficient design of PUs of which the behaviour is specified by one or more SFGs.

First, a formal model is set up, which we use to formally define the processing unit design problem. The model contains the decision variables and formally defines SFGs, PUs, and their timing behaviour. The processing unit design problem is formulated as a combinatorial optimization problem. Next, the complexity of the processing unit design problem is analyzed to determine whether the problem can be solved easily using (existing) algorithms with acceptable run times, or if it is, formally spoken, difficult. The result of this analysis is that the problem is difficult to solve; this result is essential to the work that is described further in this thesis.

The processing unit design problem is decomposed such that the determination of a retiming follows the determination of an operator assignment. The main reason for this order is that we want to exploit regularity in the source SFGs, i.e., substructures of it may be identical. First we determine an operator assignment such that the identical substructures become present in the target SFG. For example, in the target SFG shown in Figure 1.10 the operators $M1$, $M2$, and $A3$ comprise a substructure that is also present in the source SFGs shown in Figure 1.3. Because this assignment is based on a minimal number of operators, asynchronous feedback loops may be introduced. These loops will never carry data around since the data routing will be according to one of the functions that can be executed, which is free of asynchronous feedback loops. Therefore they are called *false loops*. Since a PU may not contain asynchronous feedback loops, the false loops are removed in the following step that allocates additional operators to which some operations are reassigned. To ensure that the PU designed can run at the required clock frequency, a timing analysis is performed resulting in a set of linear constraints on retiming. Finally, a feasible retiming is determined such that corresponding operations in identical substructures are, if possible, equally retimed.

Restrictions on the instances of the processing unit design problem lead to the definition of three special cases of it, i.e., the multiplexing, timefolding, and retiming problem. Each of the special cases emphasizes one of the subproblems into which the PU design problem is decomposed. The complexity of each of the subproblems is analyzed, also in special cases.

The identified subproblems can be handled using algorithms belonging to the field of local search and network flow. Local search is applied for most of the subproblems, but first in a straightforward way. If the time performance of the local search algorithm is unacceptable for problem instances of a typical size, we search for ways to improve the application of local search. Although local search algorithms can be readily applied without the use of much detailed knowledge, we

show that good solutions can only be obtained within acceptable run times by using problem-specific information.

Network flow algorithms can be applied to solve the retiming problem. Furthermore, a shortest path algorithm is used for timing analysis, and a bipartite-graph maximum-weighted matching algorithm is used to compute the register cost of a PU. Even when a problem is mathematically spoken easy and can be solved by, e.g., a network flow algorithm, the solution method deserves attention for its time performance.

1.5 Context and related work

The work described in this thesis is in the field of Electronic Design Automation (EDA). In the past decades, EDA has received increasing attention and more participants. In 1980 the first International Conference on Circuits and Computers (ICCC) [Rabbat, 1980] was held, which three years later was renamed the International Conference on Computer-Aided Design. This was followed in 1982 with the publication of the first IEEE (Institute of Electrical and Electronics Engineers) Transactions on Computer-Aided Design (CAD) of Circuits and Systems [Rohrer, 1982]. A part of EDA concentrates on the translation of an abstract behavioral specification to a register-transfer level implementation, which is called *high-level synthesis* [McFarland, Parker & Camposano, 1990]. For an overview of problems and their solution methods belonging to the field of high-level synthesis, we refer the reader to McFarland, Parker & Camposano [1990], Borriello & Detjens [1988], Martin & Knight [1993], and Stok [1994].

In the eighties, research on high-level synthesis was started at the Philips Research Laboratories. The first result was a design method for audio signal processing called Piramid [Woudsma, Beenker, Van Meerbergen & Niessen, 1990]. This method was successfully applied to, e.g., the design of an ASIC for compact disc players [De Loore, Crombez, Delaruelle, Sheridan, Woudsma, Niessen, Biesterbos, Gubbels & Repko, 1992]. In the late eighties it became apparent that digital video signal processing had become feasible for consumer products and, consequently, that it was important for Philips. Therefore, new work was directed towards video applications and resulted in the design method Phideo [Van Meerbergen, Lippens, Verhaegh & Van der Werf, 1995]. For Piramid as well as for Phideo, processing units are designed that can execute functions specified as subgraphs of the total SFG specifying the function to be implemented. These subgraphs are treated as single operations during the scheduling, allocation, and assignment phases of Phideo and Piramid.

As a result of the scheduling of the subgraphs, they can be executed at non-overlapping time intervals. Consequently, multiplexing techniques may be used

to design processing units that can execute more than one SFG. As a result of scheduling, subgraphs can be executed repetitively with a period greater than one clock cycle. This is especially the case in Phideo, which is based on a model of multidimensional periodic operations [Verhaegh, 1995]. Consequently, timefolding techniques can be used to design processing units that can execute SFGs in more than one clock cycle. An overview of the possible design techniques that can be applied in certain situations is shown in Figure 1.12.

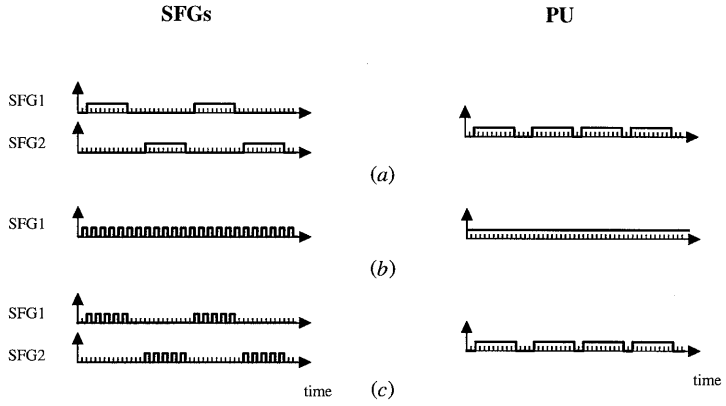


Figure 1.12. The activities of the source SFGs determine the design techniques that can be used. In (*a*) the activities of the source SFGs allow the application of the multiplexing technique. The activity of the PU on which the SFGs can be executed is shown as well, which results in well occupied hardware resources. In (*b*) the activity of one source SFGs is such that the timefolding technique can be applied. In (*c*) the activities of the source SFGs are such that both multiplexing and timefolding techniques can be applied.

To make the processing units run at a desired clock frequency, we apply retiming techniques. As a result of retiming, a processing unit can be pipelined, i.e., consumption and production times of input and output signals, respectively, can be changed with respect to the times specified by the corresponding SFG [Van der Werf, McSweeney, Van Meerbergen, Lippens & Verhaegh, 1991]. These changes can be accommodated by the scheduling and memory synthesis techniques of Piramid and Phideo.

The classical retiming technique was introduced by Leiserson, Rose & Saxe [1983]. Saxe [1985] described this work in his thesis and a refined version of the reported work appeared in 1991 [Leiserson & Saxe, 1991]. The first step in the classical retiming technique is the derivation of a number of speed constraint relations between pairs of operations. Together with causality constraints and a cost

function they comprise a special case of an (integer) linear programming problem, which can be solved in polynomial time using network-flow algorithms [Ford & Fulkerson, 1962]. To derive the speed constraints in the first step an asynchronous-timing analysis, determining the length of paths between two registers is performed for all possible register allocations. The problem of speed constraint derivation can be solved with shortest path algorithms, which are also part of the field of network flows [Ford & Fulkerson, 1962].

When using the classical retiming technique for the design of processing units that run at high throughput frequencies and in a pipelined fashion, the timing analysis step may take a very long time. This was observed by Münzner & Hemme [1991], who developed a more efficient heuristic, which may find sub-optimal solutions. An important contribution of the work described in this thesis is a speed constraint derivation algorithm that is fast and gives optimal solutions. This work was first reported in [Van der Werf, Van Meerbergen, Aarts, Verhaegh & Lippens, 1994]. About half a year later, still in 1994, Shenoy & Rudell reported a similar, but less efficient method for speed constraint derivation.

Retiming in the context of resource sharing can be viewed as scheduling. Scheduling is an important area of interest in the field of combinatorial optimization [Papadimitriou & Steiglitz, 1982]. Baker [1974] defines scheduling as the problem of allocating scarce resources to activities in time. 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]. Scheduling for processing unit design is non-preemptive, which means that the execution of an operation cannot be interrupted. Several approaches have been presented to scheduling in the context of IC design, such as list scheduling [McFarland, 1986; Haupt, 1989; Parker, Pizarro & Mlinar, 1986; Park & Parker, 1988; Verhaegh, 1995], 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; Timmer & Jess, 1993; Verhaegh, Lippens, Aarts, Korst, Van Meerbergen & Van der Werf, 1995].

Scheduling and retiming both determine the time at which operations are executed. Whereas retiming is considered to deliver a circuit that meet asynchronous timing constraints, scheduling is used to meet both synchronous timing constraints and hardware resource constraints. In this thesis a combination of both objectives is considered, which is covered by the term retiming. Related work has been reported in which the combination is often referred to as "operator chaining during scheduling"; see, e.g., [Parker, Pizarro & Mlinar, 1986; Park & Parker, 1988; Park & Kyung, 1993].

In this thesis, we present a decomposition of the processing unit design problem

in which we determine a retiming after a number of operators are allocated and the operations are assigned to them. In most other applications of scheduling and operator assignment in the context of IC design the order is reversed; see, e.g., [Parker, Pizarro & Mlinar, 1986; Park & Parker, 1988; Park & Kurdahi, 1989; Park & Kyung, 1993; McFarland, 1986; Haupt, 1989; Paulin & Knight, 1989; Timmer & Jess, 1993].

The importance of an operator assignment has also been recognized by Note, Catthoor & De Man [1989], Geurts, Catthoor & De Man [1993a], and Geurts, Catthoor & De Man [1993b], who discuss the design of processing units that can perform more than one function for ICs designed with the Cathedral III method [Note, Geurts, Catthoor & De Man, 1991]. Geurts, Catthoor & De Man [1993b] avoid the existence of false loops by taking additional constraints into account in a quadratic 0-1 programming model. Other work on operator assignment in which the importance of false loops is considered is reported by Stok [1992].

Since many of the problems related to 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.

1.6 Organization of the thesis

The organization of this thesis in chapters and sections reflects the solution approach. Chapter 2 describes a formal model that we use to describe the processing unit design problem. Chapter 3 discusses the complexity of the processing unit design problem and its decomposition. Next we have Chapter 4, which is an intermediate chapter that summarizes the concepts of local search and network flow. Chapter 5 discusses a solution method for the first subproblem, which is to determine an operator assignment. Chapter 6 discusses the next subproblem, which is to find an allocation of additional operators to which some operations are reassigned. Chapter 7 discusses the subproblem concerned with the timing analysis of SFGs, which is to efficiently derive speed constraints. Chapter 8 discusses the final subproblem, which is to determine a retiming. We end this thesis with Chapter 9, which contains the conclusions of the presented work.

2

Formal Model

In this chapter we first give the formal model of an SFG in Section 2.1, followed by a discussion on its timing model in Section 2.2. In Section 2.3 the relation between a number of source SFGs and the target SFG on which they are mapped is discussed. This is followed by a discussion in Section 2.4 on the modelling of false loops that may be introduced by the mapping. The implementation of an SFG is called a PU and its cost is discussed in Section 2.5. In Section 2.6 a formal problem definition of PU design is given. We conclude this chapter with defining three special cases of PU design in Section 2.7.

2.1 Signal flow graphs

An SFG consists of operations that are interrelated in some way. Here, we define the edges in an SFG not between the operations, as is done in most graph-based specifications, but between operation terminals. Therefore, an SFG can be represented as a directed graph in which the vertices are the operation terminals, either input or output, and the edges run from output terminals to input terminals. This leads to the following definition.

Definition 2.1 (signal flow graph). A *signal flow graph* $\mathcal{G}$ is given by a three tuple (V,T,E) in which

V is a set of *operations*, and

$T = O \cup I \subseteq V \times \mathbb{N}$	is a set of *operation terminals*, where O is a set of *output terminals*, I is a set of *input terminals*, and $O \cap I = \emptyset$, and
$E \subseteq O \times I$	is a multi-set of *signal edges* representing the signal flow,

where $\mathbb{N}$ is the set of non-negative integers. □

Input operations of an SFG are modelled as operations without input terminals and output operations are modelled as operations without output terminals.

Since we discuss the mapping of possibly more than one source SFG onto a PU, different SFGs are identified by an index, $\mathcal{G}_i = (V_i, T_i, E_i)$, where $i = 1, \ldots, N$. The union of all these SFGs is indicated by $\mathcal{G}$ and is called the *union SFG*. The union of all operations is indicated by V, i.e., $V = \cup_{i=1,\ldots,N} V_i$. Likewise, the unions of all terminals and edges are defined.

The design of a processing unit is partitioned into two steps. In the first step a number of source SFGs are mapped onto a target SFG and in the second step the target SFG is mapped onto a PU. The relationship between a target SFG and a PU is very direct. Consequently, we neglect the difference and also refer to a target SFG by the words *processing unit*. For the PU we use an apostrophe in the notation $(\mathcal{G}' = (V', T', E'))$, as well as for the variables pertaining to a PU. Furthermore, we call the operations of a PU *operators*.

To be able to formally define the relationship between SFGs and a PU, it is necessary to define the set of operation types. From a structural perspective, every operation type is characterized by its number of inputs and number of outputs. Furthermore, the cost of an operation type is important.

Definition 2.2 (operation type). $\mathcal{T}$ denotes a set of *operation types* where

$\mathcal{I} : \mathcal{T} \to \mathbb{N}$	returns the number of input terminals, and
$\mathcal{O} : \mathcal{T} \to \mathbb{N}$	returns the number of output terminals, and
$\mathcal{C} : \mathcal{T} \to \mathbb{Q}$	returns the cost of an operator of this type.

□

We consider operations and operators that can perform only one type of operation. We define the following function, so that we can determine the type of an operation.

Definition 2.3 (type function). Given are an SFG $\mathcal{G}$ and a type set $\mathcal{T}$. The *type function* $l : V \to \mathcal{T}$ returns the type of an operation. □

The number of terminals of each operation is determined by the corresponding operation type. Given an SFG $\mathcal{G}$ and a type function ℓ, the relationship between the

operation set and the terminal set is given as

$$I = \{(v,i) | v \in V \text{ and } i = 1,\ldots,\mathcal{I}(\ell(v))\}$$
$$O = \{(v,i) | v \in V \text{ and } i = \mathcal{I}(\ell(v))+1,\ldots,\mathcal{I}(\ell(v))+\mathcal{O}(\ell(v))\}.$$

2.2 Timing model

We discuss the synchronous processing of digital signals, where a processor clock synchronizes the flow of data. For synchronous circuitry, time is divided into an asynchronous component and a synchronous one. Synchronous timing is expressed in integer multiples of clock cycles, while asynchronous timing is expressed in real units, e.g., nanoseconds. We first discuss the timing model of operations and signal edges in Section 2.2.1. In Section 2.2.2 we introduce paths and discuss their timing properties.

2.2.1 The timing model of operations and signal edges

The asynchronous timing model of an operation contains a delay for every pair consisting of an input terminal and an output terminal, which is defined as follows.

Definition 2.4 (delay). Given is an SFG $\mathcal{G}$. The *delay* is a function $del : I \times O \rightarrow \mathbb{Q}^+ \cup \{\sim\}$ that returns the delay between an input terminal and an output terminal. The delay is only defined when there is a path from the input terminal to the output terminal inside one operation and without registers. Otherwise the delay is undefined, which is indicated by $\sim$. $\mathbb{Q}^+$ denotes the set of non-negative elements of $\mathbb{Q}$. □

To allow the timing model to be useful for sequential operations that take more than one clock cycle to execute, e.g., pipelined multiplications, it is extended with a number of functions. To this end, the timing model of an operation of an SFG contains information about the delay from an input terminal to an internal register and from an internal register to an output terminal. The former is called the *data available time* and the latter is called the *data ready time*. They are defined as follows.

Definition 2.5 (data available time). Given is an SFG $\mathcal{G}$. The *data available time* is a function $dat : I \rightarrow \mathbb{Q}^+$ that returns the maximum delay of an internal path from an input terminal to either an internal register or an output terminal of the corresponding operation. □

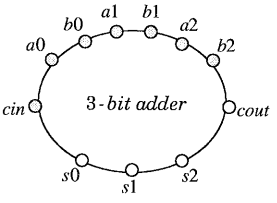
Definition 2.6 (data ready time). Given is an SFG $\mathcal{G}$. The *data ready time* is a function $drt : O \rightarrow \mathbb{Q}^+$ that returns the maximum delay of an internal path from either an internal register or an input terminal of the corresponding operation to an output terminal. □

If an operation v has a completely asynchronous behaviour, then the following holds for its input terminals (v,a) and its output terminals (v,b):

$$dat((v,a)) = \max_{o=(v,b)\in O} del((v,a),o), \text{and}$$

$$drt((v,b)) = \max_{i=(v,a)\in I} del(i,(v,b)).$$

An example of a timing model of an operation is given in Figure 2.1.



inputs / outputs			a0	a1	a2	b0	b1	b2	cin
		dat	10	5	5	10	5	5	10
	drt	del							
s0	5		5	~	~	5	~	~	5
s1	10		10	5	~	10	5	~	10
s2	5		~	~	5	~	~	5	~
cout	5		~	~	5	~	~	5	~

Figure 2.1. An example of an operation (3-bit adder) and its timing model. The adder is internally pipelined.

Below we assume that for an SFG $\mathcal{G}$, a given clock period τ, and timing functions drt, dat, and del, every operation in the SFG can be executed within one clock period, i.e.,

$drt(o) < \tau$	for each $o \in O$,
$dat(i) < \tau$	for each $i \in I$, and, consequently,
$del(i,o) < \tau$	for each $(i,o) \in I \times O$ with $del(i,o) \neq \sim$.

A signal in an SFG may be delayed a number of clock cycles between its production and consumption. To quantify this relation a weight on a signal edge, defined as follows, is applied.

Definition 2.7 (weight). Given is an SFG $\mathcal{G}$. The weight $w : E \to \mathbb{Z}$ is defined as the number of clock cycles that a signal on an edge is delayed. □

Here, we assume that both the data available times and data ready times of registers are equal to zero and that multiplexers have delays equal to zero. The timing model of operations is thus applicable for both sequential and combinational operations.

2.2.2 Paths in SFGs and their timing model

In this section we discuss some basic theory on paths in SFGs.

Definition 2.8 (path). Given is an SFG $\mathcal{G}$. A *path* p is an alternating sequence of output and input terminals, starting with an output terminal and ending with an input terminal, of the following form: $p = (o_1, i_2, o_2, ..., i_{n-1}, o_{n-1}, i_n)$, where $(o_k, i_{(k+1)}) \in E$ for each $k = 1, \ldots, n-1$ and $del(i_k, o_k) \neq \sim$ for each $k = 2, \ldots, n-1$. Furthermore, $o_k \neq o_l$, and $i_k \neq i_l$ for each $k, l = 1, \ldots, n$ with $k \neq l$. Sometimes we only need to know on what operation output the path begins and on what operation input it ends. We then denote the path p by $(o_1 \mapsto i_n)$. $P_{\mathcal{G}}$ denotes the set of all paths in $\mathcal{G}$. □

The definition of a path is such that it cannot be a cycle or contain one, because it may visit an operation terminal only once. Moreover, all operations in a path belong to one source SFG.

The notion of timing functions in the timing model and weights on signal edges is extended to the domain of paths.

Definition 2.9 (path delay). Given are an SFG $\mathcal{G}$ and timing functions *dat*, *drt*, and *del*. The *path delay* is a function $D : P_{\mathcal{G}} \rightarrow \mathbb{Q}^+$ that returns the delay of a path, which is defined as follows. For each $p = (o_1, i_2, o_2, ..., i_{n-1}, o_{n-1}, i_n) \in P_{\mathcal{G}}, D(p) = drt(o_1) + \sum_{k=2}^{n-1} del(i_k, o_k) + dat(i_n)$. □

Note that the delay of a path does not depend on the weights of the signal edges in the path. Consequently, the delay of a path can be greater than the clock period at which an SFG can be executed.

Definition 2.10 (path weight). Given are an SFG $\mathcal{G}$ and a timing function w. The *path weight* is a function $W : P_{\mathcal{G}} \rightarrow \mathbb{Z}$ that returns the weight of a path $p = (o_1, i_2, o_2, ..., i_{n-1}, o_{n-1}, i_n) \in P_{\mathcal{G}}$ and is defined as $W(p) = \sum_{k=1}^{n-1} w(o_k, i_{k+1})$. □

2.3 Mapping SFGs onto a target SFG

The mapping of a union of source SFGs onto a PU is done in two steps. First the source SFGs are mapped onto a target SFG, and then this target SFG is implemented by a PU. The first step is discussed in this section, the second step in one of the following sections, Section 2.5. First the folding factor is introduced in Section 2.3.1. This is followed by a discussion in Sections 2.3.2 to 2.3.4 on the three decision variables: allocation, assignment, and retiming.

2.3.1 Folding factor

Timefolding implies that the synchronous timing behaviour of a source SFG is expressed using a processing clock that has more than one clock cycle for every execution of the SFG. The number of clock cycles used for one execution is called the folding factor and is defined as follows.

Definition 2.11 (folding factor). Given are a source SFG $\mathcal{G}$ and a weight w. The *folding factor* of $\mathcal{G}$ is defined by a positive integer f. Furthermore, the folding factor defines a folded weight $w_f : E \rightarrow \mathbb{Z}$ as follows. For each $e \in E$, $w_f(e) = f * w(e)$. □

Below we do not consider explicitly the folded weight; we assume that it is already accounted for in the weight. The number of clock cycles that it takes to execute one iteration of a source SFG is expressed in *phases*. The notion of phase is formally defined in Section 2.3.4.

2.3.2 Allocation

The operator allocation determines the operator set V' and the terminal set T' of a PU $\mathcal{G}'$.

Definition 2.12 (operator allocation). Given is a set of operation types $\mathcal{T}$. The *operator allocation* is a function $al : \mathcal{T} \rightarrow \mathbb{N}$ that returns for each operation type the number of allocated operators. □

In order to execute each of a union of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, by a PU $\mathcal{G}'$, it must have a sufficient number of operators, i.e., there is a lower bound on $|V'|$. Since we consider operators that can perform only one type of operation, this bound is determined by the bounds of each specific operation type. Therefore, the sets V_i, $i = 1,\ldots,N$, are partitioned into subsets of the same operation type, i.e., $V_i = \bigcup_{j \in \mathcal{T}} V_{i,j}$, where $V_{i,j} = \{v \in V_i \mid \ell(v) = j\}$, $i = 1,\ldots,N$. A constraint on the allocation following from the instances is as follows.

(*i*) $al(j) \geq \max_{i=1,\ldots,N} \lceil \frac{|V_{i,j}|}{f_i} \rceil$ for each $j \in \mathcal{T}$.

2.3.3 Assignment

The assignment of the operations of a source SFGs to the operations of a target SFG is defined as follows.

Definition 2.13 (operator assignment). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, a set of operators V', a type function ℓ, and folding factors f_i. The *operator assignment* is a function $as : V \rightarrow V'$ that returns for each operation in V the operator in V' on which it is executed. □

An operator assignment must meet the constraints that each operation is assigned to an operator of the same type and that the number of operations in the same source SFG that are assigned to one operator cannot exceed the number of available phases, i.e.,

(*ii*) $\ell(v) = \ell(as(v))$ for each $v \in V$,

(*iii*) $|\{v \in V_i | as(v) = v'\}| \leq f_i$ for each $v' \in V'$, and for each $i = 1, \ldots, N$.

The edges and their weights in the PU are defined by the operator assignment as follows.

(*iv*) $e' = ((as(v), a), (as(u), b)) \in E'$ and $w(e') = w(e)$ for each $e = ((v, a), (u, b)) \in E$.

Thus each edge in the source SFGs is mapped onto only one edge in the target SFG and an edge in the target SFG has only one edge in one of the SFGs mapped onto it.

2.3.4 Retiming

Changing the times at which operations are executed is defined as follows.

Definition 2.14 (retiming). Given are an SFG $\mathcal{G}$ and a weight w. The *retiming* is a function $r : V \rightarrow \mathbb{Z}$ that returns for each operation in V the number of clock cycles the operation is delayed. Furthermore, retiming defines a retimed weight $w_r : E \rightarrow \mathbb{Z}$ as follows. For each $e = ((u, a), (v, b)) \in E$, $w_r(e) = w(e) + r(v) - r(u)$. □

As a result of retiming, the synchronous and asynchronous timing behaviour of an SFG are changed. We use the index r to identify the synchronous and asynchronous delay of paths in a retimed SFG.

In addition to retiming we also need a concept of phase, which is defined as follows.

Definition 2.15 (phase). Given are a source SFG $\mathcal{G}$, a retiming r and folding factor f. The *phase* is a function $ph : V \rightarrow \mathbb{Z}$ that returns the phase of an operation. It is defined as follows. For each $v \in V$, $ph(v) = r(v) \bmod f$. □

There are three reasons to retime operations, that each correspond with a type of constraint on the retiming. The first reason is that two or more operations of the same source SFG can be executed on one operator if they are executed in different phases according to the following requirement.

(*v*) $ph(u) \neq ph(v)$ for each $u, v \in V$ with $as(u) = as(v)$ and $u \neq v$.

Note that these constraints make Constraints (*iii*) redundant, which state that the number of operations which can be assigned to one operator cannot exceed the number of phases that are available. Initially all operations are executed in the same phase. By changing the execution time of operations we can assign every operation that is assigned to the same operator to a different phase.

The second reason to retime operations is that the execution times of the operations determine the placements of registers in the PU. The minimum throughput frequency at which consecutive executions of an SFG must be performed is specified by a clock period τ. It implies that a path between two registers in the retimed PU must have an asynchronous delay less than the clock period. A path between two registers is characterized by a path weight equal to zero, so we demand that

$$(vi)\ D(p') < \tau \qquad \text{for each } p' \in P_{G'} \text{ with } W_r(p') = 0.$$

By changing the execution times of operations in SFGs, the weights on paths are influenced and consequently the maximum achievable clock frequency can be controlled. Whether a path is executable or not is discussed in the next section, Section 2.4.

The third reason to retime operations is that the initial weights in the SFGs may be negative. Retiming changes the weights on signal edges and must make them all positive, i.e.,

$$(vii)\ w_r(e') \geq 0 \qquad \text{for each } e' \in E'.$$

Another reason to retime operations is that the register cost of a PU can be minimized by doing it. The relationship between the retiming of operations and the cost is explained in Section 2.5.

2.4 False paths and loops in PUs

One of the constraints on the PU designed is that it cannot contain asynchronous feedback loops. These loops can be introduced by the mapping of source SFGs onto a target SFG; see Figure 2.2. However, these loops will never carry data around since the data routing will be according to one of the functions that can be executed, which is free of asynchronous feedback loops. Therefore, an asynchronous feedback loop in the target SFG is false. A false asynchronous feedback loop causes problems for timing verification, since most timing verification tools cannot handle PUs that contain false asynchronous feedback loops.

A path in a target SFG is false if not all its edges correspond to the same source SFG or if the operations of the source SFG that are assigned to the operators of the path are executed in different phases. A false path in a target SFG consists of a sequence of two or more paths, connected by operators with a defined delay, each of which corresponds to a path in one of the source SFGs. A false loop in a PU is a false path with the starting output terminal and the ending input terminal belonging to the same operator and a defined delay between these terminals. A false loop can be identified by a sequence of two or more paths in one or more source SFGs. Such a sequence is called a *complex circuit*.

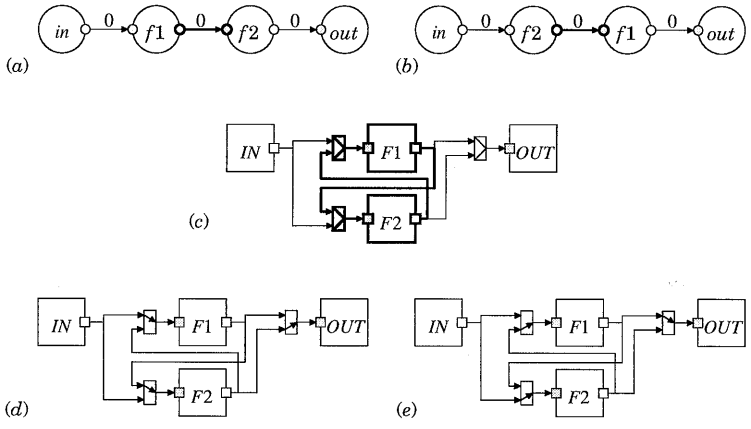


Figure 2.2. Two source SFGs, (*a*) and (*b*), and a PU, (*c*), that can execute one of the SFGs at a time. The PU contains a false loop, which is highlighted. The loop is false since one of the multiplexers in it is switched in such a way that the loop is broken. In (*d*) the selected inputs of the multiplexers are indicated corresponding with the execution of SFG (*a*). In (*e*) the selected inputs of the multiplexers are indicated corresponding with the execution of SFG (*b*). The paths of the SFGs that constitute a complex circuit are highlighted in (*a*) and (*b*).

Definition 2.16 (complex circuit). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i = (V_i, T_i, E_i)$, $i = 1,\ldots,N$, an operator allocation al, and an operator assignment as. A *complex circuit* is a sequence of paths $(p_0, p_1,\ldots, p_{m-1})$, with $p_k = ((u_k, a_k) \mapsto (v_k, b_k)) \in P_{\mathcal{G}}$, $k = 0,\ldots,m-1$, and $m \geq 2$, satisfying

$$as(v_k) = as(u_{(k+1) \bmod m}) \qquad k = 0,\ldots,m-1, \text{ and}$$

$$del((as(v_k), b_k), as(u_{(k+1) \bmod m}), a_{(k+1) \bmod m})) \neq \sim \qquad k = 0,\ldots,m-1, \text{ and}$$

$$u \neq v \qquad \text{for each } (v,a) \in p_k, \text{ and } (u,b) \in p_l \text{ with } k,l = 0,\ldots,m-1,\ k \neq l.$$

We denote the set of all complex circuits in $\mathcal{G}$ by $C_{\mathcal{G}}$. Furthermore, one of the paths p of a complex circuit $c \in C_{\mathcal{G}}$ is considered to be an element of the complex circuit, which is denoted by $p \in c$. □

Although a complex circuit visits an operation at most once, corresponding operators may be visited more than once because different operations may be assigned to the same operator. Note that a false loop may have more than one corresponding complex circuit.

We demand that on each complex circuit at least one register be present, i.e.,

(*viii*) $\sum_{p \in c} W_r(p) \geq 1$ for each $c \in C_{\mathcal{G}}$.

2.5 Mapping a target SFG onto a PU

The second step of the mapping of a union of source SFGs onto a PU consists of implementing the target SFG, which was designed in the first step, as a PU. In the second step the number of registers and the number of multiplexers are determined. The register allocation is defined as follows.

Definition 2.17 (register allocation). Given is an SFG $\mathcal{G}$. The *register allocation* is a function $ra : T \to \mathbb{N}$ that returns for each terminal in T the number of registers allocated to the terminal. □

According to the architecture of a PU, registers on different edges may be shared at operator inputs and operator outputs if edges have common terminals. The clocked delay of a signal edge must be smaller than or equal to the number of registers allocated to the terminals to which it is connected, i.e.,

(*ix*) $ra(o) + ra(i) \geq w_r(e)$ for each $e = (o,i) \in E$.

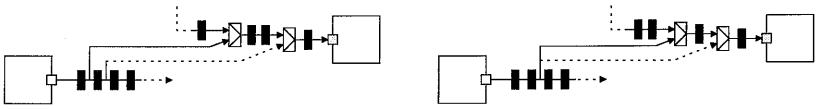
An operator output can be connected to more than one operator input by a tapped delay line without the need for additional hardware; see Figure 1.11. If out of more than two sources must be selected at an operator input, a tree of two-input multiplexers is placed at the input. The number of two-input multiplexers placed at an operator input is determined by a multiplexer allocation.

Definition 2.18 (multiplexer allocation). Given are an SFG $\mathcal{G}$ and a weight w. The *multiplexer allocation* is a function $ma : I \rightarrow \mathbb{N}$ that returns for each operator input in I the number of two-input multiplexers located at the operator input, which is defined as follows. For each $i \in I$, $ma(i) = (|\{(o, w_r((o,i))) \in O \times \mathbb{N} | (o,i) \in E\}| - 1)^+$, where $x^+ = \max\{x, 0\}$. □

The number of multiplexers can be reduced if two or more edges have identical sources and destinations and if they have the same weight. An operator assignment determines the sources and destinations. Retiming may affect the weight on an edge and, consequently, the multiplexer cost.

Having defined functions to indicate the number of operators, registers, and multiplexers, we next discuss the way they are connected. All the elements of a PU, i.e., the operators, registers, and multiplexers, are interconnected by point-to-point connections, also called wires. The number of wires is equal to the number of inputs of all the elements. Therefore we include the cost of wires by adding to each element an extra cost per input.

If a signal on an edge must be delayed a number of clock cycles that is smaller than the sum of the registers that are allocated to the output terminal and input terminal to which the edge is connected, more than one possibility exists to configure the multiplexers and registers. Since all the possible configurations have the same cost, we do not discuss this any further; see Figure 2.3.



(*a*) (*b*)

Figure 2.3. Two configurations of a (part of a) PU, which have the same cost. At the output terminal four registers are allocated. At the input terminal four registers and two multiplexers are allocated. The connection indicated by a solid line between the input and output terminals corresponds with a signal edge with a weight equal to four.

Given an SFG $\mathcal{G}$, a type set $\mathcal{T}$, an operator allocation al, a register allocation ra, and a multiplexer allocation ma, the cost of the SFG is

$$\alpha \sum_{t \in T} ra(t) + \beta \sum_{i \in I} ma(i) + \sum_{j \in \mathcal{T}} al(j)\mathcal{C}(j),$$

where $\alpha \in \mathbb{Q}^+$ denotes the cost of a register and $\beta \in \mathbb{Q}^+$ the cost of a multiplexer.

The control signals to the multiplexers are defined by the mapping of source SFGs onto a PU and are not discussed any further.

2.6 Processing unit design problem

The central problem of this thesis is formulated as follows.

Definition 2.19 (processing unit design problem (PUDP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$ with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, timing functions dat, drt, del, and w, a clock period τ, the cost of a register α, and the cost of a multiplexer β. The problem is to find a PU $\mathcal{G}'$ constructed by an operator allocation al, an operator assignment as, a retiming r, a register allocation ra, and a multiplexer allocation ma, for which

(i)	$al(j) \geq \max_{i=1,\ldots,N} \lceil \frac{\lvert V_{i,j} \rvert}{f_i} \rceil$	for each $j \in \mathcal{T}$, and
(ii)	$\ell(v) = \ell(as(v))$	for each $v \in V$, and
(iii)	$\lvert \{v \in V_i \mid as(v) = v'\} \rvert \leq f_i$	for each $v' \in V'$, and for each $i = 1,\ldots,N$, and
(iv)	$e' = ((as(v),a)(as(u),b)) \in E'$ and $w_r(e') = w_r(e)$	for each $e = ((v,a),(u,b)) \in E$, and
(v)	$ph(u) \neq ph(v)$	for each $u, v \in V_i$ with $as(u) = as(v)$, and $u \neq v$, and for each $i = 1,\ldots,N$, and
(vi)	$D(p') < \tau$	for each $p' \in P_{\mathcal{G}'}$ with $W_r(p') = 0$, and
(vii)	$w_r(e') \geq 0$	for each $e' \in E'$, and
(viii)	$\sum_{p \in c} W_r(p) \geq 1$	for each $c \in C_{\mathcal{G}}$, and
(ix)	$ra(o') + ra(i') \geq w_r(e')$	for each $e' = (o', i') \in E'$, and

that minimizes

$$\alpha \sum_{t' \in T'} ra(t') + \beta \sum_{i' \in I'} ma(i') + \sum_{j \in \mathcal{T}} al(j)C(j).$$

□

2.7 Special cases of PUDP

In the introduction of this thesis we informally introduced PUDP as a generalization of the multiplexing, timefolding, and retiming problems. Here, we formally define these problems as special cases of PUDP. A special case of PUDP is defined by additional constraints that must hold for instances of PUDP.

2.7.1 Multiplexing problem

The first special case of PUDP is called the multiplexing problem and it occurs when the following restrictions on the instances hold. Each source SFG has a folding factor of one, the required clock period is infinity and the weights are at least zero, i.e.,

$$f_i = 1 \qquad \text{for each } i = 1,\ldots,N, \text{ and}$$
$$\tau = \infty \qquad \text{and}$$
$$w(e) \geq 0 \qquad \text{for each } e \in E.$$

As a consequence of the restrictions on the instances, some of the constraints of PUDP become irrelevant. Since the folding factor is one, Constraints (*v*) are implied by Constraints (*iii*). Constraints (*vi*) are true for every instance that is restricted as described above. Furthermore, the non-negative weights make Constraints (*vii*) true. We assume that no retiming has to be determined.

The multiplexing problem is formulated as follows.

Definition 2.20 (multiplexing problem (MP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, timing functions *del* and *w*, the cost of a register α, and the cost of a multiplexer β. The problem is to find a PU $\mathcal{G}'$ constructed by an operator allocation *al*, an operator assignment *as*, a register allocation *ra*, and a multiplexer allocation *ma*, for which

(*i*) $al(j) \geq \max_{i=1,\ldots,N} |V_{i,j}|$ for each $j \in \mathcal{T}$, and

(*ii*) $\ell(v) = \ell(as(v))$ for each $v \in V$, and

(*iii*) $|\{v \in V_i | as(v) = v'\}| \leq 1$ for each $v' \in V'$, and for each $i = 1,\ldots,N$, and

(*iv*) $e' = ((as(v),a)(as(u),b)) \in E'$ and $w(e') = w(e)$ for each $e = ((v,a),(u,b)) \in E$, and

(*viii*) $\sum_{p \in c} W(p) \geq 1$ for each $c \in C_{\mathcal{G}}$, and

(*ix*) $ra(o') + ra(i') \geq w(e')$ for each $e' = (o',i') \in E'$, and

that minimizes

$$\alpha \sum_{t' \in T'} ra(t') + \beta \sum_{i' \in I'} ma(i') + \sum_{j \in \mathcal{T}} al(j)\mathcal{C}(j).$$

□

2.7.2 Timefolding problem

The second special case of PUDP is called the timefolding problem and involves the following restrictions on the instances. There is only one source SFG and the required clock period is infinity, i.e.,

$N = 1$, and

$\tau = \infty$.

As a consequence of the restrictions on the instances, some of the constraints of PUDP become irrelevant. Constraints (*iii*) are implied by Constraints (*v*). Constraints (*vi*) are true for any instance restricted as described above.

The timefolding problem is formulated as follows.

Definition 2.21 (timefolding problem (TP)). Given are a source SFG $\mathcal{G}$, a type set $\mathcal{T}$, a type function ℓ, a folding factor f, timing function w, the cost of a register α, and the cost of a multiplexer β. The problem is to find a PU $\mathcal{G}'$ constructed by an operator allocation *al*, an operator assignment *as*, a retiming r, a register allocation *ra*, and a multiplexer allocation *ma*, for which

(*i*) $al(j) \geq \lceil \frac{\lvert V_j \rvert}{f} \rceil$	for each $j \in \mathcal{T}$, and
(*ii*) $\ell(v) = \ell(as(v))$	for each $v \in V$, and
(*iv*) $e' = ((as(v), a)(as(u), b)) \in E'$ and $w_r(e') = w_r(e)$	for each $e = ((v, a), (u, b)) \in E$, and
(*v*) $ph(u) \neq ph(v)$	for each $u, v \in V$ with $as(u) = as(v)$, and $u \neq v$, and
(*vii*) $w_r(e') \geq 0$	for each $e' \in E'$, and
(*viii*) $\sum_{p \in c} W_r(p) \geq 1$	for each $c \in C_{\mathcal{G}}$, and
(*ix*) $ra(o') + ra(i') \geq w_r(e')$	for each $e' = (o', i') \in E'$, and

that minimizes

$$\alpha \sum_{t' \in T'} ra(t') + \beta \sum_{i' \in I'} ma(i') + \sum_{j \in \mathcal{T}} al(j)\mathcal{C}(j).$$

□

2.7.3 Retiming problem

The third special case of PUDP is called the retiming problem and it occurs when the following restrictions on the instances hold. There is only one source SFG and it has a folding factor equal to one, i.e.,

$N = 1$, and

$f = 1$.

As a consequence of the restrictions on the instances, some of the constraints of PUDP become irrelevant. Since the instances contain only one source SFG and no timefolding is applied, no complex circuits are present. Consequently, Constraints (*viii*) become irrelevant. Since the folding factor is one, Constraints (*v*) are

implied by Constraints (*iii*). Constraints (*viii*) are true for any instance restricted as described above.

Another consequence of the restrictions on the instances is that trivial solutions can be found for the operator allocation, operator assignment, and multiplexer allocation. For each operation in the source SFG, an operator in the PU is allocated to which it is assigned. These trivial solutions make Constraints (*i*), (*ii*), and (*iii*) superfluous. Furthermore, since the target and source SFG are identical Constraint (*iv*) can be omitted.

The retiming problem is formulated as follows.

Definition 2.22 (retiming problem). Given are a source SFG $\mathcal{G}$, timing functions dat, drt, del, and w, and a clock period τ. Then the *Retiming Problem* (*RP*) is to find a retiming r, and a register allocation ra, for which

(*vi*)	$D(p) < \tau$	for each $p \in P_{\mathcal{G}}$ with $W_r(p) = 0$, and
(*vii*)	$w_r(e) \geq 0$	for each $e \in E$, and
(*ix*)	$ra(o) + ra(i) \geq w_r(e)$	for each $e = (o, i) \in E$, and

that minimizes

$$\sum_{t \in T} ra(t).$$

□

RP was first formulated by Leiserson, Rose & Saxe [1983], although in a slightly different way. They did not model the register cost explicitly by a function like *ra*; instead they artificially added operations and edges such that they could use the sum of the edges' weights as the cost. Furthermore, they only allowed register sharing at operator outputs since they did not consider multi-functional or multicycle PUs in which more than one edge can be incident to an input terminal.

3

Complexity Analysis and Decomposition

In this chapter we discuss the computational complexity of the processing unit design problem. We first show in Section 3.1 that PUDP is NP-hard and, consequently, we must abandon the idea to develop an algorithm that solves all instances of PUDP to optimality in polynomial time. We chose to use a domain decomposition method such that PUDP is replaced by a number of (sub)problems, which must be solved in a particular order. The decomposition is discussed in Section 3.2 and it is followed by a discussion on the complexity of PUDP's subproblems and some of their special cases in Section 3.3. An overview of the complexity of PUDP and its subproblems is given in Section 3.4. Some of the problems that are presented in this chapter and a discussion on their complexity have been reported earlier by De Fluiter, Aarts, Korst, Verhaegh & Van der Werf [1996].

3.1 Complexity of the processing unit design problem

The processing unit design problem can be formulated as a *combinatorial optimization problem* [Papadimitriou & Steiglitz, 1982]. We can make use of the theory of NP-completeness [Garey & Johnson, 1979] to analyze the complexity of the problem. We consider two possibilities in the complexity analysis of a combinatorial optimization problem. The first possibility is to reduce it to a known optimization problem that can be solved in polynomial time. The second possibility is to re-

duce a known *NP-complete* problem to the *decision variant* of the combinatorial optimization problem at hand. The latter we apply here to prove that PUDP is intractable. First we define the corresponding decision problem of PUDP, which is derived from Definition 2.19 by replacing the optimization goal by a constraint.

Definition 3.1 (processing unit design problem - decision variant (PUDP-D)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$ with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, timing functions dat, drt, del, and w, a clock period τ, the cost of a register α, the cost of a multiplexer β, and a constant $K \in \mathbb{Q}$. The problem is to find a PU $\mathcal{G}'$ constructed by an operator allocation al, an operator assignment as, a retiming r, a register allocation ra, and a multiplexer allocation ma, for which

(i)	$al(j) \geq \max_{i=1,\ldots,N}\lceil \frac{\lvert V_{i,j}\rvert}{f_i} \rceil$	for each $j \in \mathcal{T}$, and
(ii)	$\ell(v) = \ell(as(v))$	for each $v \in V$, and
(iii)	$\lvert\{v \in V_i \lvert as(v) = v'\}\rvert \leq f_i$	for each $v' \in V'$, and for each $i = 1,\ldots,N$, and
(iv)	$e' = ((as(v),a),(as(u),b)) \in E'$ and $w_r(e') = w_r(e)$	for each $e = ((v,a),(u,b)) \in E$, and
(v)	$ph(u) \neq ph(v)$	for each $u, v \in V_i$ with $as(u) = as(v)$, and $u \neq v$, and for each $i = 1,\ldots,N$, and
(vi)	$D(p') < c$	for each $p' \in P_{\mathcal{G}'}$ with $W_r(p') = 0$, and
(vii)	$w_r(e') \geq 0$	for each $e' \in E'$, and
(viii)	$\sum_{p \in c} W_r(p) \geq 1$	for each $c \in C_{\mathcal{G}}$, and
(ix)	$ra(o') + ra(i') \geq w_r(e')$	for each $e' = (o', i') \in E'$, and

$$\alpha \sum_{t' \in T'} ra(t') + \beta \sum_{i' \in I'} ma(i') + \sum_{j \in \mathcal{T}} al(j)\mathcal{C}(j) \leq K.$$

□

Next, we state that the problem is NP-complete and prove this.

Theorem 3.1. PUDP-D is NP-complete.

Proof. PUDP-D is in NP since it can be checked in polynomial time that a given solution is feasible. Most of the constraints are checked in a trivial way, except for Constraints *(vi)* and *(viii)*. The number of paths and the number of complex circuits can increase exponentially with the size of the problem. However, in order to check whether the constraints related to them are satisfied, we do not need to

know all paths and all complex circuits. We can check whether the delay of a path between any pair of registers is shorter than the clock period by using a longest path algorithm that runs in polynomial time. We can check whether there exists a complex circuit with zero weight by applying a depth-first search for an operation that is already labeled over edges with zero weight starting from every operation, which runs in polynomial time. If we do not find a zero weight cycle we know that the constraints related to complex circuits are satisfied. Finally, it can be checked in polynomial time that the cost is at most K.

We give a reduction from the Directed Hamiltonian Circuit (DHC) problem, which is NP-complete [Garey & Johnson, 1979], to PUDP-D. There the problem is to find a circuit through a directed graph visiting all vertices exactly once. Let $G^* = (V^*, A^*)$ be an instance of DHC, with $|V^*| = n$ and $|A^*| = m$. Suppose $m \geq n$. This is not a restriction, since otherwise G^* cannot contain a Hamiltonian circuit. From this, we construct two source SFGs $\mathcal{G}_1$ and $\mathcal{G}_2$ and we show that they can be mapped onto a PU $\mathcal{G}'$ with a cost that is at most K if and only if G^* contains a Hamiltonian circuit.

From the DHC instance we derive an instance of PUDP-D in the following way; see Figure 3.1. We define $\mathcal{T} = \{1,2\}$, with $\mathcal{I}(1) = 0$, $\mathcal{O}(1) = 1$, $\mathcal{I}(2) = 2$, and $\mathcal{O}(2) = 0$. We construct $\mathcal{G}_1$ with for each $v \in V^*$, an operation v with $\ell(v) = 1$, and for each $a \in A^*$, an operation a with $\ell(a) = 2$. For each edge $a = (u, v) \in A^*$ we construct two edges in E_1: $((u,1),(a,1))$ and $((v,1),(a,2))$.

We construct $\mathcal{G}_2$ with V_2 consisting of two parts. The first part consists of n operations of type 1, numbered $r_1, r_2, \ldots, r_n$. The second part consists of n operations of type 2, numbered $s_1, s_2, \ldots, s_n$. E_2 contains the edge $((r_i,1),(s_i,1))$ for each $i = 1, \ldots, n$. Furthermore, E_2 contains $((r_1,1),(s_n,2))$ and $((r_{i+1},1),(s_i,2))$ for each $i = 1, \ldots, n-1$. $\mathcal{G}_2$ is the translation of an arbitrary directed cycle on n vertices.

The two source SFGs are the main part of the problem instance. Furthermore, we define the following:

$f_1 = 1$ and $f_2 = 1$,
$\mathcal{C}(1) = 1$ and $\mathcal{C}(2) = 1$,
$\alpha = 0$ and $\beta = 1$,
$w(e) = 0$ for each $e \in E$,
$del((v,a),(v,b)) = \frac{1}{3}$ for each $((v,a),(v,b)) \in I \times O$,
$\tau = 1$, and
$K = |V^*| + |A^*|$.

The complete transformation can be done in polynomial time.

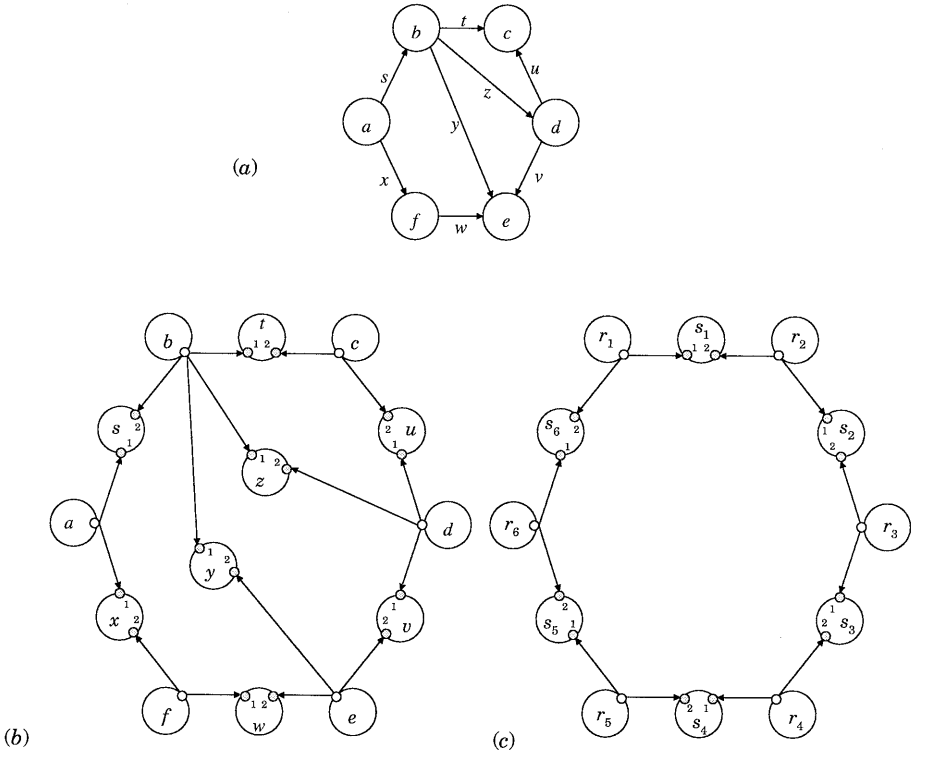


Figure 3.1. An example of a DHC graph (a) and the two source SFGs $\mathcal{G}_1$ (b) and $\mathcal{G}_2$ (c) that are derived from it.

We make the following remarks about the constructed instance of PUDP-D. A PU that will cost (at most) $|V^*| + |A^*|$ has no registers, no multiplexers, and a minimal operator allocation, i.e., for each $v \in V^*$, an operator v' of type 1 is allocated, and for each $a \in A^*$, an operator a' of type 2 is allocated. The solution has a trivial retiming and register allocation, i.e., $r(v) = 0$ for each $v \in V$ and $ra(t) = 0$ for each $t \in T$. Since an operator has only one output or two inputs, complex circuits cannot exist. All the delays of paths in the PU are smaller than the clock period. The operator assignment must be such that no multiplexers exist in the PU.

We now prove that G^* contains a Hamiltonian circuit if and only if there exists a feasible operator assignment that assigns the operations of $\mathcal{G}_1$ and $\mathcal{G}_2$ to operators of $\mathcal{G}'$ such that the number of multiplexers needed is zero.

Suppose G^* contains a Hamiltonian circuit $(v_1,...,v_n)$. Then there are edges $a_1,...,a_n \in A^*$ such that for each $i = 1,\ldots,n-1$, $a_i = (v_i, v_{i+1})$, and $a_n = (v_n, v_1)$. Then we can assign the operations of SFG $\mathcal{G}_2$ without introducing any multiplexers as follows. For each $i = 1,\ldots,n$, $as(r_i) = as(v_i)$, $as(s_i) = as(a_i)$.

Suppose as is a feasible operator assignment that does not introduce any multiplexers. Number the vertices in $V^* = \{v_1,\ldots,v_n\}$ and the edges in $A^* = \{a_1,\ldots,a_m\}$, such that $as(r_i) = as(v_i)$ and $as(s_i) = as(a_i)$ for each $i = 1,\ldots,n$. Since no multiplexers are present and $((r_i,1),(s_i,1)) \in E_2$ and $((r_{i+1},1),(s_i,2)) \in E_2$ it follows that $((v_i,1),(a_i,1)) \in E_1$ and $((v_{i+1},1),(a_i,2)) \in E_1$ for all $i = 1,\ldots,n-1$. Furthermore, since no multiplexers are present and $((r_n,1),(s_n,1)) \in E_2$ and $((r_1,1),(s_n,2)) \in E_2$ it follows that $((v_n,1),(a_n,1)) \in E_1$ and $((v_1,1),(a_n,2)) \in E_1$. Then $(v_1,...,v_n)$ is a Hamiltonian circuit of G^*, since for each $i = 1,\ldots,n-1$, $((v_i,1),(a_i,1)) \in E_1$, and $((v_{i+1},1),(a_i,2)) \in E_1$, so $(v_i, v_{i+1}) \in A^*$. Furthermore, $((v_n,1),(a_n,1)) \in E_1$ and $((v_1,1),(a_n,2)) \in E_1$, so $(v_n, v_1) \in A^*$. □

3.2 Decomposition of the processing unit design problem

Since it is difficult to find values for the decision variables in one step, we concentrate on a decomposition method that treats them one by one. The solution method uses a decomposition strategy consisting of four main steps, namely operator allocation, operator assignment, operator duplication, and retiming. In Section 3.2.1 we discuss the order in which the subproblems are solved. In Section 3.2.2 we formally define the subproblems.

3.2.1 Subproblem order

The ordering of the subproblems is motivated as follows.

Step 1: operator allocation

We start with determining an operator allocation since the other decision variables can best be determined when an operator allocation is known. The determination of an operator assignment has to follow the determination of an operator allocation since operators must exist before operations are assigned to them. The determination of a retiming also has to follow, since in the case of timefolding we need to know how many operators are allocated in order to know how many operations we can execute in the same phase. Otherwise, we may have retimed many operations in the same phase and, consequently, require many operators; this, in general, cannot lead to an optimal solution, since we expect that an optimal PU contains more operators than registers.

Step 2: operator assignment

Once an operator allocation is given, we determine an operator assignment before a retiming. The main reason for this in the case of timefolding is that the structure of a source SFG may contain regularity, i.e., a substructure of it may be equal to another substructure of it; see Figure 3.2*a*, 3.2*b*, and 3.2*c*. In that case, the operations of the substructures may be assigned to the operators in a PU such that the same substructure is created in it. This operator assignment is then part of a problem instance in which a retiming must be determined. There, the assignment very much restricts the way in which phases can be assigned to operations according to Constraints (*v*). The same substructures may exhibit the same timing behaviour, and if they do, the operations in them can be retimed such that Constraints (*vi*) and (*vii*) are satisfied and such that on corresponding edges the same weight is present. The latter results in a minimal number of multiplexers since at an input, all signals available for selection must be delayed the same number of clock cycles. To satisfy Constraints (*v*), all operations in a substructure can be easily retimed one more clock cycle, if necessary, without changing the retimed weights on the edges in the substructure. Consequently, the phase conflicts can be solved at no additional multiplexer and register cost in the substructure of the PU.

In the case of multiplexing, another appearance of regularity is the existence of common substructures in different source SFGs; see Figure 3.2*d*, 3.2*e*, and 3.2*f*. When corresponding operations in common substructures of different source SFGs are assigned to the same operator, that substructure becomes present in the PU. Equally retiming the corresponding operations results in the minimum number of multiplexers and registers in the PU.

Step 3: operator duplication

Based on the first step, we use a minimal allocation of operators to determine an operator assignment in the second step. However, after an initial operator assign-

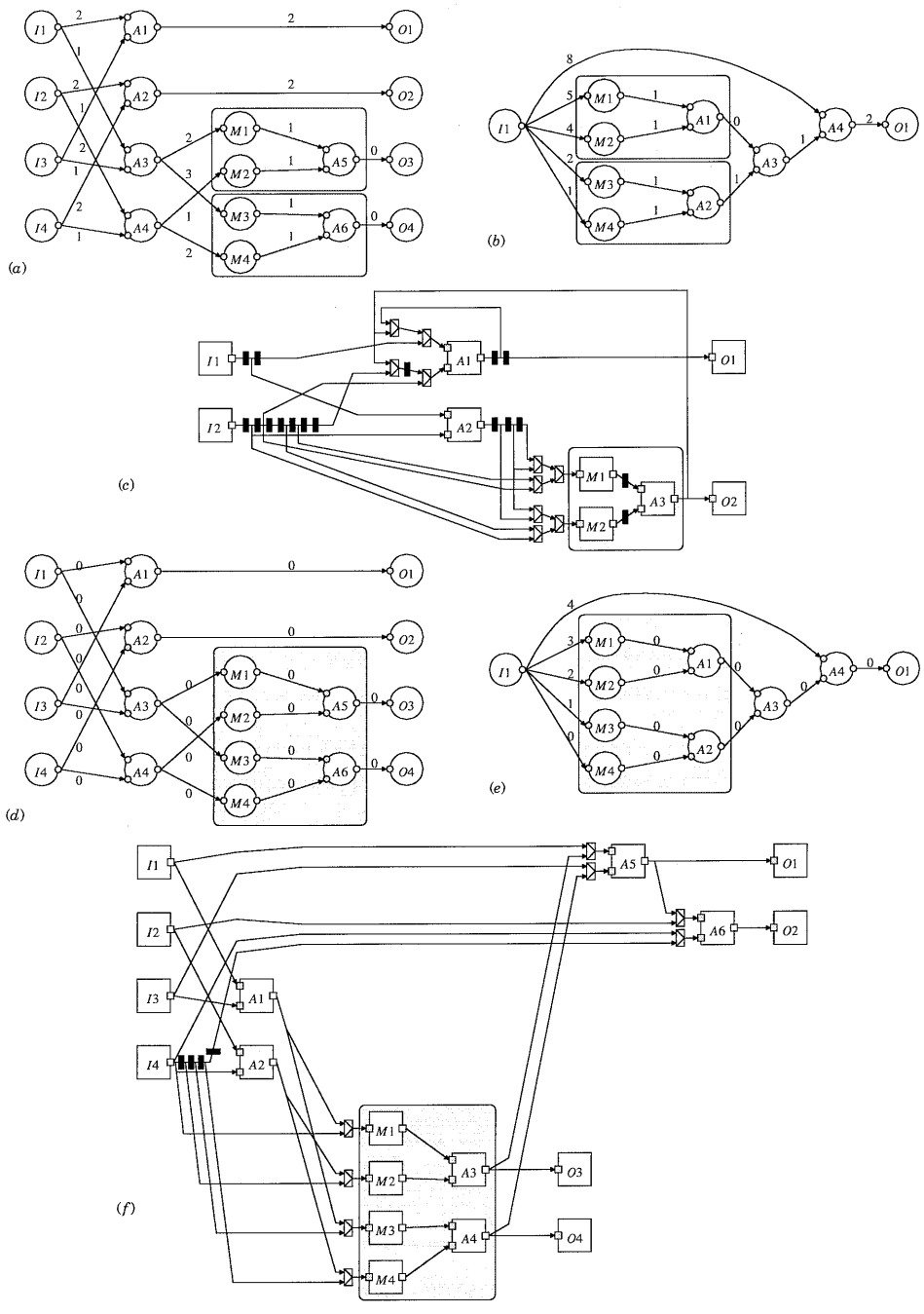


Figure 3.2. The two source SFGs of the butterfly (*a*), (*d*) and the 5-tap filter (*b*), (*e*) and their PU implementations in the case of multiplexing (*f*) and in the case of a combination of timefolding and multiplexing (*f*). A common substructure is indicated by the grey boxes.

ment has been determined, the result may not be satisfactory because the PU may be too expensive and may contain complex circuits. The PU may be too expensive because one or more multiplexers placed at the input(s) of an operator can be quite expensive with respect to the cost of the operator itself. Although the goal of the operator assignment step is to minimize the connectivity cost, it does not make a trade-off with operator cost. Complex circuits may be inevitably created by an operator assignment if a limited number of operators is allocated.

Within the decomposition strategy three methods fit to handle the problem of complex circuits; see Figure 3.3. The first method reassigns operations to a minimum number of operators. The second method allocates additional operators onto which some operations are reassigned. The third method generates constraints for the step following assignment, in which operations are retimed such that at least one register is placed on each false loop. Only the second method is guaranteed to be effective. Consequently, we introduce a step in our decomposition in which an adjustment of the initial operator allocation and the initial operator assignment is determined. We reassign operations to operators in such a way that two operations can only be (re)assigned to the same operator if they were initially assigned to that operator. Furthermore, no complex circuits may be present and the cost is minimized. We refer to this technique as *operator duplication.*

Step 4: retiming

The only decision variable that is left to be determined in this step is retiming. Suppose that the order in which an operator assignment and a retiming are determined is exchanged. Then, it becomes very difficult to determine an operator assignment that complies with Constraints (v) and still has a low overall cost, since the cost of the retiming is difficult to determine or even to estimate accurately when no operator assignment is given.

3.2.2 Formal subproblem definitions

In this section the definitions of the subproblems into which PUDP is decomposed are given.

Operator allocation problem

We exclude the contribution of registers and multiplexers to the cost since an operator assignment and a retiming are unknown. Furthermore, we do not require timing functions in the instances since no retiming needs to be determined. All constraints except for Constraints (i) are related to an operator assignment or a retiming. These constraints can be ignored without imposing problems for following steps in the decomposition. Then, the *operator allocation problem* is defined as follows.

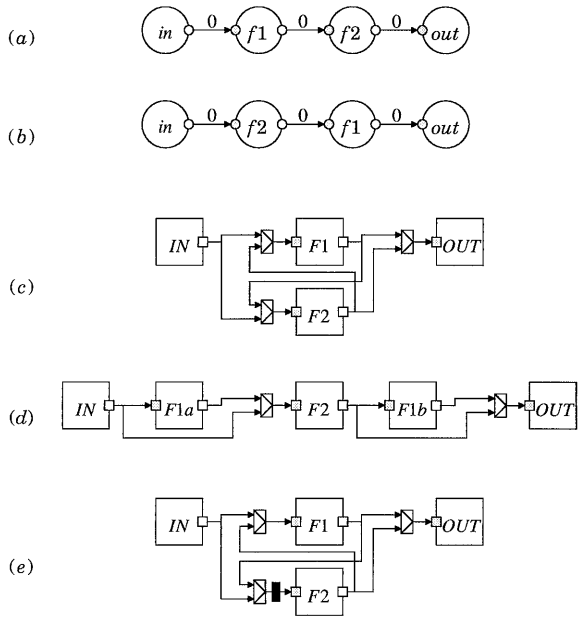


Figure 3.3. Two source SFGs (*a*), (*b*), and three PUs that can execute each of the source SFGs, one at a time. The first PU (*c*) has a minimal number of allocated operators and is not retimed, but contains a false loop. The second PU (*d*) has a non-minimal number of allocated operators, but no false loops. The third PU (*e*) is based on the same allocation and assignment as the PU of (*c*), but is retimed to prevent the existence of a false loop. Reassignment of operations cannot remove the false loop from the PU of (*c*). Note that the PU of (*d*) is cheaper than the PU of (*c*) if a multiplexer costs more than operator $F1a$.

Definition 3.2 (operator allocation problem (OALP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, and a type function ℓ. The problem is to find an operator allocation al, for which

(*i*) $al(j) \geq \max_{i=1,\ldots,N}\lceil \frac{|V_{i,j}|}{f_i} \rceil$ for each $j \in \mathcal{T}$, and

that minimizes

$$\sum_{j\in\mathcal{T}} al(j)C(j).$$

□

Operator assignment problem

As a result of the first step we have a minimal number of operators of each type. Therefore, Constraints (*i*) are irrelevant here. Retiming operations in a following step may change the weights on edges in the SFGs and, if they do, the original weights cannot be used to determine the number of multiplexers. Therefore, we ignore the weight in the instances and we assume zero weights on all edges in the SFGs. Furthermore, we do not require the asynchronous timing functions in the instances since no retiming needs to be determined. Consequently, Constraints (*v*), (*vi*), and (*vii*) are irrelevant for this subproblem. Constraints (*viii*) are taken into account in the next step, and can therefore be omitted here. Furthermore, because the weights are unknown, Constraints (*ix*) become irrelevant. We exclude the contribution of registers and operators to the cost to be minimized. Because we assume zero weights on the edges, the Constraints (*iv*) are reformulated as Constraints (*xii*) in the following definition of the *operator assignment problem.*

Definition 3.3 (operator assignment problem (OASP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, and a minimal operator allocation al. The problem is to find a PU $\mathcal{G}'$ constructed by an operator assignment as, and a multiplexer allocation ma, for which

(*ii*) $\ell(v) = \ell(as(v))$ for each $v \in V$, and

(*iii*) $|\{v \in V_i | as(v) = v'\}| \leq f_i$ for each $v' \in V'$, and for each $i = 1,\ldots,N$, and

(*xii*) $((as(v),a),(as(u),b)) \in E'$ for each $((v,a),(u,b)) \in E$, and

that minimizes

$$\sum_{i'\in I'} ma(i').$$

□

Operator duplication problem

Solutions of OALP and OASP determine the instances of the operator duplication problem. We demand that at least the same number of operators must be allocated of each operation type as in the initial allocation, thus satisfying Constraints (*i*). Moreover, we demand that two operations can be assigned to the same operator only when they are assigned to the same operator in the initial assignment (Constraints (*x*) in Definition 3.2). The timing functions do not need to be part of instances of the operator duplication problem, for the same reasons as is the case with instances of the operator assignment problem. However, we need the delay of operators since complex circuits can only pass through operations with a defined delay. Because the weights are unknown, Constraints (*viii*) imply that no complex circuits may exist, and, consequently, these constraints are reformulated as Constraints (*xi*) in Definition 3.4. We exclude the contribution of registers to the cost. Then, the operator duplication problem is defined as follows.

Definition 3.4 (operator duplication problem (ODP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, a delay *del*, the cost of a multiplexer β, and an initial operator allocation al' and an initial operator assignment as' defining a PU $\mathcal{G}'$. The problem is to find a PU $\mathcal{G}^*$ constructed by an operator allocation al^*, an operator assignment as^*, and a multiplexer allocation ma^*, for which

(*ii*)	$\ell(v) = \ell(as^*(v))$	for each $v \in V$, and
(*x*)	$as^*(v) \neq as^*(u)$	for each $u, v \in V$ with $as'(v) \neq as'(u)$, and
(*xi*)	$C_{\mathcal{G}} = \emptyset$,	and
(*xii*)	$((as^*(v), a), (as^*(u), b)) \in E^*$	for each $((v, a), (u, b)) \in E$, and

that minimizes

$$\beta \sum_{i^* \in I^*} ma^*(i^*) + \sum_{j \in \mathcal{T}} al^*(j)\, C(j).$$

□

Generalized retiming problem

Since the determination of a retiming follows the determination of the assignment, we assume that an operator allocation and an operator assignment are given. Constraints (*i*), (*ii*), (*iii*), and (*viii*) are already satisfied and, therefore, not explicitly taken into account here. Furthermore, the contribution of the operators is excluded from the cost, because it is constant. Since the problem to determine a retiming is more general than RP of Definition 2.22, which is a special case of PUDP, we call it the *generalized retiming problem* and define it as follows.

Definition 3.5 (generalized retiming problem (GRP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, a clock period τ, timing functions dat, drt, del, and w, the cost of a register α, the cost of a multiplexer β, an operator allocation al, and an operator assignment as. The problem is to find a PU $\mathcal{G}'$ constructed by a retiming r, a register allocation ra, a multiplexer allocation ma, and the given al and as, for which

(iv)	$e' = ((as(v),a),(as(u),b)) \in E'$ and $w_r(e') = w_r(e)$	for each $e = ((v,a),(u,b)) \in E$, and
(v)	$ph(u) \neq ph(v)$	for each $u, v \in V_i$ with $as(u) = as(v)$, and $u \neq v$, and for each $i = 1,\ldots,N$, and
(vi)	$D(p') < \tau$	for each $p' \in P_{\mathcal{G}'}$ with $W_r(p') = 0$, and
(vii)	$w_r(e') \geq 0$	for each $e' \in E'$, and
(ix)	$ra(o') + ra(i') \geq w_r(e')$	for each $e' = (o', i') \in E'$, and

that minimizes

$$\alpha \sum_{t \in T} ra(t) + \beta \sum_{i \in I} ma(i).$$

□

We can further decompose this subproblem into two subproblems, being the *timing analysis problem* and the *generalized synchronous retiming problem*. The first problem is to transform the constraints involving the asynchronous timing of an SFG into a set of constraints only involving synchronous timing. The second problem is to determined a retiming based on the synchronous timing constraints.

Constraints on the retiming decision variables can be expressed in the same form, i.e., the difference of two retimings of operations must be at most a certain constant. With F we denote the set of constraints of this form. The constraint set F can be represented in a graph. To this end, we extend the definition of an SFG with a constraint set F in the following way and call it an *Extended SFG*.

Definition 3.6 (extended signal flow graph). An *Extended Signal Flow Graph* $\mathcal{EG}$ is given by a four tuple (V,F,T,E) in which

(V,T,E)	is an SFG, and
$F \subseteq V \times V$	is a multi-set of *constraint edges* representing timing constraints.

□

We define a weight on a constraint edge to quantify the constraint as follows.

Definition 3.7 (weight on constraint edges). Given is an ESFG $\mathcal{EG}$. The weight $w : F \to \mathbb{Z}$ is defined as follows. For each edge $f = (u, v) \in F$, $w(f)$ is the number of clock cycles that v may be retimed earlier than u. Consequently the constraint is $w_r(f) \geq 0$. □

The Constraints (*vi*) and (*vii*) can be replaced by constraints on the edges in a set F with $w_r(f) \geq 0$ for each $f \in F$ in the following way.

Constraints (*vii*) state that a signal edge $e = ((u, a), (v, b)) \in E$ imposes a time order on the execution of the operations connected to it, i.e., the signal can only be delayed a non-negative number of clock cycles, $w_r(e) \geq 0$. Such a constraint can be represented by an $f = (u, v) \in F$ with $w(f) = w(e)$. We call constraints of this type *causality constraints*.

Constraints (*vi*) and (*iv*) state that a path between two registers in a retimed PU $\mathcal{G}$ must have a delay of less than the clock period. In other words, a path that is longer than or equal to the clock period must contain at least one register. For paths $p = ((u, a), ..., (v, b))$ that must have a constraint stating that $W_r(p) \geq 1$, a constraint edge $f = (u, v)$ with a weight $w(f) = W(p) - 1$ can represent this. We call constraints of this type *speed constraints*. An example of an ESFG with speed constraints is shown in Figure 3.4.

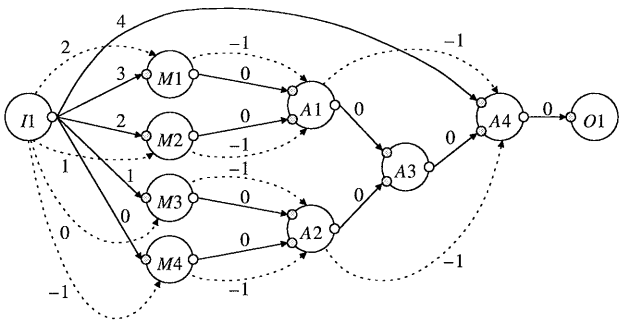


Figure 3.4. An example of an extended signal flow graph. In this case, the constraint edges shown (dashed) correspond to speed constraints only. We assume that on a path at most one multiplication or two additions can be performed in one clock cycle and that the delays of the input and output operators can be neglected.

In addition to retiming operations in the source SFGs, the retiming of operators at the target level has to be considered, because the mapping of more than one source SFG onto a target SFG can introduce many false paths. The start and end operators of such a path do not necessarily correspond with operations of the same

source SFG. Consequently, if such a path has a delay greater than the clock period, no corresponding constraint can exist in any of the source ESFGs; see Figure 3.5*a* and 3.5*b*.

That retiming of the target SFG is not sufficient can be explained as follows. A loop in the target SFG can only correspond with a path in a source SFG, since no complex circuits may be present. The number of registers on the loop cannot be changed by retiming at the target level. However, a speed constraint at the source level can introduce sufficient registers in the loop of the target SFG; see Figure 3.5*c* and 3.5*d*.

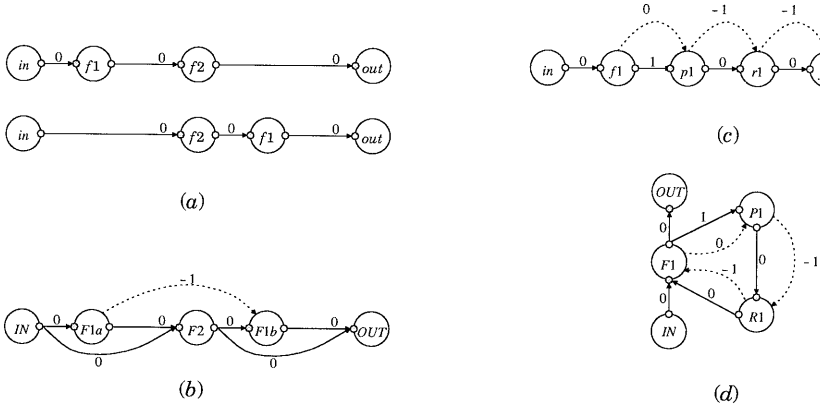


Figure 3.5. Two examples of speed constraint edges in source vs. target SFGs. The source ESFGs of (*a*) are mapped onto the target ESFG of (*b*). The constraint edge in the target ESFG of (*b*) corresponds to a false path. The source ESFG of (*c*) is mapped onto the target ESFG of (*d*) in a timefolded fashion with a folding factor equal to two. The constraint edges in the source ESFG of (*c*) can be satisfied, since the ESFG can be pipelined. However, the corresponding constraint edges in the target ESFG cannot be satisfied, since they correspond to a cycle on which the number of registers cannot be changed by retiming at the target level.

Using source and target level retiming, we can find a solution for an instance of GRP in two steps that both involve instances of the timing analysis problem and the synchronous generalized retiming problem. In the first step we generate constraint edges based on a timing analysis at the source level and use them to find a retiming of operations at the source level. The result of this step is a target SFG, which is treated in the second step in a similar way as a source SFG in the first step. During the first step only real paths are taken into account, whereas in the second step additional constraint edges are generated corresponding to false paths.

Now, the first subproblem of GRP is the *timing analysis problem* and define it as follows.

Definition 3.8 (timing analysis problem (TAP)). Given are an SFG $\mathcal{G}$, a type set $\mathcal{T}$, a type function ℓ, a clock period τ, and timing functions dat, drt, del, and w. The problem is to find a constraint set F, such that if there exists a retiming r with

(*xiii*) $w_r(f) \geq 0$ for each $f = (u, v) \in F$,

then and only then

(*vi*) $D(p) < \tau$ for each $p \in P_{\mathcal{G}}$ with $W_r(p) = 0$, and

(*vii*) $w_r(e) \geq 0$ for each $e \in E$.

□

The second subproblem of GRP is the *synchronous generalized retiming problem* (SGRP).

Definition 3.9 (synchronous generalized retiming problem (SGRP)). Given are a union $\mathcal{EG}$ of source ESFGs $\mathcal{EG}_i$ with folding factors f_i, $i = 1, \ldots, N$, a type set $\mathcal{T}$, a type function ℓ, a weight w, the cost of a register α, an operator allocation al, and an operator assignment as. The problem is to find a PU $\mathcal{G}'$ constructed by a retiming r, a register allocation ra, a multiplexer allocation ma, and the given al and as, for which

(*iv*) $e' = ((as(v), a), (as(u), b)) \in E'$ and $w_r(e') = w_r(e)$ for each $e = ((v, a), (u, b)) \in E$, and

(*v*) $ph(u) \neq ph(v)$ for each $u, v \in V_i$ with $as(u) = as(v)$, and $u \neq v$, and for each $i = 1, \ldots, N$, and

(*ix*) $ra(o') + ra(i') \geq w_r(e')$ for each $e' = (o', i') \in E'$, and

(*xiii*) $w_r(f) \geq 0$ for each $f \in F$, and

that minimizes

$$\alpha \sum_{t \in T} ra(t) + \beta \sum_{i \in I} ma(i).$$

□

3.3 Complexity of subproblems

In this section we derive the complexity of PUDP's subproblems and we give for the special cases related to MP, TP, and RP, a proof of their complexity.

3.3.1 Complexity of the operator allocation problem

Theorem 3.2. OALP can be solved in polynomial time.

Proof. An optimal solution to OALP can be computed by replacing the inequality sign in the formulation of Constraints (i) by an equality sign, i.e.,

$$al(j) = \max_{i=1,\ldots,N} \lceil \frac{V_{i,j}}{f_i} \rceil \qquad \text{for each } j \in \mathcal{T}.$$

Clearly this satisfies the constraints and gives an optimal solution. □

3.3.2 Complexity of the operator assignment problem

Two special cases of OASP exist for which we would like to know their complexity. One special case concerns instances of OASP with $N \geq 1$ and $f_i = 1$ for each $i = 1,\ldots N$, which can be considered as a subproblem of MP. We call this special case OASP-MP and refer to its decision variant as OASP-MP-D. The other special case concerns instances of OASP with $N = 1$ and $f_1 \geq 1$, which can be considered as a subproblem of TP. We call this special case OASP-TP and refer to its decision variant as OASP-TP-D.

Theorem 3.3. OASP-MP-D is NP-complete.

Proof. OASP-MP-D is in NP since it can be checked in polynomial time that a solution is feasible and that the cost is at most K.

The proof that OASP-MP-D is NP-complete is based on a reduction from the Directed Hamiltonian Circuit (DHC) problem, which is NP-complete [Garey & Johnson, 1979], analogous to the proof of Theorem 3.1 that states that PUDP-D is NP-complete. The main difference is that a minimal number of operators is given here, whereas in the proof of Theorem 3.1 this was forced by the bound on the cost. Therefore, to construct an instance of OASP-MP-D a minimal operator allocation is chosen, i.e.,

$$al(j) = \max_{i=1,\ldots,N} |V_{i,j}| \qquad \text{for each } j \in \mathcal{T}.$$

Furthermore, the cost bound K is chosen to be zero. □

Theorem 3.4. OASP-TP-D is NP-complete.

Proof. OASP-TP-D is in NP since it can be checked in polynomial time that a solution is feasible and that the cost is at most K.

To prove that OASP as a subproblem of TP is NP-complete, we give a reduction from a special case of Independent Set (IS). IS is the problem to find at least J vertices of a graph $G^* = (V^*, E^*)$ that are not connected to each other by edges. IS has been proven to be NP-complete, even for the special case with $d(v^*) \leq 3$ for each $v^* \in V^*$ and $J \leq |V^*|/2$, where $d(v^*)$ denotes the degree of vertex v^* [Garey, Johnson & Stockmeyer, 1976].

Let $G^* = (V^*, E^*)$, J be an instance of IS, with $|V^*| = n$, $J \leq n/2$, and $d(v^*) \leq 3$ for each $v^* \in V^*$. We construct a source SFG $\mathcal{G}$ and a non-negative integer K, such that there is an operator assignment with at most K multiplexers if and only if G^* contains an independent set of size J or more.

An instance of OASP-TP-D is constructed as follows. An operation type set is given by $\mathcal{T} = \{1\}$, $\mathcal{O}(1) = 1$, and $\mathcal{I}(1) = 3$. $\mathcal{G}$ consists of two disjoint parts: $\mathcal{G}_a$ and $\mathcal{G}_b$. $\mathcal{G}_a$ is constructed as follows. For each $v \in V^*$ we construct an operation $v \in V_1$, with $\ell(v) = 1$. The set E_a consists of the following edges. For each edge $\{u, v\} \in E^*$ we construct an edge $((u,4),(v,k)) \in E_a$ and an edge $((v,4),(u,l)) \in E_a$, where k and l are chosen in such a way that every input terminal has at most one incoming edge. The number of inputs is sufficient for this, since $d(v^*) \leq 3$ for each $v^* \in V^*$, which means that each operation $v \in V_a$ has at most 3 incoming edges. An example is shown in Figure 3.6.

Part $\mathcal{G}_b$ of $\mathcal{G}$ is a kind of cycle of $b = 2(n-J)+3$ operations $v_1,...,v_b$. Each operation v_j, $j = 1,\ldots,b$, has an edge to each input terminal of operation v_{j+1}, and v_b has an edge to each input terminal of operation v_1, i.e.,

$$V_b = \{v_j \mid j = 1,\ldots,b\} \text{ and}$$
$$E_b = \{((v_j,4),(v_{(j \bmod b)+1},k)) \mid j = 1,\ldots,b \text{ and } k = 1,2,3\}.$$

An example is shown in Figure 3.6.

Furthermore, we take $f = 2n - 2J + 2 = 2(n-J)+3-1 = b-1$. For the number of operations we get $|V| = n+b$. Therefore

$$\frac{|V|}{f} = \frac{n+b}{b-1} = 1 + \frac{n+1}{b-1}.$$

Since $b \geq 3$ and $n > 0$, we get $(n+1)/(b-1) > 0$. Furthermore, since $J \leq n/2$, we get $n - J \geq n/2$, so

$$\frac{n+1}{b-1} = \frac{n+1}{2(n-J)+2} \leq \frac{n+1}{2 \cdot n/2+2} = \frac{n+1}{n+2} < 1.$$

This means that we can choose $al(1) = |V'| = \lceil |V|/f \rceil = 2$. We name the two allocated operators in the target SFG $\mathcal{G}'$ u'_1 and u'_2. Finally, we take $K = 3$. The above transformation can be done in polynomial time.

We can make the following observations about the constructed instance. The cycle $\mathcal{G}_b$ has length b, and thus not all operations of this cycle can be assigned to the same operator. Furthermore, since b is odd, at least $(b+1)/2$ operations of the cycle must be assigned to one operator and at most $(b-1)/2$ operations must be assigned to the other operator. Without loss of generality, we assume that at least $(b+1)/2$ operations are assigned to u'_1. Since the number of operations of the cycle assigned to u'_1 is at least one more than the number of operations assigned to u'_2, there must be two consecutive operations in the cycle that are both assigned to

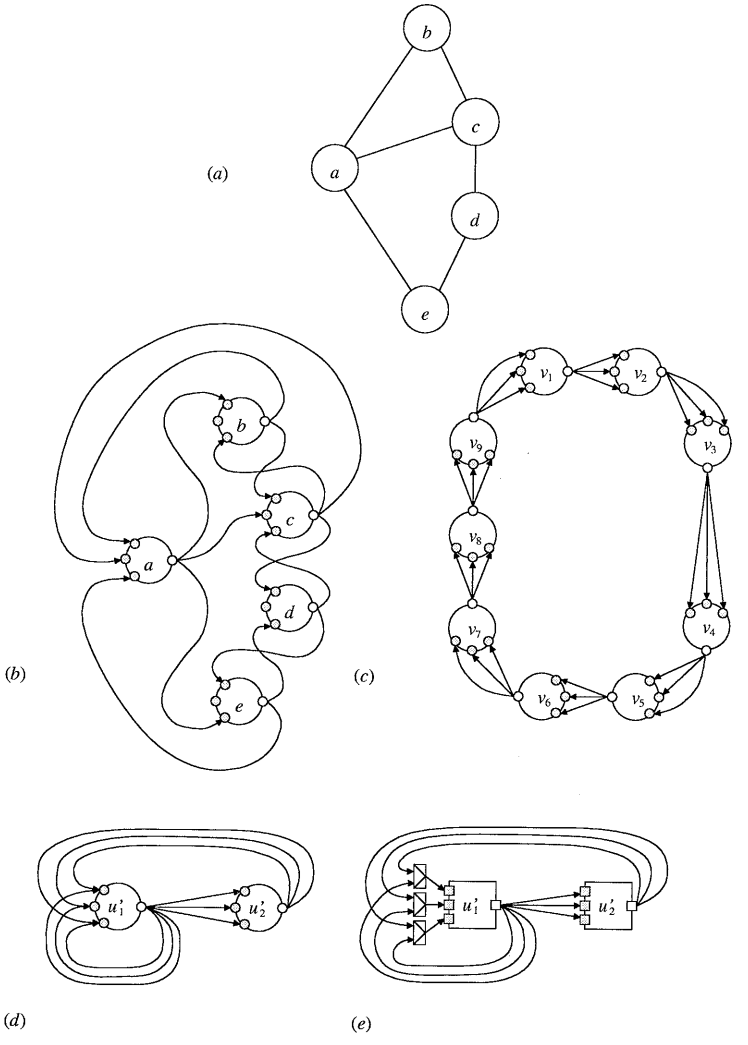


Figure 3.6. An example of an IS graph G^* (*a*) and the source SFG $\mathcal{G}$ consisting of two parts $\mathcal{G}_a$ (*b*) and $\mathcal{G}_b$ (*c*) that are derived from it. For this example $n = 5$, $J = 2$, and $b = 9$. (*d*) shows the target SFG and (*e*) the resulting PU.

u'_1. This means that the number of edges in the resulting target SFG is at least 9, and the number of multiplexers is at least 3.

We now prove that G^* has an independent set of at least size J if and only if there is an operator assignment, for which the number of multiplexers is at most $K = 3$.

Suppose G^* contains an independent set of size J or more. Take a subset $\tilde{V}$ of this independent set of size J. Now construct an operator assignment in the following way. Assign the J operations of V_a corresponding to $\tilde{V}$ to operator u'_2 in $\mathcal{G}'$ and all other $n-J$ operations of V_a to u'_1. This does not result in any edges from u'_2 to itself, since the vertices in $\tilde{V}$ are not connected and thus the corresponding operations in V_a are not connected. Consequently, no multiplexers at the inputs of u'_2 contribute to the PU cost. Next, we can still assign

$$f-(n-J) = n-J+2 = (b+1)/2$$

operations to operator u'_1 and

$$f-J = 2n-2J+2-J \geq 2n-n-J+2 = n-J+2 \geq (b-1)/2$$

operations to operator u'_2. We assign $(b+1)/2$ operations of the cycle to u'_1 and $(b-1)/2$ operations of the cycle to u'_2 in the following way.

$$as(v_j) = \begin{cases} u'_1 & \text{if } j \bmod 2 = 1, \\ u'_2 & \text{if } j \bmod 2 = 0, \end{cases}$$

for each $j = 1, \ldots, b$.

This results in a PU with exactly 3 multiplexers, since there are no edges from u'_2 to itself, but all other possible edges are present.

Now suppose that there is an operator assignment such that the number of multiplexers in the resulting PU is at most 3. We have shown before that the number of multiplexers is at least 3 and that at least $(b+1)/2$ operations of the cycle are assigned to one operator, say u'_1, and at most $(b-1)/2$ operations of the cycle are assigned to the other operator, say u'_2. This means that we can have at most $b-1-(b+1)/2 = n-J$ operations of V_a assigned to u'_1 and thus at least J operations of V_a are assigned to u'_2. Since there are already 3 multiplexers at the inputs of u'_1 as a result of the cycle in $\mathcal{G}_b$, there are no edges from u'_2 to itself, so the operations of V_a that are assigned to u'_2 cannot be interconnected. This means that the vertices in V^* corresponding to these operations are not interconnected and thus they form an independent set of size at least J. □

Corollary 3.1. OASP-D is NP-complete.

Proof. OASP-D is in NP since it can be checked in polynomial time that a solution is feasible and that the cost is at most K. Since special cases of OASP-D have been proven to be NP-complete, OASP-D is NP-complete. □

3.3.3 Complexity of the operator duplication problem

Theorem 3.5. ODP-D is NP-complete.

Proof. ODP-D is in NP since it can be checked in polynomial time that a solution is feasible and that the cost is at most K. To check whether Constraints (*xi*) are satisfied, we do not need to know all complex circuits in the instance. We can check whether a complex circuit exists by applying a depth-first search over edges starting from every operation, which runs in polynomial time. If we do not find a complex circuit we know that Constraints (*xi*) are satisfied. Note that the only cycle that can be present after operator duplication is not a complex circuit, but corresponds to a cycle in one of the source SFGs. Furthermore, it can be easily checked that the cost is at most K. Hence ODP-D is in NP.

Next we present a reduction from Graph Coloring (GC), which is NP-complete [Garey & Johnson, 1979], to ODP-D. GC is the problem to assign at most L colors to nodes in a graph such that adjacent nodes have different colors. Let $\tilde{G}=(\tilde{V},\tilde{E}),L$ be an instance of GC with $\tilde{V}=\{v_1,...,v_n\}$. We construct an instance of ODP as follows. The set of operation types is $\mathcal{T}=\{0,1,2,...,n\}$, with $\mathcal{I}(0)=0$, $\mathcal{O}(0)=1$, $\mathcal{I}(i)=2$, $\mathcal{O}(i)=1$, for each $i=1,\ldots,n$, and $\mathcal{C}(i)=1$ for each $i=0,\ldots,n$. For each $v_i\in\tilde{V}$, we construct a source SFG $\mathcal{G}_i$ with $V_i=\{v_{i1},\ldots,v_{in}\}\cup\{p_i\}$, $\ell(v_{ij})=j$ for each $j=1,\ldots,n$ and $\ell(p_i)=0$. The set of edges E_i consists of two parts E_{i1} and E_{i2}, where

$$E_{i1}=\{((v_{ii},3),(v_{ii},1))\}\ \cup\ \{((p_i,1),(v_{ik},1))\mid\{v_i,v_k\}\in\tilde{E}\},$$

$$E_{i2}=\{((v_{ik},3),(v_{i(k+1)},2))\mid k=1,\ldots,n-1\}\ \cup\ \{((v_{in},3),(v_{i1},2))\}.$$

Hence, part E_{i2} creates a cycle and is similar for each i. Figure 3.7 shows a graph and the source SFGs that are constructed from it.

The PU $\mathcal{G}'$ is constructed such that it contains one operator of type j named v'_j for each $j=1,\ldots,n$, and one operator p' of type 0. The operator assignment is as follows. $as'(p_i)=p'$, $as'(v_{ij})=v'_j$ for each $v_{ij}\in V_i$ and for each $i,j=1,\ldots,n$. Furthermore, we take all delays of operations equal to undefined, i.e., $del(i,o)=\sim$ for each $(i,o)\in I\times O$. Consequently, no complex circuits can exist. Furthermore, we take $\beta=nL+2$ and $K=nL+1$. This implies that there can be no multiplexers in $\mathcal{G}^*$, and at most $nL+1$ operators. The above transformation can be done in polynomial time.

We can make the following remarks about the constructed instance of ODP-D. It is of no use to duplicate operator p' since it does not have any input terminals. If we duplicate one operator of type j, then we have to duplicate all the other operators of type $k=1,\ldots,n$, $k\neq j$, in the same way, because otherwise the sets E_{i2} results in multiplexers in $\mathcal{G}^*$. Therefore, for two source SFGs $\mathcal{G}_s$ and $\mathcal{G}_t$, where $s\neq t$, either each operation v_{sj} of V_s is assigned to the same operator as operation v_{tj}

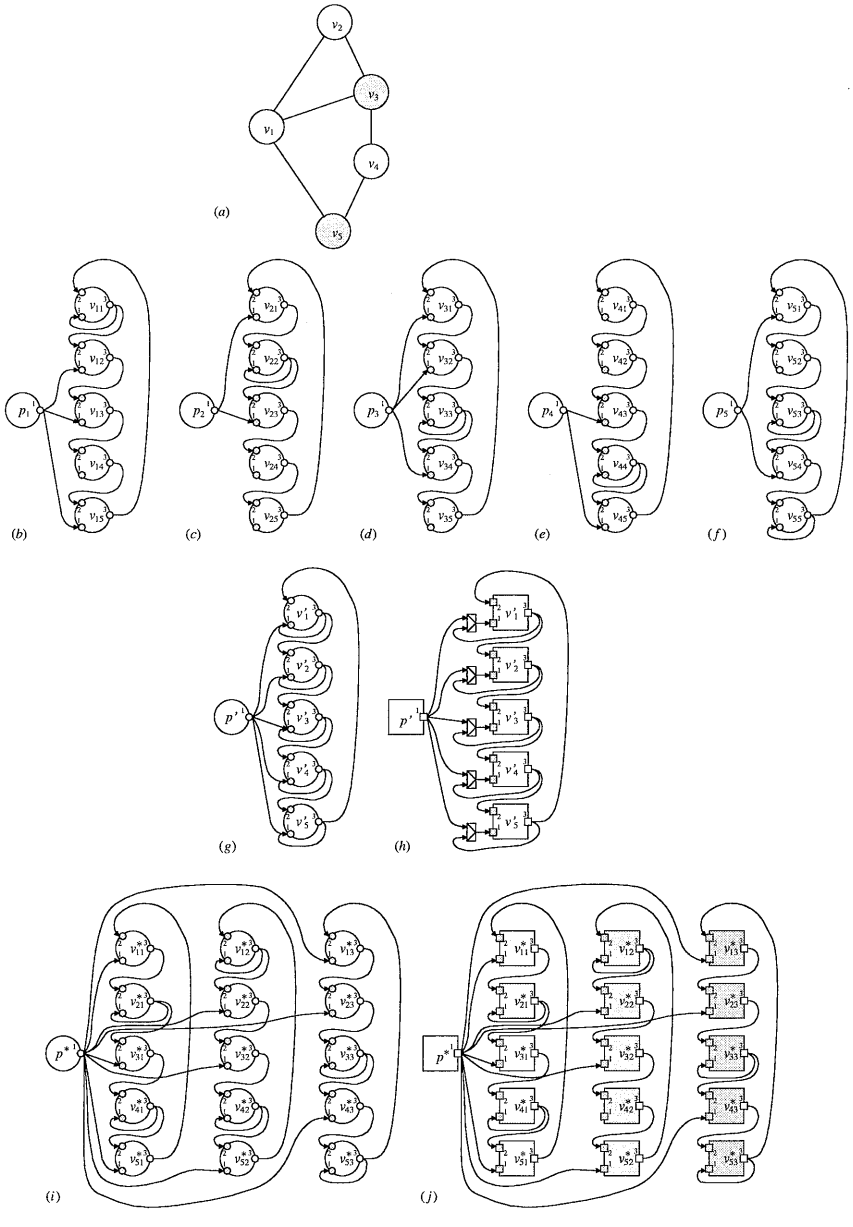


Figure 3.7. An example of a GC graph $\tilde{G}$ (*a*) and the source SFGs $\mathcal{G}_i$, $i = 1,\ldots,5$ ((*b*), (*c*), (*d*), (*e*), (*f*)) that are derived from it. (*g*) shows the initial target SFG $\mathcal{G}'$ and (*h*) shows the initial PU. (*i*) shows the target SFG $\mathcal{G}^*$ after operator duplication and and (*j*) the PU after operator duplication.

of V_t, or each operation v_{sj} of V_s is assigned to a different operator than operation v_{tj} of V_t. Hence $\mathcal{G}^*$ always has one operator of type 0 and a multiple of n other operators, such that the number of operators of each type j is equal, since the number of multiplexers is zero. Furthermore, in $\mathcal{G}'$, two source SFGs $\mathcal{G}_s$ and $\mathcal{G}_t$ introduce two multiplexers if $\{v_s, v_t\} \in \tilde{E}$. In that case, $((p_s, 1), (v_{st}, 1)) \in E_s$ and $((v_{tt}, 3), (v_{tt}, 1)) \in E_t$. Since $v'_t = as'(v_{st}) = as'(v_{tt})$ and $p' = as(p_t)$, E' contains the edges $((p', 1), (v'_t, 1))$ and $((v'_t, 3), (v'_t, 1))$. Consequently, $ma(v'_t, 1) = 1$. Likewise, it follows that $ma(v'_s, 1) = 1$ if $\{v_s, v_t\} \in \tilde{E}$.

We now prove that graph $\tilde{G}$ can be colored with L colors if and only if we can duplicate operators in the instance of ODP such that the cost is at most K.

Suppose $\tilde{G}$ is colored with L colors by $z : \tilde{V} \to \{1, 2, ..., L\}$. Then we can construct an operator duplication as follows. Duplicate operator v'_j into L operators, named $v^*_{j1}, v^*_{j2}, ..., v^*_{jL}$, and assign operation v_{ij} of V_i to operator $v^*_{jz(v_i)}$ for each $i, j = 1, \ldots, n$. Then for each $\{v_i, v_j\} \in \tilde{E}$, the operations of type i and j in $\mathcal{G}_i$ and $\mathcal{G}_j$ are assigned to different operators, so no multiplexers are present. Furthermore, the number of operators used is at most $nL + 1$. Therefore the cost is at most K.

Suppose we have an operator assignment as^* such that the cost is not more than $nL + 1$. Then no multiplexers are present and the operators of each type are duplicated into at most L operators. For each operator v'_j in $\mathcal{G}'$, we number the operators in $\mathcal{G}^*$ that originate from v'_j from v^*_{j1} to v^*_{jL} in such a way that for each $\mathcal{G}_i$, there exists some $k = 1, \ldots, L$, such that each operation $v_{ij} \in V_i$ is assigned to operator v^*_{jk}, where $i, j = 1, \ldots, n$. This is possible, since no multiplexers are used. This also means that two operations of two source SFGs $\mathcal{G}_i$ and $\mathcal{G}_j$ are not assigned to the same operators if $\{v_i, v_j\} \in \tilde{E}$. Hence we can color the vertices of $\tilde{G}$ as follows. For each source SFG $\mathcal{G}_i$, if operation v_{i1} is assigned to operator v^*_{1k}, then we color v_i with color k. □

3.3.4 Complexity of the generalized retiming problem

Since GRP is decomposed into two subproblems we discuss the complexity of each of these problems.

Complexity of the timing analysis problem

Theorem 3.6. TAP can be solved in polynomial time.

Proof. The number of Constraints (*vi*) is not necessarily polynomial in the size of the instance, because the number of paths in an SFG $\mathcal{G} = (V, T, E)$ is not necessarily bounded by a polynomial. However, the number of constraints in the constraint set F can be bounded by a polynomial, since we do not need all paths. For each $u, v \in V$, we only need a path p between an output terminal of u and an input terminal of v with the smallest weight and of these paths with smallest weight, we need the one with the largest delay [Leiserson, Rose & Saxe, 1983]. If this delay is larger

than or equal to c, then a constraint $f = (u, v)$ is generated with $w(f) = W(p) - 1$. These constraints can be generated in polynomial time by an *all-pairs shortest-path* algorithm [Floyd, 1962; Warshall, 1962] as indicated by Leiserson, Rose & Saxe [1983]. Furthermore, the causality constraints in F that correspond to Constraints (*vii*) can be trivially generated in polynomial time. Hence TAP is in P. □

Complexity of the synchronous generalized retiming problem

Three special cases of SGRP exist for which we like to know the complexity. The first special case concerns instances of SGRP with $N \geq 1$ and $f_i = 1$ for each $i = 1, \ldots, N$. We call this special case SGRP-MP and refer to its decision variant as SGRP-MP-D. The second special case concerns instances of SGRP with $N = 1$ and $f_1 \geq 1$, which can be considered as a subproblem of TP. We call this special case SGRP-TP and refer to its decision variant as SGRP-TP-D. The third special case concerns instances of SGRP with $N = 1$ and $f_1 = 1$, which can be considered as a subproblem of RP. We call this special case SGRP-RP.

Theorem 3.7. SGRP-MP-D is NP-complete.

Proof. SGRP-MP-D is in NP since it can be checked in polynomial time that a solution is feasible and that the cost is at most K.

We give a reduction from 3-satisfiability (3SAT), which is NP-complete, to SGRP-MP-D [Garey & Johnson, 1979]. 3SAT is the problem of finding a truth assignment $f : U \to \{\text{true}, \text{false}\}$ for a set of variables U that are used in a set of clauses C containing three literals of different variables. Let an instance U, C of 3SAT be given, where U is a set of variables and C a set of clauses. Then we construct an instance of SGRP-MP-D from this as follows. For each $u \in U$, we create an operation type t_u and an operation type $t_{\bar{u}}$, with $\mathcal{I}(t_u) = \mathcal{I}(t_{\bar{u}}) = \mathcal{O}(t_u) = \mathcal{O}(t_{\bar{u}}) = 2$. Next, we construct one source ESFG $\mathcal{ESG}_u$ for each $u \in U$ in the following way; see Figure 3.8. $V_u = \{u, \bar{u}\}$, with $\ell(u) = t_u$, and $\ell(\bar{u}) = t_{\bar{u}}$, $E_u = \{((u,3),(\bar{u},1)),((\bar{u},3),(u,1))\}$ and $w(((u,3),(\bar{u},1))) = 1$, $w(((\bar{u},3),(u,1))) = 0$. Furthermore, the constraint sets of these source ESFGs contain only causality constraints. These source ESFGs correspond with a truth assignment. The number of registers in each source ESFG cannot be changed since they are cycles. For each source ESFG $\mathcal{ESG}_u$, the register can be placed either on $((u,3),(\bar{u},1))$ or on $((\bar{u},3),(u,1))$ by retiming. We let the first case correspond to a truth assignment in which u is true and the second case correspond to a truth assignment in which u is false, so the weight on the first edge represents the 'value' of u and the weight on the second edge represents the 'value' of $\bar{u}$. Next, we construct a source ESFG $\mathcal{ESG}_c$ for each clause $c \in C$ consisting of literals l_1, l_2 and l_3 in the following way. $V_c = \{l_1, \bar{l}_1, l_2, \bar{l}_2, l_3, \bar{l}_3\}$, with each operation of the corresponding type, and

$$E_c = \{((l_1,3),(\bar{l}_1,1)),((l_2,3),(\bar{l}_2,1)),((l_3,3),(\bar{l}_3,1)),$$

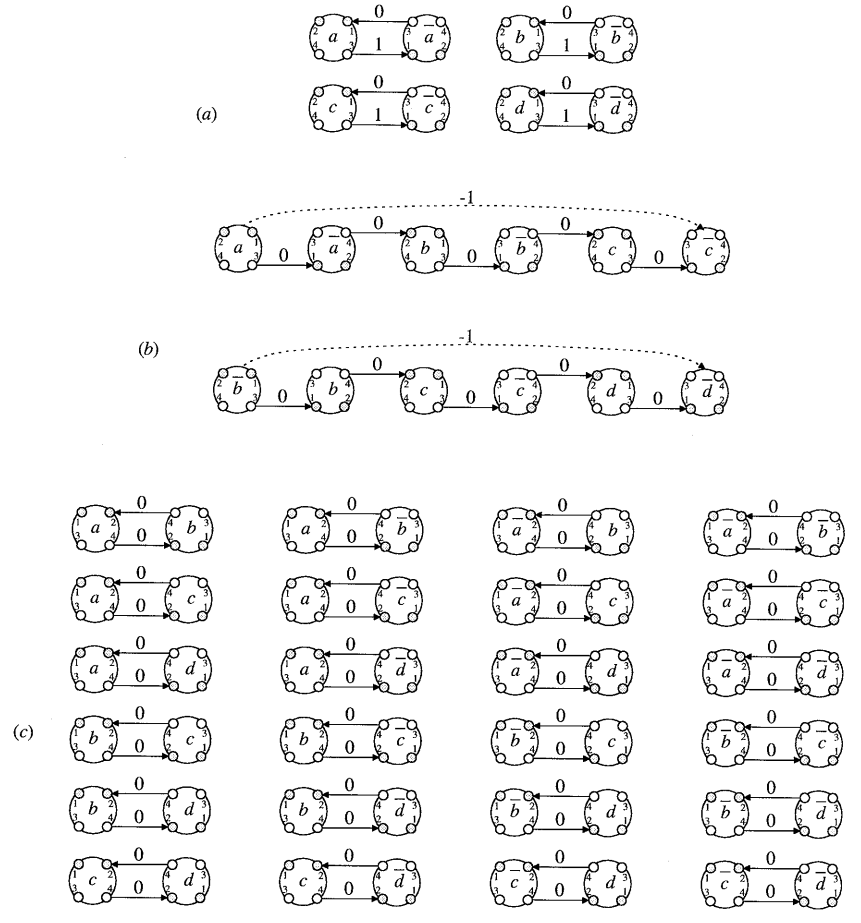


Figure 3.8. An example of a set of variables $U = \{a, b, c, d\}$ and a set of clauses $C = \{abc, \bar{b}cd\}$ and source ESFGs $\mathcal{ESG}_u$ (*a*), $\mathcal{ESG}_c$ (*b*), and $\mathcal{ESG}_{l_1 l_2}$ (*c*) that are constructed from them. In the figures, the constraint edges shown (dashed) correspond to speed constraints only.

$$((\bar{l}_1,4),(l_2,2)),((\bar{l}_2,4),(l_3,2))\}.$$

All the weights of the edges are 0.

The source ESFGs contain paths with six operations. After retiming, a weight of 1 on an edge from a literal to the complement of this literal will correspond to a truth assignment of this literal. The constraint set is such that at least one of the signal edges will have a retimed weight greater than zero, since we add a speed constraint edge

$$(l_1,\bar{l}_3)\in F_c,$$

with weight -1. Furthermore, the constraint sets of these source ESFGs contain causality constraints.

Next, we construct a source ESFG $\mathcal{ESG}_{l_1l_2}$ for each pair of literals l_1,l_2, where $l_1\neq l_2$ and $\bar{l}_1\neq l_2$, with $V_{l_1l_2}=\{l_1,l_2\}$, and each operation of the corresponding type,

$$E_{l_1l_2}=\{((l_1,4),(l_2,2)),((l_2,4),(l_1,2))\},$$

and the weights all equal to 0. The constraint sets of these source ESFGs contain only causality constraints. The weights on the edges in these source ESFGs can never be changed by retiming, since they are cycles. Thus we now have constructed three different sets of source ESFGs with a total of $|U|+|C|+2|U|^2-2|U|$ ESFGs.

The operator allocation consists of one operator of each type, and the operator assignment assigns each operation to the corresponding operator. Furthermore, we choose the register cost $\alpha=1$, the multiplexer cost $\beta=1$, and

$$K=4|U|^2-5|U|.$$

The above transformation can be done in polynomial time.

We can make the following remarks about the instance of SGRP-MP-D constructed. The instance is constructed in such a way that to obtain a cost K, a minimum number of registers $|U|$ is present as well as a minimum number of multiplexers $4|U|^2-6|U|$ is present. This number of registers is required for the implementation of the retimed weights in the source ESFGs $\mathcal{ESG}_u$. Mapping all source ESFGs $\mathcal{ESG}_{l_1l_2}$ together on one PU will result in $(2(|U|-1)-1)$ multiplexers at each second input terminal of each operator corresponding to a literal. In order not to exceed the cost K, retiming may not increase the number of multiplexers by causing different weights on edges between equal input terminals and output terminals in the target SFG. Consequently, the first, third, and fifth edge of each of the paths in $\mathcal{ESG}_c$ must be joined with edges of the source ESFGs $\mathcal{ESG}_u$ that correspond to the truth assignment in the resulting target SFG after retiming, so they may only have weight zero or one after retiming, and they must correspond to the truth assignment. The second and fourth edge of each of the paths in $\mathcal{ESG}_c$ must be joined with edges of the source ESFGs $\mathcal{ESG}_{l_1l_2}$, so these edges always have weight

zero after retiming. To satisfy the speed constraints in the constraint set F_c, one of the signal edges of the path must obtain a retimed weight greater than zero. This implies that the first, third or fifth edge of the path must have a retimed weight greater than zero, which corresponds to satisfying the corresponding clause.

We now prove that there is a truth assignment for U that satisfies all clauses in C if and only if there is a feasible retiming, a feasible register allocation, and a feasible multiplexer allocation such that the cost is at most K.

Suppose there is a truth assignment that satisfies all clauses. Then we construct a retiming as follows.

For each $\mathcal{EG}_u$

$$r(u) = 0,$$

$$r(\bar{u}) = \begin{cases} 0 & \text{if } f(u) = \text{true} \\ -1 & \text{if } f(u) = \text{false.} \end{cases}$$

For each $\mathcal{EG}_{l_1 l_2}$,

$$r(l_1) = r(l_2) = 0.$$

For each $\mathcal{EG}_c$

$$r(l_1) = 0,$$

$$r(\bar{l}_1) = \begin{cases} 1 & \text{if } l_1 \text{ is true} \\ 0 & \text{if } l_1 \text{ is false,} \end{cases}$$

$$r(l_2) = r(\bar{l}_1),$$

$$r(\bar{l}_2) = \begin{cases} r(l_2)+1 & \text{if } l_2 \text{ is true} \\ r(l_2) & \text{if } l_2 \text{ is false,} \end{cases}$$

$$r(l_3) = r(\bar{l}_2),$$

$$r(\bar{l}_3) = \begin{cases} r(l_3)+1 & \text{if } l_3 \text{ is true} \\ r(l_3) & \text{if } l_3 \text{ is false.} \end{cases}$$

Then the weights on the edges are as follows. For each $u \in U$, if $f(u) = \text{true}$, then $w_r(((u,3),(\bar{u},1))) = 1$, otherwise $w_r(((\bar{u},3),(u,1))) = 1$ in each source ESFG in which these edges occur, and all other edges have a retimed weight equal to zero. $2(|U|-1)-1$ multiplexers are needed this way for each input 2, since all edges between the same terminals have the same weight. A register allocation that allocates to each output terminal the weight on the outgoing edges of that output terminal is an optimal one. This is because no multiplexers are used at each input 1, so each output 3 in the target SFG has exactly one outgoing edge and each input 1 has exactly one incoming edge. The total number of registers is $|U|$, so the total cost is K. The timing constraints are satisfied, since each edge has non-negative weight and the paths corresponding to the clauses must have weight at least one since the clauses are satisfied.

Suppose there is a feasible retiming r, a feasible register allocation ra, and a feasible multiplexer assignment such that the cost is K. Then we can make a truth assignment f as follows: for each $u \in U$, if $w_r((u,3),(\overline{u},1)) = 1$ in $\mathcal{EG}_u$, then $f(u) =$ true, otherwise $f(u) =$ false. At least one of the first, third, and fifth edges of each source ESFG $\mathcal{EG}_c$ corresponding to a clause must have weight one, and this can only happen if this edge is joined in the target SFG with an edge of the corresponding source ESFG $\mathcal{EG}_u$ that has weight one. This means that the corresponding literal in the truth assignment is true. □

In practical cases, each input terminal in a source ESFG has one incoming edge and each output terminal has at least one outgoing edge, which is not the case in the constructed ESFGs in the proof. However, we can slightly modify the construction by making a new operation with one input terminal for each output terminal that does not have an outgoing edge and then creating an edge from the output terminal of this operation to the input terminal of the new operation. For each input terminal that does not have an incoming edge, we make a new operation that has one output terminal and an edge from that output terminal to the input terminal. The number of new operations constructed in this way is polynomial, since the total number of terminals is. The new operations are each assigned to their own types of operators. In this way, new multiplexers are introduced, but their number cannot be reduced by retiming. The new edges do not influence the possible retimings or the minimal number of registers, since we can always choose the weight on the new edges equal to zero, by choosing the retiming of each new operation equal to the retiming of the operation to which it is connected.

Theorem 3.8. SGRP-TP-D is NP-complete.

Proof. SGRP-TP-D is in NP since it can be checked in polynomial time that a solution is feasible and that the cost is at most K.

We can use almost the same construction as in the proof that SGRP-MP-D is NP-complete. In this proof the number of source ESFGs is one and the folding factor is greater than one, whereas in the proof that SGRP-MP-D is NP-complete the number of source ESFGs was greater than one and the folding factor equal to one. Given an instance of 3SAT, the transformation is modified as follows. The different source ESFGs are all taken together in one large source ESFG, called $\mathcal{EG}$. The folding factor $f = 2|U|^2 - 2|U| + 4|C| + 2$. The rest remains unchanged. We allocate for each type of operation one operator, since there are at most $2|U| + |C| + 1$ operations of the same type, and $\left\lceil \frac{2|U|+|C|+1}{2|U|^2-2|U|+4|C|+2} \right\rceil = 1$.

Suppose there is a truth assignment that satisfies all clauses. Then we construct a retiming r, a register allocation and a multiplexer allocation such that the cost is at most K, analogous to the construction in the proof that SGRP-MP-D is NP-

complete. We then construct from this a retiming r' that also satisfies the phase constraints, while the cost remains the same.

The retiming r' is constructed as follows. The operations in the parts of the source ESFG formed by the $\mathcal{EG}_u$s are not assigned to the same operator, so they may have the same retiming. Since r satisfies the timing constraints, we have $0 \leq r(u) - r(\overline{u}) \leq 1$ in each $\mathcal{EG}_u$, which means that we can take $r'(\overline{u}) = 0$ and $r'(u) = r(u) - r(\overline{u})$ for $u, \overline{u} \in V_u$ for each $u \in U$.

In $\mathcal{EG}_{l_1 l_2}$, operations l_1 and l_2 can never have a different retiming. We number these $\mathcal{EG}_{l_1 l_2}$s with the numbers $0, 1, \ldots, 2|U|^2 - 2|U| - 1$ and give the operations of the ith $\mathcal{EG}_{l_1 l_2}$ both retiming $i+2$ ($i = 0, \ldots, 2|U|^2 - 3$), so we do not use 0 or 1, which have already been used. At this point we have used retimings $0, 1, \ldots, 2|U|^2 - 2|U| + 1$.

Next we consider the $\mathcal{EG}_c$s corresponding to the clauses. If we have a clause $\{l_1, l_2, l_3\}$, then $1 \leq r(\overline{l_3}) - r(l_1) \leq 3$, since r is a retiming that satisfies the timing constraints and that has cost at most K. We number the clauses $0, 1, \ldots, |C| - 1$, and give the first operation l_1 of the path corresponding to clause j retiming $r'(l_1) = 2|U|^2 - 2|U| + 2 + 4j$, and all other operations x in this path $r'(x) = r'(l_1) + r(x) - r(l_1)$. Then two operations from different clauses never have the same retiming and the operations never have the same retiming as any other operation from another part of the source ESFG. We now have used retimings $0, 1, \ldots, 2|U|^2 - 2|U| + 4|C| + 1$, which means that operations sharing the same operator not only have different retimings, but also different phases. Therefore we can conclude that in the above way we have constructed a retiming that additionally satisfies the phase constraints. The cost is not changed, since neither the retimed weight on each edge nor the register allocation are changed.

Suppose a feasible retiming r, a feasible register allocation ra, and a feasible multiplexer assignment exist such that the cost is K. Then we can make a truth assignment f in the same way as in the proof that SGRP-TP-D is NP-complete, which is as follows: for each $u \in U$, if $w_r((u, 3), (\overline{u}, 1)) = 1$ in $\mathcal{EG}_u$, then $f(u) =$ true, otherwise $f(u) =$ false. At least one of the first, third, and fifth edges of each part of the source ESFG $\mathcal{EG}_c$ corresponding to a clause must have weight one, and this can only happen if this edge is joined in the target SFG with an edge of the corresponding part of the source ESFG $\mathcal{EG}_u$ that has weight one. This means that the corresponding literal in the truth assignment is true. □

Again we can modify the construction of the proof such that each input terminal in the source ESFG has exactly one incoming edge and each output terminal has at least one outgoing edge.

Corollary 3.2. SGRP-D is NP-complete.

Proof. SGRP-D is in NP since it can be checked in polynomial time that a solution is feasible and that the cost is at most K. Since the special cases of SGRP-D, SGRP-MP-D and SGRP-TP-D, have been proven to be NP-complete, SGRP-D is NP-complete. □

Theorem 3.9. GRP-D is NP-complete.

Proof. The proof is analogous to both the proof that SGRP-MP-D is NP-complete and the proof that SGRP-TP-D is NP-complete, which are based on a reduction from 3SAT. Here, we limit ourselves to giving the main difference in the proof, which is the following. Instead of constructing a speed constraint for each $\mathcal{EG}_u$, we choose

$$\tau = 5$$

$$dat(i) = 1 \qquad \text{for each } i \in I, \text{ and}$$

$$drt(o) = 1 \qquad \text{for each } o \in O, \text{ and}$$

$$del((v,1),(v,3)) = 1,$$
$$del((v,1),(v,4)) = 1,$$
$$del((v,2),(v,3)) = 1 \qquad \text{for each } v \in V \text{ and,}$$

$$del((u,a),(v,b)) = \sim \qquad \text{for each } u, v \in V_u$$
$$\text{with } u \neq v,\ (u,a) \in O, \text{ and}$$
$$(v,b) \in I, \text{ or}$$
$$\text{with } u = v,\ a = 2, \text{ and } b = 4.$$

The delay on the path from the output of the first operation to the input of the last operation in an SFG $\mathcal{G}_c$ is equal to 6. Since the clock period is equal to 5, at least one of the signal edges must get a retimed weight greater than zero. □

Although SGRP is NP-complete, the third special case, SGRP-RP, in which an instance of SGRP is restricted to one source ESFG that is not timefolded or multiplexed, can be solved in polynomial time. We call this problem the *synchronous retiming problem* (SRP). SRP can be reformulated as the *dual* of the *network flow* problem [Ford & Fulkerson, 1962], which can be solved in polynomial time. We reformulate SRP in two steps. In the first step it is reduced into an *integer linear programming* (ILP) problem [Papadimitriou & Steiglitz, 1982] and in the second step it is reduced into the network flow problem.

By writing the feasibility constraints as inequalities and expanding the cost we show that SGRP can be written as an ILP problem.

Theorem 3.10. Given are an ESFG $\mathcal{EG}$, a weight w, an operator allocation al, and an operator assignment as. Then the *synchronous retiming problem* (*SRP*) is equivalent to the following ILP problem. Find a retiming r, and a register allocation ra, for which

$$
\begin{aligned}
& w_r(f) \geq 0 && \text{for each } f \in F, \text{ and} \\
& ra(o) + ra(i) \geq w_r(e) && \text{for each } e = (o, i) \in E, \text{ and} \\
& ra(t) \geq 0 && \text{for each } t \in T, \text{ and} \\
& w_r(f) = w(f) + r(v) - r(u) && \text{for each } f = (u, v) \in F, \text{ and} \\
& w_r(e) = w(e) + r(v) - r(u) && \text{for each } e = ((u, a), (v, b)) \in E, \text{ and}
\end{aligned}
$$

that maximizes

$$-\sum_{t \in T} ra(t).$$

Proof. Straightforward substitution of the constraints yield the above linear (in)equalities. Furthermore, the cost function is linear. □

We cast this ILP problem into the dual of the network flow problem such that we can select an efficient solution method. In the above ILP problem formulation, constraints are present as inequalities containing up to four variables. The dual of the network flow problem has constraints in which a difference between two variables is at most a constant. To cast the ILP formulation of SRP into the dual of the network flow problem, we need the help of a substitution function.

Definition 3.10 (substitution function). Given are an SFG $\mathcal{G}$, a retiming r, and a register allocation ra. The *substitution function* $n : T \rightarrow \mathbb{Z}$ is defined as follows:

$$
\begin{aligned}
& n(i) = r(v) - ra(i) && \text{for each } i = (v, b) \in I, \text{ and} \\
& n(o) = r(u) + ra(o) && \text{for each } o = (u, a) \in O.
\end{aligned}
$$

□

Theorem 3.11. Given are an ESFG $\mathcal{EG}$, a weight w, an operator allocation al, and an operator assignment as. Then the *synchronous retiming problem* (*SRP*) is equivalent to the following dual of the network flow problem. Find a retiming r and a substitution function n, for which

$$
\begin{aligned}
& r(u) - r(v) \leq w(f) && \text{for each } f = (u, v) \in F, \text{ and} \\
& n(i) - n(o) \leq -w(e) && \text{for each } e = (o, i) \in E, \text{ and} \\
& r(u) - n(o) \leq 0 && \text{for each } o = (u, a) \in O, \text{ and} \\
& n(i) - r(v) \leq 0 && \text{for each } i = (v, b) \in I, \text{ and}
\end{aligned}
$$

that maximizes

$$\sum_{v \in V} r(v)[\mathcal{O}(\ell(v)) - \mathcal{I}(\ell(v))] + \sum_{i \in I} n(i) - \sum_{o \in O} n(o).$$

Proof. Substituting the register allocation in the ILP formulation of Theorem 3.10 by the given combination of the retiming and the substitution function yields this

ILP formulation, which is identified as the dual of the network flow problem. The inequalities are transformed by a simple substitution while the cost is transformed as follows:

$$\begin{aligned} -\sum_{t\in T} ra(t) &= -\sum_{i\in I} ra(i) - \sum_{o\in O} ra(o) \\ &= -\sum_{i=(v,b)\in I} [r(v) - n(i)] - \sum_{o=(u,a)\in O} [n(o) - r(u)] \\ &= \sum_{v\in V} r(v)[\mathcal{O}(\ell(v)) - \mathcal{I}(\ell(v))] + \sum_{i\in I} n(i) - \sum_{o\in O} n(o). \end{aligned}$$

□

The network flow problem is defined on a graph, which has nodes corresponding to the retiming of operations of V and the substitution function of terminals of T. The edges in this graph correspond to the inequalities in the problem formulation.

3.4 Overview of the complexity of PUDP and its subproblems

In the introductory chapter, Chapter 1, we discussed the multiplexing, timefolding, and retiming techniques and introduced a generalized technique, which combines them. In the previous chapter, Chapter 2, we expressed the processing unit design problem in a formal way. By restricting the instances of PUDP we derived three special cases of PUDP, namely the multiplexing problem, the timefolding problem, and the retiming problem, each of which is related to a design technique discussed in Chapter 1.

Table 3.1. Overview of the complexity of PUDP and its subproblems. When a trivial solution can be found to a problem it is indicated by T, when a problem can be solved in polynomial time it is indicated by P and when a problem is NP-complete it is indicated by NPC.

Special Cases			Subproblems' complexity				
N	f_i	complexity	OALP	OASP	ODP	GRP	
						TAP	SGRP
≥ 1	≥ 1	NPC	T	NPC	NPC	P	NPC
≥ 1	1	NPC	T	NPC	NPC	P	NPC
1	≥ 1	NPC	T	NPC	NPC	P	NPC
1	1	P	T	T	T	P	P

In this chapter we decomposed PUDP into four subproblems, i.e., the operator allocation problem (OALP), the operator assignment problem (OASP), the operator duplication problem (ODP) and the the generalized retiming problem (GRP), of which the latter is subdivided into the timing analysis problem (TAP) and the synchronous generalized retiming problem (SGRP). We considered special cases of the subproblems in our complexity analysis. The special cases of the subprob-

lems are related to the special cases of PUDP. However, to match the special cases of PUDP with the design techniques as discussed in Section 1.2, the data in the instances are more restricted than is the case with the special cases of the subproblems of PUDP. An overview of the complexity of PUDP and its subproblems as given in Table 3.1.

4

Basic Solution Methods

In the previous chapter we introduced a decomposition of PUDP. We next have to design algorithms that can handle the corresponding subproblems. These algorithms are based on two basic solution methods, i.e., *local search* and *network flow*, which we describe in this chapter. In Section 4.1 we discuss local search algorithms and in Section 4.2 we discuss network flow algorithms.

4.1 Local search

For NP-hard subproblems of PUDP the cost function can be computed quickly. This is one of the main reasons to choose a class of generally applicable algorithms that we know as *local search* algorithms. These algorithms can easily be implemented, are flexible in use, and proved to be quite successful in many applications [Yannakakis, 1990; Aarts & Lenstra, 1997]. Local search algorithms are based on a simple concept, namely the change in cost by a stepwise exploration of a solution space. The use of a local search algorithm presupposes the definition of a solution set $\mathcal{S}$, a cost function $f : \mathcal{S} \rightarrow \mathbb{Q}$, and a neighborhood structure $\mathcal{N} : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$, which defines for each solution a set of solutions that can be reached in one step of the algorithm, where $\mathcal{P}(\mathcal{S})$ denotes the set of all subsets of $\mathcal{S}$. Furthermore, a start solution must be given. Solutions and their neighborhoods can be represented graphically by a neighborhood graph, which is defined

as follows.

Definition 4.1 (neighborhood graph). A *neighborhood graph* is a directed graph with a vertex set W given by the solution set $\mathcal{S}$ and an edge set E, defined by $E = \{(s_u, s_v) \in W \times W \mid s_v \in \mathcal{N}(s_u)\}$. □

A run of a local search algorithm can be regarded as a walk on a neighborhood graph, repeatedly from a solution to one of its neighbors, which we refer to as transitions. We consider a transition to consist out of two parts: the generation of a solution and the evaluation of its acceptance. Every neighborhood of a given solution can be generated with a not necessarily equal probability. After a solution is generated from the neighborhood it is accepted depending on its cost and on the type of transition mechanism. Various local search algorithms exist that differ predominantly in the way they organize the walk on the neighborhood graph. The strongly-connectedness of a neighborhood graph plays an important role for local search algorithms, since it enables them to walk from any solution to every other solution.

Here we discuss the *iterative improvement* and *simulated annealing* algorithms, which only differ in their acceptance part. Over the years many other kind of local search algorithms have been presented, among which *tabu search* and *genetic* algorithms are probably the best known ones. For more details on these algorithms we refer to Aarts & Lenstra [1997].

4.1.1 Iterative improvement

The ITERATIVE_IMPROVEMENT algorithm uses a simple transition mechanism. It randomly generates a neighboring solution and steps to this solution if it leads to an improvement in the cost or if the cost remains unchanged. The algorithm terminates when no improvement in cost has been achieved after a number of passes. A description of the ITERATIVE_IMPROVEMENT algorithm in a pseudo programming language is shown in Figure 4.1.

4.1.2 Simulated annealing

The ITERATIVE_IMPROVEMENT algorithm stops in the first local minimum that is found. This is a drawback, because this may not be the solution with the lowest cost. In order to be able to escape from a local minimum, an algorithm must accept cost deteriorating transitions. An algorithm that accepts deteriorations, to a limited extent, is the SIMULATED_ANNEALING algorithm [Aarts & Korst, 1989; Van Laarhoven & Aarts, 1987; Aarts & Lenstra, 1997].

A description of the SIMULATED_ANNEALING algorithm is shown in Figure 4.2. In the SIMULATED_ANNEALING algorithm, the temperature is modelled

```
procedure ITERATIVE_IMPROVEMENT
in  a solution set S,
    a cost function f : S → Q,
    a neighborhood structure N : S → P(S);
out a solution s_i;
begin INITIALIZE(s_start);
    s_i=s_start;
    repeat
        GENERATE(s_j from N(s_i));
        if f(s_j) ≤ f(s_i) then s_i=s_j;
        else skip;
    until some stop criterion;
end;
```

Figure 4.1. The ITERATIVE_IMPROVEMENT algorithm.

by a control parameter c_k. Generated solutions with a cost deterioration are accepted with a high probability for large values of c_k and with low probability for small values of c_k. The algorithm accepts a solution if it leads to an improvement in the cost or if the cost remains unchanged.

A run of the SIMULATED_ANNEALING algorithm is controlled by a cooling schedule that determines the values of a number of parameters in the algorithm, such as the control parameter c_k and the length L_k [Aarts & Korst, 1989; Van Laarhoven & Aarts, 1987]. The cooling schedule also accounts for the stop criterion. We use the adaptive cooling schedule presented by Aarts, De Bont, Habers & Van Laarhoven [1985]. This schedule sets the chain length L_k equal to the size of the neighbourhood for each value of k. In the beginning of the walk on a neighbourhood graph, large deteriorations in the cost are accepted. As the walk proceeds, fewer deteriorations are accepted and they are of smaller size. In the end only solutions are accepted that improve the cost. The algorithm stops if no improvement is obtained for a given number of subsequent iterations in k and gives as a result the best visited solution.

It has been proved that the SIMULATED_ANNEALING algorithm finds an optimal solution if an infinitely large amount of run time were available [Aarts & Korst, 1989; Van Laarhoven & Aarts, 1987]. In practice, one can only resort to finite time implementations resulting in approximations of optimal solutions, as shown in Figure 4.2.

```
procedure SIMULATED_ANNEALING
in  a solution set S,
    a cost function f : S → Q,
    a neighborhood structure N : S → P(S);
out a solution s_i;
begin INITIALIZE(s_start, c_0, L_0);
    k=0;
    s_i=s_start;
    repeat
        for t=1 to L_k
        begin GENERATE(s_j from N(s_i));
            if f(s_j) ≤ f(s_i) or exp((f(s_i)-f(s_j))/c_k) > random[0,1) then s_i=s_j;
            else skip;
        end;
        k=k+1;
        CALCULATE_LENGTH(L_k);
        CALCULATE_CONTROL(c_k);
    until some stop criterion;
end;
```

Figure 4.2. The SIMULATED_ANNEALING algorithm.

4.2 Network flow

Some of the subproblems of PUDP that can be solved in polynomial time are reduced to the network flow problem in Chapter 3. Many special cases of the network flow problem exist among which the well known shortest path and maximum-flow problem. In this section we give a condensed overview of network flow problems and their solution methods. For a more elaborate overview we refer to an excellent overview given by Ahuja, Magnanti & Orlin [1989]. In Section 4.2.1 we introduce a formal description of a network and the network flow problem. Special cases of this problem are treated in Section 4.2.2 to Section 4.2.4. In Section 4.2.5 we discuss the maximum-weight bipartite matching problem and show how it is mapped onto the dual of the minimum-cost network flow problem.

4.2.1 Network flow problems

A network is defined as follows.

Definition 4.2 (network). A *network* is a directed graph $G = (V, E)$ with the following labels.

$c : E \rightarrow \mathbb{R}$	the *cost* of one unit flow over an edge, and
$u : E \rightarrow \mathbb{R}$	the flow *capacity* of an edge, and
$b : V \rightarrow \mathbb{R}$	the flow *supply* to a node in the network.

□

A node in a network with a positive supply is called a *source* of the network. A node in a network with a negative supply, or demand, is called a *sink* of the network. Furthermore, the number of nodes in V is denoted by n, and the number of edges in E by m.

The (minimum-cost) network flow problem is formulated as follows.

Definition 4.3 (network flow problem (NFP)). Given are a network $G = (V, E)$, a cost function c, a capacity u, and a supply b. The problem is to find a flow $f(e) \in \mathbb{R}$ for each $e \in E$, for which

$$\sum_{u:e=(v,u)\in E} f(e) - \sum_{u:e=(u,v)\in E} f(e) = b(v) \qquad \text{for each } v \in V, \text{ and}$$
$$0 \leq f(e) \leq u(e) \qquad \text{for each } e \in E, \text{ and}$$

that minimizes

$$\sum_{e\in E} c(e) f(e).$$

□

From its definition it follows that the network flow problem is a special case of the *linear programming* (LP) problem because linear expressions in the decision variables are used to formulate the cost function and the constraints. It is characteristic for LP problems that a dual problem can be formulated [Papadimitriou & Steiglitz, 1982]. The dual variables for the network flow problem are defined as the *potentials* of the nodes. A difference in potential allows the flow of mass from one node to another. The dual network flow problem is formulated as follows.

Definition 4.4 (dual network flow problem (NFPD)). Given are a network $G = (V, E)$, a cost function c, a capacity function u, and a supply function b. The problem is to find potentials $p(v) \in \mathbb{R}$ for each $v \in V$ and slack variables $d(e) \in \mathbb{R}$ for each $e \in E$, for which

$$p(u) - p(v) - d(e) \leq c(e) \qquad \text{for each } e = (u, v) \in E, \text{ and}$$
$$d(e) \geq 0 \qquad \text{for each } e \in E, \text{ and}$$

that maximizes

$$\sum_{v\in V} b(v) p(v) - \sum_{e\in E} u(e) d(e).$$

□

The decision variables $d(e)$ are absent for an uncapacitated network flow problem. Furthermore, if a problem instance has only integer data, then a solution, if it exists, is integer.

Optimal flows correspond with optimal potentials if the following complementary slackness conditions are satisfied:

$$f(e) > 0 \Rightarrow p(u) - p(v) - d(e) = c(e)$$

$$d(e) > 0 \Rightarrow f(e) = u(e).$$

These conditions can be transformed into the following conditions by using the *reduced cost*, which is defined for each edge $e = (u, v) \in E$ as $\tilde{c}(e) = c(e) - p(u) + p(v)$, and by using the feasibility condition of the dual network flow problem,

$$\tilde{c}(e) > 0 \Rightarrow f(e) = 0,$$

$$\tilde{c}(e) = 0 \Rightarrow 0 \leq f(e) \leq u(e), \text{and}$$

$$\tilde{c}(e) < 0 \Rightarrow f(e) = u(e).$$

A primal-dual pair of solutions is optimal if the complementary slackness conditions hold and both the primal and dual solutions are feasible.

Some network flow algorithms operate on a so-called *residual network*, which is constructed from the normal network by adding for each edge with a flow greater than zero a reversing edge and by removing the edge if the flow is maximum. The capacity of the reversing edge is equal to the negative flow and the capacity of the normal edge becomes the maximum minus the flow. Note that the normal edge disappears when its flow is maximum.

Before discussing solution methods for the minimum-cost network flow problem we first discuss special cases of it and dedicated solution methods for these cases.

4.2.2 Shortest path algorithms

If an instance of the network flow problem is such that $b(s) = |V| - 1$ for some node $s \in V$, $b(v) = -1$ for all other nodes $v \in V$, and $u(e) = |V|$ for each $e \in E$, then the solution contains a shortest path from the start node s to all other nodes, with the cost of an edge representing the distance between the two incident nodes. In the solution to the minimum-cost network-flow problem, a flow of one unit is sent over the shortest path from s to every other node. Although we can use solution algorithms for the network flow problem, special algorithms are developed for the shortest path problem.

Label-setting algorithms

The first class of algorithms we discuss is called label-setting algorithms. During the execution of the algorithm, a label pertaining to a node is set that represents the shortest distance from the start node to that node. This class of algorithms is applicable for network flow problems having non-negative costs of edges. A well

known and widely applicable example of a label-setting algorithm was introduced by Dijkstra [1959]; see Figure 4.3.

During the execution of the DIJKSTRA'S algorithm the nodes are partitioned into two sets: permanently labeled nodes and temporarily labeled nodes. The algorithm starts with all nodes belonging to the temporarily labeled set and labels them infinite, except for the start node which has a label equal to zero. At every step of the algorithm a node with the smallest label from the temporarily labeled set is selected and moved to the permanently labeled set. An adjacent node succeeding the selected one gets its label updated if a shortest path to the adjacent node goes along the selected one. Together with a label update, an index to the node is set to the selected node, which is then preceding the node on the shortest path. The algorithm ends when no temporarily labeled nodes are left.

---

procedure DIJKSTRA'S
in a network $G = (V, E)$ with labels c,
a start node $s \in V$;
out potentials $p(v)$ representing for each $v \in V$ the shortest distance
from the start node,
predecessors $pred(v)$ indicating for each $v \in V$ the shortest path
back to the start node;
begin $P = \emptyset$;
$T = V$;
$p(s) = 0$;
$p(j) = \infty$ for each $j \in V \setminus s$;
$pred(s) = 0$;
while $P \neq V$
begin select $i \in T$ with $p(i) = \min_{j \in T} p(j)$;
$P = P \cup \{i\}$;
$T = T \setminus \{i\}$;
for each $(i, j) \in E$
if $p(j) > p(i) + c((i, j))$ **then**
begin $p(j) = p(i) + c((i, j))$;
$pred(j) = i$;
end;
end;
end;

Figure 4.3. The DIJKSTRA'S algorithm.

The time spent by the algorithm can be split into two parts: the updating of labels and the selection of nodes. In one iteration, the algorithm requires $O(n)$ time to select a best temporarily-labeled node by linearly traversing the nodes. This can be improved to $O(log(n))$ when a more advanced algorithm is used, in which the

temporarily-labeled nodes are kept sorted. The algorithm requires $O(m)$ time for updating temporary labels. Therefore the algorithm runs in total in $O(nlog(n)+m)$ time.

When the network is acyclic, the nodes can be ordered in topological order in $O(m)$ time. Compared with the DIJKSTRA'S algorithm, this algorithm has the advantage that the node selection can be done much faster because the ordering of the temporarily labeled nodes can now remain fixed during the execution of the algorithm. Then, the algorithm requires $O(n+m)$ time.

Label-correcting algorithms

The second class of algorithms is called label-correcting algorithms. The concept on which this class of algorithms is based was introduced by Bellman [1958] and Ford & Fulkerson [1962]. Only at the end of the execution of the algorithm is the label pertaining to each node set, representing the shortest distance from the start node to that node. Although the labels are set during the algorithm, they can be corrected several times. These algorithms can handle networks with negative costs. However, if a network contains negative cycles, many nodes cannot be part of a shortest path. Therefore, many label-correcting algorithms have built-in means to detect the existence of negative cycles. An example is the *modified label-correcting* algorithm, which we discuss here in more detail; see Figure 4.4.

The MODIFIED_LABEL_CORRECTING algorithm starts with all nodes labeled infinite except for the start node, which is labeled zero. During the execution of the algorithm a list is kept of the nodes for which the labels have been updated but hose of adjacent nodes ahve not. At the start of the algorithm the list contains the start node only. In each iteration of the algorithm a node is fetched and removed from the list. All the adjacent nodes succeeding the fetched one get their label updated if the shortest path goes along the fetched node. The updated nodes are added to the list and treated in succeeding iterations of the algorithm. The MODIFIED_LABEL_CORRECTING algorithm runs in $O(nm)$ time.

To detect negative cycles, we also keep track of the number of times that each node's label is updated. If this number exceeds nm, then a negative weight cycle must be present.

A problem related to the shortest path problem is the longest path problem. It can easily be mapped onto the shortest path problem by negating the costs in the network. Then it is very probable that the network contains edges with negative costs and, consequently, a label-correcting algorithm is to be used.

4.2.3 Maximum-flow algorithms

The maximum-flow problem is to send the maximum possible flow in a network containing edges with equal costs from a source node to a sink node. If an instance

```
procedure MODIFIED_LABEL_CORRECTING
in  a network G = (V, E) with labels c and
    a start node s ∈ V;
out potentials p(v) representing for each v ∈ V the shortest distance
    from the start node, and
    predecessors pred(v) indicating for each v ∈ V the shortest path
    back to the start node;
begin p(s) = 0;
    p(j) = ∞ for each j ∈ V \ {s};
    pred(s) = 0;
    L = {s};
    while L ≠ Ø
    begin get first i ∈ L;
        L = L \ {i};
        for each (i, j) ∈ E
            if p(j) > p(i) + c((i, j)) then
                begin
                    p(j) = p(i) + c((i, j));
                    pred(j) = i;
                    L = L ∪ {j};
                end;
    end;
end;
```

Figure 4.4. The MODIFIED_LABEL_CORRECTING algorithm without negative weight cycle detection.

of the network flow problem is such that an edge from the sink to the source is present with an infinite capacity and a negative cost, then the solution maximizes the flow through the edge and consequently through the network. Furthermore, the nodes have a supply equal to zero.

We discuss the MAXIMUM_FLOW algorithm that operates on a residual network [Ford & Fulkerson, 1962]; see Figure 4.5. A straightforward algorithm to achieve maximum flow is to send flow along a directed path from the source to the sink in the residual network. The flow is added to the flow that was already flowing and in the case of a reversing edge the flow is decreased. As a consequence of the change in flow, the residual network is then updated. The algorithm repeats the augmentation until no path from the source to the sink can be found anymore.

For each flow augmentation we apply a so-called labeling algorithm, which starts at the only labeled node, the source. It performs a search from a labeled node to the sink node by searching for an adjacent unlabeled node to which it can send more flow via an edge. If such an unlabeled node is found it is labeled and the search is continued from that node. Otherwise the algorithm selects a labeled node from which not yet all adjacent nodes were searched. In this way a tree of labeled nodes is constructed. If the sink gets labeled, then the flow is increased, all labels are erased and the search is started again at the source. If the sink cannot be reached by the labeling process, then the maximum flow has been reached. During the labeling process, the predecessor of each labeled node is stored such that we can easily trace back the path from the sink to the source. The flow augmentation is determined by the smallest capacity of edges on the path from the source to the sink in the residual network.

At the end of the execution of the MAXIMUM_FLOW algorithm a *minimum cut* set of edges is derived, which is a bottleneck for the flow from source to sink. All the labeled nodes are in front of the minimum cut set and all the unlabeled ones are behind it. Each edge from a labeled node to an unlabeled node has a maximum flow, while each edge in the other direction has zero flow.

4.2.4 Minimum-cost flow algorithms

Here we discuss the network flow problem for uncapacitated networks only. In this case the variable $d(e)$ is excluded from the dual problem formulation. We discuss the *minimum-cost maximum-flow* algorithm and a more general applicable algorithm, the *out-of-kilter* algorithm [Ford & Fulkerson, 1962]. A *reweighting* algorithm is discussed that transforms an arbitrary network into a network with non-negative edge costs, because that is presupposed by the minimum-cost maximum-flow algorithm [Edmonds & Karp, 1972].

We construct a new network from the network given in an instance of the network flow problem by adding two additional nodes to the network to model the

```
procedure MAXIMUM_FLOW
in  a (residual) network G = (V,E) with labels c, u, and
    a source and a sink node;
out flows f(e) for each e ∈ E;
begin f((i,j)) = 0 for each (i,j) ∈ E;
    repeat
        pred(i) = 0 and unlabel i for each i ∈ V;
        L = {source};
        while L ≠ Ø and sink is unlabeled
        begin get first i ∈ L;
            L = L \ {i};
            for each (i,j) ∈ E
                if j is unlabeled then
                    begin L = L ∪ {j};
                        label j;
                        pred(j) = i;
                    end;
        end;
        increase flow from source to sink on path indicated by pred labels;
        update residual network;
    until there exists no path from source to sink in residual network;
end;
```

Figure 4.5. The MAXIMUM_FLOW algorithm.

supply and demand; see Figure 4.6. A source node is added to the network and from this source node an edge is connected to each node with a positive supply. The capacity of the edge is made equal to the specified supply and its cost is made equal to zero. Furthermore, a sink node is added and from each node with a negative supply an edge is connected to the sink. The capacity of the edge is made equal to the specified demand and its cost is made equal to zero. Nodes with zero supply do not need to be connected to the sink or the source. Clearly, a feasible flow in the original network corresponds with a maximum flow on the newly added edges. Therefore the algorithm that achieves this is called the minimum-cost maximum-flow algorithm. To force a maximum flow through the newly created network from source to sink, the network is extended with a return edge from sink to source. In this way the network becomes a closed circuit. The return edge is given an enormous negative cost to make it act like a pump. As long as the flow through the network is not maximal, the cost can be decreased by increasing the flow over the return edge. When this is done, the flow through the subnetwork consisting of the original edges and nodes will have minimal cost.

Minimum-cost maximum-flow algorithm

The MINIMUM_COST_MAXIMUM_FLOW algorithm solves the primal and dual network flow problem simultaneously. As a start solution we make the potentials equal to zero and the flow equal to zero. Furthermore, we restrict the cost on the edges in the original network to be non-negative. In other words, the dual start solution corresponding to the nodes in the original network must be feasible. Consequently, all the complementary slackness conditions are satisfied except for the one corresponding to the return edge. In order to satisfy its complementary slackness conditions, the flow is increased and the potentials are updated in alternating order.

The flow is increased such that complementary slackness conditions corresponding to edges other than the return edge do not become violated. To this end, flow may only increase or decrease on edges with $\tilde{c}(e) = 0$, which are called *admissable*. We use the MAXIMUM_FLOW algorithm, modified for admissable edges, to determine a maximum flow through these edges. Then, a minimum cut divides the nodes into two sets: one set with labeled nodes and the other with unlabeled nodes. Without violating the complementary slackness conditions of any edge except for the return edge, we can increase the potential of the labeled nodes, including the source, by one. Now edges in the minimum cut set that had $\tilde{c}(e) = 1$ get $\tilde{c}(e) = 0$, and thus become admissable. In that case their flow may become positive in a following step of the algorithm without violating the complementary slackness condition of an edge.

The increase in flow and the update of potentials continues until only the source

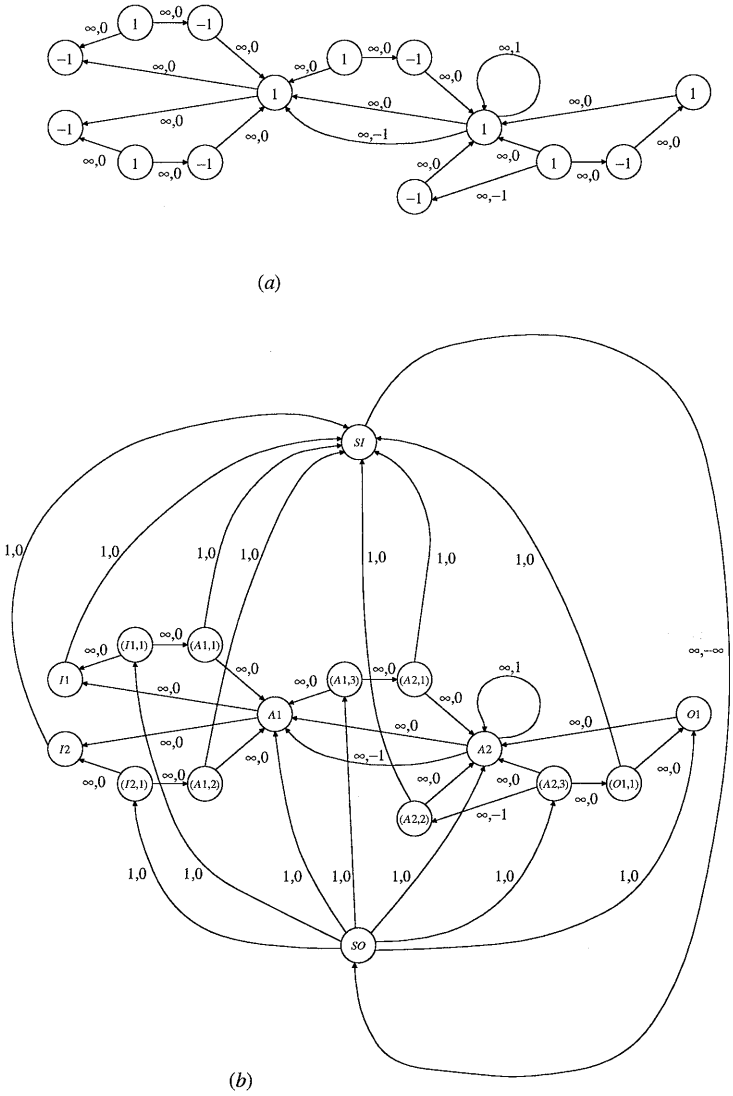


Figure 4.6. An example of an original network (*a*) and its extended version (*b*) for the application of the MINIMUM_COST_MAXIMUM_FLOW algorithm. For each edge in the network the capacity and the cost of an edge are shown and for the nodes in the original network of (*a*) the supply is shown. The network corresponds with an instance of SRP of which the SFG is shown in Figure 1.1*a* with one speed constraint present. In (*b*) the names of the operations and terminals to which respectively the retiming and register allocations belong are shown. Furthermore, *SI* denotes the sink and *SO* denotes the source.

node can be reached for an increase in flow. At this point an optimal solution of an instance of the network flow problem has been found.

In every step of the algorithm we have a dual feasible solution available for the nodes in the original network. The dual cost decreases with every update of the potentials. The algorithm iteratively improves the dual solution and only visits dual feasible solutions of the nodes in the original network.

Out-of-kilter algorithm

The MINIMUM_COST_MAXIMUM_FLOW algorithm is a special case of the OUT_OF_KILTER algorithm. This algorithm is more generally applicable since it requires neither a feasible start solution nor an uncapacitated original network. Furthermore, it does not distinghuish a sink or a source from other nodes. However, it maintains the mass balance by requiring that the amount of mass flowing into a node added to its supply be equal to the amount of mass flowing out of the node.

An edge is called out-of-kilter if its complementary slackness conditions are violated. To repair these conditions, the out-of-kilter algorithm iteratively satisfies the conditions for one edge in one step of the algorithm; see Figure 4.7. In one step, the MINIMUM_COST_MAXIMUM_FLOW algorithm is applied with the nodes at both ends of the out-of-kilter edge acting as source and sink nodes. The flow on the out-of-kilter edge must be either increased or decreased, depending on the relation between flow and reduced cost on the edge. If the flow has to be decreased, the originating node acts as the sink, otherwise it acts as the source.

```
procedure OUT_OF_KILTER
in  a network G = (V, E) with labels c, u,
    a source and a sink node;
out flows f(e) for each e ∈ E;
begin while there exists an e = (i, j) ∈ E that is out-of-kilter
    begin assign i and j as source and sink,
        depending on relation between flow and reduced cost;
        while e is out-of-kilter
        begin
            force maximum flow from source to sink on admissable edges;
            increase potentials of labeled nodes such that
                no edge in the cut set becomes out-of-kilter;
        end;
    end;
end;
```

Figure 4.7. The OUT_OF_KILTER algorithm.

Ideally, the OUT_OF_KILTER algorithm can be applied to incrementally compute solutions. When the cost of an edge is changed in a network, or when an edge is added to a network of which the optimal solution was known, the OUT_OF_KILTER algorithm can be used to quickly compute the optimal solution of the new network.

Reweighting

The potentials $p(v)$ in an instance of the network flow problem express changes with respect to an initial situation. For instance, were an optimal solution to be offered to a network flow algorithm, than the decision variables could all be zero. We use this property to apply the MINIMUM_COST_MAXIMUM_FLOW algorithm to networks with negative edge costs. First we determine potentials for the nodes in a network G such that

$$p(u) - p(v) \leq c(e) \qquad \text{for each } e = (u, v) \in E.$$

The shortest path algorithm MODIFIED_LABEL_CORRECTING for networks with negative distances (costs) can be used for this purpose. As a consequence, the reduced network gets edges with positive reduced costs.

In the second step we use the resulting reduced network as input to the MINIMUM_COST_MAXIMUM_FLOW algorithm. To compute the final potential of a node, we must add the potential computed in the second step to the potential computed in the first step.

4.2.5 Maximum-weight bipartite matching problem

A matching of a bipartite graph is a subset of edges such that each node from either set of nodes is incident to at most one edge of the matching. If a weight is associated with each edge, a matching has a maximum weight if the sum of the weights of the edges of the matching is maximal. The maximum-weight bipartite matching problem is defined as follows [Kuhn, 1955].

Definition 4.5 (maximum-weight bipartite matching problem (MWBMP)). Given are a bipartite graph $G = (V_1, V_2, E)$ and a weight function c. The problem is to find variables $x(e) \in \mathbb{N}$ for each $e \in E$, for which

$$\sum_{u:e=\{u,v\}\in E} x(e) \leq 1 \qquad \text{for each } u \in V_1, \text{ and}$$

$$\sum_{v:e=\{u,v\}\in E} x(e) \leq 1 \qquad \text{for each } v \in V_1, \text{ and}$$

$$x(e) \geq 0 \qquad \text{for each } e \in E, \text{ and}$$

that maximizes

$$\sum_{e\in E} c(e)x(e).$$

□

Although MWBMP is also known as the *assignment problem*, we do not use this latter name, to avoid confusion with the assignment problem defined in Chapter 3.

Special algorithms are available to solve instances of MWBMP in polynomial time [Kuhn, 1955]. Here, we use the dual problem formulation to reduce it to the network flow problem.

Definition 4.6 (dual maximum-weight bipartite matching problem). (MWBMPD).
Given are a bipartite graph $G = (V_1, V_2, E)$ and a weight function c. The problem is to find variables $p(v) \in \mathbb{Z}$ for each $v \in V_1 \cup V_2$ for which,

$$p(u) + p(v) \geq c(e) \qquad \text{for each } e = \{u, v\} \in E, \text{ and}$$
$$p(v) \geq 0 \qquad \text{for each } v \in V_1 \cup V_2, \text{ and}$$

that minimizes

$$\sum_{v \in V_1 \cup V_2} p(v).$$

□

To reduce MWBMPD to NFPD we extend the graph with an *offset* node w and introduce a substitution function $t : V_1 \cup V_2 \cup \{w\}$ that is defined as follows:

$$t(u) = t(w) - p(u) \qquad \text{for each } u \in V_1, \text{ and}$$
$$t(v) = t(w) + p(v) \qquad \text{for each } v \in V_2.$$

Now we can reformulate the constraints of MWBMPD into

$$t(u) - t(v) \leq -c(e) \qquad \text{for each } e = (u, v) \in E, \text{ and}$$
$$t(u) - t(w) \leq 0 \qquad \text{for each } u \in V_1, \text{ and}$$
$$t(w) - t(v) \leq 0 \qquad \text{for each } v \in V_2.$$

Furthermore, we can reformulate the cost of MWBMPD into

$$\sum_{v \in V_2} t(v) - \sum_{u \in V_1} t(u) + (|V_1| - |V_2|)t(w).$$

After the above reformulation, MWBMPD has become equivalent to the dual of a network flow problem. The set of nodes is $V_1 \cup V_2 \cup \{w\}$. The set of edges is E extended with edges corresponding to the constraints involving the potential of the offset node. The edges have become directed from the node in V_1 to the node in V_2 to which it is incident. Consequently, the maximum-weight bipartite matching problem can be solved in polynomial time. To solve one of its instances, we can use, for example, the MINIMUM_COST_MAXIMUM_FLOW algorithm.

5

Operator Assignment

From Chapter 3 we recall that the operator assignment problem (OASP) can be formulated as follows.

Definition 5.1 (operator assignment problem (OASP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, with folding factors f_i, $i = 1, \ldots, N$, a type set $\mathcal{T}$, a type function ℓ, and a minimal operator allocation al. The problem is to find a PU $\mathcal{G}'$, constructed by an operator assignment as and a multiplexer allocation ma, for which

(ii)	$\ell(v) = \ell(as(v))$	for each $v \in V$, and
(iii)	$\lvert\{v \in V_i \lvert as(v) = v'\}\rvert \leq f_i$	for each $v' \in V'$, and for each $i = 1, \ldots, N$, and
(xii)	$((as(v), a), (as(u), b)) \in E'$	for each $((v, a), (u, b)) \in E$, and

that minimizes

$$\sum_{i' \in I'} ma(i').$$

□

The NP-hardness of OASP, as stated by Theorem 3.4, and the large size of practical instances force us to search for effective and efficient approximation algorithms. Here we present an approach based on local search algorithms, namely the ITERATIVE_IMPROVEMENT and SIMULATED_ANNEALING algorithms as discussed in

Section 4.1. This approach has been reported before by Van der Werf, Peek, Aarts, Van Meerbergen, Lippens & Verhaegh [1992].

The organization of this chapter is as follows. In Section 5.1 a straightforward application of local search is discussed and some results are presented. In Section 5.2, we show that the run times for large source SFGs are unacceptable when the local search algorithms are applied in a straightforward fashion, and that the performance can be substantially improved by reducing the neighborhood structures based on detailed knowledge of the connectivity in PUs.

5.1 Local search

5.1.1 Solution space, cost, and neighborhood structure

For OASP, the solution space $\mathcal{S}$ is defined as the set of all PUs obtained by two functions, i.e., an operator assignment and a multiplexer allocation. However, when an operator assignment is given, an optimal multiplexer allocation can be found in a trivial way following directly from the definition of the multiplexer allocation, Definition 2.18.

A neighborhood structure can be chosen according to a *two-exchange* mechanism. Examples of the concept of *exchanges* can be found in [Kernighan & Lin, 1970; Aarts & Korst, 1989; Van Laarhoven & Aarts, 1987; Yannakakis, 1990]. The two-exchange mechanism that we use is called *swap* and is defined as follows.

Definition 5.2 (swap-neighborhood structure). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$ with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, and a minimal operator allocation al. The *swap-neighborhood structure* ($\mathcal{N}_{\text{swap}}$) is given as follows. The neighborhood $\mathcal{N}_{\text{swap}}(as)$ of an operator assignment as is the set of all operator assignments $as^\bullet$ for which there is an $i = 1,\ldots,N$ and one pair of operations $u, v \in V_i$ with $\ell(u) = \ell(v)$, such that

- $as^\bullet(u) = as(v)$ and $as^\bullet(v) = as(u)$ and
- $as^\bullet(w) = as(w)$ for each $w \in V \setminus \{u, v\}$.

□

This means that two operations of one source SFG may exchange their operator assignments, while for all the other operations the assignment remains the same. The swap-neighborhood structure is defined such that Constraints (*ii*) remain satisfied. If the start solution satisfies Constraints (*iii*), it remains satisfied when the above two-exchanges are applied.

The definition of the swap-neighborhood structure implies that some solutions in the neighborhood graph cannot be reached from all of the other solutions in the

graph. This occurs when different source SFGs have different numbers of operations of the same type. In this situation, there can be an operator to which no operation of a certain source SFG is assigned. Then a sequence of two-exchanges cannot get an operation from that source SFG to be assigned to that operator. To ensure strongly-connectedness of the neighborhood graph, we add the following neighborhood, which is based on a *one-exchange* mechanism called *switch.*

Definition 5.3 (switch-neighborhood structure). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$ with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, and a minimal operator allocation al. The *switch-neighborhood structure* ($\mathcal{N}_{\text{switch}}$) is given as follows. The neighborhood $\mathcal{N}_{\text{switch}}(as)$ of an operator assignment as is the set of all operator assignments $as^\bullet$ for which there is an $i = 1,\ldots,N$, an operation $v \in V_i$, and an operator $v' \in V'$ with $\ell(v') = \ell(v)$ and $|\{u \in V_i | as(u) = v'\}| < f_i$, such that

- $as^\bullet(v) = v'$ and
- $as^\bullet(w) = as(w)$ for each $w \in V \setminus \{v\}$.

□

Now we can combine the swap- and the switch-neighborhood structures into one neighborhood structure $\mathcal{N}$ given by $\mathcal{N}(as) = \mathcal{N}_{\text{swap}}(as) \cup \mathcal{N}_{\text{switch}}(as)$. This neighborhood structure is strongly connected.

An example of two source SFGs and a corresponding PU on which they are mapped is shown in Figure 5.1. The solution on the left in this example shows the existence of a local minimum that is not a global minimum, since every possible two-exchange for the left solution results in a solution with a larger cost. Note that for this example one-exchanges are not applicable since the two source SFGs have the same number of operations of each type. For example, a two-exchange of operations $a1$ and $a6$ in $\mathcal{G}_2$ in the solution on the left results in the PU that is shown in Figure 5.1*d*. The cost has increased with 2 multiplexers in the PU. Since the source SFGs are identical in this trivial example, a global optimal solution has no multiplexers.

To start a walk in the neighborhood graph, a start solution has to be generated. This solution can be computed by a straightforward assignment of operations to operators of the same type such that it satisfies Constraints (*ii*) and (*iii*).

5.1.2 Implementation

We implemented the generation of a neighboring solution from the union of $\mathcal{N}_{\text{swap}}$ and $\mathcal{N}_{\text{switch}}$ in the following way. First, the sets of operations of each source SFG are augmented with a number of *dummy* operations, such that every operator has the maximum number of operations of each source SFG assigned to it, i.e.,

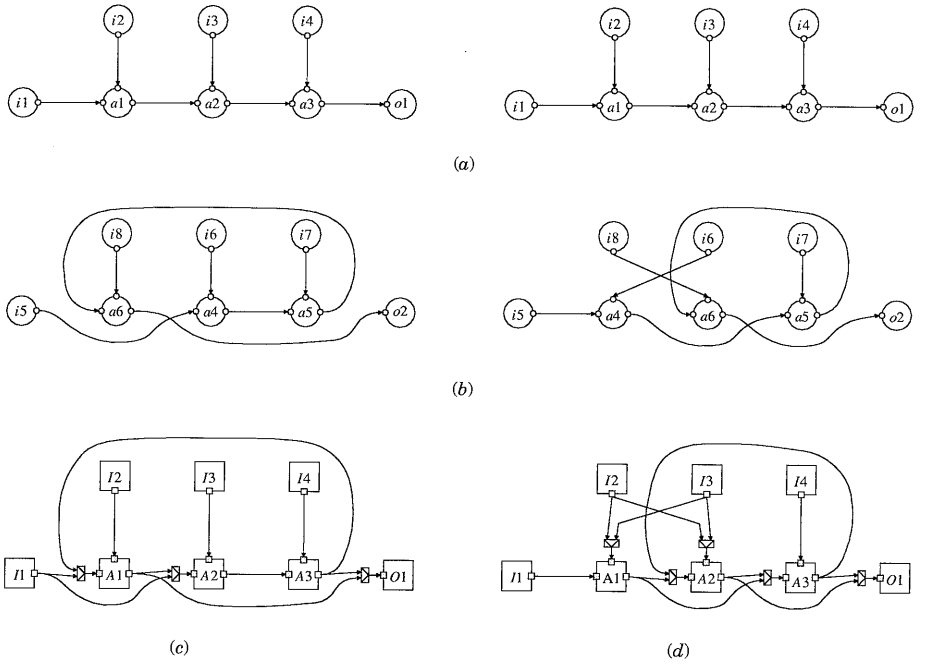


Figure 5.1. An example of two source SFGs, $\mathcal{G}_1$ (a) and $\mathcal{G}_2$ (b), and a PU, $\mathcal{G}'$ (c), on which they are mapped. In (a) and (b) the source SFGs are shown twice, such that operations in $\mathcal{G}_1$ and $\mathcal{G}_2$ are placed above the operator in the PU to which they are assigned. For example, $A1 = as(a1) = as(a6)$ in the solution on the left. A two-exchange of operations $a4$ and $a6$ in $\mathcal{G}_2$ results in the PU of (d).

$|\{as(v_i) = v' | v_i \in V_i\}| = f_i$ for each $v' \in V'$ and each $i = 1, \ldots, N$. Like all operations, each dummy operation belongs to a certain operation type corresponding to the operator to which it is assigned. Dummy operations have no connections and consequently do not influence the cost of solutions. Second, we apply two-exchanges only, where selected operations can be dummy operations. The generation of a neighboring solution starts with the random selection of an operation from the union SFG. Next, another operation that is of the same type and belongs to the same source SFG is randomly selected.

The computation of the cost function of a proposed solution is performed by actually executing the two-exchange. If the new solution is rejected, the two-exchange is performed again to return to the current solution.

In the implementation we deviate slightly from the model presented in Chapter 2, by replacing edges with identical sources and destinations in the target SFG by one single edge. Constraints (*iv*) are used to construct the set of edges E' in a target SFG. Since a minimal operator allocation is given, each operator has at least one operation assigned to it. Furthermore, we assume that each input terminal has at least one edge incident to it, whereas an output terminal can be unconnected. Consequently, there is a linear dependency between the number of two-input multiplexers and the number of edges in E', i.e.,

$$\begin{aligned}\sum_{i' \in I'} ma(i') &= \sum_{i' \in I'} (|\{o' \in O' | (o', i') \in E'\}| - 1)^+ \\ &= \sum_{i' \in I'} (|\{o' \in O' | (o', i') \in E'\}| - 1) \\ &= |E'| - |I'|.\end{aligned}$$

To compute the cost, we use the linear relation between the number of two-input multiplexers and the number of edges in E'. We do not need to compute the cost of every visited solution from scratch, since only the number of edges connected to the involved operators may change. So, the difference in cost is given by

$$|\{((u', a), (v', b)) \in E' \mid as'(u) = u' \text{ or } as'(v) = v'\}| -$$
$$|\{((u^\bullet, a), (v^\bullet, b)) \in E^\bullet \mid as^\bullet(u) = u^\bullet \text{ or } as^\bullet(v) = v^\bullet\}|,$$

where $as^\bullet$ is a neighbour of as by exchanging the operator assignment of operations u and v, and $E^\bullet$ is the set of edges in the target SFG determined by $as^\bullet$. Using consecutive cost differences, the cost can be efficiently computed in an incremental way. Using the above we implemented two local search algorithms, namely SIMULATED_ANNEALING and ITERATIVE_IMPROVEMENT. Note that these local search algorithms require only the difference in cost.

5.1.3 Experimental results

As an example we use instance IOASP1 of OASP, which is defined as follows. Given are two source SFGs of a Discrete Cosine Transform (DCT) $\mathcal{G}_1$ and an Inverse Discrete Cosine Transform (IDCT) $\mathcal{G}_2$ [Rao & Yip, 1990]; see also Figure 5.2. The two functions are used in a digital VCR application in which either the DCT is performed during recording or the IDCT is performed during play back [Borgers, Heijnemans, De Niet & De With, 1988].

Both SFGs contain the following operations: 16 multiplications ($M1,\ldots,M16$), 32 additions/subtractions ($A1,\ldots,A32$), 8 inputs ($I1,\ldots,I8$), and 8 outputs ($O1,\ldots,O8$). The constant inputs of the multipliers are internally connected. The addition and subtraction operations are considered to be of the same type. The SFGs differ in connectivity between the operations, although the number of edges is equal, i.e., $|E_1| = |E_2| = 88$. Furthermore, the minimal allocation of operators is given, consisting of 16 multipliers, 32 adders/subtracters, 8 inputs, and 8 outputs.

Table 5.1. Results of local search for the mapping of a DCT and an IDCT onto a PU. Average values were obtained for 100 runs.

local search variant	cost (multiplexers)			average CPU time (sec.)
	best	average	worst	
SIMULATED_ANNEALING	43	46	52	159
ITERATIVE_IMPROVEMENT	44	48	55	100

Results of the SIMULATED_ANNEALING and the ITERATIVE_IMPROVEMENT algorithms for this example are listed in Table 5.1. Average values were obtained for 100 runs to average out statistical fluctuations of both algorithms. The best solution obtained requires 43 multiplexers. This is about half the number of multiplexers required in the case of the average of arbitrary feasible solutions, which is 84 multiplexers. From these results we conclude that local search gives good solutions. Moreover, we see that the SIMULATED_ANNEALING algorithm has a slightly better average performance than the ITERATIVE_IMPROVEMENT algorithm. The run times of both algorithms for this example are within acceptable limits.

5.2 Large SFGs

In this section we discuss the application of local search with the straightforward neighborhood structures as defined in Section 5.1 to map large source SFGs onto a PU. First some experimental results are presented, from which we conclude that too much run time is needed and that solutions are of low quality. We present an analysis of the causes of these bad results, which led us to use a reduced neighborhood structure.

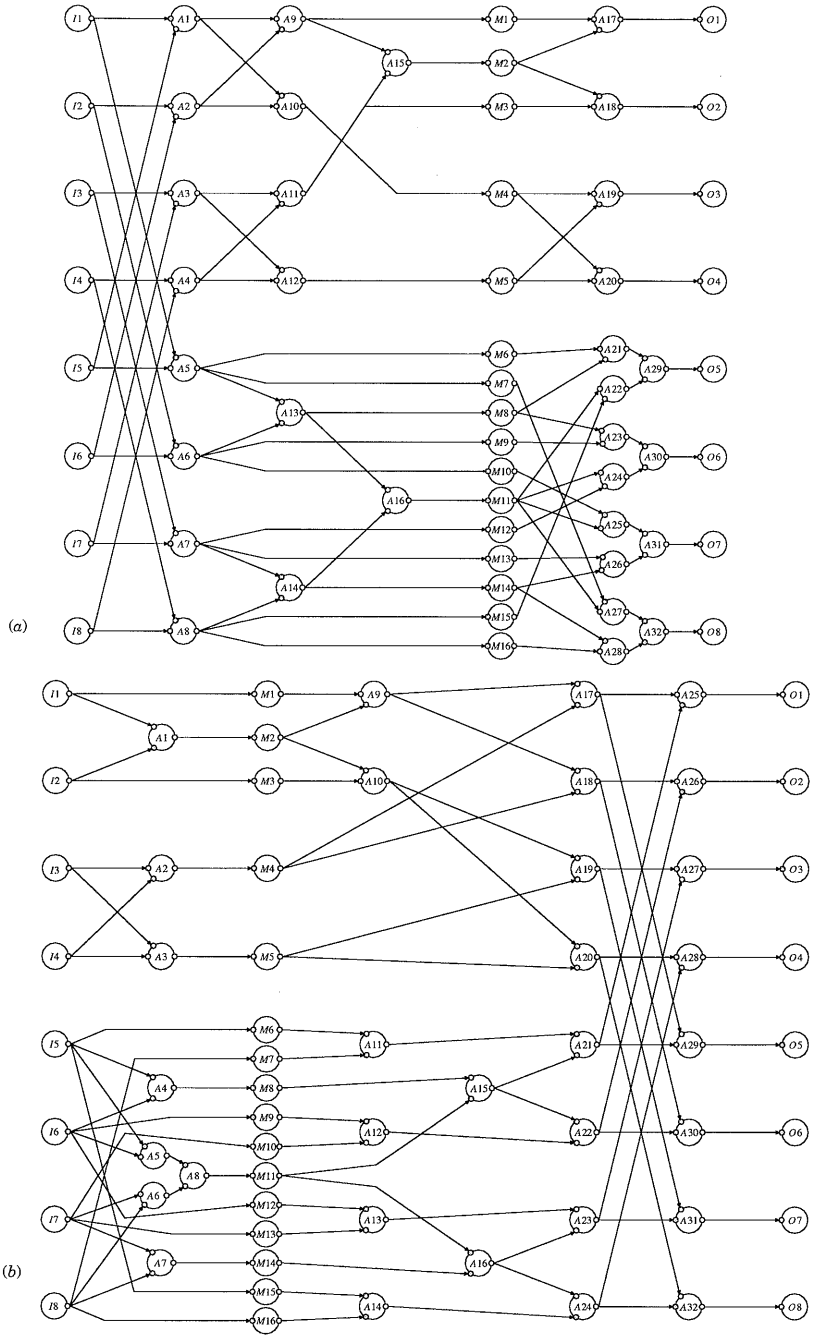


Figure 5.2. The source SFGs $\mathcal{G}_1$ and $\mathcal{G}_2$ of a DCT (a) and an IDCT (b), respectively. The weights of all signal edges are equal to zero.

5.2.1 First experimental results

We consider the following instance IOASP2 of OASP. Given are two large, identical SFGs $\mathcal{G}_3$ of multiplications, that are 12x12 bit, modified Booth, and modelled at the bit level [Booth & Booth, 1965]. The operations consist of *full adder*s, *and*s, and other logic gates. Also given is the minimal number of required operators. This instance has been selected because we know the cost of an optimal solution, namely that the target SFG has the same number of edges as each of the source SFGs, i.e., $|E'| = |E_3| = 1526$, and no multiplexers are allocated. We applied the SIMULATED_ANNEALING algorithm, which ran for about three days (CPU time) on a workstation and resulted in a solution with a cost of about 500 multiplexers. The cost as a function of time, measured as the number of two-exchanges executed, is shown in Figure 5.3 as curve (*a*).

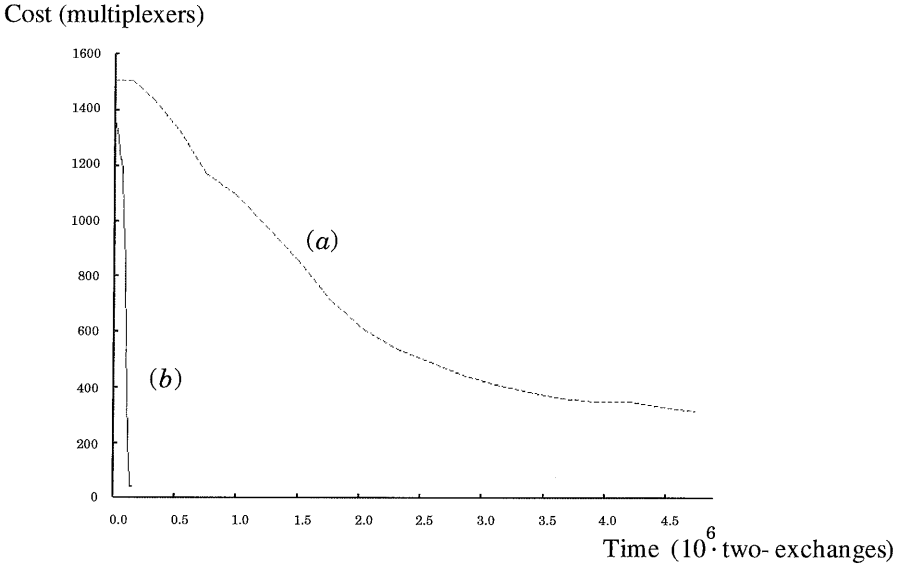


Figure 5.3. Cost as a function of time for the straightforward (*a*) and the reduced neighborhood structures (*b*) during a run of the SIMULATED_ANNEALING algorithm for problem instance IOASP2.

We conclude that the run time of the program as well as the quality of the solution are unacceptable for this large instance of OASP.

5.2.2 Analysis

We first analyze the acceptance ratio for a constant value of c_k, which is defined as the number of accepted solutions divided by the number of generated solutions.

The latter is equal to the chain length L_k. For the experiment of Section 5.2.1, which uses problem instance IOASP2, the acceptance ratio as a function of the cost is shown in Figure 5.4 as curve (*a*).

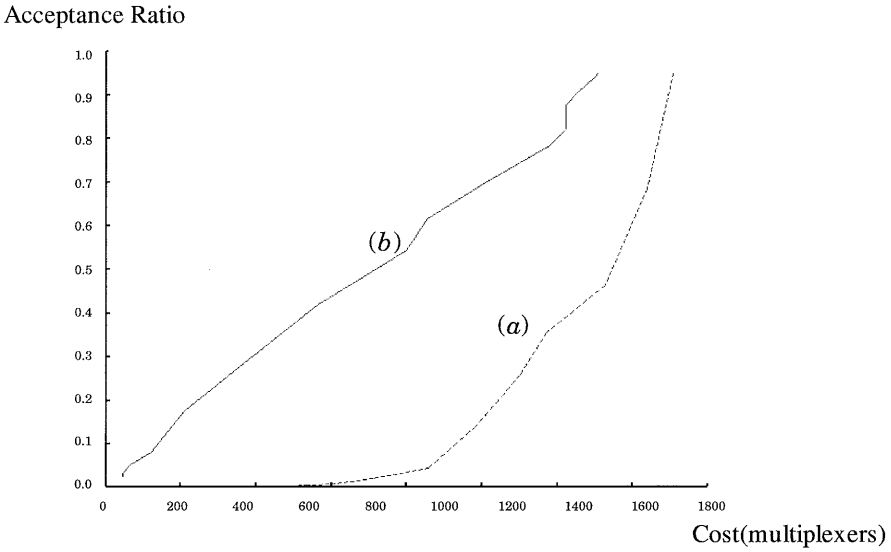


Figure 5.4. Acceptance ratio as a function of the cost for the straightforward (*a*) and the reduced neighborhood structures (*b*) during a run of SIMULATED_ANNEALING algorithm for problem instance IOASP2.

For solutions with about 500 multiplexers, which is much larger than the optimal number of zero, the acceptance ratio is about 0.002, which is very low. Therefore only a few solutions are accepted and much run time is wasted by consecutively executing two identical two-exchanges as a result of a rejected solution.

5.2.3 Neighborhood reductions

We reduce the neighborhoods $\mathcal{N}_{\text{swap}}$ and $\mathcal{N}_{\text{switch}}$ by excluding a number of transitions of which we know in advance that they cannot result in a cost improvement. A decrease of the number of edges can only be achieved by stepping to a neighboring solution that assigns yet another edge in E to an already existing edge in E'. In case of a two-exchange, the number of edges can only decrease by one of the three types of two-exchanges shown in Figure 5.5.

We reduce the neighborhoods to solutions that may lead to a cost improvement by using knowledge about the edge configuration in the target SFG. To this end, the three examples in Figure 5.5 are formalized and a reduced swap-neighborhood structure is defined as follows.

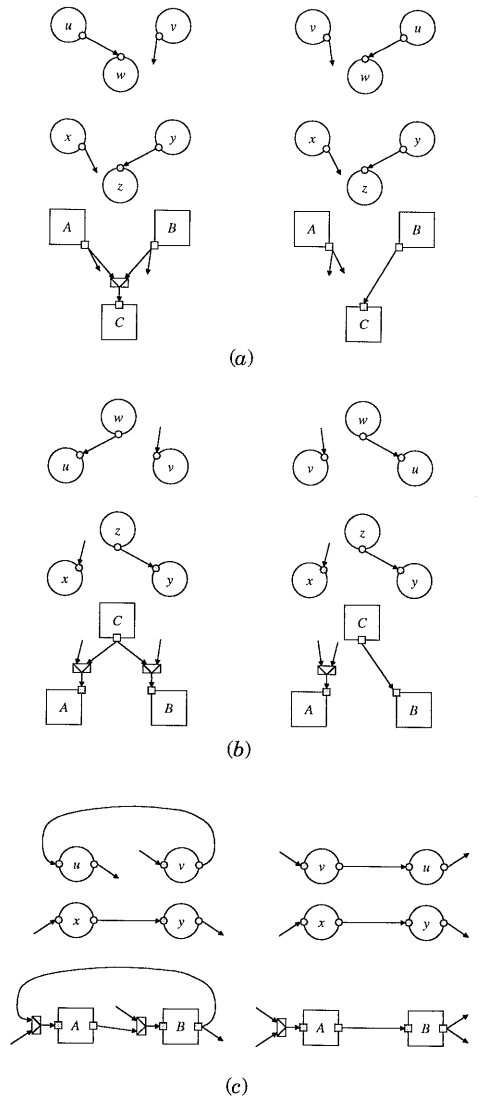


Figure 5.5. The three types, (*a*), (*b*), and (*c*), of two-exchanges of operations u and v in a source SFG that may result in better solutions. At the left, the solutions before the two-exchange, and at the right, the better solutions after the two-exchange. Operations that are assigned to the same operator are aligned vertically. Only relevant operations, signal edges, and terminals of the source SFGs and PUs are shown.

Definition 5.4 (reduced swap-neighborhood structure). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$ with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, and a minimal operator allocation al. The *reduced swap-neighborhood structure* ($\mathcal{N}_{\text{reduced swap}}$) is given as follows. The neighborhood $\mathcal{N}_{\text{reduced swap}}(as)$ of an operator assignment as is the set of all operator assignments $as^\bullet$ for which there is an $i = 1,\ldots,N$ and one pair of operations $u, v \in V_i$ with $\ell(u) = \ell(v)$, and

(*a*)	there exist such that	$(w',b) \in I'$, $a \in \mathcal{O}(\ell(u))$, $((as(u),a),(w',b)) \in E'$ and $((as(v),a),(w',b)) \in E'$, or
(*b*)	there exist such that	$(w',a) \in O'$, $b \in \mathcal{I}(\ell(u))$, $((w',a),(as(u),b)) \in E'$ and $((w',a),(as(v),b)) \in E'$, or
(*c*)	there exist such that	$b \in \mathcal{I}(\ell(u))$, $a \in \mathcal{O}(\ell(v))$ $((as(v),a),(as(u),b)) \in E'$ and $((as(u),a),(as(v),b)) \in E'$,

such that

- $as^\bullet(u) = as(v)$ and $as^\bullet(v) = as(u)$ and
- $as^\bullet(w) = as(w)$ for each $w \in V \setminus \{u, v\}$.

□

Note that the extra constraints (*a*), (*b*), and (*c*) for the operations of which the assignment is exchanged correspond with the three types shown in Figure 5.5*a*, 5.5*b*, and 5.5*c*, respectively.

Similarly, we reduce the switch-neighborhood structure into a reduced one, which is used together with the reduced swap-neighborhood structure.

Definition 5.5 (reduced switch-neighborhood structure). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$ with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, and a minimal operator allocation al. The *reduced switch-neighborhood structure* ($\mathcal{N}_{\text{reduced switch}}$) is given as follows. The neighborhood $\mathcal{N}_{\text{reduced switch}}(as)$ of an operator assignment as is the set of all operator assignments $as^\bullet$ for which there is an $i = 1,\ldots,N$, an operation $v \in V_i$, and an operator $v' \in V'$ with $\ell(v') = \ell(v)$ and $|\{u \in V_i | as(u) = v'\}| < f_i$, and

(*a*)	there exist such that	$(w',b) \in I'$, $a \in \mathcal{O}(\ell(v'))$, $((v',a),(w',b)) \in E'$ and $((as(v),a),(w',b)) \in E'$, or
(*b*)	there exist such that	$(w',a) \in O'$, $b \in \mathcal{I}(\ell(v'))$, $((w',a),(v',b)) \in E'$ and $((w',a),(as(v),b)) \in E'$, or

(c) there exist such that $b \in \mathcal{I}(\ell(v')), a \in \mathcal{I}(\ell(v))$ $((as(v), a), (v', b)) \in E'$ or $((v', a), (as(v), b)) \in E'$,

such that

- $as^{\bullet}(v) = v'$ and
- $as^{\bullet}(w) = as(w)$ for each $w \in V \setminus \{u, v\}$.

□

Now we can combine the reduced swap- and the reduced switch-neighborhood structures into one reduced neighborhood structure $\mathcal{N}_{\text{reduced}}$ given by $\mathcal{N}_{\text{reduced}}(as) = \mathcal{N}_{\text{reduced swap}}(as) \cup \mathcal{N}_{\text{reduced switch}}(as)$. The reduced neighborhood of a solution contains all solutions of the straightforward neighborhood that have a cost improvement. Since additional edges connected to other operator terminals may be created by the reduced one- or two-exchange, a reduced neighborhood may still contain solutions with cost deteriorations. The sizes of reduced neighborhoods are in general much smaller than the sizes of straightforward neighborhoods. Therefore, it is expected and has been confirmed by experiments, as shown in the next section, that both drawbacks of the straightforward defined swap- and switch-neighborhood structures, namely large CPU times and low acceptance ratios, are overcome by applying the reduced swap- and reduced switch-neighborhood structures.

5.2.4 Experimental results

Two multiplications. We use instance IOASP2 of OASP, as given in Section 4.1, to show that the application of the reduced neighborhood structure is effective when using the SIMULATED_ANNEALING algorithm. The cost as a function of time, which is measured in the number of executed two-exchanges, is shown in Figure 5.3 for the straightforward neighborhood structure as curve (*a*), and for the reduced neighborhood structure as curve (*b*). In this figure the dramatic improvement in run time by more than a factor of fifty and in the cost of the obtained solution are clearly visible. The acceptance ratio as a function of the cost is shown in Figure 5.4 as the curve (*b*). There is a remarkable improvement in the acceptance ratio compared to the straightforward neighborhood. Nevertheless, the optimal solution was not reached, since the run of the SIMULATED_ANNEALING algorithm got stuck at a low temperature in a local minimum that was not the optimal solution. The average size of the reduced neighborhoods was 6000, while for the straightforward neighborhoods it was 135,000.

DCT and IDCT. Using the reduced neighborhood structures, we preformed the same experiment as that of Section 5.1.3 again, using IOASP1, which is an instance of OASP.

Table 5.2. Results of local search for the mapping of a DCT and an IDCT onto a PU, using the reduced neighborhood structure. Average values were obtained for 100 runs.

local search variant	cost (multiplexers)			average CPU time (sec.)
	best	average	worst	
SIMULATED_ANNEALING	43	46	52	71
ITERATIVE_IMPROVEMENT	43	51	82	53

The results are shown in Table 5.2. Compared to the results in Table 5.1, the CPU time has decreased by a factor of two for this relatively small example. However, the decrease is less dramatic than it was for the merging of larger source SFGs in the previous example. The average result for the ITERATIVE_IMPROVEMENT algorithm has increased slightly and the worst number of multiplexers has increased significantly from 55 to 82. The latter can be explained as follows. Many edges in the neighborhood graph are removed by applying the reduced neighborhood structure. Many of these removed edges represent transitions without a change in cost. Since an iterative improvement walk on this neighborhood graph now has fewer possibilities to walk around without a change in cost, it can reach fewer solutions and consequently the probability that it will be stuck in a bad local minimum has increased. Since the results of the SIMULATED_ANNEALING algorithm in Table 5.1 and Table 5.2 are about equal, the advantage of the SIMULATED_ANNEALING algorithm over the ITERATIVE_IMPROVEMENT algorithm is clearly demonstrated when the reduced neighborhood structure is used.

FFT-butterfly and multiplication. As a third example, we use instance IOASP3, which is given as follows. Given are SFGs of an FFT butterfly $\mathcal{G}_4$ (radix-2, decimation in frequency, 8 bits) [Oppenheim & Schafer, 1975; Rabiner & Gold, 1975; Roberts & Mullis, 1987] and a multiplication $\mathcal{G}_5$ (18x18 bits, modified Booth) [Booth & Booth, 1965], and given is the minimal number of required operators. We expanded both SFGs to the bit level. This resulted in source SFGs with $|E_4| = 2213$, and $|E_3| = 1891$. The average cost of feasible solutions is 2550 multiplexers. Applying the SIMULATED_ANNEALING algorithm resulted in a PU with 1228 multiplexers. This is an improvement of 51% with respect to the average cost.

6

Operator Duplication

From Chapter 3 we recall that the operator duplication problem (ODP) can be formulated as follows.

Definition 6.1 (operator duplication problem (ODP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1, \ldots, N$, a type set $\mathcal{T}$, a type function ℓ, a delay del, the cost of a multiplexer β, an initial operator allocation al' and an initial operator assignment as' defining a PU $\mathcal{G}'$. The problem is to find a PU $\mathcal{G}^*$ constructed by an operator allocation al^*, an operator assignment as^*, and a multiplexer allocation ma^*, for which

(*ii*)	$\ell(v) = \ell(as^*(v))$	for each $v \in V$, and
(*x*)	$as^*(v) \neq as^*(u)$	for each $u, v \in V$ with $as'(v) \neq as'(u)$, and
(*xi*)	$C_{\mathcal{G}} = \emptyset$,	and
(*xii*)	$((as^*(v), a), (as^*(u), b)) \in E^*$	for each $((v, a), (u, b)) \in E$, and

that minimizes

$$\beta \sum_{i^* \in I^*} ma^*(i^*) + \sum_{j \in \mathcal{T}} al^*(j) C(j).$$

□

ODP is an NP-hard problem, as stated by Theorem 3.5, and consequently one may not expect that an efficient algorithm can be constructed that solves practical instances of ODP, which have a large size, to optimality. In this chapter we present an approximation method based on the SIMULATED_ANNEALING algorithm as discussed in Section 4.1.

The organization of this chapter is as follows. Section 6.1 reformulates ODP. In Section 6.2 the basic ingredients of a local search algorithm are discussed. In Section 6.3 the implementation of the algorithm is discussed, in particular the part in which complex circuits are to be found. In Section 6.4 we show experimental results.

6.1 Reformulating the operator duplication problem

For each complex circuit we can identify operators that link two paths. If operations assigned to such an operator are reassigned to different operators by operator duplication, the complex circuit disappears. This set of pairs of operations is called the *split set* of a complex circuit.

Definition 6.2 (split set). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, an initial operator assignment as' and a set of complex circuits $C_\mathcal{G}$. The *split set* is a function $SP : C_\mathcal{G} \rightarrow \mathcal{P}(V \times V)$, defined as $SP((((u_0,a_0) \mapsto (v_0,b_0)),\ldots,((u_{m-1},a_{m-1}) \mapsto (v_{m-1},b_{m-1})))) = \{(v_k, u_{(k+1) \bmod m}) \mid k = 0,\ldots,m-1\}$. □

We have to find an allocation and assignment such that one or more pairs of operations from the split set are reassigned to different operators. To compactly represent this we use the notion of *split number*, which is defined as follows.

Definition 6.3 (split number). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, an initial operator assignment as', a set of complex circuits $C_\mathcal{G}$, and an operator reassignment as^* satisfying

(xvi) $as^*(u) \neq as^*(v)$ for each $u, v \in V$ with $as'(u) \neq as'(v)$.

The *split number* is a function $sn : C_\mathcal{G} \rightarrow \mathbb{N}$, which is defined as

$$sn(c) = \mid \{(v,u) \in SP(c) \mid as^*(u) \neq as^*(v)\} \mid \qquad \text{for each } c \in C_\mathcal{G}.$$

□

Figure 6.1 shows two source SFGs with a complex circuit and the PU on which they are mapped with a corresponding complex circuit. Using the split number, Constraints (xi) are replaced by Constraints (xiv) in the following definition of ODP.

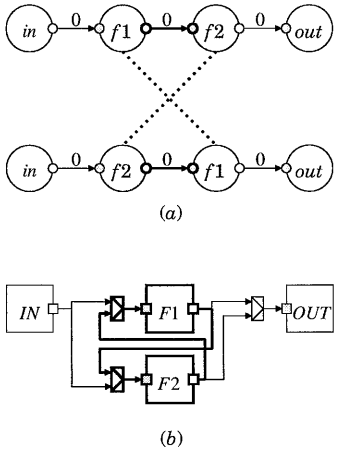


Figure 6.1. An example of two source SFGs (a) and the PU (b) on which they are mapped. A false loop in the PU is present, which is highlighted. The corresponding complex circuit in the SFGs is highlighted. The complex circuit's split set is indicated by shaded operations of which pairs are connected by dashed lines.

Definition 6.4 (operator duplication problem (ODP)). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, a delay del, the cost of a multiplexer β, an operator allocation al' and an operator assignment as' defining a PU $\mathcal{G}'$. The problem is to find a PU $\mathcal{G}^*$ constructed by an operator reallocation al^*, an operator reassignment as^*, and a multiplexer allocation ma^*, for which

(*ii*)	$\ell(v) = \ell(as^*(v))$	for each $v \in V$, and
(*x*)	$as^*(v) \neq as^*(u)$	for each $u, v \in V$ with $as'(v) \neq as'(u)$, and
(*xii*)	$((as^*(v),a),(as^*(u),b)) \in E^*$	for each $((v,a),(u,b)) \in E$, and
(*xiv*)	$sn(c) \geq 1$	for each $c \in C_{\mathcal{G}}$, and

that minimizes

$$\beta \sum_{i^* \in I^*} ma^*(i^*) + \sum_{j \in \mathcal{T}} al^*(j)\,C(j).$$

□

6.2 Solution space, cost, and neighborhood structure

For ODP, we define the solution space $\mathcal{S}$ as the set of all possible PUs defined by an operator reallocation, an operator reassignment, and a multiplexer allocation.

However, when an operator allocation and an operator assignment are given, an optimal multiplexer allocation can be found in a trivial way following directly from the definition of the multiplexer allocation. Therefore we limit ourselves to the decision variables *al* and *as* only. The solutions may result in complex circuits; however, the search for an efficient solution is started at a solution that contains no complex circuits. This solution can be easily found by fully duplicating all operators, i.e., for each operation of a union SFG an operator of the same type is allocated and the operation is reassigned it. A transition from one solution to another is performed by reassigning one operation to an operator, which may be newly allocated. We call this step a *switch*. If the operator to which the operation was assigned is left unoccupied, the operator may be discarded and the allocation can be decreased accordingly.

Definition 6.5 (switch-neighborhood structure). Given are a union $\mathcal{G}$ of source SFGs $\mathcal{G}_i$, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, a delay *del*, an initial operator allocation al' and an initial operator assignment as' defining a PU $\mathcal{G}'$. The *switch-neighborhood structure* ($\mathcal{N}_{\text{switch}}$) is given as follows. The neighborhood $\mathcal{N}_{\text{switch}}(al^*, as^*)$ of an operator allocation al^* and an operator assignment as^* is the set of all operator allocations $al^\bullet$ and operator assignments $as^\bullet$ for which there is a $v \in V$ and a $v' \in V'$, with $v' \neq as^*(v)$ and $\ell(v') = \ell(v)$, satisfying

$$
\begin{array}{ll}
as^\bullet(v) = v' & \text{and} \\
as^\bullet(u) = as^*(u) & \text{for each } u \in V \setminus \{v\} \text{ and} \\
as^\bullet(u) \neq as^\bullet(w) & \text{for each } u, w \in V \\
 & \text{with } as'(u) \neq as'(w) \text{ and} \\
al^\bullet(l(v)) = al^*(l(v)) + 1 & \text{if } |\{u \in V \setminus \{v\} | as^\bullet(v) = as^*(u)\}| = 0, \text{ or} \\
al^\bullet(l(v)) = al^*(l(v)) - 1 & \text{if } |\{u \in V \setminus \{v\} | as^*(v) = as^*(u)\}| = 0, \text{ and} \\
al^\bullet(j) = al^*(j) & \text{for each } j \in \mathcal{T} \setminus \{l(v)\}.
\end{array}
$$

□

The switch-neighborhood structure implies that one can step from one solution to another by duplicating, which increases the number of allocated operators, as well as by merging, which decreases the number of allocated operators. We still refer to the technique as operator duplication because every solution can be reached from the initial solution without merging.

The SIMULATED_ANNEALING algorithm only requires that the difference in cost be computed. The change in the operator allocation follows directly from a generated switch. Moreover, we do not need to compute the multiplexer cost of every visited solution from scratch, since only the number of edges connected to the operators involved may change; see also Section 5.1.2.

6.3 Implementation of local search

The SIMULATED_ANNEALING algorithm as shown in Figure 4.2 is modified into the OPERATOR_DUPLICATION algorithm to handle the existence of complex circuits; see Figure 6.2. In a temperature step, only solutions are generated that do not reintroduce complex circuits that have already been detected and removed. Other complex circuits, however, may be introduced. Therefore, at the end of every temperature step a detection of complex circuits is performed. If any are found they are added to the set of detected complex circuits. Furthermore, a solution is created that is free of complex circuits. Note that only complex circuits can be detected that were also present in the initial solution, because operator duplication cannot introduce new complex circuits. Furthermore, no complex circuits can be detected in the start solution.

```
procedure OPERATOR_DUPLICATION
in  a union 𝒢 of source SFGs 𝒢_i, i = 1,...,N,
    a delay del,
    an initial operator allocation al',
    an initial operator assignment as';
out operator allocation al*, operator assignment as*;
begin k = 0;
    INITIALIZE(c_0, L_0);
    fully duplicate operators resulting in al* and as*;
    C* = Ø;
    repeat
        for t=1 to L_k
        begin generate (al•, as•) ∈ N_switch(al*, as*)
                    for which sn(c) > 0 for each c ∈ C*;
            compute delta_cost;
            if delta_cost ≤ 0 or exp(−delta_cost / c_k) > random[0,1) then
                begin al* = al•;
                    as* = as•;
                end;
            else skip;
            update split numbers sn(c) for each c ∈ C*;
        end;
        k=k+1;
        COMPLEX_CIRCUIT_REMOVAL(𝒢,del,al*,as*,C*);
        CALCULATE_LENGTH(L_k);
        CALCULATE_CONTROL(c_k);
    until stop criterion;
end;
```

Figure 6.2. The OPERATOR_DUPLICATION algorithm.

We efficiently verify as follows that no complex circuit that had already been

detected and removed is created again. At every operation we link a list of references to complex circuits. A complex circuit is referenced if the operation belongs to the split set of the complex circuit. The split number for each detected complex circuit is stored. When a pair of the split set of a complex circuit is merged, then its split number decrements by one, and when a pair is duplicated it increments by one. Only solutions are generated that do not result in detected complex circuits with a split number equal to zero. In this way, it is easily guaranteed that previously detected complex circuits do not occur again, without it having been necessary to perform a more time-consuming complex-circuit detection algorithm.

Given an operator reallocation and an operator reassignment, all complex circuits can only be found with an algorithm that traverses all paths. Such an algorithm runs in exponential time and consequently is unacceptable, since our goal is to construct an efficient approximation algorithm. We use the complex-circuit detection algorithm called COMPLEX_CIRCUIT_DETECTION, which checks whether complex circuits exist; see Figure 6.3. If complex circuits exist, the detection algorithm returns a subset of the set of all complex circuits. Stok [1992] claims that if we start an algorithm like COMPLEX_CIRCUIT_DETECTION from every operation in a union of SFGs, it will find all complex circuits. It is easy to understand that this cannot be true, since the total number of circuits may be exponential, and the algorithm runs in polynomial time.

The algorithm COMPLEX_CIRCUIT_DETECTION continually forms a sequence of paths from a starting operation. It recursively extends a sequence of paths with an operation terminal in such a way that it may detect an operation that is already contained by one of the paths in the sequence at that moment. In that case a circuit is detected which is a complex circuit if it consists of two or more paths that do not share operations. In order to limit the time complexity of the algorithm, operations are visited only once. To this end, an operation is labeled after it has been visited and is consequently never visited again during the execution of COMPLEX_CIRCUIT_DETECTION. Without this labeling, COMPLEX_CIRCUIT_DETECTION would find all complex circuits.

Generating a solution that is free of complex circuits with this fast detection algorithm is done in an incremental way; see Figures 6.4 and 6.5. Repeatedly, the algorithm detects a number of complex circuits and duplicates sufficient operators such that the detected complex circuits are removed. The algorithm COMPLEX_CIRCUIT_REMOVAL ends when no complex circuits are detected anymore. Note that duplicating an operator may increase the split number of more than one complex circuit, even of a complex circuit that has not been detected. Since the probability of this is very high, we favour this approach over one in which all complex circuits are detected in one step. This is confirmed by an experiment given in

```
procedure COMPLEX_CIRCUIT_DETECTION
in  a union G of source SFGs G_i, i = 1,...,N,
    a delay del,
    a feasible operator assignment as;
out C* ⊆ C_G, a subset of all complex circuits;
begin C* = ∅;
    repeat take (v,a) ∈ O with label(v) = false;
        p = ((v,a));
        TRAVERSE_INPUTS(G,as,p,(v,a),C*,del);
    until label(v) = true for each (v,a) ∈ O;
end;
procedure TRAVERSE_INPUTS
in  a union G of source SFGs G_i, i = 1,...,N,
    a delay del, a feasible operator assignment as,
    a path p, an output terminal (v,a);
inout C* ⊆ C_G, a subset of all complex circuits;
begin for each ((v,a),(u,b)) ∈ E with label(u) = false
    begin if u ≠ w for each (w,c) ∈ p with (w,c) ∈ O then
            begin extend p with (u,b);
                TRAVERSE_OUTPUTS(G,as,p,(u,b),C*,del);
                remove (u,b) from p;
            end;
        else if del((w,b),(w,c)) ≠~ and
                the detected circuit, from w to u, is a complex circuit then
            construct complex circuit and add it to C*;
    end;
    label(u) = true;
end;
procedure TRAVERSE_OUTPUTS
in  a union G of source SFGs G_i, i = 1,...,N,
    a delay del, a feasible operator assignment as,
    a path p, an input terminal (u,b);
inout C* ⊆ C_G, a subset of all complex circuits;
begin for each (v,a) ∈ O with as(v) = as(u) and del((u,b),(u,a)) ≠~ and label(v) = false
    begin if v ≠ w for each (w,c) ∈ p then
            begin extend p with (v,a);
                TRAVERSE_INPUTS(G,as,p,(v,a),C*,del);
                remove (v,a) from p;
            end;
        else if the detected circuit, from w to v, is a complex circuit then
            construct complex circuit and add it to C*;
    end;
    label(v) = true;
end;
```

Figure 6.3. The COMPLEX_CIRCUIT_DETECTION algorithm.

the next section.

```
procedure COMPLEX_CIRCUIT_REMOVAL
in  a union G of source SFGs G_i, i = 1,...,N,
    a delay del;
inout an operator allocation al*, operator assignment as*;
    C' ⊆ C_G, a subset of all complex circuits;
begin stop = false;
    repeat COMPLEX_CIRCUIT_DETECTION(G,del,as,C*);
        C' = C* ∪ C';
        if C* = ∅ then stop = true;
        else repeat get c ∈ C* with sn(c) = 0;
                get (v,u) ∈ SP(c);
                as*(v) = v'_empty;
                al*(l(v)) = al'(l(v)) + 1;
                update split numbers sn(c) for each c ∈ C';
            until sn(c) ≥ 1 for each c ∈ C'
    until stop;
end;
```

Figure 6.4. The COMPLEX_CIRCUIT_REMOVAL algorithm. The algorithm repeatedly allocates an additional operator v'_{empty} to which it assigns an operation of the split set of a detected complex circuit with a split number equal to zero.

6.4 Experimental results

As an example we use instance IOASP1 of OASP as defined in Section 5.1.3, concerning the mapping of a Discrete Cosine Transform (DCT) and an Inverse Discrete Cosine Transform (IDCT) onto one PU; see Figure 5.2. We use the two source SFGs to create three different instances of ODP, namely IODP1, IODP2, and IODP3, which all have the same initial allocation, but different initial assignments. The initial operator allocation is minimal, consisting of 16 multipliers, 32 adders/subtractors, 8 inputs, and 8 outputs. The cost of an adder equals 15, the cost of a multiplier equals 45, the cost of an input equals 0, the cost of an output equals 0, and the cost of a multiplexer equals 7. The results are listed in Table 6.1.

From the results we can conclude the following. The total number of complex circuits is large when the quality of the initial assignment is poor. In addition, the amount of redundancy in the set of all constraints is high when the initial assignment is bad. Moreover, it takes a large amount of time to find all complex circuits that are present in the initial bad solution. The instances do not show an improvement in cost compared to the initial solution since the operators are expensive with respect to the cost of a multiplexer. Executing an additional run of the OPERA-

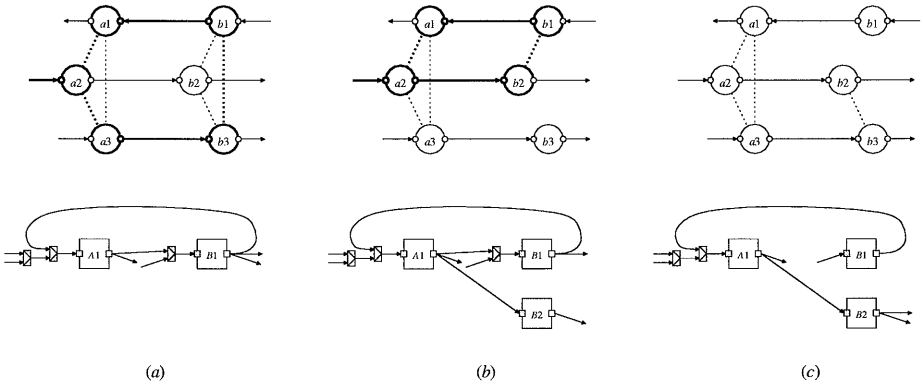


Figure 6.5. An example of removing complex circuits, considering three parts of a union SFG the operations of which are assigned to the same operator in a PU. In the first step (*a*) a complex circuit is detected, which is highlighted. After this complex circuit is removed another complex circuit is detected, which is shown in (*b*). In the final step this complex circuit is removed too, and a PU without a false loop is created, which is shown in (*c*). Only pairs of split sets are indicated that are still assigned to one operator.

Table 6.1. Results of the OPERATOR_DUPLICATION algorithm for mapping a DCT function and an IDCT function onto one PU. Three different instances of ODP are considered, namely IODP1, IODP2, and IODP3, which differ in the quality of the initial assignment: bad, good, and excellent, respectively. The start costs are, however, all equal to 2632. The column "final" lists the results after running the OPERATOR_DUPLICATION algorithm. The large column "total" lists the number of detected complex circuits and the CPU time that are obtained by a method that first derives all constraints concerning complex circuits and next finds an operator duplication satisfying them. The large column "incremental" lists results obtained by a method that incrementally derives only those constraints that were violated by a visited solution. The initial and greedy costs are listed for comparison. The greedy costs were obtained by only applying the COMPLEX_CIRCUIT_REMOVAL algorithm on the initial PU.

instance	cost			total		incremental	
	initial	final	greedy	complex circuits	CPU time	complex circuits	CPU time
IODP1	1936	1946	1955	53703	4h36'11.4"	26	13.8"
IODP2	1768	1781	1791	603	33.3"	14	10.2"
IODP3	1624	1662	1741	42	7.8"	15	7.9"

TOR_DUPLICATION algorithm omitting the check for complex circuits resulted in a PU with a cost equal to the initial one. We also notice that the difference between a greedy solution and a more refined one is greater when the initial assignment is better.

Next we consider an example of mapping two larger source SFGs, $\mathcal{G}_3$ and $\mathcal{G}_4$, as given in Section 5.2.4, onto one PU. We call this instance IODP4. Some characteristics of the large SFGs are: $|V_3| = 959$, $|E_3| = 1891$, $|V_4| = 1163$, $|E_4| = 2213$. The operators represent standard cells of different types with different costs. An initial allocation and assignment result in a PU with a cost of 11,500. Applying a run of the OPERATOR_DUPLICATION algorithm with the check for complex circuits results in a PU with a cost of 9,883. The cost as a function of time, expressed as the number of temperature steps, is shown in Figure 6.6. The accumulated number of complex circuits detected during the run as a function of time is shown in Figure 6.7. In the beginning of the run most complex circuits are detected. This is a very good characteristic, because when a complex circuit is found at a low temperature, the probability is small that a good solution is found in which the right operation of its split set is reassigned.

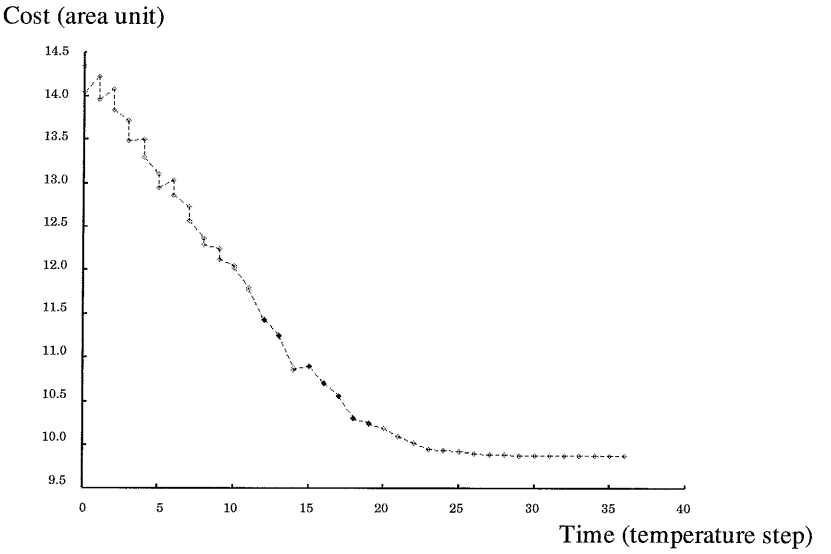


Figure 6.6. Cost of as a function of time for a run of the OPERATOR_DUPLICATION algorithm on instance IODP4. Two points are shown for each temperature step; one represents the cost before complex circuits have been removed, and the other the cost after complex circuits have been removed.

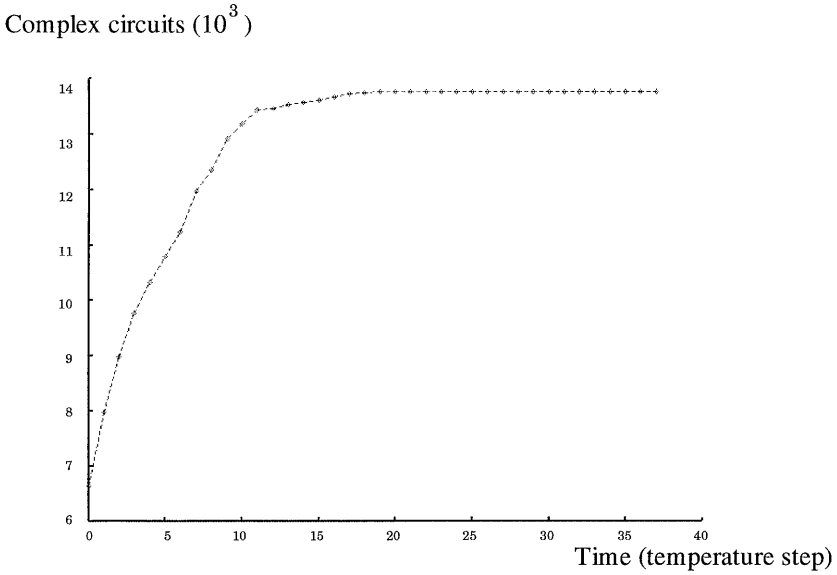


Figure 6.7. Accumulated number of detected complex circuits as a function of time for the same run of the OPERATOR_DUPLICATION algorithm on instance IODP4 as is used for Figure 6.6. Each point represents the number at the end of a temperature step.

7

Timing Analysis

From Chapter 3 we recall that the timing analysis problem (TAP) can be formulated as follows.

Definition 7.1 (timing analysis problem (TAP)). Given are an SFG $\mathcal{G}$, a type set $\mathcal{T}$, a type function ℓ, a clock period τ, and timing functions dat, drt, del, and w. The problem is to find a constraint set F, such that if there exists a retiming r with

(xiii) $w_r(f) \geq 0$	for each $f = (u, v) \in F$,
then and only then	
(vi) $D(p) < \tau$	for each $p \in P_{\mathcal{G}}$ with $W_r(p) = 0$, and
(vii) $w_r(e) \geq 0$	for each $e \in E$.

□

TAP can be solved in polynomial time, as stated by Theorem 3.6. In this chapter, algorithms are given to derive constraint edges $f \in F$, each of which imposes a bound on the difference of the retiming of two operations. The constraint set F contains two types of constraints, namely *causality* constraints corresponding to Constraints *(vii)* and *speed* constraints corresponding to Constraints *(vi)*.

The derivation of causality constraints is straightforward; for every signal edge $e = ((u, a), (v, b)) \in E$ a constraint edge $f = (u, v) \in F$ with $w(f) = w(e)$ is generated. Because of the trivial nature of causality constraints they are not discussed

any further; a constraint set F is assumed to contain at least the required causality constraints, and in figures portraying ESFGs they will not be shown.

The speed constraints can also be derived in a straightforward way. A constraint (*vi*) implies that a path that is longer than or equal to the clock period must contain at least one register, i.e.

$$W_r(p) \geq 1 \qquad \text{for each } p \in P_{\mathcal{G}} \text{ with } D(p) \geq \tau.$$

Thus, for every path $p = ((u,a) \mapsto (v,b)) \in P_{\mathcal{G}}$ with $D(p) \geq \tau$, a constraint edge $f = (u,v)$ with $w(f) = W(p) - 1$ is generated. However, better algorithms for constraint derivation can be found when redundancy of constraints has been taken into account. This was reported for the first time by Van der Werf, Van Meerbergen, Aarts, Verhaegh & Lippens [1994].

The organization of this chapter is as follows. In Section 7.1 we discuss basic timing properties of paths in SFGs. Section 7.2 discusses redundancy properties of timing constraints. Based on the properties presented, in Section 7.3 we present three different constraint derivation algorithms. The performance of these algorithms is discussed in Section 7.4.

7.1 Basic timing properties

The relation between delays and weights of paths in an SFG before and after retiming is expressed in the following property.

Property 7.1. Given are an SFG $\mathcal{G}$, timing functions *del*, *dat*, *drt*, w, and a retiming r. Then for each path $p = ((u_1,a_1) \mapsto (u_n,a_n)) \in P_{\mathcal{G}}$ it follows that $D_r(p) = D(p)$ and $W_r(p) = W(p) + r(u_n) - r(u_1)$.

Proof. $D_r(p) = D(p)$ because *del*, *dat*, and *drt* do not change by retiming; see Definitions 2.4, 2.5, and 2.6. Next,

$$\begin{aligned}
W_r(p) &= \sum_{k=1}^{n-1} w_r((u_k,a_k),(u_{k+1},a_{k+1})) \\
&= \sum_{k=1}^{n-1} [w((u_k,a_k),(u_{k+1},a_{k+1})) + r(u_{k+1}) - r(u_k)] \\
&= \sum_{k=1}^{n-1} w((u_k,a_k),(u_{k+1},a_{k+1})) + \sum_{k=2}^{n} r(u_k) - \sum_{k=1}^{n-1} r(u_k) \\
&= \sum_{k=1}^{n-1} w((u_k,a_k),(u_{k+1},a_{k+1})) + r(u_n) - r(u_1) \\
&= W(p) + r(u_n) - r(u_1).
\end{aligned}$$

□

Therefore, the number of registers on a path after retiming is given by the number of registers present before retiming plus what is added at the end of the path minus what is removed at the beginning of the path. The retimings of all the other, intermediate operations on a path determine where the registers are placed on the path, but do not influence the total number of registers on it.

Next we discuss subpaths and reconvergent paths, which are two structure properties of SFGs that we need in order to discuss redundancy among timing constraints.

Property 7.2. Given are an SFG $\mathcal{G}$ and functions w, del, dat, and drt. If $p_2 = (o_1 \mapsto i_n) \in P_\mathcal{G}$ is a path consisting of two subpaths $p_0 = (o_1 \mapsto i_l) \in P_\mathcal{G}$ and $p_1 = (o_l \mapsto i_n) \in P_\mathcal{G}$, denoted by $p_2 = p_0 + p_1$, then $W(p_2) = W(p_0) + W(p_1)$ and $D(p_2) = D(p_0) - dat(i_l) + del(i_l, o_l) - drt(o_l) + D(p_1)$.

Proof. Applying the definition of the path weight, Definition 2.10, it follows that

$$\begin{aligned} W(p_2) &= \sum_{k=1}^{n-1} w(o_k, i_{k+1}) \\ &= \sum_{k=1}^{l-1} w(o_k, i_{k+1}) + \sum_{k=l}^{n-1} w(o_k, i_{k+1}) \\ &= W(p_0) + W(p_1). \end{aligned}$$

Applying the definition of the path delay, Definition 2.9, it follows that

$$\begin{aligned} D(p_2) &= drt(o_1) + \sum_{k=2}^{n-1} del(i_k, o_k) + dat(i_n) \\ &= drt(o_1) + \sum_{k=2}^{l-1} del(i_k, o_k) + del(i_l, o_l) + \sum_{k=l+1}^{n-1} del(i_k, o_k) + dat(i_n) \\ &= D(p_0) - dat(i_l) + del(i_l, o_l) - drt(o_l) + D(p_1). \end{aligned}$$

□

Property 7.3. Given are an ESFG $\mathcal{EG}$, a weight w, and a retiming r with $w_r(f) \geq 0$ for each $f \in F$. Then $W_r(p) \geq 0$ for each $p = (o_1 \mapsto i_n) \in P_\mathcal{G}$.

Proof. Since for each $e = ((v, a), (u, b)) \in E$ there exists an $f = (v, u) \in F$ with $w(f) = w(e)$, it follows that $w_r(e) \geq 0$ for each $e \in E$. Since $W_r(p) = \sum_{k=1}^{n-1} w_r((o_k, i_{k+1}))$ it follows that $W_r(p) \geq 0$ for each $p \in P_\mathcal{G}$. □

Definition 7.2. Given is an SFG $\mathcal{G}$. Two paths $p_0 = (o_i \mapsto i_j), p_1 = (o_k \mapsto i_l) \in P_\mathcal{G}$ are *reconvergent*, which is denoted by $p_0 \| p_1$, if and only if $o_i = o_k$ and $i_j = i_l$. □

In words, two paths are reconvergent if they start at the same output terminal and end at the same input terminal.

Property 7.4. Given are an SFG $\mathcal{G}$ and a weight w. Then for every two paths $p_0, p_1 \in P_\mathcal{G}$ with $p_0 \| p_1$ it follows that $W_r(p_0) - W_r(p_1) = W(p_0) - W(p_1)$.

Proof. Let $p_0 = ((u_1, a_1) \mapsto (u_n, a_n))$ and $p_1 = ((u_1, a_1) \mapsto (u_n, a_n))$. Then

$$\begin{aligned} W_r(p_0) - W_r(p_1) &= (W(p_0) + r(u_n) - r(u_1)) - (W(p_1) + r(u_n) - r(u_1)) \\ &= W(p_0) - W(p_1). \end{aligned}$$

□

7.2 Redundancy

When a constraint edge is generated for every path in an SFG that is longer than the clock period, the set of constraint edges can be quite large. Fortunately, such a set will contain much redundancy, i.e., many constraints can be left out without any consequences for the retimed SFG. The redundancy is caused by the structure of the SFG and the functions W_r and D_r. We distinguish the following types of redundancy: *subpath*, *reconvergent path*, and *reconvergent n-long path* redundancy.

Theorem 7.1. Given are an ESFG $\mathcal{EG}$, a weight w and a retiming r with $w_r(f) \geq 0$ for each $f \in F$. Then for each $p_0, p_1, p_2 \in P_\mathcal{G}$ with $p_2 = p_0 + p_1$ and $W_r(p_1) \geq 1$ or $W_r(p_0) \geq 1$ it follows that $W_r(p_2) \geq 1$.

Proof. Combining Property 7.2, which states that if

$$p_2 = p_0 + p_1$$

then

$$W_r(p_2) = W_r(p_1) + W_r(p_0),$$

and Property 7.3, which states that for each $p \in P_\mathcal{G}$,

$$W_r(p) \geq 0,$$

the theorem follows. □

The theorem is illustrated by the example of Figure 7.1. Theorem 7.1 implies that during the generation of speed constraints we can restrict ourselves to considering paths that are longer than the clock period, but have no subpath longer than the clock period, i.e., we only have to consider paths that are just longer than the clock period.

If two reconvergent paths have different path weights in an SFG, then the path with the largest weight contains at least one register after a feasible retiming, which is stated by the following theorem.

Theorem 7.2. Given are an ESFG $\mathcal{G}$, a weight w, a clock period τ, and a retiming r with $w_r(f) \geq 0$ for each $f \in F$. For every two paths $p_l, p_v \in P_\mathcal{G}$ with $p_l \| p_v$, if

(*a*) $W(p_l) < W(p_v)$, or

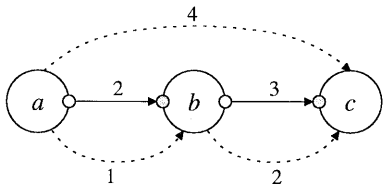


Figure 7.1. An example of an SFG with three constraint edges, indicated by the dashed lines. The constraint edge from a to c is redundant with respect to the one from a to b and it is redundant with respect to the one from b to c.

(b) $W(p_l) = W(p_v)$ and $D(p_l) \geq \tau$

then it follows that $W_r(p_v) \geq 1$.

Proof. From Property 7.4 we have

$$W_r(p_v) = W(p_v) - W(p_l) + W_r(p_l)$$

In case (a) this gives

$$\begin{aligned} W_r(p_v) &\geq 1 + W_r(p_l) \\ &\geq 1, \end{aligned}$$

since $W_r(p_l) \geq 0$ by Property 7.3.

In case (b) we have $D(p_l) \geq \tau$ and, consequently, a retiming r with $w_r(f) \geq 0$ for each $f \in F$ that obeys $W_r(p_l) \geq 1$. Hence

$$\begin{aligned} W_r(p_v) &= 0 + W_r(p_l) \\ &\geq 1. \end{aligned}$$

□

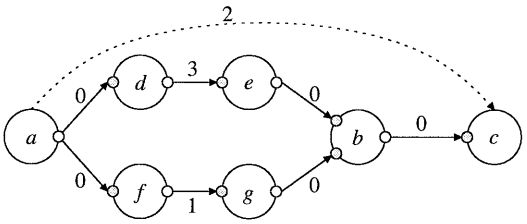


Figure 7.2. An example of an SFG with a redundant constraint edge. In the SFG, there are two reconvergent paths from the output terminal of operation a to the input terminal of operation c. Because the lower path has a weight smaller than the weight of the upper path, the constraint edge, corresponding to the upper path, is redundant.

Theorem 7.2 is illustrated by the example of Figure 7.2. The theorem serves as the basis for some of the work done by Leiserson, Rose & Saxe [1983], which is further explained in Section 7.3.2. The theorem implies that one is only interested in paths between two operations on which no registers are trapped by the structure, i.e., the paths with the smallest weight. If one of these paths is longer than the clock period, then a constraint edge between the two end operations of the paths may be required. However, even such a constraint edge may be redundant with respect to other speed constraint edges on reconvergent paths. To explain this we consider paths that have a delay greater than a multiple of the clock period, called *n-long paths*.

Lemma 7.1. Given are an ESFG $\mathcal{G}$, timing functions w, del, dat, drt, a clock period τ, and a retiming r such that $W_r(p) \geq 1$ for each $p \in P_\mathcal{G}$ with $D(p) \geq \tau$. Then for each $p \in P_\mathcal{G}$ with $D(p) \geq n\tau$, $n \in \mathbb{N}^+$, it follows that $W_r(p) - n \geq 0$.

Proof. For $n = 1$ the proof is trivial. For $n > 1$ it is possible to partition p into subpaths p_i, with $i = 1, \ldots, n$, such that $p = \sum_{i=1}^{n} p_i$ and $D(p_i) \geq \tau$ in the following way. We divide $p = (o_1, i_2, o_2, \ldots, i_n)$ into two subpaths p_i and p_j with $p = p_i + p_j$ and $D(p_i) \geq \tau$. Two subpath configurations are possible:

(*a*) $p_i = (o_1, \ldots, i_{l-1}, o_{l-1}, i_l)$ and $p_j = (o_l \mapsto i_n)$ with $D((o_1, \ldots, i_{l-1})) < \tau$. Since $D((o_1, \ldots, i_{l-1})) = D(p_i) - dat(i_l) - del(i_{l-1}, o_{l-1}) + dat(i_{l-1}) < \tau$ and $del(i_{l-1}, o_{l-1}) \leq dat(i_{l-1})$ it follows that $D(p_i) - dat(i_l) < \tau$.

(*b*) $p_i = (o_1, i_2)$ and $p_j = (o_2 \mapsto i_n)$.
Since $D(p_i) = drt(o_1) + dat(i_2)$ and $drt(o_1) < \tau$ and $dat(i_2) < \tau$ it follows that $D(p_i) - dat(i_l) < \tau$.

Thus in both cases $D(p_i) - dat(i_l) < \tau$. Furthermore, $del(i_l, o_l) \leq drt(o_l)$. Starting from Property 7.2, it follows that if $D(p_0) \geq n\tau$, then

$$\begin{aligned} D(p_j) &= D(p_0) - D(p_i) + dat(i_l) - del(i_l, o_l) + drt(o_l) \\ &\geq D(p_0) - D(p_i) + dat(i_l) \\ &> D(p_0) - \tau \\ &\geq (n-1)\tau. \end{aligned}$$

By using induction, the path p_j can be split into $n-1$ subpaths with a delay of at least τ each. For every subpath p_i, with $i = 1, \ldots, n$, it is required that $W_r(p_i) \geq 1$ such that, according to Property 7.2, $W_r(p) = W_r(\sum_{i=1}^{n} p_i) = \sum_{i=1}^{n} W_r(p_i) \geq n$. □

This lemma derives a constraint on the retimed weight of a path from constraints on the retimed weights of its subpaths. This constraint is called the *corresponding constraint* of a path. The corresponding constraint of a path $p \in P_\mathcal{G}$ can also be formulated as $W_r(p) \geq \lfloor \frac{D(p)}{\tau} \rfloor$ or $W_r(p) > \frac{D(p)}{\tau} - 1$. Furthermore, corresponding constraints may result in constraint edges in F in the following way. For each

path $p = ((u,a) \mapsto (v,b)) \in P_{\mathcal{G}}$ with $(n+1)\tau > D(p) \geq n\tau$ there exists a constraint edge $f = (u,v) \in F$ with $w(f) = W(p) - n$, where $n \in \mathbb{N}^+$. Note that one can presume $n \in \mathbb{N}$, i.e., that n may be zero, but then many additional constraints arise that are obviously redundant since, according to Property 7.3, $W_r(p) \geq 0$ for each $p \in P_{\mathcal{G}}$. Consequently, we do not consider the corresponding constraints of paths that have a smaller delay than the clock period. Furthermore, paths with $n \geq 2$ have corresponding constraints that are redundant with respect to corresponding constraints of their subpaths.

Theorem 7.2 can be generalized to n-long paths. A constraint to add at least k registers to a path may be redundant if the path reconverges with a path on which k or more registers must be added. However, we present here a slightly different theorem that combines subpath redundancy and reconvergent path redundancy for n-long paths. To this end we need the following definition.

Definition 7.3. Given are an SFG $\mathcal{G}$, timing functions w, del, dat, drt, and a clock period τ. A *relevant path* $p_r \in P_{\mathcal{G}}$ is a path such that for each $p_v \in P_{\mathcal{G}}$ with $p_r \| p_v$, $W(p_r) - \frac{D(p_r)}{\tau} \leq W(p_v) - \frac{D(p_v)}{\tau}$. □

Theorem 7.3. Given are an ESFG $\mathcal{G}$, timing functions w, del, dat, drt, a clock period τ, and a retiming r such that $W_r(p) \geq 1$ for each $p \in P_{\mathcal{G}}$ with $D(p) > \tau$. For each relevant path p_r and each path $p_v \in P_{\mathcal{G}}$ we have that p_v has a redundant corresponding constraint if

(*a*) $p_r \| p_v$, or

(*b*) there exists a subpath $p_t \in P_{\mathcal{G}}$ of p_v such that $p_r \| p_t$.

Proof. In case (*a*) we have to prove that every reconvergent path p_v has a redundant corresponding constraint.

$$\begin{aligned} W_r(p_v) &= W_r(p_r) - W(p_r) + W(p_v) \\ &\geq W_r(p_r) - \frac{D(p_r)}{\tau} + \frac{D(p_v)}{\tau} \\ &> -1 + \frac{D(p_v)}{\tau}. \end{aligned}$$

Since $W_r(p_v) \in \mathbb{Z}$ it follows that

$$W_r(p_v) \geq \left\lfloor \frac{D(p_v)}{\tau} \right\rfloor$$

In case (*b*) we have to prove that every path p_v that has a subpath p_t reconverging with the relevant path p_r has a redundant corresponding constraint. For brevity, we discuss only one subpath configuration, $p_v = p_t + p_a$, where $p_a \in P_{\mathcal{G}}$. This configuration implies that a path $p_u = p_r + p_a$ also exists, where $p_u \in P_{\mathcal{G}}$. Furthermore,

$d = D(p_v) - D(p_t) = D(p_u) - D(p_r)$. Now we derive

$$\begin{aligned} W_r(p_v) &= W_r(p_t) + W_r(p_a) \\ &= W_r(p_t) - W_r(p_r) + W_r(p_u) \\ &= W(p_t) - W(p_r) + W_r(p_u) \\ &\geq \frac{D(p_t)}{\tau} - \frac{D(p_r)}{\tau} + \left\lfloor \frac{D(p_r)+d}{\tau} \right\rfloor \\ &> \frac{D(p_t)}{\tau} - \frac{D(p_r)}{\tau} + \frac{D(p_r)+d}{\tau} - 1 \\ &= \frac{D(p_t)+d}{\tau} - 1 \\ &= \frac{D(p_v)}{\tau} - 1. \end{aligned}$$

Since $W_r(p_v) \in \mathbb{Z}$ it follows that

$$W_r(p_v) \geq \left\lfloor \frac{D(p_v)}{\tau} \right\rfloor.$$

For other subpath configurations similar proofs exist. □

According to Theorem 7.3 the relevant path may be different from the one with minimal weight of all reconvergent paths that was put forward by Leiserson, Rose & Saxe [1983]. A corresponding constraint edge of a relevant path is shown in Figure 7.3.

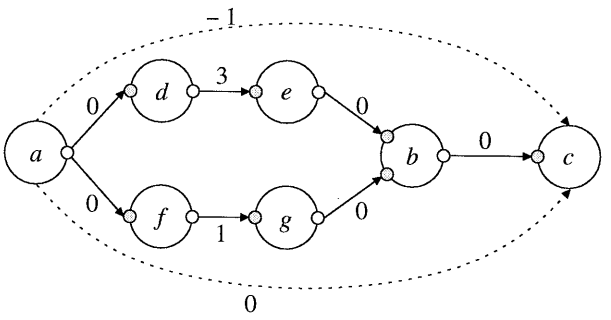


Figure 7.3. An example of an SFG with two constraint edges. In the SFG two reconvergent paths exist from the output terminal of operation a to the input terminal of operation c. The lower path has a smaller weight than the weight of the upper path, but since on the upper path, which is the relevant path, more registers must be placed than originally are present the corresponding constraint edge of the lower path is redundant.

Conclusion

The above presented theory can be summarized as follows. If a constraint is generated for every path that is longer than the clock period, the set of constraints contains much redundancy. To obtain a set of constraints with less redundancy, according to Theorem 7.1 we can restrict ourselves to paths that do not contain a subpath longer than the clock period. If such a path is longer than the clock period, then a constraint needs to be generated. However, if the path reconverges with another path on which fewer registers are initially present, according to Theorem 7.2 the constraint does not need to be generated. Furthermore, according to Theorem 7.3 the constraint does not need to be generated, if the path reconverges with another path on which retiming must add more registers, even when that path initially contains more registers.

Discussion

The set of corresponding constraint edges of all relevant paths that are just longer than the clock period can still contain redundancy. So far, redundancy has been considered using properties of paths in an SFG, i.e., corresponding constraint edges of paths can be redundant if paths have terminals in common, like a path and one of its subpaths, or reconvergent paths. However, a constraint edge is defined on a pair of operations that can have more than one input and one output terminal. Consequently, corresponding constraint edges of paths that have no terminals but do have operations in common may be redundant. We do not discuss this kind of redundancy any further, but the theory presented can be easily adapted to it.

Even the causality constraints can contain redundancy with respect to speed constraints. This is the case when a path $p = ((u,a),(v,b))$, consisting of one output terminal (u,a) and one input terminal (v,b), has a delay greater than the clock period, i.e.,

$$\begin{aligned} D(p) &= drt((u,a)) + dat((v,b)) \\ &> \tau. \end{aligned}$$

Then the corresponding constraint is

$$r(v) - r(u) \geq W((u,a),(v,b)) - 1 = w((u,a),(v,b)) - 1,$$

which makes the causality constraint

$$r(v) - r(u) \geq w((u,a),(v,b))$$

redundant. Note that this is the case in Figure 7.1, although it is not mentioned there.

A different argument for the removal of redundant constraint edges comes from the network flow theory [Ahuja, Magnanti & Orlin, 1989]. In a solution of an

instance of the uncapacitated minimum-cost flow problem there only exists non-zero flow between two operations over shortest paths between them. Consequently, a constraint edge between two operations can be removed if it is not on the only shortest path between the two operations. Figure 7.4 shows some operations in an SFG and the constraint edges between them. In this figure, the dashed lines indicate constraint edges that are redundant.

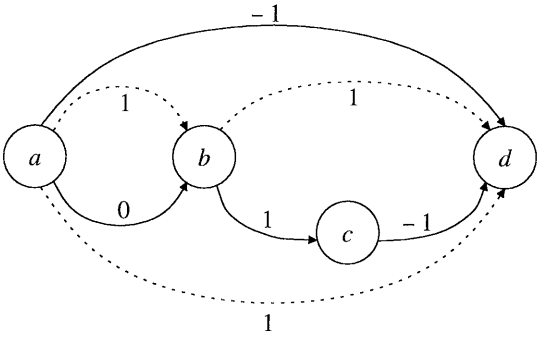


Figure 7.4. An example of operations and constraint edges. Redundant constraint edges are indicated by dashed lines.

7.3 Implementation of speed constraint derivation

For constraint derivation we make a trade-off between the number of derived constraints that must be stored and the run time of the derivation algorithm. The number of constraints also affects the run time of the algorithm to solve instances of SGRP, but that topic is not considered here. Constraint derivation can be viewed as a special form of path length analysis in which distances are defined in terms of weights and delays. Since SFGs have a sparse structure, derivation is performed starting from each output terminal of an operation in an SFG. The derivation starting at each output terminal is partitioned into two steps. In the first step, input terminals are labeled with delay labels (D) and weight labels (W) and in the second step, these labels are used to generate constraint edges to be included in the set F of an SFG $\mathcal{G}$. A description of the CONSTRAINT_DERIVATION algorithm is shown in Figure 7.5. The result of the first step is that input terminals are labeled with delays and weights, which represent for each input terminal the delay and weight of a certain path from the starting output terminal to it. To which of the reconvergent paths the labels belong depends on the type of redundancy that is taken into account. Before discussing three different labeling methods, we first discuss constraint generation, because some of its characteristics are important for the discussion of the

first step.

7.3.1 Constraint generation

The constraint generation is implemented as a depth first search for input terminals labeled with a delay greater than the clock period, as follows. At every input terminal the labels are used to select a path from the set of reconvergent paths from the starting output terminal to that input terminal, for which a corresponding constraint edge may be generated. More specifically, during the generation of constraint edges, paths are followed by only traversing from an input terminal of an operation $i_1 = (u,a) \in I$ via an output terminal of the same operation $o_1 = (u,b) \in O$, with $del(i_1,o_1) \neq \sim$, to an input terminal of another operation $i_2 \in I$, with $e = (o_1,i_2) \in E$, if $W(i_2) = W(i_1) + w((o_1,i_2))$ and $D(i_2) - dat(i_2) = D(i_1) - dat(i_1) + del(i_1,o_1)$. If an input terminal is labeled with a delay greater than the clock period, then a path with that delay runs between the starting output terminal and the input terminal. Consequently, a constraint edge must be generated from the operation to which the output terminal belongs to the operation to which the input terminal belongs. The weight of the constraint edge becomes equal to the weight label. According to Theorem 7.1 each path that has a subpath with a delay greater than the clock period has a redundant corresponding constraint edge such that the depth of the search can be limited to paths with a delay just greater than the clock period. The CONSTRAINT_GENERATION algorithm is shown in Figure 7.6.

```
procedure CONSTRAINT_DERIVATION
in  an SFG G,
    timing functions w, del, dat, drt,
    a clock period τ;
out a constraint set F;
begin F = Ø;
    generate causality constraints into set F;
    for each o = (u,a) ∈ O
        begin INPUT_LABELING(o,G,w,del,dat,drt,τ,W,D);
            CONSTRAINT_GENERATION(o,G,w,del,dat,drt,τ,F,W,D);
        end;
end;
```

Figure 7.5. The CONSTRAINT_DERIVATION algorithm.

7.3.2 Delay and weight labeling

The task of the labeling step is to label an input terminal with a delay and a weight that represent the delay and weight of a certain path from the starting output terminal to the input terminal. We discuss three different labeling methods, which differ

```
procedure CONSTRAINT_GENERATION
in      an SFG G,
        timing functions w, del, dat, drt,
        a clock period τ,
        labels W and D,
        an output terminal o;
inout   a constraint set F;
begin I = ∅
    for each i = (v,b) ∈ I : (o,i) ∈ E
        if D(i) ≥ τ then
            begin generate constraint edge f = (u,v) with w(f) = W(i) − 1;
                F = F ∪ {f};
            end;
        else I = I ∪ {i};
    while I ≠ ∅
    begin get i1 ∈ I;
        I = I \ {i1};
        for each i2 = (v,b) ∈ I for which there exists an o1 ∈ O with
                        e = (o1,i2) ∈ E and
                        del(i1,o1) ≠∼ and
                        W(i2) = W(i1) + w(e) and
                        D(i2) − dat(i2) = D(i1) − dat(i1) + del(i1,o1)
            if D(i2) ≥ τ then
                begin generate a constraint edge f = (u,v) with w(f) = W(i2) − 1;
                    F = F ∪ {f};
                end;
            else I = I ∪ {i2};
    end;
end;
```

Figure 7.6. The CONSTRAINT_GENERATION algorithm.

in how they select one of the reconvergent paths to which the labels belong. We first discuss the *relevant-path labeling* method that derives as few constraint edges as possible. Secondly, the *classical labeling* method is presented based on the work by Leiserson, Rose & Saxe [1983]. The third method is the *clock-period-limited labeling* method, which takes the clock period into account during derivation. Of all the methods, this method produces the most redundant constraint edges.

Relevant-path labeling In order to generate as few constraint edges as possible, each input terminal is labeled with the delay and weight of a relevant path to it according to Definition 7.3. According to Theorem 7.3, corresponding constraints of reconvergent paths are redundant. The delay and weight of relevant paths starting from one output terminal to all the input terminals can be derived by a shortest path algorithm. The distances from the output terminal to input terminals are determined as follows. Traversing a signal edge e increases the distance with $w(e)$, while traversing through an operation, from an input terminal i to an output terminal o, decreases the distance by $\frac{del(i,o)}{\tau}$. The data ready time of the output terminal and the data available time of the input terminal must be added at respectively the beginning of a path and the end of a path. Because of the negative distances, a straightforward application of an efficient shortest path algorithm like DIJKSTRA'S, as discussed in Section 4.2.2, is impossible here.

To explain the above labeling method, we use an example SFG as shown in Figure 7.7. The labels resulting from a path length analysis starting from the output terminal of operation a and the constraint edges generated for that starting output terminal are shown in the figure. Note that the delay label of an input terminal does include the data available time of the input terminal and the data ready time of the starting output terminal. The labels define the paths that the constraint generation will follow. In the figure, the signal edges and the operation delay edges of these paths are indicated by bold lines.

Classical labeling In this section a method is presented that primarily builds on the work by Leiserson, Rose & Saxe [1983] by using Theorem 7.2. Of all reconvergent paths, either the path with the smallest weight or, if there is more than one of these paths, the one of these with the greatest delay is considered the shortest. Like the previously discussed method, the labeling method must label each input terminal for every output terminal. In other words, an all-pairs shortest-path algorithm [Floyd, 1962; Warshall, 1962] is required. Compared to the previous method, many more constraint edges may be generated since n-long path redundancy is not taken into account.

Figure 7.8 shows the labels and generated constraint edges that correspond to the classical labeling method. Compared to the relevant-path labeling method,

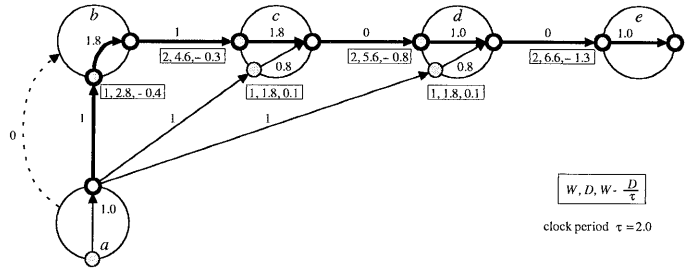


Figure 7.7. An example of an SFG with labels corresponding to a timing analysis starting at the output terminal of operation a. The first two labels, W and D, are produced by the relevant-path labeling method, while the third, $W - \frac{D}{\tau}$ is added here to identify the shortest paths. Constraint generation follows the paths indicated by bold lines to which the labels correspond. One constraint edge is generated from a to b, which is indicated by the dashed arrow.

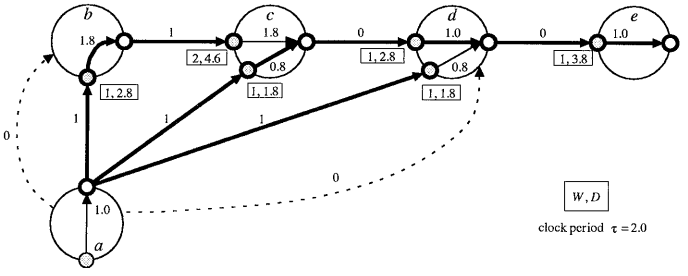


Figure 7.8. An example of an SFG with labels corresponding to a timing analysis starting at the output terminal of operation a. The labels are produced by the classical labeling method. Constraint generation follows paths indicated by bold lines to which the labels correspond. Two constraint edges are generated, which are indicated by the dashed lines, of which the constraint from a to d is redundant.

one additional constraint edge is generated; cf. Figure 7.7. This constraint edge corresponds to the path from the output terminal of operation a via operation c to an input terminal of operation d. This path is different from the relevant path from the output terminal of operation a via operation b and c to the same input terminal of d.

Clock-period-limited labeling A disadvantage of both methods presented above is that an instance of an all-pairs shortest path problem must be solved. When the delay of the longest path in an SFG is (considerably) greater than the clock period, it is very likely that not all input terminal labels are used, because during constraint generation the search depth is limited by the clock period. We present a new labeling algorithm that is based on subpath redundancy as stated by Theorem 7.1. The algorithm avoids unnecessary labeling by taking the clock period into account such that the number of visited input terminals during labeling is reduced. The algorithm labels only input terminals that have a delay smaller or just greater than the clock period. Although this labeling method is more run-time efficient, it may generate more redundant constraint edges than do the previously presented methods. This can be explained as follows. Not all paths incident with an input terminal are known, only those that have a delay smaller or just greater than the clock period. If for one of the known paths a corresponding constraint edge is generated, it is possible that this constraint edge is redundant with respect to one of the corresponding constraint edges of the unknown paths to the input terminal. Therefore, redundant constraint edges may be generated if we only search for paths that are just longer than the clock period. Since the search depth of the CONSTRAINT_GENERATION algorithm is limited by the clock period, unlabeled input terminals are not visited in that step. Figure 7.9 shows the labels and generated constraint edges that correspond to the clock-period-limited labeling method. Compared to the previous two labeling methods, additional constraint edges are generated; cf. Figures 7.7 and 7.8.

7.3.3 Efficient labeling

The run-time efficiency of both the classical labeling algorithm as well as the clock-period-limited labeling algorithm can be improved by using more efficient path analysis algorithms. Here, we discuss the efficient CLOCK_PERIOD_LIMITED_LABELING algorithm; see Figure 7.10. It can easily be generalized to a classical labeling algorithm. Both labeling methods compare labels in two steps, first the weight label and then the delay label. The weight labels are based on the signal edge weights, which in many cases are restricted to positive values. This implies that the unretimed SFG already must have a causal behaviour, i.e., for each $e \in E, w(e) \geq 0$. Consequently, an efficient algorithm like

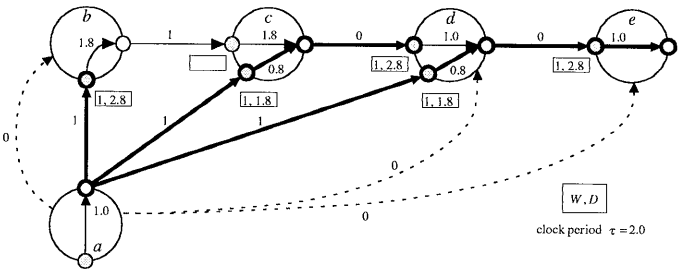


Figure 7.9. An example of an SFG with labels corresponding to a timing analysis starting at the output terminal of operation a. The labels are produced by the clock-period-limited labeling method. Note that one input terminal is left unlabeled. Constraint generation follows paths indicated by bold lines to which the labels correspond. Three constraint edges are generated and are indicated by the dashed line; two of them are redundant.

DIJKSTRA'S, as discussed in Section 4.2.2, can be used to label the input terminals with weight labels. Only an input terminal is labeled that has a shortest path to it with a delay smaller or just greater than the clock period. Therefore, when an input terminal is labeled with the weight of a shortest path to it, it is also labeled with the delay of that shortest path. If an input terminal is labeled with a delay greater than the clock period, the analysis can be stopped from that input terminal. Note that at this stage, the delay label of an input terminal does not necessarily represent the delay of the minimum-weight path to it that has the largest delay. It represents only one of the delays of the minimum-weight paths to it.

The delay labels that will be used by constraint generation have to be determined in the second step. As a result of the first step, all paths from the starting output terminal with a minimum weight and a delay smaller or just greater than the clock period are known. From a set of reconvergent paths with minimum weight, the one with the largest delay must determine the delay label of the input terminal at which the reconvergent paths end. The delay labeling only needs to be performed on a part of an SFG, as can be seen as follows. The delay labeling part of the algorithm only has to walk on minimum-weight paths in an SFG. Since we assume that no negative-weight feedback loops are present in an SFG, the operations on the minimum-weight paths can be sorted in topological order. Moreover, only those operations that have input terminals that are labeled with weights, i.e., input terminals that have a finite weight label, have to be sorted. For topological sorting, only those edges need to be considered that lie on a minimum-weight path. Therefore the delay labeling can be done efficiently by a longest path algorithm on

a topologically ordered set of a reduced number of operations traversing a reduced number of edges.

Shenoy & Rudell [1994] also apply clock-period-limited labeling. They recognize the use of the DIJKSTRA'S algorithm for the weight labeling, but for the delay labeling they make no use of the a-cyclic nature of the graph consisting of nodes with equal weight labels. Instead of topological sorting followed by labeling, they use the less efficient MODIFIED_LABEL_CORRECTING algorithm, as discussed in Section 4.2.2.

The CLOCK_PERIOD_LIMITED_LABELING algorithm can be generalized to a classical labeling algorithm by assuming an infinite value for the clock period. Then, the real clock period value can be accounted for during constraint generation.

7.4 Experimental results

We implemented the efficient variants of the classical labeling and clock-period-limited labeling methods and used them for generating some results. Although we do not have an implementation of the relevant-path labeling method, we can bound the number of constraint edges it would generate in the following way. First, we use the CLOCK_PERIOD_LIMITED_LABELING algorithm followed by the CONSTRAINT_GENERATION algorithm to determine a set of constraint edges. Secondly, we remove all constraint edges from the set that do not lie on a shortest path between two operations, where a path consists of constraint edges and its length is the total weight of the edges. The resulting number of constraint edges is an upper bound on the number of constraint edges that a relevant-path labeling method would have derived. The three methods are applied to two SFGs: $\mathcal{G}_1$ and

Table 7.1. Characteristics of two SFGs and the results of applying three constraint derivation methods to them. The CPU time of the relevant-path derivation run corresponds to a bound computation, which consists of the CLOCK_PERIOD_LIMITED_LABELING followed by a shortest path algorithm to remove redundant constraints.

	$\mathcal{G}_1$		$\mathcal{G}_2$	
$\|V\|$	7512		13002	
clock period (ns.)	21		24	
longest path delay (ns.)	159		150	
	$\|F\|$	CPU time	$\|F\|$	CPU time
relevant-path (upper bound)	73,260	45'40"	186,303	2h11'49"
classical	302,818	34'41"	800,784	1h45'19"
clock-period-limited	302,994	2'30"	803,188	5'05"
$\|V\| \times \|V\|$	56,430,144		169,052,004	

$\mathcal{G}_2$. Table 7.1 shows the results. From the ratios between the clock period and the delay of a longest path in an SFG it can be concluded that both retimed SFGs must

```
procedure CLOCK_PERIOD_LIMITED_LABELING
in  an SFG G,
    timing functions w, del, dat, drt,
    a clock period τ,
    an output terminal o;
out labels W and D;
begin for each i ∈ I
    begin D(i) = 0.0;
        W(i) = ∞;
    end;
    I = Ø;
    for each i ∈ I with e = (o, i) ∈ E
        begin D(i) = drt(o) + dat(i);
            W(i) = w(e);
            if D(i) < τ then I = I ∪ {i};
        end;
    while I ≠ Ø
    begin get i1 ∈ I with W(i3) ≥ W(i1) for each i3 ∈ I;
        I = I \ {i1};
        for each i2 ∈ I for which there exists an o1 ∈ O with
                e = (o1, i2) ∈ E and del(i1, o1) ≠∼)
            if W(i2) > W(i1) + w(e) or
                (W(i2) = W(i1) + w(e) and
                D(i2) < D(i1) − dat(i1) + del(i1, o1) + dat(i2)) then
                begin D(i2) = D(i1) − dat(i1) + del(i1, o1) + dat(i2);
                    if D(i2) < τ and W(i2) = ∞ then I = I ∪ {i2};
                    W(i2) = W(i1) + w(e);
                end;
    end;
    topologically sort labeled input terminals into list L;
    while L ≠ Ø
    begin get first i1 ∈ L;
        L = L \ {i1};
        for each i2 ∈ I for which there exists an o1 ∈ O with
                e = (o1, i2) ∈ E and
                del(i1, o1) ≠∼ and
                W(i2) = W(i1) + w(e))
            if D(i2) < D(i1) − dat(i1) + del(i1, o1) + dat(i2) then
                D(i2) = D(i1) − dat(i1) + del(i1, o1) + dat(i2);
    end;
end;
```

Figure 7.10. The CLOCK_PERIOD_LIMITED_LABELING algorithm.

be pipelined. The CPU times are obtained on a 124 MIPS workstation. The size of a matrix representation of the shortest paths between all pairs of operations as originally proposed by Leiserson, Rose & Saxe [1983] is also given in the table.

From the results we conclude that for the two SFGs, clock-period-limited labeling is about 20 times faster than classical labeling at the cost of a marginal number of additional redundant constraints. Furthermore, relevant-path labeling will derive more than 4 times fewer constraints than clock-period-limited labeling and classical labeling. Representing constraints between all pairs of operations requires almost 1000 times more memory space.

8

Retiming

From Chapter 3 we recall that the synchronous generalized retiming problem (SGRP) can be formulated as follows.

Definition 8.1 (synchronous generalized retiming problem (SGRP)). Given are a union $\mathcal{EG}$ of source ESFGs $\mathcal{EG}_i$ with folding factors f_i, $i = 1,\ldots,N$, a type set $\mathcal{T}$, a type function ℓ, a weight w, the cost of a register α, an operator allocation al, and an operator assignment as. The problem is to find a PU $\mathcal{G}'$ constructed by a retiming r, a register allocation ra, a multiplexer allocation ma, and the given al and as, for which

(iv)	$e' = ((as(v),a),(as(u),b)) \in E'$ and $w_r(e') = w_r(e)$	for each $e = ((v,a),(u,b)) \in E$, and
(v)	$ph(u) \neq ph(v)$	for each $u, v \in V_i$ with $as(u) = as(v)$, and $u \neq v$, and for each $i = 1,\ldots,N$, and
(ix)	$ra(o') + ra(i') \geq w_r(e')$	for each $e' = (o',i') \in E'$, and
(xiii)	$w_r(f) \geq 0$	for each $f \in F$, and

that minimizes

$$\alpha \sum_{t \in T} ra(t) + \beta \sum_{i \in I} ma(i).$$

□

Because SGRP is NP-hard, it is believed that no polynomial time algorithm can be constructed that solves to optimality practical instances of SGRP, which have a large size. This implies that we need to search for effective and efficient approximation algorithms. On the other hand, we know that SRP, which is the special case of SGRP for retiming one unfolded source ESFG, can be reduced to the dual of the network flow problem and, consequently, it can be solved in polynomial time by, e.g., the MINIMUM_COST_MAXIMUM_FLOW algorithm. This motivates us to search for applications of this kind of algorithms to find an approximation in reasonable time. In this chapter we present an approximation method using the SIMULATED_ANNEALING algorithm based on network-flow algorithms. The use of the SIMULATED_ANNEALING algorithm to find approximations for a special case of SGRP, with $f_i = 1$ for each $i = 1, \ldots N$, has been reported before by Van der Werf, Aarts, Heijnen, Van Meerbergen, Verhaegh & Lippens [1993].

The use of the SIMULATED_ANNEALING algorithm presupposes the definition of a solution set, a cost function, and a neighborhood structure, which are discussed in Section 8.1, 8.2, and 8.3, respectively. Section 8.4 discusses the use of the SIMULATED_ANNEALING algorithm based on network-flow algorithms. In Section 8.5 we show experimental results.

8.1 Solution space

For SGRP, we define the solution space $\mathcal{S}$ as the set of all possible PUs constructed by a feasible retiming, i.e., a retiming that satisfies the timing constraints (*xiii*) and the phase constraints (*v*). Although the register allocation and the multiplexer allocation have to be determined as well, this is considered a part of the cost evaluation as is discussed in Section 8.2. There it is shown that, given a retiming, an optimal multiplexer allocation and an optimal register allocation can be found in polynomial time.

Straightforward scheduling or retiming algorithms can be used to determine a start solution. However, in practice, applying the SIMULATED_ANNEALING algorithm starting at far from optimal solutions yields bad results. Therefore, we make use of network-flow algorithms to determine a start solution, taking the cost of the PU into account. This is done in a way similar to that for SRP, which can be solved in polynomial time using network-flow algorithms. In our approach instances of the network flow problem are identified for retiming at the *source* level and for retiming at the *target* level.

A start solution is generated in two steps. In the first step the timing constraints are satisfied. In the second step the phase constraints are satisfied, while maintaining the timing constraints. Although the solution space is defined such that the multiplexer allocation and register allocation are determined in the cost evaluation step, they are taken into account for the determination of a start solution.

8.1.1 Timing constraint satisfaction

A retiming that satisfies the timing constraints is determined by a combination of retiming at the source level and retiming at the target level as is discussed in more detail at the end of this section. To this end, we alternate between retiming a source ESFG and retiming the target ESFG. This enables us to take the result of previously retimed source ESFGs into account for the retiming of a next source ESFG.

Source level retiming

The first problem that we identify and map onto the network flow problem concerns the retiming of one of the source ESFGs that are mapped onto a target ESFG. In this case we disregard Constraints (v), so the resulting retiming may cause phase conflicts. The register sharing between edges of different source ESFGs is taken into account since we assume that other source ESFGs may determine the register allocation in the target ESFG. Since these registers are already required in the target ESFG, they come for free for the source ESFG under consideration, and determine a lower bound for the register allocation of corresponding terminals in that source ESFG as follows.

$$ra(t) \geq ra_c(t) \qquad \text{for each } t \in T,$$

where an ESFG $\mathcal{EG}$ and a register allocation bound ra_c, which is determined by other source ESFGs, are given.

The multiplexer cost is difficult to express explicitly in the network flow problem; therefore we introduce a cost term which is based on the *multiplicity* of an edge, which is defined as follows.

Definition 8.2 (multiplicity). Given are a union $\mathcal{EG}$ of source ESFGs $\mathcal{EG}_i$, $i = 1,\ldots,N$, an operator allocation al, and an operator assignment as. The *multiplicity* is a function $\theta : E \rightarrow \mathbb{N}$, which is defined as follows.

$$\theta(e) = |\{e = ((w,a),(x,b)) \in E | as(w) = as(u) \text{ and } as(x) = as(v)\}| - 1 \qquad \text{for each } e = (u,a),(v,b)) \in E.$$

□

Now we can express the cost to be minimized in the network flow problem as

$$\sum_{t \in T} ra(t) + \sum_{e=((u,a),(v,b)) \in E} \theta(e)(r(v) - r(u)).$$

The first cost term is the total number of registers in the PU. The second cost term is the sum of the changes in the weights of the source ESFG's signal edges, each of which multiplied by its degree of multiplicity. Minimizing the second term of the cost function reduces the retimed weights of the edges, especially of those with higher multiplicity. As a side effect, in practice, it is more likely that edges with equal weights will be obtained, thus saving some multiplexers. Note that when only one source ESFG is present, the multiplicity of the edges is equal to zero. As a result, only the first term of the cost expression is present, which is identical to the cost expression of SRP.

The problem of retiming one source ESFG when a register allocation bound and multiplicities are given and of using the substitution function n as defined in Definition 3.10, is defined as follows.

Definition 8.3 (synchronous retiming one ESFG problem (SREP)). Given are an ESFG $\mathcal{EG}$, a weight w, a register allocation ra_c, and multiplicities θ. The problem is equivalent to the following dual of the minimum cost flow problem. Find a retiming r and a substitution function n, for which

$$\begin{aligned}
&r(u) - r(v) \leq w(f) && \text{for each } f = (u,v) \in F, \text{ and}\\
&n(i) - n(o) \leq -w(e) && \text{for each } e = (o,i) \in E, \text{ and}\\
&r(u) - n(o) \leq -ra_c(o) && \text{for each } o = (u,a) \in O, \text{ and}\\
&n(i) - r(v) \leq -ra_c(i) && \text{for each } i = (v,b) \in I, \text{ and}
\end{aligned}$$

that maximizes

$$\sum_{v \in V} r(v)[\mathcal{O}(\ell(v)) - \mathcal{I}(\ell(v))] + \sum_{i \in I} n(i) - \sum_{o \in O} n(o) + \sum_{e=((u,a),(v,b)) \in E} \theta(e)(r(u) - r(v)).$$

□

The expression of the cost function is based on the substitution function n and retiming r instead of on the register allocation ra. It results from a reformulation that is analogous to the one used in the proof of Theorem 3.11.

Target level retiming

The second problem that we identify and map onto the network flow problem is the retiming of the operators of the target ESFG, which is equivalent to SRP. In Chapter 3 we showed that SRP can be mapped onto a network flow problem. There, it was assumed that the constraint set F resulted from a timing analysis of the SFG. Here, we derive the constraint set for the target ESFG $\mathcal{EG}'$ from the constraint set of the union ESFG $\mathcal{EG}$ of source ESFGs, by creating an edge $f' = (as(u), as(v)) \in F'$ with $w(f') = w(f)$ for each $(u,v) \in F$, where as is the operator assignment used

to construct the target ESFG. In this way the speed constraints only correspond to real paths. In order to determine a retiming of the target ESFG such that each path, whether false or real, between two registers must have an asynchronous delay less than the clock period, a timing analysis of the target ESFG must follow after a solution has been found of SGRP. This was discussed in more detail in Chapter 3 on page 45.

It is characteristic for retiming at the target level that the multiplexer cost remains the same. A disadvantage of target level retiming is that it can leave registers redundant; see Figure 8.1. This is due to the possible presence of reconvergent paths at the target level that do not correspond to real paths in the source ESFGs. If the reconvergence of paths is caused by the operator assignment, source level retiming can change the retimed weights of the paths independently from each other, whereas target level retiming equally changes the retimed weights of the paths.

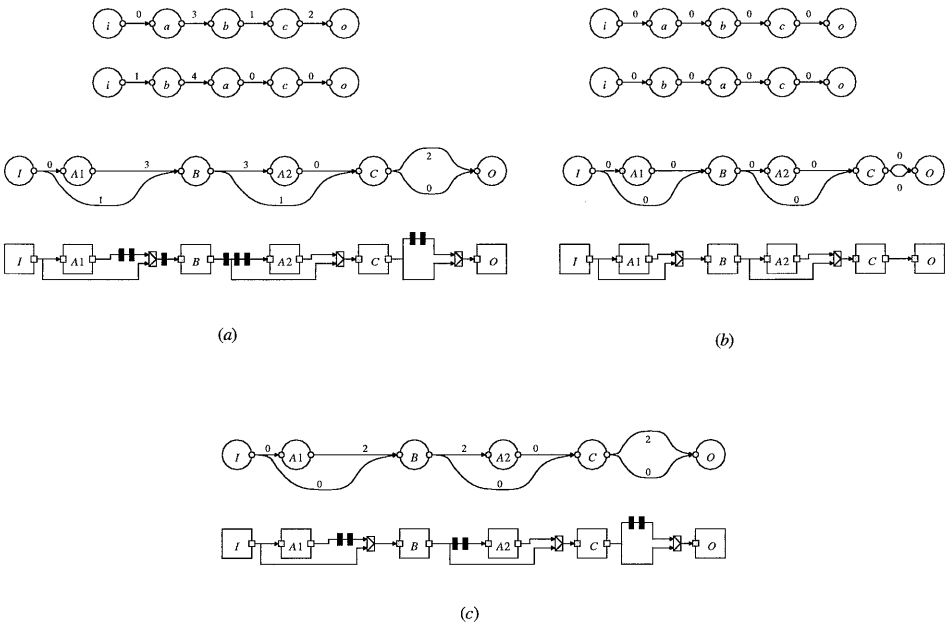


Figure 8.1. An example of source and target level retiming with two source ESFGs that are mapped on a target ESFG. In (*a*) the initial source ESFGs as well as the PU are shown. In (*b*) the source and target ESFG are shown after retiming at the source level. In (*c*) the result is shown after retiming the target ESFG of (*a*). The cost after retiming at the target level is greater than the cost after retiming at the source level, since two registers are "trapped" by each pair of reconvergent paths.

Combining source and target level retiming

We determine a retiming that satisfies the timing constraints by sequentially retiming the source ESFGs, one at a time; see Figure 8.2. To this end, we alternate between retiming a source ESFG, by solving an instance of SREP, and the target ESFG, by solving and instance of SRP. The source ESFGs are handled in a round-Robin fashion. We continue with solving instances of SREP and SRP until no improvement in cost is achieved. The register allocation that results from a target level retiming is used as the register allocation in the subsequent instance of SREP to retime the next source ESFG. Iteratively retiming the operations of a source ESFG results in retimed weights as follows:

$$w_r(e) = w(e) + \sum_{j=1}^{R}(r_j(v) - r_j(u)) + \sum_{j=1}^{P}(r_j(as(vi)) - r_j(as(u))) \quad \text{for each } e = (u,v) \in E_i,$$

where $r_j(u)$ is the retiming of an operation $u \in V_i$ in iteration j, R is the number of times an instance of SREP corresponding to $\mathcal{EG}_i$, with $i = 1,\ldots,N$, is solved and P is the number of times an instance of SRP is solved.

```
procedure TIMING_CONSTRAINT_SATISFACTION
in  a union EG of N source ESFGs EG_i,
    a weight function w,
    a target ESFG EG' defined by an operator allocation al, and
    an operator assignment as;
out a retiming r of operations of the union and target ESFGs;
begin new_cost = ∞;
    repeat old_cost = new_cost;
        for i = 1 to N
        begin solve instance of SREP to obtain a retiming r for source ESFG EG_i;
            solve instance of SRP to obtain a retiming r for target ESFG EG';
        end;
        determine new_cost;
    until old_cost ≤ new_cost;
end;
```

Figure 8.2. The TIMING_CONSTRAINT_SATISFACTION algorithm.

The iterative nature of solving instances of SREP and SRP makes the application of the OUT_OF_KILTER algorithm of Figure 4.7 suitable. Using this algorithm, a solution is not computed from scratch in every iteration; only the changes in the instances with respect to a previous computation of a solution are taken into account. The changes for the instances of SREP are in the register allocation, which may have been altered by a target level retiming. The weights of the edges are also affected by a target level retiming, but this can be considered as a reweighting, as

discussed in Section 4.2.4. The changes for the instances of SRP are in the weights of the constraint and signal edges with respect to the original weights w_{or}, i.e.,

$$w(e') = w_{or}(e') + r(v) - r(u) \qquad \text{for each } e' = ((as(u), a), (as(v), b)) \in E,$$
$$w(f') = w_{or}(f') + r(v) - r(u) \qquad \text{for each } f' = (as(u), as(v)) \in F.$$

The application of the OUT_OF_KILTER algorithm requires that during a run of the TIMING_CONSTRAINT_SATISFACTION algorithm the primal and dual solutions be stored for instances of the network flow problems SREP for every source ESFG and for an instance of SRP for the target ESFG.

After iteratively retiming using the OUT_OF_KILTER algorithm, the retimed weights of the signal edges of the source ESFGs are determined as follows:

$$w_r(e) = w_{or}(e) + r(v) - r(u) + r(as(v)) - r(as(u))) \qquad \text{for each } e = (u, v) \in E.$$

Note that here we have only one retiming of an operation at each level, whereas if every instance of SREP and SRP were computed from scratch the retiming of every iteration would have to be accumulated.

8.1.2 Phase constraint satisfaction

Starting from a solution that satisfies the timing constraints (*vi*), we next retime each operation to an execution time at which it does not cause a phase conflict, in the following way. We confine the possible retiming of an operation to a set called *freedom*. The freedom of an operation can be informally defined as the range in which the operation can be retimed without violating the timing constraints. We extend the freedom of an operation if the freedom contains only retimings that cause phase conflicts. To extend the freedom, we make use of network-flow algorithms for retiming at the source level and retiming at the target level, while taking the cost of the PU into account. This is done in a way similar to that in the previous section, but here we create additional constraint edges, in order to extend the freedom of an operation. The retiming at the source and target levels by means of network-flow algorithms is performed in such a way that phase constraints that are satisfied are not violated again. First we discuss the application of network-flow algorithms. Next we discuss the freedom extension and the phase conflict removal.

Applying network-flow algorithms

For retiming at the source level, we restrict the source level retiming to multiples of f, the folding factor of the source ESFG $\mathcal{EG}$ at hand. This restriction implies that phase conflicts that are satisfied are not violated again. This retiming is indicated by a subscript f. Consequently, the following equalities hold:

$$w_r(e) = w(e) + f(r_f(u) - r_f(v)) \qquad \text{for each } e = ((u, a), (v, b)) \in E, \text{ and}$$
$$w_r(g) = w(g) + f(r_f(u) - r_f(v)) \qquad \text{for each } g = (u, v)) \in F.$$

A problem that can be mapped onto the network flow problem is to retime a source ESFG in the same way as described in Section 8.1.1, but with the restriction that only multiples of the folding factor can be retimed. To this end, we introduce a substitution function n_f, which is defined for each terminal, in a way similar of that for the reduction of SRP to a network flow problem as discussed in Section 3.3.4. The relation between a register allocation ra, a retiming (of multiples) r_f and the substitution function n_f is the following:

$$fn_f(i) = fr_f(v) - ra(i) \qquad \text{for each } i = (v,b) \in I, \text{ and}$$
$$fn_f(o) = fr_f(u) + ra(o) \qquad \text{for each } o = (u,a) \in O.$$

The problem is defined similarly to SREP, although the cost function is a factor f smaller.

Definition 8.4 (synchronous multiple retiming one ESFG problem (SMREP)). Given are an ESFG $\mathcal{EG}$, a weight w, a folding factor f, a register allocation ra_c, and multiplicities θ. The problem is equivalent to the following dual of the minimum cost flow problem. Find a retiming r_f and a substitution function n_f, for which

$$r_f(u) - r_f(v) \leq \lceil \tfrac{w(g)}{f} \rceil \qquad \text{for each } g = (u,v) \in F, \text{ and}$$
$$n_f(i) - n_f(o) \leq \lceil \tfrac{-w(e)}{f} \rceil \qquad \text{for each } e = (o,i) \in E, \text{ and}$$
$$r_f(u) - n_f(o) \leq \lceil \tfrac{-ra_c(o)}{f} \rceil \qquad \text{for each } o = (u,a) \in O, \text{ and}$$
$$n_f(i) - r_f(v) \leq \lceil \tfrac{-ra_c(i)}{f} \rceil \qquad \text{for each } i = (v,b) \in I, \text{ and}$$

that maximizes

$$\sum_{v \in V} r_f(v)[\mathcal{O}(\ell(v)) - \mathcal{I}(\ell(v))] + \sum_{i \in I} n_f(i) - \sum_{o \in O} n_f(o) +$$
$$\sum_{e=((u,a),(v,b)) \in E} \theta(e)(r_f(v) - r_f(u)).$$

□

Multiple retiming one source ESFG was identified before by Potkonjak & Rabaey [1989] as a transformation to be used in combination with other scheduling techniques.

The second application of network-flow algorithms in the context of phase constraint satisfaction is to solve an instance of SRP for a target ESFG, as discussed in Section 8.1.1. It changes the phase of operations, but phase constraints that have been satisfied, do not become violated again.

Freedom

To determine a retiming that satisfies the phase constraints, a retiming of an operation at either level is not allowed to violate a timing constraint. This is ensured by confining the possible retiming of an operation to a set called freedom, which is defined as follows.

Definition 8.5 (freedom). Given are an ESFG $\mathcal{EG}$ and a weight w. The freedom is a set fr, such that $fr(u) = \{r_{\text{low}}(u), ..., r_{\text{high}}(u)\}$, where $r_{\text{low}}(u) = -\min_{f=(v,u)\in F} w(f)$ and $r_{\text{high}}(u) = \min_{f=(u,v)\in F} w(f)$. □

Note that when u is an input operation, $r_{\text{low}}(u) = -\infty$, and when it is an output operation, $r_{\text{high}}(u) = \infty$.

To determine a retiming that satisfies the phase constraints, we may need to retime an operation such that its execution is assigned to a given phase. If the required retiming is not inside the operation's freedom, then we extend the freedom.

Freedom extension for an operation can be done by introducing some extra constraint edges into the ESFG that embodies it. The extra constraints are placed between every predecessor and every successor of the operation of which the freedom must be extended. Suppose that we extend the freedom of an operation u from $\{r_{\text{low}}(u), ..., r_{\text{high}}(u)\}$ to the new range $\{r'_{\text{low}}(u), ..., r'_{\text{high}}(u)\}$, with $r'_{\text{high}}(u) - r'_{\text{low}}(u) = h + r_{\text{high}}(u) - r_{\text{low}}(u)$. Then the extra constraint edge $f = (q, s)$ between a preceding operation q of operation u and one of its successors s must get a weight $w(f) = -h$. After the extended freedom has been used, the extra constraint edges become superfluous.

Figure 8.3 presents in a pseudo programming language the algorithm that moves an operation from a phase with conflicts to a free phase. The algorithm picks out of the operations that cause phase conflicts the one with the highest priority. The priority of an operation is the sum of the multiplicities of all signal edges that are incident to the operation. With this definition, it follows that an operation has a higher priority if its retiming is expected to affect to a larger extent the multiplexer cost. Freedom extension for the retiming of an operation is continued until a free phase for the operation has been found.

The iterative nature of solving instances of SMREP and SRP during a run of PHASE_CONFLICT_REMOVAL makes the application of the OUT_OF_KILTER algorithm suitable. For SMREP the change in an instance with respect to a previous computation of a solution is in the register allocation and in the retimed weights of edges, which may be altered by a target level retiming and by the retiming that removes a phase conflict. Furthermore, the weights are adjusted with respect to the original weights w_{or} because of the retiming of operations to a non-occupied phase, i.e.,

```
procedure PHASE_CONFLICT_REMOVAL
in  a union ℰ𝒢 of N source ESFGs ℰ𝒢_i,
    folding factors f_i,
    a weight function w,
    a target ESFG ℰ𝒢' defined by an operator allocation al, and
    an operator assignment as;
out a retiming r_f of operations of a source ESFG ℰ𝒢_i and
    a retiming r of operations of the target ESFG;
begin while a phase conflict exists
    begin select operation u that causes a phase conflict and which has a maximum priority ;
        while u cannot be retimed to a free phase
        begin add extra constraint edges to extend freedom;
            solve instance of SMREP to obtain
                a retiming r_f for source ESFG ℰ𝒢_i with u ∈ V_i;
            solve instance of SRP to obtain
                a retiming r for target ESFG ℰ𝒢';
        end;
        retime u to a free phase;
        remove extra constraint edges used to extend freedom;
    end;
end;
```

Figure 8.3. The PHASE_CONFLICT_REMOVAL algorithm.

$$w(e) = w_{\text{or}}(e) + r(v) - r(u) + r(as(v)) - r(as(u))$$
$$\text{for each } e = ((u,a),(v,b)) \in E, \text{ and}$$
$$w(f) = w_{\text{or}}(f) + r(v) - r(u) + r(as(v)) - r(as(u))$$
$$\text{for each } f = (u,v) \in F.$$

For SRP the change is in the weights of the constraint and signal edges corresponding to each of the source ESFGs $\mathcal{EG}_i$, with $i = 1,\ldots,N$, i.e.,

$$w(e') = w_{\text{or}}(e') + r(v) - r(u) + f_i(r_{\text{f}}(v) - r_{\text{f}}(u))$$
$$\text{for each } e' = ((as(u),a),(as(v),b)) \in E',$$
$$w(f') = w_{\text{or}}(f') + r(v) - r(u) + f_i(r_{\text{f}}(v) - r_{\text{f}}(u))$$
$$\text{for each } f' = (as(u),as(v)) \in F',$$

where $u, v \in V_i$. The influence of the target level retiming cannot in this case be considered as a reweighting, as it was discussed in Section 4.2.4. Retiming an operation of an ESGF $\mathcal{EG}_i$ can only be done in multiples of f_i, and these may only become possible when target level retiming increases the retimed weight of an edge to at least a next multiple of f_i.

The application of the OUT_OF_KILTER algorithm requires that the primal and dual solutions be stored for instances of the network flow problems SMREP for every source ESFG and for an instance of SRP for the target ESFG. By applying the OUT_OF_KILTER algorithm, a new retiming can be efficiently computed at the source and target levels such that the extra constraints are satisfied and, consequently, give the operation the required freedom.

After iteratively retiming, using the OUT_OF_KILTER algorithm, the retimed weights of the signal edges of each of the source ESFGs $\mathcal{EG}_i$, with $i = 1,\ldots,N$, are determined as follows:

$$w_r(e) = w_{\text{or}}(e) + r(v) - r(u) + f_i(r_{\text{f}}(v) - r_{\text{f}}(u)) + r(as(v)) - r(as(u)))$$
$$\text{for each } e = (u,v) \in E_i.$$

Note that the retimed weight must be used to determine the freedom when the source and target ESFG have been retimed. Furthermore, the weight of an extra constraint edge $f = (q,s) \in F_i$ must be adjusted for an already existing retiming as follows:

$$w(f) = -h + r(q) - r(s) + f_i(r_{\text{f}}(q) - r_{\text{f}}(s)) + r(as(q)) - r(as(s)),$$

where the contribution of all three kinds of retiming, i.e., retiming and multiple retiming at the source level and retiming at the target level, are taken into account and the extra freedom is h.

8.2 Cost

The cost of a solution is determined by a register allocation and a multiplexer allocation. In case we have solved an instance of SRP to determine a retiming of

the operations in the target ESFG, the register allocation follows directly from the resulting solution. However, it is also possible to compute efficiently the register cost without retiming the operations of the target ESFG. From SRP the problem of finding a minimal register allocation given the retiming of operations can be derived in a straightforward fashion, which results in the following problem.

Definition 8.6 (register allocation problem (RAP)). Given are an SFG $\mathcal{G}$ and a weight w. The problem is to find a register allocation ra, for which

$$ra(i) + ra(o) \geq w(e) \qquad \text{for each } e = (o, i) \in E, \text{ and}$$
$$ra(t) \geq 0 \qquad \text{for each } t \in T, \text{ and}$$

that minimizes

$$\sum_{t \in T} ra(t).$$

□

RAP is equivalent to the dual of the maximum-weight bipartite matching problem (MWBMPD) of Definition 4.6. Consequently, it can be reduced to NFPD and, as discussed in Section 4.2.5, it can be solved in polynomial time by, e.g., the MINIMUM_COST_MAXIMUM_FLOW algorithm. Moreover, when slightly different instances of RAP have to be solved repeatedly, the OUT_OF_KILTER algorithm can be applied in order to compute the solutions more efficiently, in an incremental way.

Note that the instance of RAP that corresponds to an ESFG does not necessarily contain a connected graph G. Therefore, it may be partitioned into a number of instances of RAP, each of which contains a connected graph, corresponding with a component of G.

The multiplexer cost can be computed by inspecting the input terminals of the operators, according to Definition 2.18. In conclusion, the cost can be computed in polynomial time.

8.3 Neighborhood structure

The neighborhood structure is defined in such a way that a cost improvement can be achieved and the timing and phase constraints are maintained. The local search algorithm may step from one solution to a neighbor with two types of moves. The first one is done by retiming a single operation with a retiming within its freedom and to an unoccupied phase. The second type of move consists of retiming two operations from the same source ESFG that are assigned to the same operator, such that they switch their phases. The retiming values of the two operations must be inside their freedom. The neighborhood structure is formally defined as follows.

Definition 8.7 (retiming neighborhood structure). Given are a union $\mathcal{EG}$ of source ESFGs $\mathcal{EG}_i$ with folding factors f_i, $i = 1,\ldots,N$, a weight w, a retiming r, and an operator assignment as. The *retiming neighborhood structure* ($\mathcal{N}_{\text{retiming}}$) is given as follows. The neighborhood $\mathcal{N}_{\text{retiming}}(r)$ of a retiming r is the set of all retimings $r^\bullet$, implying phases $ph^\bullet$ and freedoms $fr^\bullet$,

(i) for which there is an operation $v \in V_i$ with $i = 1,\ldots,N$, satisfying

$r^\bullet(v) \in fr(v)$ and
$ph^\bullet(w) \neq ph^\bullet(v)$ for each $w \in V_i \setminus \{v\}$.
$r^\bullet(w) = 0$ for each $w \in V \setminus \{v\}$, or

(ii) for which there is one pair of operations $u, v \in V_i$ with $u \neq v$, $as(u) = as(v)$, and $i = 1,\ldots,N$, satisfying

$r^\bullet(u) \in fr(u)$ and $r^\bullet(u) \in fr^\bullet(u)$ and
$r^\bullet(v) \in fr(v)$ and $r^\bullet(v) \in fr^\bullet(v)$ and
$ph^\bullet(u) = ph(v)$ and
$ph^\bullet(v) = ph(u)$ and
$r^\bullet(w) = 0$ for each $w \in V \setminus \{u, v\}$.

□

A straightforward generation of a solution from a neighborhood is to randomly select one or two operations from one of the source ESFGs and randomly select a retiming that meets the constraints of the neighborhood. When applying such a generation, a careful examination of the results shows that many steps are made that do not change the cost. The cause of this is that the change in the cost is affected mainly by retiming operations incident to an edge of which the (retimed) weight determines a register allocation; see Figure 8.4. This motivates us to search for ways to increase the probability that a neighbor is selected that improves the cost. In Chapter 5 it was discussed how the experience obtained with applying the SIMULATED_ANNEALING algorithm for the operator assignment problem led to a reduced neighborhood structure. There it was possible to exclude neighbors that cannot lead to a cost improvement. Here we do not exclude neighbors, but we bias the generation of neighbors that are more likely to have a better cost. Therefore, the generation of a neighboring solution is done such that operations are favoured that are incident to an edge of which the (retimed) weight determines a register allocation. To this end, we partition the set of signal edges of the source ESFGs into three disjoint sets in the following way.

Definition 8.8 (edge partition). Given are a union $\mathcal{EG}$ of source ESFGs $\mathcal{EG}_i$, $i = 1,\ldots,N$, a register allocation ra, and an assignment as. The sets of critical edges, non-critical, and zero-weight edges are given as follows:

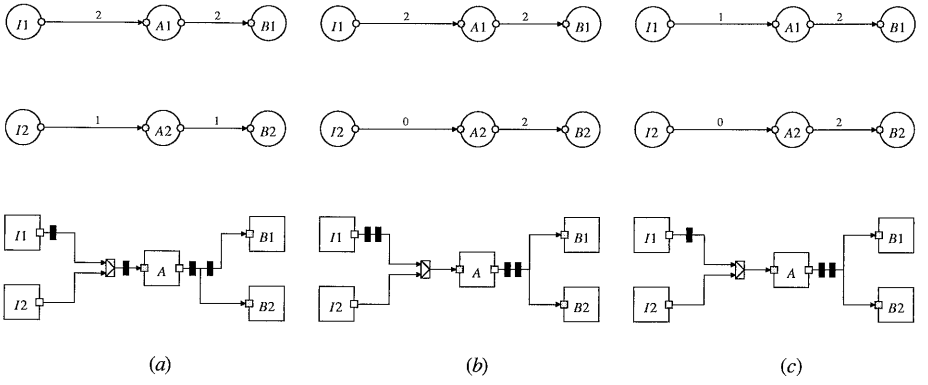


Figure 8.4. An example of two source ESFGs and the PU on which they are mapped. The initial situation is shown in (*a*). After retiming operation *A*2 of the bottom source ESFG one clock cycle earlier, the PU of (*b*) results. This PU has the same register cost as the PU of (*a*), since the weights of the edges connected to operation *A*2 of the bottom source ESFG do not determine the register allocation. After retiming operation *I*1 of the top source ESFG in (*b*) one clock cycle later, the PU of (*c*) results. This PU has a lower register cost than the PU of (*a*), since the weight of the edge connected to operation *I*1 determines the register allocation.

$$E_c = \{e = ((u,a),(v,b)) \in E | w(e) = ra((as(u),a)) + ra((as(v),b)) \neq 0\},$$
$$E_{nc} = \{e = ((u,a),(v,b)) \in E | 0 < w(e) < ra((as(u),a)) + ra((as(v),b))\},$$
$$E_{zw} = \{e \in E | w(e) = 0\}.$$

□

The generation of a neighboring solution is now performed in a number of steps. In the first step one of the edge sets E_c or E_{nc} is randomly selected. In the next step an edge is randomly selected from this set. After an edge is selected, an operation is chosen at random from one of its ends. The algorithm searches a free target phase that is reachable within the freedom of the operation. If this fails, then the algorithm tries to find another operation that is assigned to the same operator and to retime both the operations such that they switch phases. These retimings must lay within their freedoms. If this is not possible, then the algorithm returns without performing any retiming.

There are two reasons not to restrict the neighborhood to operations connected to a critical edge, but still to generate solutions by retiming operations connected to non-critical edges. The first reason is that the multiplexer allocation can only be changed by retiming operations if it affects the weights of edges running between the same input and output terminals. If the weights are different, retiming an operation connected to one of the edges may make the weights equal, resulting in a reduction of the number of multiplexers; see Figure 8.5. The criticalness of such an edge is of no influence here.

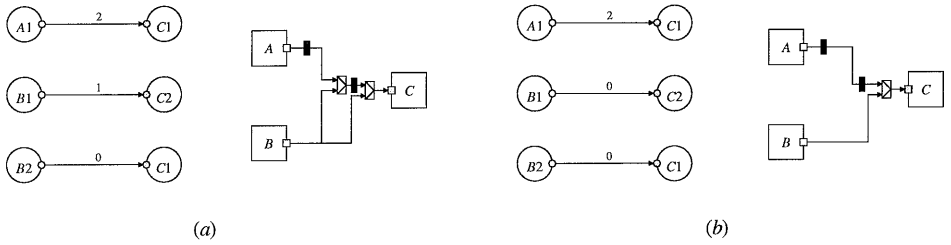


Figure 8.5. An example of three source ESFGs that are mapped on a PU before retiming the middle source ESFG (*a*) and after retiming the middle source ESFG (*b*). Changing the weight of the non-critical edge of the middle source ESFG by retiming affects the multiplexer cost of the PU.

The second reason to consider non-critical edges is that, given a retiming, different register allocations may exist with the same cost, but resulting in a different partition of the edges in critical, non-critical, and zero weight edges. An example of two register allocations resulting in different sets E_c, E_{nc}, and E_{zw} is given

in Figure 8.6. If an edge is non-critical, a reduction of its weight does not influence the register allocation. However, if another register allocation for the same retiming would make the edge critical, then its weight must be reduced in order to decrease the register cost. Therefore, the selection of non-critical edges increases the probability that the register cost is reduced.

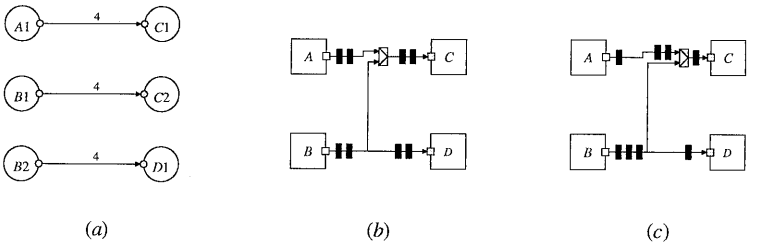


Figure 8.6. An example of three source ESFGs (*a*) that can be mapped on two PUs (*b*) and (*c*) with the same cost but different register allocations. The PU of (*b*) has a register allocation such that the edge of the middle source ESFG of (*a*) is critical, whereas the PU of (*c*) has a register allocation such that the edge of the middle source ESFG is not critical.

We do not have to start a generation at a zero weight edge, since no registers are allocated to implement its (retimed) weight. Maybe its retiming can save a multiplexer, but then another edge between the same input and output terminals is certainly critical or non-critical. Such an edge may be a starting point for the generation of a neighboring solution and its weight may be decreased. Consequently, the possibility that a multiplexer is saved is not affected by not starting a generation at the set of zero weight edges.

8.4 Simulated annealing

Now that we have given the solution space, the generation of a start solution, the cost computation, and the neighborhood structure we can apply the SIMULATED_ANNEALING algorithm to approximate an optimal solution for SGRP. The SIMULATED_ANNEALING algorithm as shown in Figure 4.2 is modified into the RETIMING algorithm, which uses network-flow algorithms; see Figure 8.7. Throughout a run of the RETIMING algorithm, a retiming solution consists of three components: retiming and multiple retiming at the source level, and retiming at the target level. The multiple retiming at the source level and the retiming at the target level are results from network-flow algorithms. Since they are determined in an iterative way, the OUT_OF_KILTER algorithm is ideally applied. The weights of edges in the input for the runs of the OUT_OF_KILTER algorithm are determined in

the same way as for PHASE_CONFLICT_REMOVAL, as discussed in Section 8.1.2. Moreover, the OUT_OF_KILTER algorithm can be applied for all instances of SRP within TIMING_CONSTRAINT_SATISFACTION, PHASE_CONFLICT_REMOVAL, and RETIMING. The instances of SMREP within PHASE_CONFLICT_REMOVAL and RETIMING can also be solved with the OUT_OF_KILTER algorithm.

```
procedure RETIMING
in  a union ℰ𝒢 of N source ESFGs ℰ𝒢_i,
    folding factors f_i,
    a weight function w,
    a target ESFG ℰ𝒢' defined by an operator allocation al, and
    an operator assignment as;
out a retiming r of operations of the union and target ESFGs;
begin k = 0;
    INITIALIZE(c_0, L_0);
    TIMING_CONSTRAINT_SATISFACTION(ℰ𝒢, w, ℰ𝒢', as, r);
    PHASE_CONFLICT_REMOVAL(ℰ𝒢, f, w, ℰ𝒢', as, r_f1, ..., r_fN, r);
    repeat
        for t=1 to L_k
        begin
            generate r• ∈ 𝒩_retiming(r);
            solve instances of RAP to obtain register allocation ra;
            compute multiplexer allocation;
            compute delta_cost;
            if delta_cost < 0 or exp(−delta_cost / c_k) > random[0,1) then r = r•
            else skip
        end;
        for i = 1 to N
        begin
            solve instance of SMREP to obtain a retiming r_fi for source ESFG ℰ𝒢_i;
            solve instance of SRP to obtain a retiming r for target ESFG ℰ𝒢';
        end;
        k=k+1;
        CALCULATE_LENGTH(L_k);
        CALCULATE_CONTROL(c_k);
    until stop criterion;
end;
```

Figure 8.7. The RETIMING algorithm.

To compute the register cost, the register allocation can be computed by solving instances of RAP. The operation assignment determines a partition of the graph such that each part is a connected graph. An instance of RAP corresponds to each part, which can be solved incrementally by the OUT_OF_KILTER algorithm. To compute the cost of a generated solution, only those instances of RAP have to be

solved of which the weight of an edge has been affected. At the end of a temperature step all weights can be affected and, consequenlty, all instances of RAP have to be solved. Note that when more than one optimal solution exists, the register allocation computed in this way may differ from the one following from a solution to the instances of SRP.

The multiplexer cost can be computed incrementally during one temperature step of the RETIMING algorithm. For a generated solution, the multiplexers are only changed at input terminals that are incident to edges that are connected to the retimed operations. Only when finding a solution to SMREP at the end of a temperature step, a complete new computation of the multiplexer cost is performed, since then all operations may have obtained a different retiming.

The generation of a neighboring solution requires an overview of the sets E_c and E_{nc}. These sets can also be maintained in an iterative way, since only edges can change that are part of updated clusters of edges, each corresponding with an instance of RAP.

To apply the OUT_OF_KILTER for the various instances of SMREP, SREP, SRP, and RAP, the primal and dual solutions of the corresponding instances of the network flow problem have to be stored throughout a run of the RETIMING algorithm.

8.5 Experimental results

As an example we use ISGRP1, which contains the same SFGs as used in instance IOASP1 of OASP as defined in Section 5.1.3, concerning the mapping of a Discrete Cosine Transform (DCT) and an Inverse Discrete Cosine Transform (IDCT) onto one PU; see Figure 5.2. The two ESFGs are mapped onto one PU, each with a folding factor of two. From the solutions of two previous steps, solving an instance of OASP and solving an instance of ODP, it follows that the cost of the allocated operators is 780 and the cost of multiplexers will be at least 322. The latter gives us a lower bound of 322 area units. The clock period is such that no speed constraint edges are present. If the two ESFGs are mapped without timefolding onto separate PUs, then the cost, which consist of merely operators, is 1200 area units for each PU. So, in that case the total total cost is 2400 area units, which can be considered as an upper bound.

Results of the RETIMING algorithm for this example are listed in Table 8.1. Experiments were performed using a fast cooling schedule starting at a low temperature as well as a slow cooling schedule starting at a high temperature. The former schedule more or less results in an iterative improvement because the probability that cost deteriorations are accepted is very low. Average values were obtained for 50 runs to average out statistical fluctuations of both algorithms. After the removal of phase conflicts, the start solution has a cost of 839. The best solution has a cost

of 687, which is an improvement of 18%. If we subtract the lower bound from these cost, the improvement is 29%. From the results listed in Table 8.1 we conclude that the RETIMING algorithm gives good solutions. Moreover, we see that the acceptance of cost deteriorations, as is the case with a slow cooling schedule starting at a high temperature, is important to give good solutions.

Table 8.1. Results of the RETIMING algorithm concerning the mapping of a DCT and an IDCT with each a folding factor of two, onto a PU. Average values were obtained for 50 runs.

	cost (area units)			cost − lower bound (area units)		
cooling settings	best	average	worst	best	average	worst
slow cooling	687	731	800	365	409	478
fast cooling	767	813	839	445	491	517

9

Conclusions

We considered the design of area-efficient *Processing Units* (PUs) that can execute one or more functions, specified as *Signal Flow Graphs* (SFGs), in one or more clock cycles and at a high clock frequency. An approach to designing PUs is presented that uses a combination of multiplexing, timefolding, and retiming techniques. Although the effect of each of the techniques, and certainly that of a combination of them, on the area of the PU and the clock frequency at which it can execute the SFGs cannot be well predicted in general, an efficient solution method is presented that leads to near-optimal solutions.

In Chapter 2 processing unit design is formally presented as a combinatorial optimization problem that is called the *processing unit design problem* (PUDP). The formalization is supported by models of SFGs, PUs, and their timing behaviour. Furthermore, the relationship between SFGs and a PU on which they can be executed is modelled in two parts. First the SFGs are related to a target SFG by an assignment of operations in the source SFG to operations that are allocated in the target SFG. The execution of an operation of a source SFG is retimed relative to its initial execution time, which determines the number of clock cycles a signal must be delayed. The allocation, assignment, and retiming of operations are the main decision quantities in the processing unit design problem. In the second part the target SFG is implemented by a PU through an allocation of registers and multiplexers for the communication between the allocated operators. This allocation

can be considered as a sophisticated cost evaluation.

In Chapter 3 we proved that PUDP is NP-hard. To approach the problem, we decomposed PUDP into the subproblems OALP, OASP, ODP, and GRP, of which the latter is again subdivided into TAP and SGRP. These subproblems are informally defined as follows. OALP is the problem of finding an allocation of operations in the target SFG such that the cost of operators is minimal. OASP is the problem of finding an assignment of operations in the source SFGs to operations in the target SFG such that the cost of multiplexers is minimal. ODP is the problem of finding a refined assignment and allocation based on an initial allocation and assignment, while taking into account constraints concerning the absence of false loops in the PU. The optimization goal of ODP is a minimum weighted cost of multiplexers and operators. GRP is the problem of finding a retiming of operations, such that two operations of one source SFG are executed in different phases if they are executed on the same operator, and such that the PU can operate at the required clock frequency. TAP translates the constraints expressed using the asynchronous timing behaviour into constraints concerning the synchronous timing behaviour involving the retiming of two operations in an SFG. SGRP is the problem to find a retiming of operations, such that two operations of one source SFG are executed in different phases if they are executed on the same operator, and such that the synchronous timing constraints are satisfied. The optimization goal of SGRP is a minimum weighted cost of multiplexers and registers.

OASP, ODP, GRP and SGRP are NP-hard, even for special cases where either no multiplexing or no timefolding is involved. TAP and OALP can be solved in polynomial time; the latter in a trivial way. TAP can be solved using a combination of longest and shortest paths algorithms, which are a special kind of network flow algorithms. Furthermore, SGRP has an interesting special case, SRP, concerning the retiming of a single source SFG that is mapped on its own on a target SFG. This special case can be solved in polynomial time, since it can be reduced to the network flow problem.

In Chapter 4 we describe basic solution methods that we apply to PUDP. We discussed two classes of algorithms: local search algorithms and network flow algorithms.

Chapter 5 presents an approximation algorithm for OASP. It finds a near-optimal operator assignment using simulated annealing or iterative improvement. For the mapping of relatively small source SFGs onto a PU, both local search algorithms give good results within acceptable run times by using straightforward neighborhood structures based on one- and two-exchanges. We introduced reduced neighborhood structures to handle large problem instances that are based on knowledge of the connectivity of PUs and careful examination of experimental

results. These demonstrate the advantage of simulated annealing over iterative improvement when using the reduced neighborhoods. Then, the probability of getting stuck in a local minimum is larger than when using the straightforward neighborhoods.

Chapter 6 presents an approximation algorithm for ODP. The method is based on a combination of simulated annealing and an algorithm that derives constraints that prevent the occurrence of false loops in a PU. These constraints are derived in an incremental way because the way constraints are defined makes many of them redundant such that derivation of all constraints takes too much run time. Although not all constraints are known at the beginning of a simulated annealing run, this does not affect the quality of the solution. Since most complex circuits are detected at high temperatures steps with high acceptance ratios, simulated annealing can still balance the cost of operators and multiplexers very well.

In Chapter 7, solution methods for TAP are presented. We presented two alternatives in addition to a classical constraint derivation algorithm following the theory of Leiserson, Rose & Saxe [1983]. Both algorithms are based on the redundancy of constraints. One algorithm derives fewer redundant constraints than the other and consequently is more memory efficient, while the other allows more redundancy but is more run-time efficient. The advantages of the new algorithms over the classical one become more pronounced when the clock period is small compared to the largest path delay in the SFG. We show that for instances where the clock period is substantially smaller than the longest path in the SFG, the new timing analysis for retiming is substantially faster than existing methods.

In Chapter 8 an approximation algorithm for SGRP is presented. The method is based on simulated annealing in combination with network flow algorithms. Feasible start solutions are obtained using network flow algorithms. Also the cost of neighboring solutions is calculated by network flow algorithms. Since subsequent instances of network flow problems must be solved that differ only slightly, the efficient out-of-kilter algorithm is applied. Straightforward generation of solutions from a neighborhood results in simulated annealing runs in which many steps are made that do not change the cost. To make the algorithm more greedy, we increased the probability of generating a neighboring solution that leads to a cost improvement. In this case, we did not apply a reduced neighborhood structure, since it is very hard to determine accurately cost improving neighbors.

We could have considered a different implementation of simulated annealing in which a neighboring solution is generated and unconditionally accepted based on an accurate overview of all neighboring solutions and their costs [Greene & Supowit, 1986]. This is considered more complex, since it requires much computational effort to maintain such an overview for large neighborhoods. Furthermore,

the neighborhood reduction for OASP and the biased generation for SGRP make the application of simulated annealing very practical, such that there was no need to search for different local search algorithms. The run times of simulated annealing of the experiments described in this thesis are typical for the way we used simulated annealing. However, simulated annealing is flexible in the sense that a designer of processing units can make a trade-off between run times and quality of results. Either better quality or faster run times can be achieved. Furthermore, simulated annealing is complementary to heuristic constructive algorithms. These algorithms can be used to generate start solutions in a first step, which simulated annealing may improve in a second step.

Since the optimization goal of processing unit design is minimum area, whereas the clock frequency is a constraint, the retiming technique is considered the most important one since it yields at least a feasible PU. On the other hand, the retiming step is the only step that can fail if a feasible solution cannot be found. In many cases, an area reduction that can be achieved by resource sharing may not be significant. In such situations, an IC designer may want to reduce the multiplexing and/or the folding factor in order to find trade-offs between operator cost on the one hand, and register and multiplexer cost on the other hand.

Bibliography

AARTS, E.H.L., F.M.J. DE BONT, E.H.A. HABERS, AND P.J.M. VAN LAARHOVEN [1985], Statistical cooling: a general approach to combinatorial optimization problems, *Philips Journal of Research* **4**, 193–226.

AARTS, E.H.L., AND J.H.M. KORST [1989], *Simulated Annealing and Boltzmann Machines*, John Wiley & Sons, Chichester.

AARTS, E.H.L., AND J.K. LENSTRA (eds.) [1997], *Local Search in Combinatorial Optimization*, John Wiley & Sons, Chichester.

AHUJA, R.K., T.L. MAGNANTI, AND J.B. ORLIN [1989], Network flows, in: G.L. Nemhouser, A.H.G. Rinnooy Kan, and H.J. Todd (eds.), *Handbooks in Operations Research and Management Sciences*, Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 211–370.

BAKER, K.R. [1974], *Introduction to Sequencing and Scheduling*, John Wiley & Sons, New York.

BELLMAN, R. [1958], On a routing problem, *Quarterly Applied Mathematics* **16**, 87–90.

BOOTH, AND BOOTH [1965], *Automatic Digital Calculators*, Butterworth, London.

BORGERS, S.M.C., W.A.L. HEIJNEMANS, E. DE NIET, AND P.H.N. DE WITH [1988], An experimental digital VCR with 40mm drum, single actuator and dct-based bit-rate reduction, *IEEE Transactions on Consumer Electronics* **34**, 597–605.

BORRIELLO, G., AND E. DETJENS [1988], High-level synthesis: current status and future directions, *Proceedings of the Design Automation Conference*, 477–482.

COFFMAN, JR., E.G. (ed.) [1976], *Computer and Job-Shop Scheduling Theory*, John Wiley & Sons, New York.

CONWAY, R.W., W.L. MAXWELL, AND L.W. MILLER [1967], *Theory of Scheduling*, Addison-Wesley, Reading, MA.

DE LOORE, B.J.S., P. CROMBEZ, A. DELARUELLE, P. SHERIDAN, R. WOUDSMA, C. NIESSEN, J. BIESTERBOS, W. GUBBELS, AND W. REPKO [1992], The design of a competitive ASIC for the consumer market using the Piramid design system, *Proceedings of Fifth Annual IEEE International ASIC*

Conference and Exhibit, 520–524.

DIJKSTRA, E.W. [1959], A note on two problems in connexion with graphs, *Numerische Mathematik* **1**, 269–271.

EDMONDS, J., AND R.M. KARP [1972], Theoretical improvements in algorithmic efficiency for network flow problems, *Journal of the Association for Computing Machinery* **19**, 248–264.

FLOYD, R.W. [1962], Algorithm 97: Shortest path, *Communications of the ACM* **5**, 345.

FLUITER, B.L.E. DE, E.H.L. AARTS, J.H.M. KORST, W.F.J. VERHAEGH, AND A. VAN DER WERF [1996], The complexity of generalized retiming problems, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **15**, 1340–1353.

FORD, L.R., AND D.R. FULKERSON [1962], *Flows in Networks*, Princeton University Press, Princeton, NJ.

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.

GAREY, M.R., D.S. JOHNSON, AND L. STOCKMEYER [1976], Some simplified NP-Complete graph problems, *Theory of Computer Science* **1**, 237–267.

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.

GEURTS, W., F. CATTHOOR, AND H. DE MAN [1993a], Heuristic techniques for the synthesis of complex functional units, *Proceedings of the European Conference on Design Automation*, 552–556.

GEURTS, W., F. CATTHOOR, AND H. DE MAN [1993b], Quadratic zero-one programming based synthesis of application specific data paths, *Proceedings of the International Conference on Computer-Aided Design*, 522–525.

GREENE, J.W., AND K.J. SUPOWIT [1986], Simulated annealing without rejected moves, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **5**, 221–228.

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 in high level synthesis, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **10**, 464–475.

KERNIGHAN, B.W., AND S. LIN [1970], An efficient heuristic approach proce-

dure for partitioning graphs, *Bell System Technical Journal* **49**, 291–308.

KUHN, H.W. [1955], The Hungarian method for the assignment problem, *Naval Res. Log. Quart.* **2**, 83–97.

LAARHOVEN, P.J.M. VAN, AND E.H.L. AARTS [1987], *Simulated Annealing: Theory and Applications*, Reidel, Dordrecht.

LEISERSON, C.E., F.M. ROSE, AND J.B. SAXE [1983], Optimizing synchronous circuitry by retiming, *Proceedings of the Third Caltech Conference on Very Large Scale Integration.*

LEISERSON, C.E., AND J.B. SAXE [1991], Retiming synchronous circuitry, *Algorithmica* **6**, 5–35.

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, *Proceedings of VLSI Signal Processing VII*, 24–32.

MARTIN, R.S., AND J.P. KNIGHT [1993], Operations research in the high-level synthesis of integrated circuits, *Computers and 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 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.

MÜNZNER, A., AND G. HEMME [1991], Converting combinational circuits into pipelined data paths, *Proceedings of the International Conference on Computer-Aided Design*, 368–371.

NEMHAUSER, G.L., AND L.A. WOLSEY [1988], *Integer and Combinatorial Optimization*, John Wiley & Sons, New York.

NOTE, S., F. CATTHOOR, AND H. DE MAN [1989], Definition and assignment of complex data-paths suited for high throughput applications, *Proceedings of the International Conference on Computer-Aided Design*, 108–111.

NOTE, S., W. GEURTS, F. CATTHOOR, AND H. DE MAN [1991], Cathedral III: Architecture-driven high-level synthesis for high throughput dsp applications, *Proceedings Design Automation Conference*, 597–602.

OPPENHEIM, A.V., AND R.W. SCHAFER [1975], *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

PAPADIMITRIOU, C.H., AND K. STEIGLITZ [1982], *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs.

PARK, I.C., AND C.M. KYUNG [1993], Famos: An efficient scheduling algorithm for high-level synthesis, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **12**, 1437–1448.

PARK, N., AND F.J. KURDAHI [1989], Module assignment and interconnect sharing in register-transfer synthesis of pipelined data paths, *Proceedings of the International Conference on Computer-Aided Design*, 16–19.

PARK, N., AND A.C. PARKER [1988], Sehwa: A software package for synthesis of pipelines from behavioral specifications, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **7**, 356–370.

PARKER, A.C., J. PIZARRO, AND M. MLINAR [1986], Maha: A program for datapath synthesis, *Proceedings of the Design Automation Conference*, 461–466.

PAULIN, P.G., AND J.P. KNIGHT [1989], Force-directed scheduling for the behavioural synthesis of ASICs, *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* **8**, 661–679.

PINEDO, M. [1995], *Scheduling: Theory, Algorithms, and Systems*, Prentice-Hall, Englewood Cliffs, NJ.

POTKONJAK, M., AND J. RABAEY [1989], A scheduling and resource allocation algorithm for hierarchical signal flow graphs, *Proceedings of the Design Automation Conference*, 7–12.

RABBAT, G. (ed.) [1980], *Proceedings of the International Conference on Circuits and Computers*, IEEE, New York.

RABINER, L.R., AND B. GOLD [1975], *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

RAO, K.R., AND P. YIP (eds.) [1990], *Discrete Cosine Transform: Algorithms, Advantages, and Applications*, Academic Press, New York.

ROBERTS, R.A., AND C.T. MULLIS [1987], *Digital Signal Processing*, Addison-Wesley, Reading, MA.

ROHRER, R.A. (ed.) [1982], *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, IEEE, New York.

SAXE, J.B. [1985], *Decomposable Searching Problems and Circuit Optimization by Retiming: Two Studies in General Transformations of Computational Structures*, Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University.

SCHRIJVER, A. [1986], *Linear and Integer Programming*, John Wiley & Sons, Chichester.

SHENOY, N., AND R. RUDELL [1994], Efficient implementation of retiming, *Proceedings of the International Conference on Computer-Aided Design*, 226–233.

SONDERVAN, J. [1993], Retiming and logic synthesis, *Electronic Engineering* **65**, 33–36.

SONDERVAN, J. [1994], Minimize time delays and reduce circuit density by retiming a design, *EDN (European Edition)* **39**, 107–108.

STOK, L. [1992], False loops through resource sharing, *Proceedings of the International Conference on Computer-Aided Design*, 345–348.

STOK, L. [1994], Data path synthesis, *Integration, the VLSI Journal* **18**, 1–71.

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.

VAN MEERBERGEN, J.L., P.E.R. LIPPENS, W.F.J. VERHAEGH, AND A. VAN DER WERF [1995], Phideo: high-level synthesis for high throughput applications, *Journal of VLSI Signal Processing* **9**, 89–104.

VERHAEGH, W.F.J. [1995], *Multidimensional Periodic Scheduling*, Ph.D. thesis, Eindhoven University of Technology.

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 of Integrated Circuits and Systems* **14**, 945–960.

WARSHALL, S. [1962], A theorem on boolean matrices, *Journal of the ACM* **9**, 11–12.

WERF, A. VAN DER, E.H.L. AARTS, E.W. HEIJNEN, J.L. VAN MEERBERGEN, W.F.J. VERHAEGH, AND P.E.R. LIPPENS [1993], A new method for retiming multi-functional processing units, *Proceedings of the VLSI 93 Conference*, 5.2.1–5.2.10.

WERF, A. VAN DER, W.H.A. BRÜLS, R.P. KLEIHORST, E. WATERLANDER, M.J.W. VERSTRAELEN, AND T. FRIEDRICH [1997], I.McIC: A single-chip MPEG2 video encoder for storage, *Proceedings of the International Solid-Sate Circuits Conference*, 254–255.

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.

WERF, A. VAN DER, M.J.H. PEEK, E.H.L. AARTS, J.L. VAN MEERBERGEN, P.E.R. LIPPENS, AND W.F.J. VERHAEGH [1992], Area optimization of multi-functional processing units, *Proceedings of the International Conference of Computer-Aided Design*, 292–299.

WERF, A. VAN DER, J.L. VAN MEERBERGEN, E.H.L. AARTS, W.F.J. VERHAEGH, AND P.E.R. LIPPENS [1994], Efficient timing constraint derivation for optimally retiming high speed processing units, *Proceedings of the 7th International Symposium on High-Level Synthesis*, 48–53.

WOUDSMA, R., F. BEENKER, J.L. VAN MEERBERGEN, AND C. NIESSEN [1990], Piramid: an architecture-driven silicon compiler for complex DSP applications, *Proceedings IEEE International Symposium on Circuits and Systems*,

2696–2700.

YANNAKAKIS, M. [1990], The analysis of local search problems and their heuristics, *Lecture Notes in Computer Science*, Number 415, 298–310.

Author index

Aarts, E.H.L., 14, 33, 65–67, 82, 108, 128
Ahuja, R.K., 68, 115

Baker, K.R., 14
Beenker, F., 12
Bellman, R., 72
Biesterbos, J., 12
Bont, F.M.J. de, 67
Booth, 88, 93
Borgers, S.M.C., 86
Borriello, G., 12
Brüls, W.H.A., vii

Camposano, R., 12
Catthoor, F., 15
Coffman, Jr., E.G., 14
Conway, R.W., 14
Crombez, P., 12

Delaruelle, A., 12
Detjens, E., 12
De Loore, B.J.S., 12
De Man, H., 15
Dijkstra, E.W., 71

Edmonds, J., 74
Elmasry, M.I., 14

Floyd, R.W., 55, 119
Fluiter, B.L.E. de, 33
Ford, L.R., 14, 61, 72, 74
French, S., 14
Friedrich, T., vii
Fulkerson, D.R., 14, 61, 72, 74

Garey, M.R., 15, 33, 35, 48, 52, 55
Gebotys, C.H., 14
Geurts, W., 15
Gold, B., 4, 93
Grant, D.M., vii
Greene, J.W., 149
Gubbels, W., 12

Habers, E.H.A., 67
Haupt, R., 14, 15
Heijnemans, W.A.L., 86
Heijnen, E.W., 128
Hemme, G., 14
Hsu, Y.C., 14
Hwang, C.T., 14

Jess, J.A.G., 14, 15
Johnson, D.S., 15, 33, 35, 48, 52, 55

Karp, R.M., 74
Kernighan, B.W., 82
Kleihorst, R.P., vii
Knight, J.P., 12, 14, 15
Korst, J.H.M., 14, 33, 66, 67, 82
Kuhn, H.W., 79, 80
Kurdahi, F.J., 15
Kyung, C.M., 14, 15

Laarhoven, P.J.M. van, 66, 67, 82
Lee, J.H., 14
Leiserson, C.E., vii, 13, 31, 54, 55, 112, 114, 119, 125, 149
Lenstra, J.K., 65, 66
Lin, S., 82
Lippens, P.E.R., vii, 12–14, 82, 108, 128

Magnanti, T.L., 68, 115
Martin, R.S., 12
Maxwell, W.L., 14
McFarland, M.C., 12, 14, 15

McSweeney, B.T., 13
Miller, L.W., 14
Mlinar, M., 14, 15
Mullis, C.T., 4, 93
Münzner, A., 14

Nemhauser, G.L., 15
Niessen, C., 12
Niet, E. de, 86
Note, S., 15

Oppenheim, A.V., 4, 93
Orlin, J.B., 68, 115

Papadimitriou, C.H., 14, 15, 33, 61, 69
Park, I.C., 14, 15
Park, N., 14, 15
Parker, A.C., 12, 14, 15
Paulin, P.G., 14, 15
Peek, M.J.H., 82
Pinedo, M., 14
Pizarro, J., 14, 15
Potkonjak, M., 134

Rabaey, J., 134
Rabbat, G., 12
Rabiner, L.R., 4, 93
Rao, K.R., 86
Repko, W., 12
Roberts, R.A., 4, 93
Rohrer, R.A., 12
Rose, F.M., vii, 13, 31, 54, 55, 112, 114, 119, 125, 149
Rudell, R., 14, 123

Saxe, J.B., vii, 13, 31, 54, 55, 112, 114, 119, 125, 149
Schafer, R.W., 4, 93
Schrijver, A., 15
Shenoy, N., 14, 123
Sheridan, P., 12
Sondervan, J., vii
Steiglitz, K., 14, 15, 33, 61, 69
Stockmeyer, L., 48
Stok, L., 12, 15, 100
Supowit, K.J., 149

Timmer, A.H., 14, 15

Van Meerbergen, J.L., vii, 12–14, 82, 108, 128
Verhaegh, W.F.J., vii, 12–14, 33, 82, 108, 128
Verstraelen, M.J.W., vii

Warshall, S., 55, 119
Waterlander, E., vii
Werf, A. van der, vii, 12–14, 33, 82, 108, 128
With, P.H.N. de, 86
Wolsey, L.A., 15
Woudsma, R., 12

Yannakakis, M., 65, 82
Yip, P., 86

Summary

The work described in this thesis concerns the design of area-efficient processing units that can execute one or more functions in one or more clock cycles at a high clock frequency. The functions are part of a signal-processing algorithm specifying the behaviour of a digital electronic system. The processing units find their application in video signal processors as part of an integrated circuit in silicon.

A formal model has been set up which is used to formally define the processing unit design problem. Functions to be executed are modelled as signal flow graphs consisting of operations that communicate via signals. A processing unit is modelled as operators connected by a network of multiplexers and registers. Using this model, the processing unit design problem is formulated as a combinatorial optimization problem, where the implementation cost of the processing unit is minimized. An analysis of the complexity of the processing unit design problem shows that the problem is NP-hard. Consequently, the problem is decomposed such that each of its subproblems is mainly involved with one of the three main decision variables: allocation of operators, assignment of operations onto operators, and retiming (changing the time at which an execution takes place) of operations. The complexity of each of the subproblems is analyzed, also for some special cases.

Near-optimal solutions of the subproblems can be found using algorithms belonging to the field of local search and network flow. Although the local search algorithm simulated annealing can be readily applied without the use of much detailed knowledge, we show that good solutions can only be obtained within acceptable run times by using problem specific information. The minimum-cost maximum-flow algorithm can be applied to determine a retiming. Furthermore, a shortest path algorithm is used for timing analysis, and a maximum-weight bipartite matching algorithm is used to compute the register cost of a processing unit. Even when a subproblem of the processing unit design problem is mathematically spoken easy and can be solved by, e.g., a network-flow algorithm, the solution method is given attention for its time performance.

The algorithms described in this thesis are implemented in software. Results from experiments are included in the thesis. Part of the solution method has been successfully made into a product, a computer-aided design tool, for use by designers of integrated circuits within Philips.

Samenvatting

Het werk beschreven in dit proefschrift betreft het ontwerpen van oppervlakte-efficiente verwerkingseenheden die in één of meer klokslagen één of meer functies kunnen uitvoeren op een hoge klokfrekwentie. De functies maken deel uit van een signaalbewerkingsalgoritme dat een digitaal electronisch systeem specificeerd. De verwerkingseenheden worden toegepast in videosignaalbewerkers als onderdelen van schakelingen die geintegreerd worden in silicium.

Een formeel model is opgezet dat is gebruikt om het verwerkingseenheidontwerpprobleem te definieren. De functies die moeten worden uitgevoerd zijn gemodelleerd als signaalstroomdiagrammen bestaande uit operaties die met elkaar communiceren via signalen. Een verwerkingseenheid is gemodelleerd als operatoren die met elkaar worden verbonden door een netwerk van multiplexers en registers. Gebruik makend van het model drukken we het verwerkingseenheidontwerpprobleem uit als een combinatorisch optimaliseringsprobleem, waarbij de implementatiekosten van een verwerkingseenheid geminimaliseerd dienen te worden. Nadat we d.m.v. een complexitieitsanalyse hebben aangetoond dat dit probleem NP-lastig is, verdelen we het in deelproblemen zodanig dat een deelprobleem voornamelijk betrekking heeft op één van de drie klassen van beslissingsvariabelen: de allocatie van operatoren, de toewijzing van operaties aan operatoren, en de verandering van de tijd waarop een operatie wordt uitgevoerd. De complexiteit van de deelproblemen is geanalyseerd, ook voor speciale gevallen van deze.

Bij benadering optimale oplossingen van instanties van de deelproblemen kunnen worden gevonden door gebruik te maken van algoritmen uit het gebied van „local search" en „network flow". Ofschoon we het local search algoritme „simulated annealing" direct kunnen toepassen zonder gebruik te maken van gedetailleerde kennis, tonen we aan dat goede oplossingen alleen binnen aanvaardbare rekentijden kunnen worden verkregen door gebruik te maken van probleemspecifieke informatie. Het „minimum-cost maximum-flow" algoritme kan worden toegepast om veranderingen in de tijd waarop operaties worden uitgevoerd te vinden. Bovendien wordt een „shortest path" algoritme gebruikt voor een tijdsanalyse en een „maximum-weight bipartite matching" algoritme om de registerkosten van een verwerkingseenheid te berekenen. Zelfs als een deelprobleem van het verwerkingseenheidontwerpprobleem wiskundig eenvoudig is en bijvoorbeeld kan worden op-

gelost met een algoritme uit het gebied van „network flow", wordt aandacht besteed aan de rekentijd van het algoritme.

De beschreven algoritmen zijn geimplementeerd in software. Resultaten van experimenten staan in dit proefschrift vermeld. Een gedeelte van de methode is succesvol ingezet als een ontwerpgereedschap voor gebruik door ontwerpers van geintegreerde schakelingen binnen Philips.

Curriculum Vitae

Albert van der Werf was born on June 2, 1964 in Leeuwarden, the Netherlands. He received the M.Sc. degree in electrical engineering with honors for research in the area of telecommunications in 1987 from the University of Twente, the Netherlands. From 1987 to 1989 he followed a postgraduate course on the design of VLSI circuits at the Graduate School Twente, the Netherlands. Since 1989 he has been employed by Philips Electronics at its research laboratories in Eindhoven, the Netherlands. His present areas of interest include the development of computer-aided design methodologies for the design of integrated circuits and the design of complex heterogeneous systems on silicon.

Stellingen

behorende bij het proefschrift

Processing Unit Design

van

Albert van der Werf

I

Retiming is hetzelfde als *scheduling*.

II

De synchrone lengte W en de asynchrone lengte D van een pad in een digitaal circuit kunnen worden gecombineerd tot één lengte $\frac{D}{\tau} - W$, waarbij τ de periode van het kloksignaal is.
[Dit proefschrift, hoofdstuk 7]

III

Algoritmen voor lastige problemen die gebaseerd zijn op iteratief verbeteren kunnen eenvoudig worden aangepast tot *simulated annealing* wat de kans op betere resultaten verhoogt.

IV

Het oranje licht van verkeerslichten kan dermate verwarrend zijn met straatverlichting dat het vervangen zou moeten worden door een andere kleur.

V

Als mensen het sprekenderwijs over een probleem hebben betreft het meestal een instantie.

VI

In het verplichte deel van het curriculum van de universitaire opleiding electrotechniek moet meer discrete wiskunde.

VII

Alle optimalisatie algoritmen vallen in de klasse *local search*.

VIII

De kwaliteit van gereedschap wordt verhoogd als de maker van het gereedschap het ook zelf gebruikt.

IX

Wijzigingen in rentetarieven zouden relatief i.p.v. absoluut moeten worden uitgedrukt om een beter inzicht te geven in hun effect op de aandelenbeurs en de huizenmarkt.