The application of genetic algorithms to high-level synthesis

The Application of Genetic Algorithms to High-Level Synthesis

The Application of Genetic Algorithms to High-Level Synthesis

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 van Dekanen in het openbaar te verdedigen op woensdag 23 oktober 1996 om 16.00 uur

door

Marcus Josephus Maria Heijligers

geboren te Eindhoven

Dit proefschrift is goedgekeurd door de promotoren:

prof.Dr.-Ing. J.A.G. Jess
prof.dr.ir. W.M.G. van Bokhoven

en door de copromotor:

dr.ir. J.T.J. van Eijndhoven

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Heijligers, Marcus Josephus Maria

The application of genetic algorithms to high-level synthesis / by Marcus Josephus Maria Heijligers. - Eindhoven : Technische Universiteit Eindhoven, 1996. - X, 144 p.
Proefschrift. - ISBN 90-386-0190-5
NUGI 832, 851
Trefw.: grote geïntegreerde schakelingen; CAD / digitale systemen ; CAD / combinatorische optimalisering.
Subject headings: VLSI / high level synthesis / scheduling / genetic algorithms.

Abstract

The increasing complexity of Very Large-Scale Integrated (VLSI) circuits together with the economical pressure to issue new VLSI circuit designs very quickly, results in a progressive requirement to design circuits on higher levels of abstraction. High-level synthesis offers the circuit designer the possibility to automatically generate a digital network from a functional high-level description of a circuit, combined with the possibility to use constraints and objectives. The key problems within high-level synthesis are scheduling (determining the cycle step in which particular tasks of the functional description start their execution) and allocation (determining the amount of hardware units required to implement the functional specification). Scheduling and allocation belong to the class of problems which are hard to solve for practical high-level synthesis problems. Therefore, efficient scheduling and allocation strategies are needed, capable of producing good quality solutions with respect to the objectives, satisfying the constraints, and generated within reasonable time.

Before discussing new methodologies, the scheduling and allocation problems are formally introduced. On behalf of this, high-level synthesis related objects are introduced, and an object-oriented implementation of these objects is presented. The way synthesis related data is stored within these objects, allows a flexible way of handling synthesis constraints and objectives, and hence doesn't restrict the order and the way in which solutions are generated.

Next, it is shown how infeasible solutions can be excluded from the search space, without excluding all optimal solutions. This results in the notion of the schedule range, describing for each operation the interval of cycles in which it can be scheduled. Extra attention is paid to the throughput rate, specifying the distance between the arrival of successive input data. If the functional description contains cyclic structures, a lower bound on the throughput rate results. A new efficient algorithm will be presented, which given an arbitrary functional description, determines the minimal distance between successive arrival of input data. Furthermore, it will be shown how constraints regarding time and hardware can be integrated in a unified model, and how these constraints can be exchanged using accurate estimation techniques.

Then, various ways to construct schedules will be investigated. A new constructive scheduling method is presented, which determines a schedule by using a permutation of tasks to determine the order in which tasks are scheduled, in combination with a cycle step selection strategy, scheduling operations in their first free cycle step. Given a resource allocation, it is proven that there exists at least one optimal schedule solution (in other words a schedule with minimal completion time), obtained by scheduling tasks using a topologically sorted strategy. Statistical results applied to some examples

show that the ratio of optimal solutions with respect to the total number of solutions using such a strategy is quite large, increasing the probability that an optimal solution will be found. Finally, it will be shown that the construction of (loop) pipelined schedules is a more difficult problem, and a new strategy based on permutations is presented to construct these kind of schedules.

Additionally, this thesis describes how genetic algorithms can be used to search for good quality solutions with respect to the scheduling and allocation problem, by searching for a permutation resulting in a good quality solution. A theoretical analysis of genetic algorithms will be given, indicating how genetic algorithms should be applied to obtain efficient convergence, supported by empirical results. Different kind of encodings are presented, resulting in a new efficient strategy in which genetic algorithms use a permutation encoding of a schedule, combined with topological construction techniques. Finally, the genetic approach is extended with the possibility to allocate additional resources, to compensate lower bound resource allocations, which for a time constrained scheduling problem might have been estimated too low.

Using these methods, optimal results have been found for all cases tested, and comparisons with other heuristic search methods show that the genetic approach provides an efficient way to generate good quality solutions to the high-level synthesis scheduling and allocation problem.

Samenvatting

De toenemende complexiteit van de hedendaagse chips en de economische druk om snel met nieuwe ontwerpen te komen, zorgen ervoor dat er een toenemende behoefte bestaat om op een hoger niveau van abstractie te ontwerpen. Hoog-niveau synthese biedt de chip ontwerper de mogelijkheid om vanuit een functionele beschrijving geautomatiseerd een digitaal netwerk te genereren, met daarbij de mogelijkheid om hierbij allerlei restricties en doelstellingen mee te geven. De centrale problematiek binnen de hoog-niveau synthese bestaat uit het tijdsplanning en allocatie probleem, waarbij taken uit de functionele beschrijving toegewezen worden aan hardware welke deze taken kan uitvoeren, plus de tijdmomenten waarop deze hardware zo'n taak uitvoert. Tijdsplanning en allocatie probleem behoren tot een klasse van problemen die in de praktijk moeilijk oplosbaar zijn, en daarom moet naar methodes gezocht worden die in korte tijd goede kwaliteit oplossingen met betrekking tot de doelstellingen genereren, en welke voldoen aan de restricties die aan het ontwerp opgelegd zijn.

Alvorens over methodieken te praten, wordt het tijdsplanning probleem formeel gedefinieerd. Hiervoor worden eerst de aan de tijdsplanning en allocatie gerelateerde hoog-niveau synthese objecten geïntroduceerd, en een object georiënteerde implementatie van deze objecten gepresenteerd. Het doel hiervan is flexibel met restricties en doelstellingen om te kunnen gaan, zodat de volgorde waarin en de manier waarop oplossingen gegenereerd worden, niet beperkt wordt door de representatie en opslag van deze synthese objecten.

Ten tweede is onderzocht hoe een groot deel van niet geldige oplossingen van het zoekproces uitgesloten kunnen worden, zonder daarbij alle optimale oplossingen uit te sluiten. Dit leidt tot de introductie van het begrip tijdsplanning interval, welke voor iedere operatie een interval van tijdstippen aangeeft waarin deze geplaatst mag worden. Extra aandacht wordt besteed aan de doorstroom snelheid, welke de grootte van het interval tussen de aan de chip aangeboden data weergeeft. Indien de functionele beschrijving cyclische structuren bevat, dan impliceert dit een ondergrens voor deze doorstroom snelheid. Er wordt een nieuw efficiënt computer programma besproken welke de minimale doorstroom snelheid voor een willekeurige functionele beschrijving bepaalt. Vervolgens wordt aangetoond dat verschillende soorten restricties met betrekking tot tijd en hardware in een enkel model geïntegreerd kunnen worden, en hoe deze met behulp van nauwkeurige schattingen op eenvoudige manier naar elkaar toe te vertalen zijn.

Dan wordt gekeken op wat voor een verschillende manieren tijdsplanningen gecreëerd kunnen worden. Een nieuwe constructieve methode wordt gepresenteerd, waarbij een permutatie van taken bepaalt in welke volgorde taken geplaatst worden, hetgeen in combinatie met een selectie mechanisme bepaalt waar taken in hun tijdsplanning inter-

val geplaatst zullen worden. Bij een gegeven restrictie met betrekking tot de maximaal te gebruiken hoeveelheid hardware wordt bewezen dat indien men taken op een topologische gesorteerde manier in hun vroegst mogelijke tijdstip plaatst, er tenminste een permutatie bestaat die leidt tot een optimale oplossing met betrekking tot het laatste tijdstip van de tijdsplanning. Statistische analyse aan de hand van enkele voorbeelden toont aan dat de verhouding van optimale oplossingen ten opzichte van het totaal aantal oplossingen in zo'n geval groot is, hetgeen de kans op het vinden van een optimale oplossing vergroot. Tot slot wordt aangetoond dat het genereren van pipelined tijdsplanningen en loop pipelined tijdsplanningen voor cyclische functionele beschrijvingen met behulp van de voorgaande methode een moeilijker probleem is, en wordt een oplossing aangedragen om ook dit soort tijdsplanningen met behulp van permutaties te genereren.

Vervolgens beschrijft het proefschrift hoe genetische computer programma's toegepast kunnen worden om de tijdsplanning en allocatie problematiek op een efficiënte manier op te lossen, door te zoeken naar een permutatie die resulteert in een goede oplossing. Een theoretische analyse van genetische computer programma's geeft een indicatie over hoe een genetisch computer programma zo efficiënt mogelijk naar een oplossing van goede kwaliteit convergeert, hetgeen met empirische resultaten wordt gesteund. Verschillende soorten coderingen zijn onderzocht, resulterend in een nieuwe efficiënte tijdsplanning strategie waarbij genetische computer programma's een permutatie codering van een tijdsplanning combineren met een topologische sortering. De genetische zoekmethode is tot slot uitgebreid met de mogelijkheid om extra hardware te alloceren, om zodoende te lage hardware schattingen te compenseren met een additionele allocatie van hardware.

Vele voorbeelden van tijdsplanningen tonen aan dat de methodiek in alle geteste gevallen optimale oplossingen genereert. Een vergelijking met andere heuristieken toont aan dat de genetische zoekmethode een efficiënte manier oplevert voor het genereren van oplossingen voor het hoog-niveau synthese tijdsplanning en allocatie probleem.

Preface

This Ph.D. thesis is a result of research that has been performed at the Design Automation Section at the faculty of Electrical Engineering of the Eindhoven University of Technology in the Netherlands, under the supervision of prof.Dr.-Ing. J.A.G. Jess.

First of all, I'd like to thank prof. Jess for giving me the opportunity to perform this research in his group. He gave me many valuable comments with respect to my work and the first drafts of this thesis. I also want to thank the reading committee, Bart Mesman, Luiz dos Santos, and Sabih Gerez for their valuable comments on (parts of) the first drafts of my thesis.

Secondly, I would like to thank Harm Arts, Ric Hilderink, Wim Philipsen and Adwin Timmer for the cooperation in the field of high-level synthesis, which has resulted in a successful implementation of the NEAT system.

Thirdly, I would like to thank all M.Sc. students and trainee students who performed valuable tasks related to the research presented in this thesis. I'm especially impressed by the work performed by Bart Mesman and Luc Cluitmans, which had a great impact on the research presented in this thesis.

Furthermore, I would like to thank all other members of the Design Automation Section for their contributions to all kinds of discussions on various topics, and their support with respect to the computer system and related software.

I also want to thank Leon Stok for introducing me to the field of high-level synthesis, and Jef van Meerbergen from Philips Research to give me the opportunity to have a look at the use of high-level synthesis methodologies within an industrial environment.

Last but not least, I want to thank Christine and my family for all their support during my research.

Contents

Chapter 1 High-Level Synthesis

1.1 Introduction

The increasing complexity of Very Large Scale Integration (VLSI) Circuits causes a substantial increase of the design time of chips. Because the time-to-market is one of the key factors to make a chip profitable, a short design time is of great importance. This must coincide with the generation of efficient designs in terms of performance, design costs, and manufacturing yield. For large applications, such as RISC processors and MPEG-2 compression algorithms, the impact of small design decisions is difficult to grasp for a human designer. It is therefore important that he can design at high abstraction levels. Computer-aided design (CAD) tools can be used to provide the designer with efficient design methodologies, which show the impact of his design decisions very quickly.

A rough sketch of the process of automatically synthesizing digital integrated circuits, also called a silicon compiler, can be found in Figure 1.1. The whole process starts with a specification of an integrated circuit, which has to be captured in a behavioural description language suitable for handling by computer programs. The behavioural description consists of high-level operations (such as addition and multiplication) and high-level control structures (such as branches, loops and procedure calls). The language in which such a description is given, is called a hardware description language, of which VHDL [IEEE88], Verilog [Thom91], Hardware C [DeMi88], and Silage [Hilf85] are examples. A behavioural description can be written by a designer, but it can also be generated by design-automation tools operating at higher levels of abstraction (for example system-level abstraction or hardware-software co-design).

High-level synthesis, also called architectural synthesis, is a process which adds structural information to a functional description at the same abstraction level. This results in a so-called data-path and a controller description. The data-path consists of building blocks such as functional units, memory, and an interconnection structure among them. The controller describes how the flow of data inside the data-path is managed, and is described in terms of states and state transitions. The controller description is translated into an implementation at the abstraction level of gates by using logic synthesis.

Building blocks inside a data-path are created by using so-called module generators. There are several possibilities to generate modules. The desired functionality can be described by boolean functions, and logic synthesis can be used to optimize and map the equations on a gate library, called behavioural generation. Structural generation

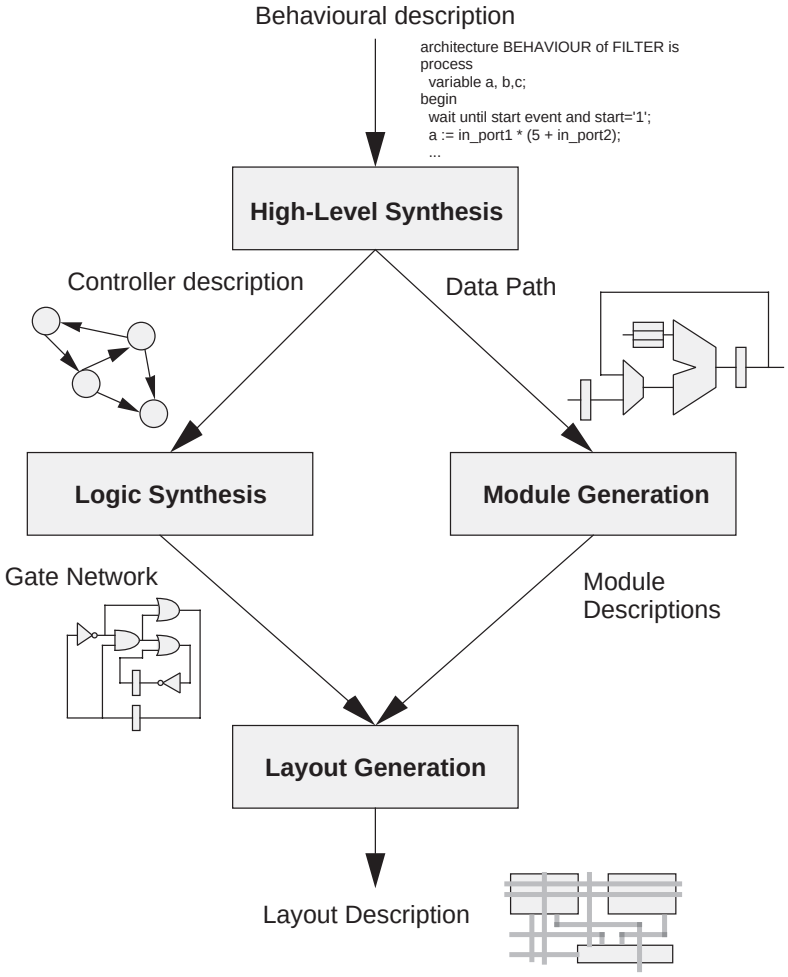


Figure 1.1 Silicon Compiler Overview.

uses knowledge of a possibly efficient structural implementation, and therefore generates such a structure directly. Finally, if a layout of a module is very regular (such as RAMs, ROMs, and register files), the layout description can be generated directly.

The final synthesis step, called layout synthesis, creates a geometrical description of the layout using placement and routing techniques. The result is a layout mask, which is a description of the IC at the physical abstraction level.

1.2 High-level synthesis problem definition

High-level synthesis translates a behavioural description of a chip into a data-path. A behavioural description specifies the functions the chip has to perform and the way the chip interacts with its environment. The structural description describes an implementation of the functions, and consists of a data-path and a controller. A data-path consists of functional units (such as for instance adders, multipliers, ALUs, and logic units), memory to store data (such as RAMs, ROMs, registers, and register files), and interconnect to transport data between functional units and memory (such as buses, wires, and multiplexers). The collective noun for functional units, memory, and interconnect is resources.

The controller describes how the flow of data in the data-path is managed, and is described in terms of states and state transitions. Each state of the controller specifies the assignment of functional units to operations, of data to registers, and the way how multiplexers should direct their data. Furthermore, given some state, a state transition function defines the subsequent state, which may or may not depend on data produced inside the data-path.

Given a behavioural description, together with a set of constraints and goals, the high-level synthesis problem is to find the best architectural solution. Some possible goals are minimal area, minimal power consumption, or maximum throughput. Constraints are often defined in terms of execution order, completion time, throughput rate, and area. The constraints and goals depend on the kind of application which needs to be synthesized. In the case of micro-processors, speed is the main goal at the expense of area and power consumption. Furthermore specialized techniques, such as the application of cache memory and branch prediction, can be used to improve the performance In case of DSP algorithms, throughput will be the main constraint, and the goal is to find an implementation with a small area or power consumption. Optimization techniques can be used to search for high quality designs in all these cases.

Almost all optimization problems associated with chip design are difficult to solve. Most of these problems are member of the class of so-called NP-hard problems [Gare79], and no polynomial-time algorithms are known that solve each instance of these problems to optimality. For a synthesis system to be efficient, trade-offs must be made to obtain acceptable solutions in an acceptable amount of time. Heuristics can be used, which usually are fast, but will return solutions which are not guaranteed to be optimal. On the other hand enumeration algorithms can be used which always give optimal solutions, but generally need exponential run-time, and hence can handle only problem instances with small input data size. The development of algorithms which obtain acceptable solutions in an acceptable amount of time, is an important topic of research in high-level synthesis, and will be the main topic of this thesis.

1.3 High-level synthesis problem partitioning

When generating a data-path from a behavioural description, four kinds of problems associated with resources must be solved:

- Selection — What kind of resources are used in the data-path?
- Allocation — How many resources are needed in the data-path?
- Scheduling — When will operations from the functional description be executed?
- Binding — To which resources will operations, values, and value transfers be assigned?

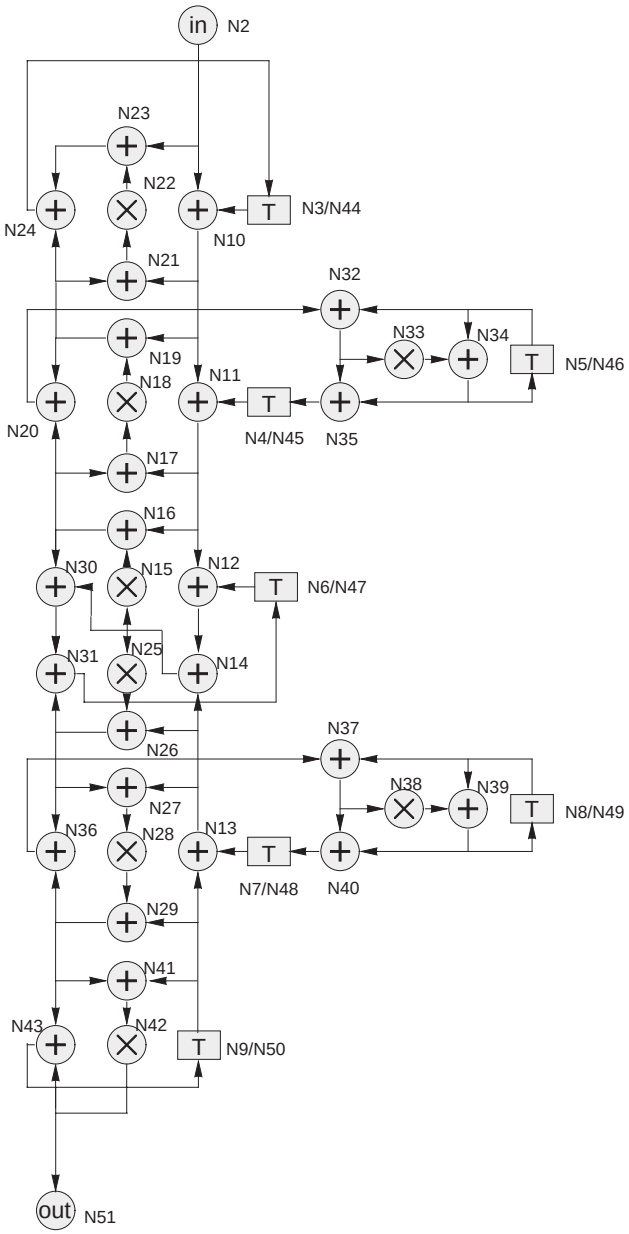


Figure 1.2 Data-flow graph of a 5th order wave digital filter.

These four problems are interrelated, but are difficult to solve simultaneously. Therefore, high-level synthesis strategies solve each problem or a small combination of these problems separately.

1.4 High-level synthesis design flow impression

In this section a simplified overview will be given concerning the translation of a description of a digital filter into a synchronous (clocked) data-path.

In Figure 1.2 a behavioural description of a 5th order wave digital filter [DeWi85] can be found, specified by a so called data-flow graph. In a data-flow graph nodes represent

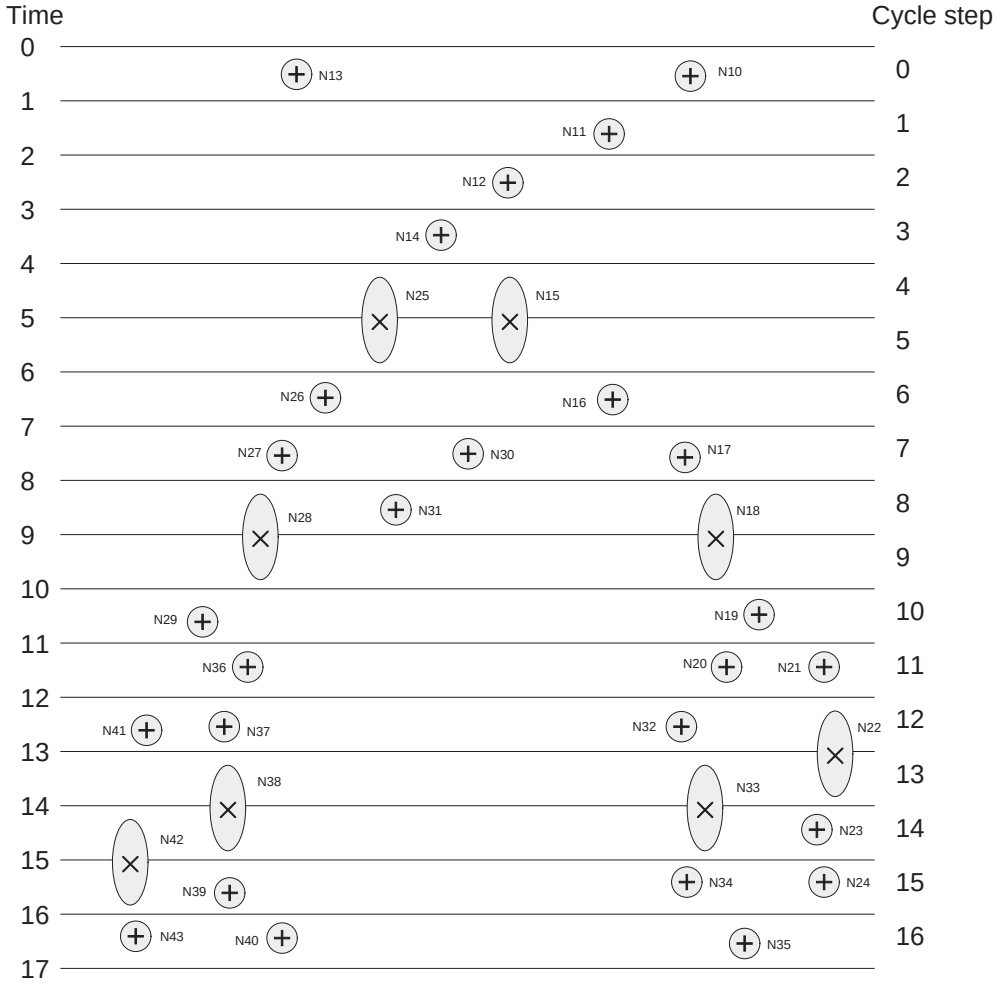


Figure 1.3 Schedule of 5th order wave digital filter.

operations, and edges represent the transfer of data values. If values are available on all incoming edges of a node, the node will execute by consuming these values, and subsequently generates an output value on all outgoing edges. A data-flow graph explicitly shows the order of execution of operations, and hence also shows which operations may be executed simultaneously. This makes data-flow graphs very suitable as a starting point for high-level synthesis scheduling and allocation.

In Figure 1.3, a schedule of the 5th order wave digital filter can be found, in which operations have been assigned to cycle steps. In this schedule an addition is assumed to require 1 cycle step for execution on an adder module, and a multiplication is assumed to require 2 cycle steps for execution on a multiplier module. In the schedule of Figure 1.3, at most 3 multiplications and 3 additions are scheduled simultaneously. This induces a functional unit allocation of at least 3 multipliers and 3 adders. The amount of cycle steps needed is 17.

After selection, allocation, and binding (based on the schedule of Figure 1.3) of functional units, memory, and interconnect, a data-path as given in Figure 1.4 can be gener-

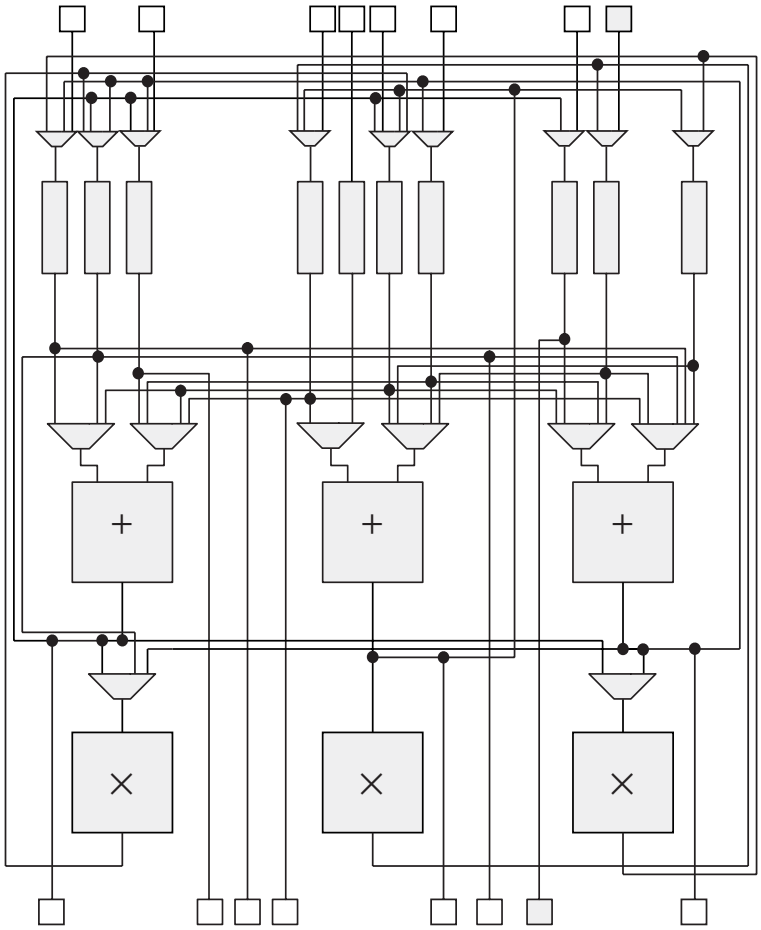


Figure 1.4 Data-path of 5th order wave digital filter.

ated. It consists of 3 adders and 3 multipliers, connected to a bunch of registers and multiplexers to store and select data respectively.

1.5 High-level synthesis scheduling

Because the result of a schedule induces a completion time and a resource allocation, scheduling is considered to be the central task of high-level synthesis. During scheduling, functional units, storage, and interconnect must be allocated over time (cycle steps) to perform a set of operations (specified by a behavioural description). In most practical cases high-level synthesis scheduling is subject to constraints, such as precedence constraints (derived from a behavioural description), resource constraints (derived from a network structure), and time constraints (completion time, data arrival rate). A schedule is called a feasible schedule if it satisfies its constraints. Performance measures or optimality criteria are used to distinguish good schedules from bad schedules. High-level synthesis performance measures, such as (a combination of) overall execution time and resource requirements [McFa90], can be used to serve as optimality criteria. The goal of a scheduling algorithm would be to find an optimal schedule, in other words, it should return the best feasible schedule with respect to the performance measures.

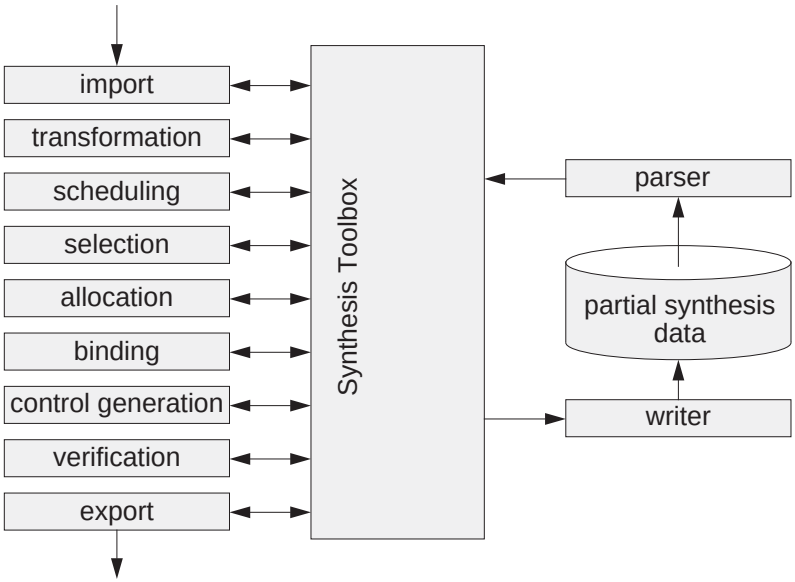


Figure 1.5 Synthesis Toolbox

1.6 Area of this thesis

This thesis addresses the high-level synthesis scheduling and allocation problem, originating from the translation of DSP algorithms into synchronous clocked circuits.

Chapter 2 introduces the components needed to describe the high-level synthesis of digital circuits, with an emphasis on the components needed to define the scheduling problems presented in Chapter 3. Secondly, an object oriented implementation of these components, called NEAT (New Eindhoven Architectural synthesis Toolbox), is presented, which serves as a software platform for all algorithms presented in this thesis. It provides the tool designer an environment in which he can develop a collection of interacting tools, for which the order and the way in which these tools are applied is not predetermined (see Figure 1.5).

In Chapter 3 two scheduling problems, the key problems of this thesis, are formally introduced.

Chapter 4 discusses the influence of constraints on the scheduling problem. It will be shown how precedence constraints, time constraints, and resource constraints have an influence on the schedule range of operations, and how these constraints can be integrated into a single scheduling model. Some efficient algorithms are presented which determine and update the scheduling range of operations during scheduling. Furthermore, a new algorithm will be presented to determine the lower bound on the minimal throughput rate of a data-flow graph containing loop structures. Finally, it is shown how constraints and goals can be exchanged using lower-bound estimations, resulting in a re-definition of the original scheduling problem.

Chapter 5 presents several ways to construct schedules under different sets of constraints. The central theme of this chapter is about permutations of operations, which provide a mechanism to classify the way schedulers construct their solution. Furthermore, in Chapter 5 a topological method of scheduling is defined. It is shown that a topological construction of schedules increases the probability that feasible and higher quality solutions are created, without excluding all optimal solution from the search space. Finally it is shown how the construction of (loop) pipelined schedules can be performed.

Chapter 6 focuses on the application of genetic algorithms, with the aim to find good quality results to the scheduling problem. The main idea is to improve the results produced by existing scheduling heuristics, and to shorten the execution time of exhaustive search methods, and hence to fill the gap between exhaustive search methods and 'plain' heuristics. First of all, a theoretical framework is presented to obtain some insight how to efficiently apply genetic algorithms to combinatorial optimization. Then genetic algorithms are applied to search for permutations, using the scheduling strategies presented in Chapter 5. Results show that genetic search is only successful when there is a good relation between the genetic encoding of the scheduling problem and the way solutions are constructed with respect to the constraints and goals imposed. If a topologically schedule strategy is applied, comparison of the results with many other heuristic approaches show that the genetic approach finds better results in acceptable computation times. Furthermore, the genetic scheduling strategy is extended with the possibility to allocate extra resources to be able to deal with a synthesis strategy based on lower-bound resource estimations. Finally, the scheduler is extended with memory allocation costs to show that, to a certain extent, the genetic search is capable to minimize more general cost functions.

Chapter 7, finally, presents conclusions together with a discussion about future work.

Chapter 2 High-Level Synthesis Components

2.1 Introduction

This chapter introduces the components needed to describe the high-level synthesis of digital circuits, with an emphasis on the scheduling problems presented in Chapter 3. To solve the whole high-level synthesis problem, a collection of interacting high-level synthesis tools is needed. Each tool retrieves, manipulates, and stores intermediate results, which should be made accessible by using a so called synthesis data interface. An object oriented implementation of such a data interface, called NEAT (New Eindhoven Architectural synthesis Toolbox), is presented, which serves as a software platform for all the algorithms presented in this thesis.

2.2 Domains

High-level synthesis generates structural information (in terms of modules and their interconnections, also called a data-path) and control information (describing how to control the data-path), derived from a behavioural description (described in terms of operations and special constructs). During the high-level synthesis process, mainly three domains of data representations can be distinguished, which are behaviour, control, and structure. Each domain provides a different view of the design, and are therefore also called design views. In the following subsections, additional information about the different domains of data and their representations will be given.

2.2.1 Behavioural domain

The input description, which specifies the behaviour of a design, can be defined by using ASCIS data-flow graphs [Eijn92]. The ASCIS data-flow graph is intended as an intermediate form between user oriented interfaces (languages, schematics) and synthesis or verification tools. The advantage of applying synthesis or verification directly to a data-flow graph is that it resolves the different nature of input languages. Data-flow graphs can be automatically obtained from hardware description languages such as VHDL [IEEE88], Verilog [Thom91], Hardware C [DeMi88], or Silage [Hilf85] by the use of data-flow analysis techniques.

Definition 2.1 (Data flow graph). A data-flow graph is a tuple (V, E), in which V is a set of nodes (representing operations), and E is a set of directed edges $V \times V$ (representing flow of data).

The execution of a data-flow graph follows the concept of a token-flow mechanism. In this mechanism a data value instance is defined to be a token. This data value instance can be a single scalar, but can also consist of more complex data-types, such as arrays, records, or user-defined data-types. The execution of an operation is defined as removing tokens from the input edges of that operation, and producing new tokens containing the result of the calculation of the operation on all output edges. The semantic behaviour of a node, which determines the translation of input values to output values, is defined by a so called operation type.

Definition 2.2 (Operation Type τ). Let V be a set of data-flow nodes, $v \in V$, *OpType* be a given set of operation types. Operation type τ: $V \rightarrow$ *OpType* is a function, with $\tau(v)$ the operation type of operation v.

A comprehensive classification of different operation types can be found in [Eijn91], such as arithmetic operations (+, -, $\times$, /), boolean operations (and, or, not), and relational operations (<, >, $\neq$). The interface of a data-flow graph to the outside world is defined by means of input and output nodes. To support special language constructs, such as loops and conditionals, nodes with a special execution mechanism have been defined, which originate from demand graphs as described in [Veen85]. An example of a data-flow graph containing a loop structure is given in Figure 2.1, accompanied with a textual description characterizing its semantic behaviour. The loop structure uses nodes with operation type *entry* and *exit* to describe the controlling mechanism of the loop. An entry node has two (or more) data inputs, one control input, and one data output. An entry node accepts a token at one of its data inputs (the choice of which depends on the value of the control input), and copies this token to its output. An exit node has one data input, one control input, and two (or more) data outputs. It accepts a token at its data input, and depending on the value of its control input copies this token to one of its outputs. Entry and exit nodes provide a mechanism for a token to enter, to rotate, and finally to leave the loop structure.

The data-flow graph in Figure 2.1 shows that control-flow is described by means of special data-flow nodes. The integration of data-flow and control-flow in one model is what makes data-flow graphs differ from most other input formats used in high-level synthesis systems (see [Walk92] for an overview). By separation of data-flow and control-flow in other formats, code inside these special constructs is moved to basic blocks or straight line code, which may impose undesirable restrictions with respect to scheduling and allocation algorithms.

The transfer of tokens inside a data-flow graph is represented by directed edges. There is no notion of variables or assignment in a data-flow graph, as they might impose timing and mapping restrictions, and therefore reduce the search space before synthesis starts. The execution order of nodes inside a data-flow graph is constrained by the structure of the data-flow graph, inducing a partial order, denoted by $\prec$. This partial order explicitly models the concurrency of the behaviour; if two nodes are not related accord-

```
for i := 0 to 9 do
  x := x + 1;
endfor;
```

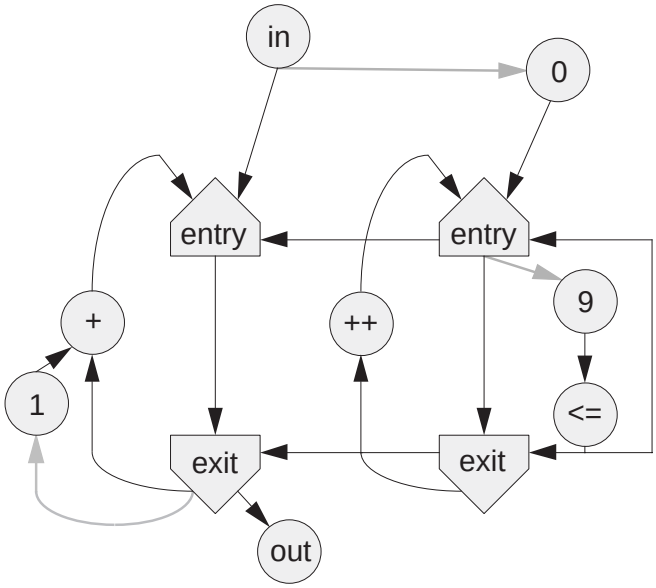


Figure 2.1 Data-flow graph containing a loop structure.

ing to partial order $\prec$ (in other words there is no flow of data between these two nodes), then they can be executed in parallel. Depending on the accuracy of the data-flow analysis, a data-flow graph represents a maximal parallel representation of the behaviour.

To enforce a specific order to the execution of a set of nodes, so called sequence edges can be used, denoted by grey-coloured edges. See for instance Figure 2.1, where sequence edges are used to activate operations of type constant.

Behaviour preserving transformations, such as tree height reduction [Hout94] and retiming [Fran94], can be applied to change the structure of a data-flow graph. The goal of these transformations is to restrict or widen the search space structure of the synthesis process.

In principle data-flow graphs impose no limitations onto a particular architectural solution. Therefore, they are highly suitable as a starting point for high-level synthesis.

2.2.2 Control domain

The control domain is used to describe the result of scheduling, inducing the way the data-path is controlled. The control domain can be described by a (set of) finite state machine(s).

Formally, a finite state machine can be described by a quintuple (I , O , S , δ , λ), in which I is the input alphabet (or set of inputs from the data-path), O is the output alphabet (or set of outputs to the data-path), S is the set of states, δ: $I \times S \rightarrow S$ is the state transition function, and λ: $S \rightarrow O$ is the output function [Hill81]. The definition of this so called Moore automaton can be extended to a Mealy automaton by extending the output function λ: $I \times S \rightarrow O$. A finite state machine can be described by a so called control graph.

Definition 2.3 (Control graph). A control graph is a quadruple (S , E , I , O), in which S is a set of nodes (representing states), and E is a set of directed edges $S \times S$ (representing state transitions). Each state transition $e \in E$ is labelled with a set of input symbols from 2^I, in which 2^I denotes the power-set of I. In case of a Mealy machine each edge $e \in E$ is labelled with $o \in O$, whereas in case of a Moore machine each state $s \in S$ is labelled with $o \in O$.

Similar to the definition of operation types for data-flow nodes, the semantic behaviour of control nodes is defined by a so called control type.

Definition 2.4 (Controller Type τ). Let S be a set of control nodes, $s \in S$, and *ConType* be a set of control types. Control type τ: $S \rightarrow$ *ConType* is a function, with $\tau(s)$ the control type of control node s.

In [Hild93] a model is suggested which is based on control graphs, extended with special type of nodes (such as join and split), to be able to describe some flows of control, such as for instance parallel executing loops, in a more compact way.

During scheduling, operations of a data-flow graph are assigned to states of a control graph. The schedules discussed in this thesis assign operations to cycle steps, which can be represented by a control graph consisting of a chain of states.

2.2.3 Structural domain

The structural domain is described by a set of network graphs.

Definition 2.5 (Network Graph). A network graph is a tuple (M , E), where M is the set of nodes (representing modules such as functional units, memory, multiplexers), and E is the set of undirected edges $M \times M$ (representing interconnections such as buses and wires).

Similar to the definition of operation types for data-flow nodes, the semantic behaviour of network nodes is defined by a so called module type (such as multiplier or register).

Definition 2.6 (Module Type τ). Let M be a set of network nodes, $m \in M$, and *ModType* be a set of module types. Module type τ: $M \rightarrow$ *ModType* is a function, with $\tau(m)$ the module type of module m.

Inside a network graph (M , E), a controller c is modelled as a node $c \in M$. Let $M_C \subseteq$ M be the set of controllers in (M , E), in other words $M_C = \{m \in M \mid \tau(m) =$ ‘control’$\}$. A data-path consists of the sub-graph (M_D , E_D), where $M_D = M \setminus M_C$ and $E_D = M_D \times M_D$. Let E_C be the set of edges $E \setminus E_D$, in other words the edges between M_C and data-path modules M_D. These edges transport input symbols I and output symbols O, used to exchange control vectors between modules of the data-path M_D and M_C.

2.3 Domain relations

During high-level synthesis, relations between different synthesis objects are generated. These relations have to be passed from one tool to another, so they should be incorporated in the synthesis interface and data exchange format. Two important kinds of relations can be distinguished, called intra-domain relations and inter-domain relations, which will be presented in the following two subsections.

2.3.1 Intra-domain relations

The semantic behaviour of nodes is described by types, as has been shown in Definition 2.2, Definition 2.4, and Definition 2.6. The semantic behaviour of a type can be described by a graph in the same domain. This means that the set of operation types is represented by a set of data-flow graphs, the set of control types is represented by a set of control-flow graphs, and the set of module types is represented by a set of network graphs. Hence, the semantics of an operation, state, or module can be described by referring to a data-flow, control-flow, or network graph respectively. When a synthesis object is created, it always inherits the semantics of the graph it refers to.

Depending on the abstraction level, graphs describing a particular type can be specified in different ways:

- If the abstraction level of the graph is at the level of high-level synthesis primitives, a graph can be described by a collection of nodes and edges. An example is a data-flow graph of a filter section which is used as a data-flow node (or operation) in another data-flow graph describing a DSP algorithm. The mechanism to break down a design into smaller parts allows hierarchical designs and partitioning, and hence offers support for bottom-up and top-down synthesis methods.

- If the abstraction level of the type is such that it cannot be described in terms of operations, states, or modules of the same abstraction level, other kind of descriptions are used. In case of primitive operations such as additions and multiplication, documentation such as [Eijn91] can be used to describe the semantic behaviour and its interface (inputs and outputs) in more detail. In case of primitive module types, such as adders, multipliers, memory, logic gates, and others, computer programs called module generators [Thee93, Arts91] can be used to specify their contents on other abstraction levels. An advantage of using module generators is that they can be parameterized (speed, size, power), which avoids the need to store each possible implementation of each module type separately. In all these cases, a type can be described by a graph consisting of input and output nodes, defining its interface, together with a reference to documentation or to computer programs. Because this mechanism uses the same interface as for hierarchical designs, no special conversion tools or functionality is needed to retrieve library information.

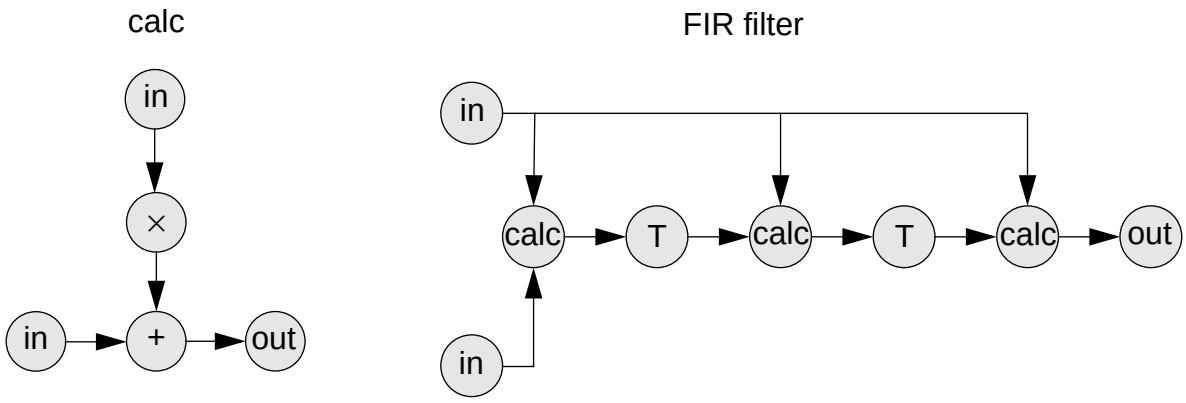


Figure 2.2 Intra-domain relation example.

In Figure 2.2, an example of an intra-domain relation can be found. A data-flow graph called *calc* is used as a node in a data-flow graph representing the behaviour of a Finite Impulse Response (FIR) filter. The addition, multiplication, and delay nodes on its turn are described by an addition graph, multiplication graph, and delay graph respectively (not drawn). Input and output are so-called primitive types, and can only be described in terms of each other, and therefore must be treated as special cases.

2.3.2 Inter-domain relations

Links are used to describe relations among objects of different domains. Links can be specified partially to represent intermediate synthesis results. Mainly two kinds of links can be distinguished, graph links and node links.

Definition 2.7 (Graph Link). Let n be the number of domains. Let G_0 , G_1 , ... , $G_{n\text{-}1}$ be a set of graphs from n different domains. A graph link is an n-tuple from $G_0 \times G_1 \times ... \times G_{n\text{-}1}$.

If the synthesis interface is restricted to the three domains mentioned in the previous section, a graph link can be described by a triple $DFG \times CTG \times NWG$, in which *DFG* represents the set of data-flow graphs, *CTG* represents the set of control graphs, and *NWG* represents the set of network graphs. A graph link relates graphs between these three different domains. Links between graphs can represent synthesis information, such as ‘this network graph is an implementation of this data-flow graph’. Hence, graph links can also be used to represent synthesis library information, such as an operation type which can be implemented on particular module types.

Definition 2.8 (Operation Type Mapping μ). Let *OpType* be a set of operation types, *ModType* be a set of module types, and $t \in OpType$. Let L be a set of graphlinks, and $X \in ConType$. Operation Type Mapping $\mu: OpType \rightarrow 2^{ModType}$ is a function, with $\mu(t)$ the set of module types that can execute operation type t, given by $\mu(t) = \{nwg \in ModType \mid (t , X , nwg) \in L\}$.

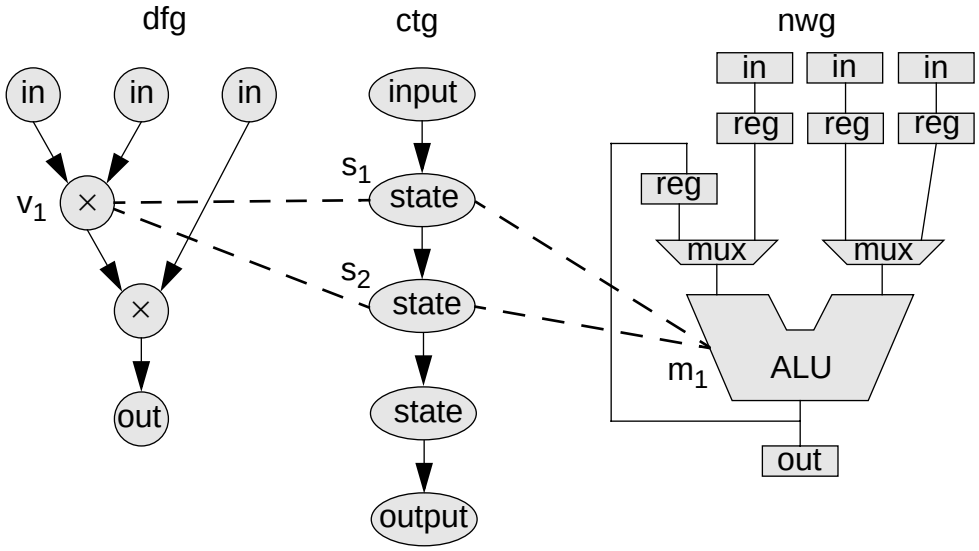


Figure 2.3 Graphical example of inter-domain relation.

To be able to describe the fine-grain information between different domains, node links are defined.

Definition 2.9 (Node link). Let n be the number of domains. Let V_0 , V_1 , ... , V_{n-1} be sets of nodes from graphs in n different domains. A node link is an n-tuple from $2^{V_0} \times 2^{V_1} \times ... \times 2^{V_{n-1}}$.

A node link relates nodes between different domains. Links between nodes can represent synthesis information such as ‘this data-flow node is related to these states (schedule information)’. Because node links denote the fine-grain relations among graphs, they can only occur within the context of graph links, relating nodes which are member of graphs, which on its turn are member of the graph links.

In Figure 2.3 an example of inter-domain relations can be found. A nodelink which relates data-flow node v_1 from *dfg*, states s_1, s_2 from *ctg*, and module m_1 from *nwg* is depicted by dashed lines.

Inside links the kind and status of the relation that it represents can be defined, which makes it easy for tools to decide whether particular links should be used, and to decide how particular links should be used. Links can for instance be used to describe constraints such as ‘this operation should be assigned to this module’, without the need for conversion tools or special access functions. Links can be tagged, which can be useful when information needs to be exchanged between different synthesis tools.

By using links, complex and detailed synthesis information is separated from the graph descriptions themselves. Nevertheless, synthesis information is still gently incorporated into the data interface. Different designs can be constructed by creating graph links and

node links using the same graph descriptions. Links can also be used to describe parameterized libraries in a compact way.

2.4 NEAT

The New Eindhoven Architectural synthesis Toolbox [Heij94], also called NEAT, is an implementation of the components discussed in the previous sections. It supports (but is not restricted to) three design views as discussed in Section 2.2, together with the domain relations as discussed in Section 2.3.

A standard interface to synthesis data, used by each synthesis tool, makes the maintainability of these tools much easier. Therefore, a standard interface has been defined [Arts92] and implemented. The standard synthesis interface can be used to manipulate synthesis objects, such as adding or deleting synthesis objects, create relations among synthesis objects, add design specific information to synthesis objects, and so on. Also, functionality such as storing and retrieving intermediate synthesis data to disk, command line parsing, consistency checking, obtaining synthesis status, providing abstract data types (lists, sets, arrays, strings), and more has been included in the standard interface to save the tool developer unnecessary work.

The relations between objects of NEAT are drawn in Figure 2.4, in which a directed edge (*x* , *y*) denotes a set relation (i.e. *x* contains or consists of a set *y*). The main object is a database. A database contains a set of graphs and a set of graphlinks. A graph contains a set of nodes and a set of edges. A node contains a set of ports, and finally, a graphlink, contains a set of nodelinks.

Each synthesis tool produces specific kind of results, and hence needs a specific data interface to store these results inside the existing synthesis objects. This data should be hidden from other tools to prevent visibility of irrelevant data, visibility of irrelevant manipulation functions, unnecessary re-compilation of the synthesis interface, and unnecessary re-compilation of tools which rely on this interface. This can be accomplished by extending the synthesis interface using object oriented programming techniques. Inheritance can be used to extend existing synthesis objects with specific information without any restrictions, and without interfering with the common NEAT synthesis interface.

For a class of synthesis objects (graphs, nodes, edges, and ports), default inheritance relations exist, which are modelled in Figure 2.5. By changing the string *object* in this figure by one of the common synthesis objects (graphs, nodes, edges, and ports), the inheritance structure for such an object can be obtained. The choice for this specific inheritance structure has been inspired by the fact that synthesis objects from different domains share common data (for example a graph in general consists of nodes and

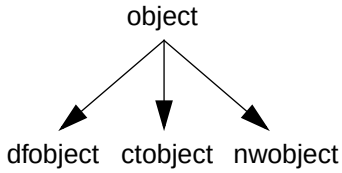


Figure 2.5 Inheritance structure of NEAT.

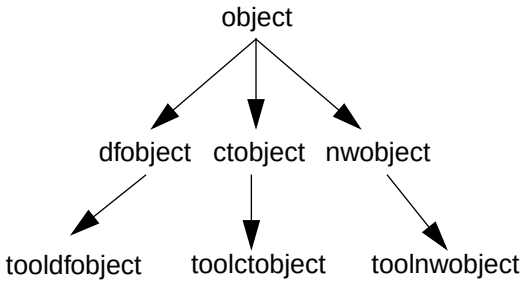


Figure 2.6 Tool-specific inheritance.

edges), but also differ in many aspects (a data-flow node may contain specific schedule information, which is not applicable for control and network nodes).

Tool specific data and functionality can be added by using inheritance, as shown in Figure 2.6. The standard synthesis interface remains unaltered, hence other tools are not disturbed with tool specific data, and the standard synthesis interface does not have to be re-compiled. The new interface that has been obtained, has the same look and feel as the standard interface.

Synthesis tool frameworks using the NEAT interface are generated automatically by means of templates. A tool programmer adds his functionality to these tools using an object oriented programming style. This prevents the programmer from building tools and programming environments from scratch.

NEAT is implemented using the C++ programming language [Elli90], which has been chosen because of its object-oriented facilities and its overall use in CAD development.

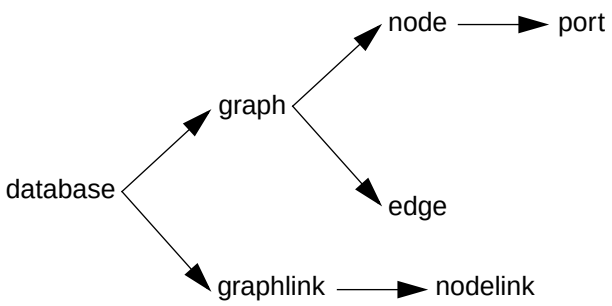


Figure 2.4 Graphical overview of standard synthesis object relations.

To be able to store intermediate synthesis results which have to be exchanged between tools or research platforms, an exchange format has been developed based on plain ASCII files. The syntax of these files consists of a balanced nested parenthesis structure (such as LISP), which only requires simple LL-(1) parsing techniques (see Example 2.1).

The intermediate data format can be extended by defining new keywords, which goes hand-in-hand with the object-oriented extensions of the standard synthesis interface. Tools which are not interested in the information attached to a particular keyword, can skip this information by just counting parentheses, which is taken care of by standard parsing functions. This implies that new extensions to the format will never disturb existing tools which don't understand the underlying semantics of these new extensions. Hence the format is both upward and downward compatible.

The ESCAPE environment [Fleu93] can be been used to display synthesis information (see Figure 2.7). Separate windows display different domains, and links can be visualized by clicking on nodes. The graphical interface gives designers the capability to analyse and manipulate (intermediate) synthesis results [Hild94].

Example 2.1 Partial example of textual format

```
(dfg-view
  (graph example
    (node N0
      (type input)
      (varname in2)
      (out-edges E1))
    (node N2
      (type output)
      (in-edges E3))
    ....
    (edge E1
      (type data)
      (width 8)
      (varname in2)
      (destination N14 (port N-1))
      (origin N0 (port out)))
      ....
))
```

2.5 Related work

In [Black88], an alternative notion of links is presented. Synthesis results are recorded as tags inside graph descriptions, and special programs (Coral) extract this information and translate them into links. Hence the links only depict relations among objects of different domains, and do not contain any synthesis information. Synthesis information is stored in tool-specific data structures, and no support is given for development or integration of new synthesis tools. In [Lann91] and [Rund93], object-oriented techniques are used in a similar way as NEAT, to extend a common synthesis interface into a tool specific interface. However, these systems store synthesis information as tags inside graph definitions, instead of using links. This may restrict the complexity of rela-

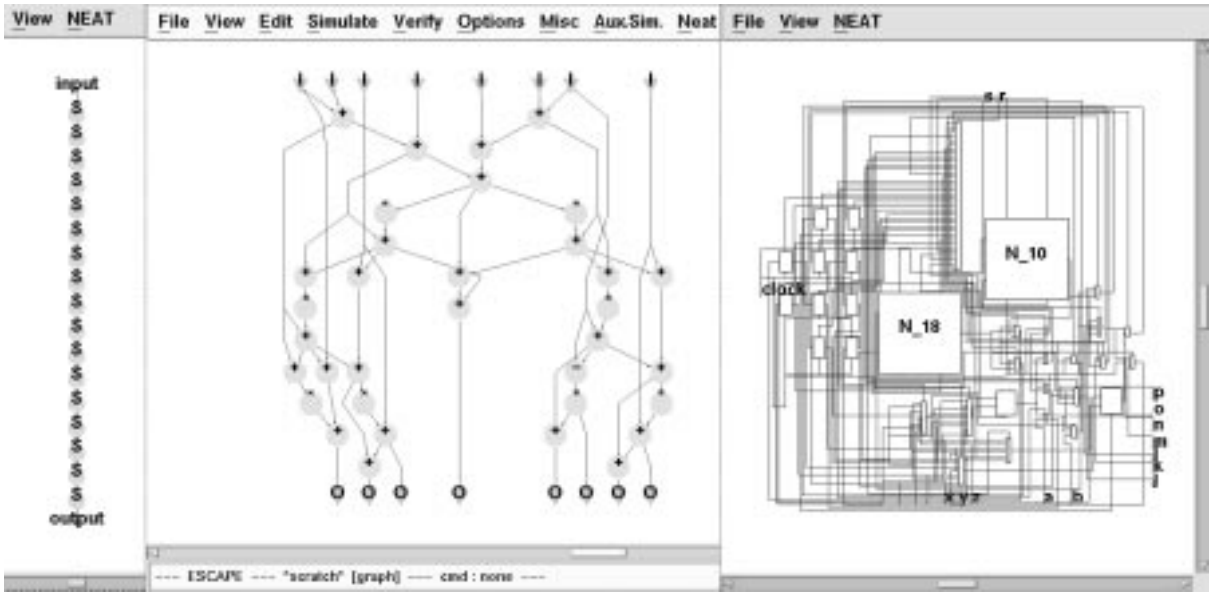


Figure 2.7 ESCAPE shows control, behaviour, and structure design views

tions that can be described. Complex inheritance mechanisms are used to describe libraries, and special techniques are needed to retrieve library information. NEAT uses links to describe comprehensive libraries very efficiently. No special conversion tools or access functions are needed to support these libraries. The NEAT interface is very closely related to the mathematical structure of synthesis objects. Inheritance is only used to make the system extendible, without losing the original structure of the interface.

To our knowledge no high-level synthesis system supports the use of extendible ASCII data to store synthesis data, or the incorporation of links inside the common synthesis interface to represent synthesis results.

2.6 Conclusions

In this chapter a synthesis toolbox system called NEAT has been presented. This system provides a flexible way of developing synthesis tools with minimal programming effort, by providing developers with a common functional interface containing common synthesis functionality, standard object manipulation functions, search functions, and common data structures. Some synthesis strategies have been successfully implemented using NEAT. The overall experience with NEAT is that it highly improves the design and maintainability of high-level synthesis tools. It has contributed significantly to the ease of incorporating new research ideas in existing synthesis trajectories.

Chapter 3 High-Level Synthesis Scheduling

3.1 Introduction

In this chapter the key problems of this thesis, a collection of high-level synthesis scheduling problems, are formally defined. Before introducing these scheduling problems, scheduling constraints and goals which are of specific interest for the high-level synthesis scheduling problem will be presented.

3.2 Scheduling and allocation definitions

This thesis is about the generation of synchronous digital circuits, in other words circuitry which is synchronized by a central clock. This introduces the notion of the so-called cycle step, which is equal to the duration of one period of the clock. The execution of an operation type on a particular module type can be measured in the amount of cycle steps, denoted as the execution delay.

Definition 3.1 (Execution delay d). Let *ModType* be the set of module types, *OpType* be the set of operation types, $l \in$ *ModType*, $t \in$ *OpType*, μ be an operation type mapping, and $l \in \mu(t)$. Execution delay d: *OpType* $\times$ *ModType* $\rightarrow \mathbb{R}$ is a function, with $d(t, l)$ the number of cycle steps an operation type t needs when it is executed on a module having module type l.

When an operation type $t \in$ *OpType* can be executed upon several module types $l \in \mu(t)$, various execution delays $d(t, l)$ may be associated with an operation $v \in V$, for which $\tau(v) = t$. In this thesis, for reasons of simplicity, the operation type mapping τ is restricted in such a way that each operation v will be associated with exactly one module type l, and hence each operation can be associated with a unique execution delay.

Definition 3.2 (Operation mapping ξ). Let $G = (V, E)$ be a data-flow graph, $v \in V$, and *ModType* be a set of module types. Operation mapping ξ: $V \rightarrow$ *ModType* is a function with $\xi(v)$ the module type upon which operation v will be implemented. Notice that $\xi(v) \in \mu(\tau(v))$.

When the operation mapping is known beforehand, the execution delay of an operation is given by $d(\tau(v), \xi(v))$, for which the following short hand notation is used:

Definition 3.3 (Operation execution delay δ). Let $G = (V, E)$ be a data-flow graph, $v \in V$, $\tau(v)$ be the operation type of v, and $\xi(v)$ be the operation mapping of v. Operation

execution delay $\delta: V \rightarrow \mathbb{R}$ is a function, with $\delta(v)$ the operation execution delay of operation v, given by $\delta(v) = d(\tau(v), \xi(v))$.

Modules inside a data-path will occupy a certain amount of area. To be able to optimize the module area of the data-path, the cost of a module type is defined as follows:

Definition 3.4 (Module type cost *cost*). Let *ModType* be a set of network graphs, and $l \in ModType$. The module type cost *cost*: $ModType \rightarrow \mathbb{R}$ is a function, with *cost(l)* the area of module type l.

A schedule of an operation can be defined as follows.

Definition 3.5 (Schedule φ). Let $G = (V, E)$ be a data-flow graph, and $v \in V$. The schedule $\varphi: V \rightarrow \mathbb{N}$ is a function, with $\varphi(v)$ the cycle step where operation v starts its execution. The schedule of each operation v induces an interval $[\varphi(v), \varphi(v) + \delta(v)]$, often written as $[begin(v), end(v)]$, which represents the range of cycles in which v is executing.

A schedule φ induces a schedule length (also called makespan) and a resource allocation.

Definition 3.6 (Schedule length *M*). Let (V, E) be a data-flow graph, S be a set of schedules, and $\varphi \in S$. The completion time C_{max}: $S \rightarrow \mathbb{N}$ is a function, with $C_{max}(\varphi) = \text{MAX}_{v \in V \mid \tau(v) = \text{`output'}}\, end(v)$. The start time $C_{min}: S \rightarrow \mathbb{N}$ is a function, with $C_{min}(\varphi) = \text{MIN}_{v \in V \mid \tau(v) = \text{`input'}}\, begin(v)$. The interval $[C_{min}(\varphi), C_{max}(\varphi)]$ is called the schedule range of schedule φ. The schedule length M: $S \rightarrow \mathbb{N}$ is a function with $M(\varphi)$ given by $M(\varphi) = C_{max}(\varphi) - C_{min}(\varphi)$. In the remainder of this thesis it is assumed, without loss of generality, that $C_{min}(\varphi) = 0$, in other words $C_{max}(\varphi) = M(\varphi)$.

Definition 3.7 (Resource Allocation *RA*). Let $G = (V, E)$ be a scheduled data-flow graph, S be a set of schedules, and $\varphi \in S$. Let $C(\varphi) = [C_{min}(\varphi), C_{max}(\varphi)]$ be the schedule range of schedule φ, and $c \in C$. Let *ModType* be the set of module types, and $l \in ModType$. Distribution function *DF*: $S \times ModType \times C \rightarrow \mathbb{R}$ is a function, given by $DF(\varphi, l, c) = |\{v \in V \mid \xi(v) = l \wedge c \in [\varphi(v), \varphi(v) + \delta(v)]\}|$, which denotes how resources are used over time. The resource allocation $RA(\varphi, l)$ for each $l \in ModType$ is given by $RA(\varphi, l) = MAX_{c \in C(\varphi)}\, DF(\varphi, l, c)$, which denotes the number of resources of type l, needed to implement schedule φ.

Definition 3.8 (Resource Allocation Costs *RA*). Let S a set of schedules and $\varphi \in S$. The resource allocation costs *RA*: $S \rightarrow \mathbb{R}$ is a function, given by $RA(\varphi) = \sum_{l \in ModType} cost(l) \cdot RA(\varphi, l)$.

3.3 Constraint sets and performance measures

The task of a high-level synthesis system is to find an optimal solution with respect to the performance measures, while satisfying the constraints specified by a designer. Most high-level synthesis (sub-)problems can be defined as an optimization problem.

Definition 3.9 (Combinatorial optimization problem). A combinatorial optimization problem is a collection of instances (F, c). An instance of an optimization problem is a pair (F, c), where F is a set of candidate solutions and c: $F \rightarrow \mathbb{R}$ is a cost function. The problem is to find an $f \in F$, for which $\forall_{y \in F}\, c(f) \leq c(y)$ in case of a minimization problem, and $\forall_{y \in F}\, c(f) \geq c(y)$ in case of a maximization problem.

In practice, high-level synthesis tasks like selection, allocation, scheduling, and binding are performed with certain goals and constraints in mind. To be able to distinguish good solutions from bad solutions, the goals can be described by the use of performance measures, resulting in a cost function c. Performance measures which are commonly found in high-level synthesis publications, are global completion time (optimize the number of cycle steps between consumption of input data and production of output data), throughput rate (find a schedule such that input data can be offered as fast as possible), and resource allocation (find a schedule which induces a minimal resource allocation). Properties such as mutual exclusion [Camp91], chaining, multi-cycling [Stok91], and time shapes [Werf91,Eijn91] can be used to find schedules with even better performance measures. Other performance measures like testability [Gebo92], power consumption [Chan92], interconnect allocation [Rim92, Rama92], placement and routing [Weng91,Pang91], system clock optimization [Park85], and more can be found in several publications, but are very hard to quantify accurately at the abstraction level used for high-level synthesis.

The set of candidate solutions F consists of solutions which don't violate the constraints imposed on the problem (also called feasible solutions). In this thesis constraints like precedence constraints, time constraints, throughput rate constraints, and resource constraints will be discussed in more detail in Section 3.4. In Section 3.5 some typical high-level synthesis scheduling problems will be formally defined.

3.4 High-level synthesis scheduling constraints and goals

3.4.1 Data-flow graphs and execution order

Section 2.2.1 told that the execution order of nodes inside a data-flow graph (V, E) is constrained by the structure of the data-flow graph, inducing a partial order $\prec$ on the execution of the nodes.

The partial order $\prec$ induced by the structure of an acyclic data-flow graph (V, E) is rather straightforward to determine. Let $u,v \in V$ and $(u, v) \in E$. Edge (u, v) denotes transfer of data from operation u to v, in other words data produced by operation u is

consumed by operation *v*. This implies that operation *v* can start its execution after operation *u* has produced data for *v*. The number of cycle steps operation *u* requires to produce data for operation *v* is described by the intra-iteration distance.

Definition 3.10 (Intra-iteration distance δ). Let (V, E) be a data-flow graph, $u,v \in V$, and $(u, v) \in E$. The intra-iteration distance $\delta(u, v) \in \mathbb{R}$ between *u* and *v* is defined as the number of cycle steps *u* needs to translate its input data into input data for *v*. In case $(u, v) \notin E$, $\delta(u, v) = -\infty$, and in case $u = v$, $\delta(u, v) = 0$.

Hence the cycle step in which *u* produces data for *v* is given by $\varphi(u) + \delta(u, v)$. In general the intra-iteration distance $\delta(u, v)$ for each outgoing edge (u, v) of operation *u* to operation $v \in V$, equals the operation execution delay $\delta(u)$ of operation *u*. Nevertheless, the situation in which an operation generates tokens at different cycle steps for each output edge, resulting in different values for the intra-iteration distance between operations, can also be modelled (also called time-shapes).

For each edge $(u, v) \in E$, the following relation can be derived:

$$\varphi(u) + \delta(u, v) \leq \varphi(v)$$

Such a relation is called a distance relation, and describes a constraint with respect to the relative distance between the schedule of operations. Because in practical situations $\delta(u, v) \geq 0$ (hence for each edge $(u, v) \in E$, $\varphi(u) \leq \varphi(v)$), the edges in a data-flow graph impose restrictions on the order of execution of operations. For each edge $(u, v) \in E$ such an execution order constraint is denoted by $u \prec v$, also called a dependence relation. The relation $\prec$ is irreflexive (in other words $u \not\prec u$), anti-symmetric (in other words $u \prec v \Rightarrow v \not\prec u$) and transitive (in other words $u \prec v \wedge v \prec w \Rightarrow u \prec w$). Hence an acyclic data-flow graph (V, E) induces a structure $(V, \prec)$, which can be obtained by taking the transitive closure of *E*, denoted by E^*. Structure $(V, \prec)$ imposes a strict partial order among the execution order of operations of *V*, and constraints the schedule of the data-flow graph. In Chapter 4 some algorithms will be presented, to determine the distance between each arbitrary pair of operations from *V*. These distances will be used by scheduling algorithms to guarantee the construction of feasible schedules with respect to the constraints induced by the structure of the data-flow graph.

In case a data-flow graph contains loop structures, it establishes a cyclic flow of data. In that case a strict partial order (which is irreflexive and anti-symmetric by definition) cannot be derived directly from the structure of such a data-flow graph. Take for instance the example shown in Figure 3.1. In this data-flow graph a transitive closure E^* of edges would result in $(v_1, v_1) \in E^*$ and $(v_2, v_2) \in E^*$.

In data-flow graphs, data-flow and control-flow are integrated into one model. The controlling part takes care that the loop is executed the correct number of times. It should

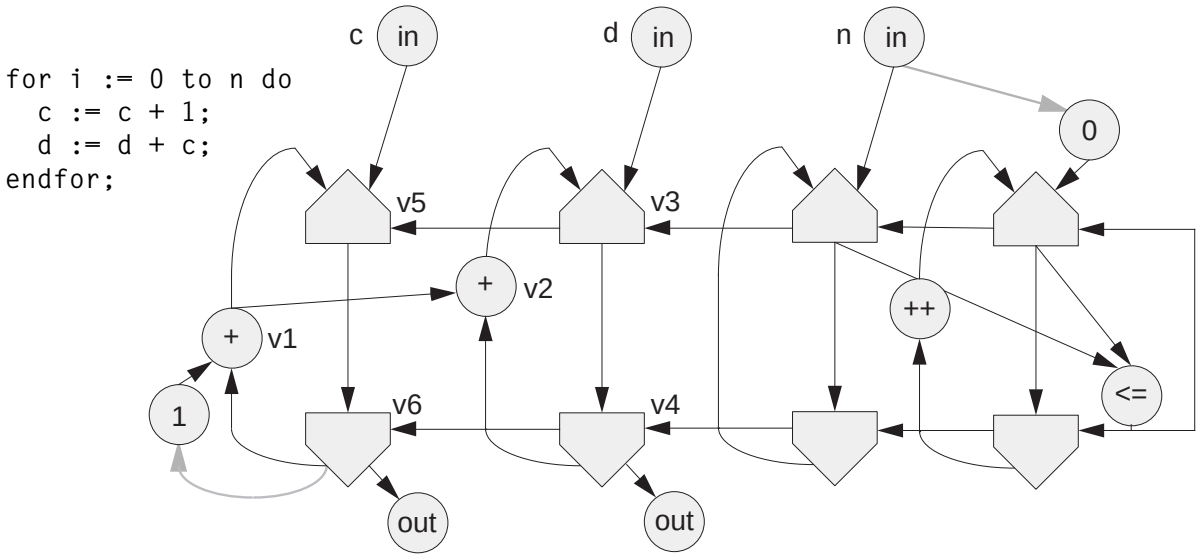


Figure 3.1 Data-flow graph loop example.

be noticed however that a data-flow graph loop structure is a shorthand description for successive executions of the operations inside this loop structure. In Figure 3.2, the loop structure is explicitly unfolded, resulting in an acyclic data-flow graph.

Unfolding a loop is not an efficient way to derive the execution order of operations. First of all, the number of operations increases with each unfolding, increasing the input size of the problem for synthesis. Secondly, the regularity induced by the loop structure is not explicitly visible any more, and greedy synthesis methods might produce irregular data-paths, hence special analysis techniques are needed to be able to construct efficient data-paths. Finally, the number of unfoldings might be unknown at compile time (for example in the case of while loops), hence in general it is not possible to eliminate the circular structure of the loop by using unfolding.

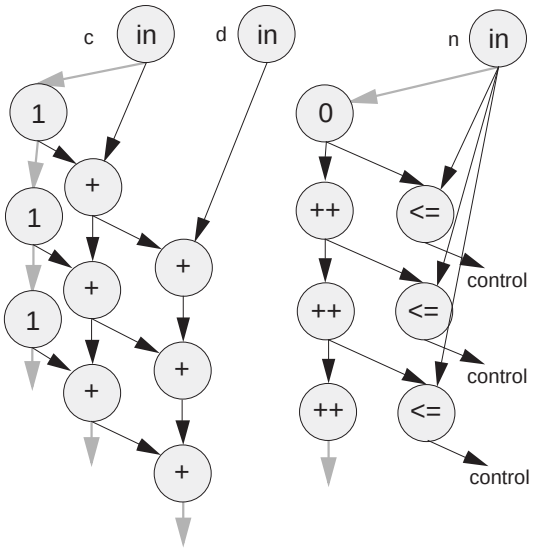


Figure 3.2 Partially unfolded loop.

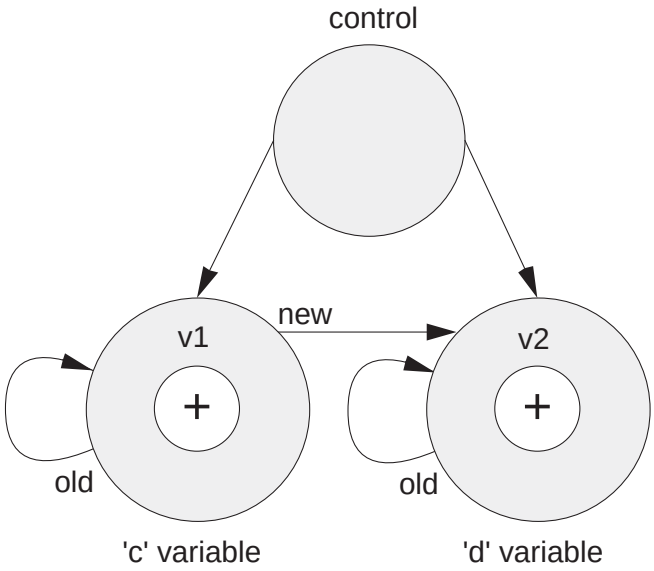


Figure 3.3 Control-flow and data-flow execution order.

Another method is to consider the loop body as a basic block, and synthesize it separately. This limitation eliminates the possibility of loop pipelining (see Section 3.4.4 for more information about pipelining), and therefore unacceptably restricts the solution space of synthesis.

In Figure 3.3 the order of execution of operations of the data-flow graph given in Figure 3.1, is drawn in terms of the production and consumption of old and new values with respect to the current loop iteration. An old value of *c* is translated to a new value of *c* (by execution of addition v_1), which together with an old value of *d* is used to calculate a new value of *d* (by execution of addition v_2). Control is used to repeat the execution of this behaviour a specified number of times.

The moment of execution of operation v_1 in the current iteration is restricted by previous executions of v_1. The execution order of operation v_2 in the current iteration is restricted by previous executions of v_2, and the execution of operation v_1 in the current iteration. This means that new *c*-values and *d*-values can only be generated consecutively with respect to their previously generated values. This imposes an execution order constraint with respect to consecutive executions of the addition operations among different iterations of the loop structure.

In Figure 3.3, two different kind of dependencies can be distinguished. Dependencies referring to data produced and consumed in the current iteration are called intra-iteration dependencies. Dependencies referring to data produced in previous iterations and consumed in the current iteration are called inter-iteration dependencies. One way to make execution order constraints with respect to loop constructs explicit for scheduling, is by the use of so-called delay-nodes [Eijn91]. A delay node (denoted with symbol ‘T’) has one input and one output. Any token arriving on the input is copied unaltered to the

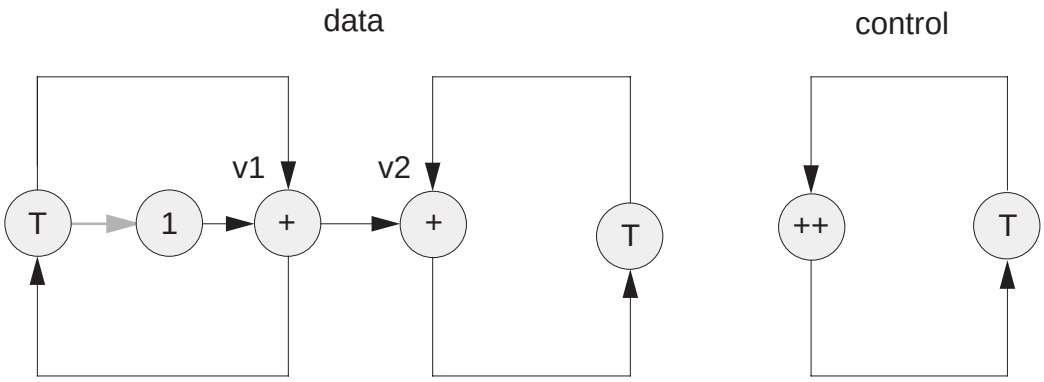


Figure 3.4 Execution order constraints of the example of Figure 3.1.

output. The special property of a delay node is the fact that it contains one (or more) initial token(s), which hold initial values used for the first iteration(s) of the loop.

3.4.2 Dependence and distance graphs

In Figure 3.4, the data-flow graph (V , E) of Figure 3.1 is modelled by using delay nodes. The resulting graph is called a dependence graph (T , F), where T denotes the set of operations, consisting of operations from V, excluding the nodes combining data-flow and control-flow, such as entry and exit nodes, and extended with delay nodes to explicitly model data dependencies between loop iteration. F is the set of edges, which model dependencies between the operations of T.

A dependence graph can be interpreted as a self-executing entity, also called a process. Just like with data-flow graphs, the execution mechanism of a process can be described by a token-flow model. During the execution of a process, each operation is executed exactly once. The mechanism to start the execution of a process is called the invocation of a process. A process can be repeatedly invocated, leading to successive process executions. A process iteration refers to the execution of a particular process invocation. Details about processes and how to obtain a dependence graph from a data-flow graph can be found in [Kost95].

The dependence graph of Figure 3.4 contains two strongly-connected components, called cycles (not to be confused with the definition of a cycle step!). The number of delay nodes inside a cycle describes the maximum number of pipeline stages that can be distributed among this cycle to obtain more efficient schedules. This leads to the notion of the so-called inter-iteration distance.

Definition 3.11 (Inter-iteration distance λ). Let (T , F) be a dependence graph, and (u , v) $\in F$. The inter-iteration distance $\lambda(u , v) \in \mathbb{N}$ between u and v is defined as the number of loop iterations between the production of data by u and the consumption of that data by v.

In a dependence graph (T , F) the boundary between loop iterations is denoted explicitly by delay nodes. Let $u,v \in T$, $(u , v) \in F$, and $\tau(u)$ = 'delay', then $\lambda(u , v)$ is equal to the number of initial tokens inside u. In other cases $\lambda(u , v) = 0$.

Let $\varphi_n(u)$ denote the time that $u \in T$ starts its execution in the n^{th} iteration of a loop. For each $(u , v) \in F$ it can be derived that:

$$\varphi_n(v) \geq \varphi_{n - \lambda(u , v)}(u) + \delta(u , v) \qquad (3.1)$$

Let $dii \in \mathbb{N}$ represent the distance between two successive process invocations (also called data introduction interval). When the distance between every two process invocations is constant, equation (3.1) can be rewritten into:

$$\varphi_n(v) \geq \varphi_n(u) - \lambda(u , v) \cdot dii + \delta(u , v) \qquad (3.2)$$

A short-hand notation $\varphi(v)$ can be used for $\varphi_n(v)$, which abstracts from the notion of iteration. In that case equation (3.2) can be rewritten as:

$$\varphi(v) \geq \varphi(u) - \lambda(u , v) \cdot dii + \delta(u , v) \qquad (3.3)$$

Relations as in equation (3.3) are called distance relations. Distance relations can be visualized by labelling each edge (u , v) of a dependence graph (T , F) by a tuple $(\delta(u , v) , \lambda(u , v))$. A labelled dependence graph is also called a distance graph.

In Chapter 4 it will be shown that cycles in a dependence graph will impose time constraints with respect to the operations inside these cycles, resulting in a lower bound and upper bound on the range of cycle steps in which these operations can be scheduled. Algorithms will be presented to determine and update these bounds, and hence guarantee the construction of feasible schedules with respect to the constraints induced by the structure of the data-flow graph. Furthermore it will be shown that cycles in a dependence graph impose a lower bound value of the invocation distance, and a new algorithm will be presented to derive this lower bound efficiently.

3.4.3 Data-flow graphs, arrays and dependence analysis

In data-flow graphs the contents of an array is modelled by a single token. Scalar values are written to and retrieved from the array by using so-called update and retrieve nodes respectively, which provide a way to index the array. A retrieve node has two inputs, an array input which accepts a token holding an array a, and an index input i to address the array. It has two outputs, one array output which passes the token associated with array a unaltered, and a scalar output which return $a[i]$. An update node has three inputs, an array input which accepts a token holding an array a, an index input i to address the

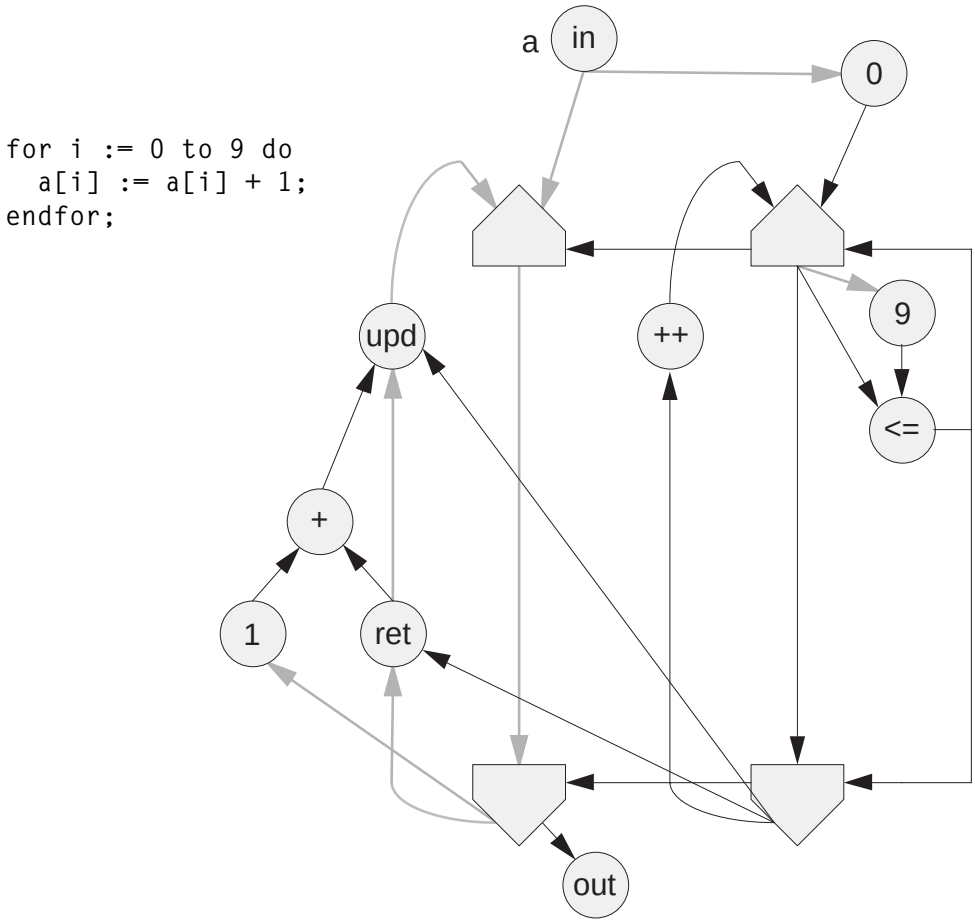


Figure 3.5 Loop with array, corresponding data-flow graph.

array, and a scalar input *d*. It has one array output which returns a token associated with array *a*, for which *a*[*i*] is equal to value *d*.

The order in which an array is accessed is modelled by a sequence of edges among update and retrieve nodes, which induces a linear order on the array accesses in such a way that no array assignment and retrieval conflicts arise. In case loop structures are used in combination with arrays, the linear sequence of edges may impose severe restrictions with respect to loop pipelining, as will be shown by the following examples.

An example of a loop structure in combination with an array is shown in Figure 3.5. Because the values of *a*[*i*] are independent for different values of *i*, and hence new *a*-values can be generated independently from the generation of other *a*-values, there are no execution order constraints between successive executions of the addition operation, despite the structure of the data-flow graph edges caused by sequencing the array. A direct relationship between the structure of data-flow edges and order constraints is not explicitly obvious. Dependence analysis is needed to extract the exact order constraints from the data-flow graph structure [Bane93,Zima90]. The execution order constraints, obtained from such an analysis, are shown in Figure 3.6. From this figure it is obvious that among the loop-body operations, no execution order restrictions exist

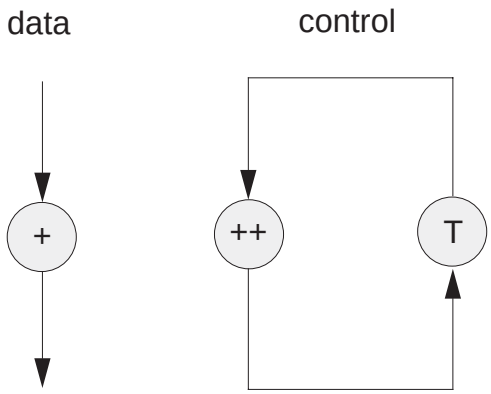


Figure 3.6 Execution order constraints of the example of Figure 3.5.

regarding previous loop executions. The control part must generate a sequence of index values, and it must terminate the loop.

An example of a loop structure in which dependencies among different loop iterations exist, is shown in Figure 3.7. The algorithmic behaviour of this loop structure is shown in Figure 3.7.

Just like in the previous example shown in Figure 3.5, the array sequence in Figure 3.7 introduces order restrictions which are unnecessary for scheduling. Additional dependence analysis is needed to investigate the index-space structure, which leads to the execution order constraints as visualized in Figure 3.9. In this figure, delay nodes are used to provide a reference to scalar values of the array produced in previous loop iterations.

The translation of the array index calculations from the original data-flow graph to the control part of Figure 3.9 isn't very efficient. In Figure 3.10 a more efficient calculation of index expressions can be found, using delay nodes to make values produced in previous iterations accessible in current iterations.

With respect to control it is a mistake to assume that the controlling part of the loop should always be executed synchronous to the loop body. In the example of Figure 3.1 and Figure 3.2 it can be observed that the controlling part of the loop (represented in the program by variable *i*) is used to manage the number of loop iterations, and to provide indexing for arrays (which in a data-path should be translated into memory addresses management). Despite the data-flow edges from the controlling part of the loop towards the entry and exit nodes, and despite some addressing aspects, there are no data depend-

```
for i := 2 to 9 do
  a[i] := b[i - 1] + c[i - 2];
  b[i] := a[i] + 1;
  c[i] := a[i] * b[i];
endfor;
```

Figure 3.7 Algorithmic behaviour of data-flow graph in Figure 3.7

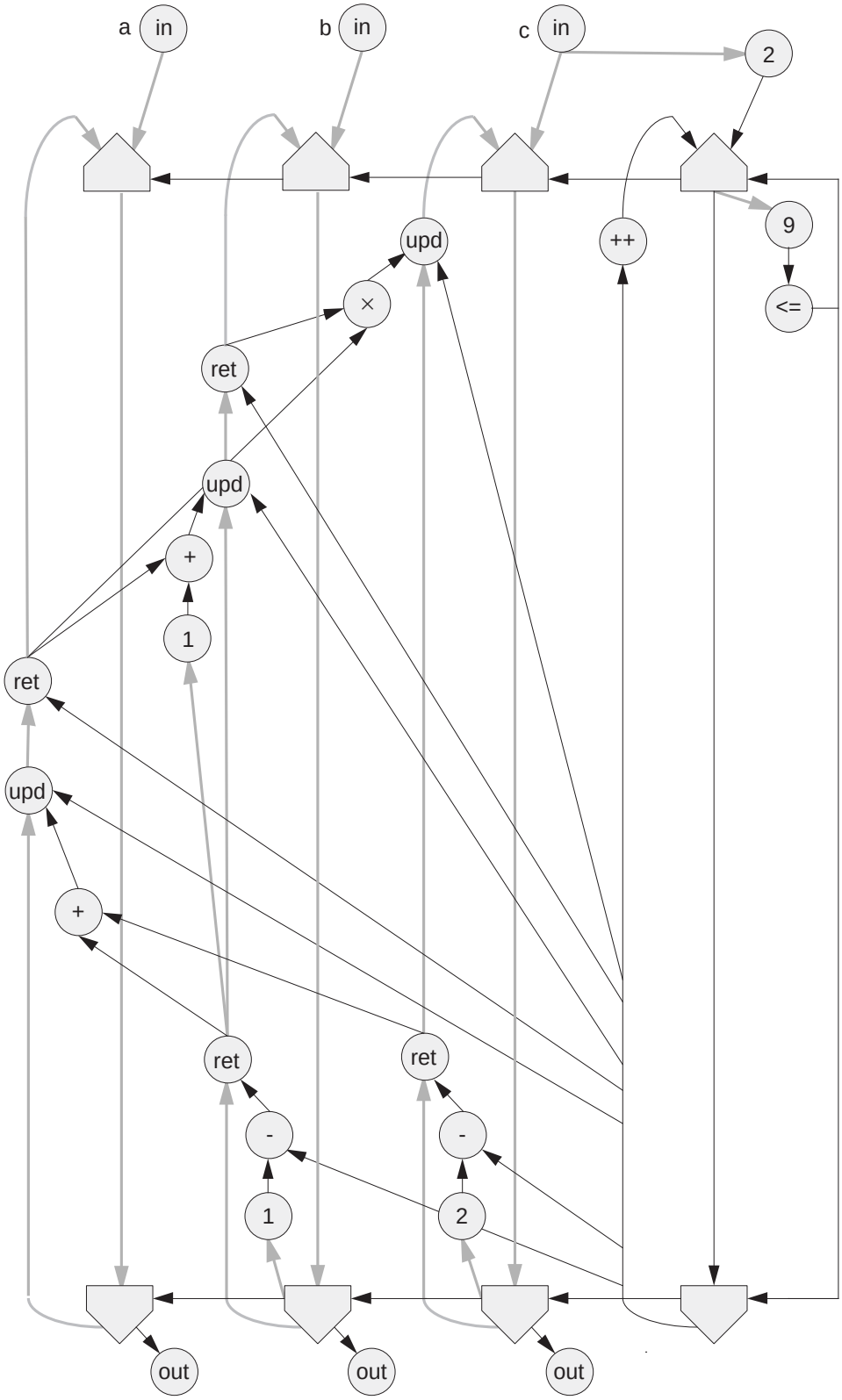


Figure 3.8 Loop construction containing array accesses.

encies whatsoever between the controlling part and the values calculated inside the loop body, which has been made explicitly visible in Figure 3.2. Schedules can be created in which loop bodies start their execution before a control value has been determined, or the other way around, in which successive control values are determined before the

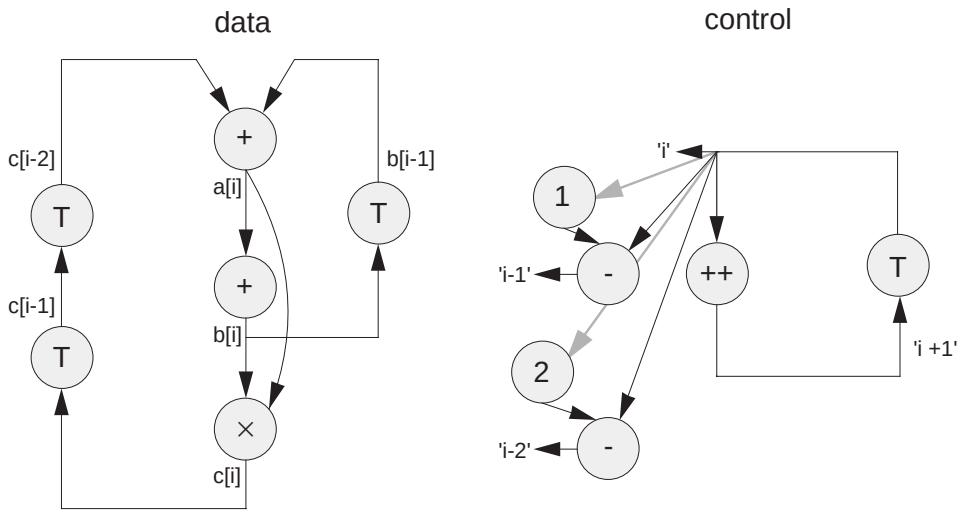


Figure 3.9 Execution order constraints of the example of Figure 3.7.

loop body has finished its execution. In the data-flow graph of Figure 3.5 one can see that a data value obtained by the controlling part is used to index an array. This implies that the control value should be calculated before the corresponding loop body can be executed. Nonetheless successive control values can be calculated before the loop body has finished its execution.

Another discussion with respect to the control part of the loop is about the implementation. It can be synthesized just like the loop-body, and be integrated inside the data-path. It can also be separated from the data-flow part of the loop, and dedicated implementations can be created by using special techniques, such as for instance for address generation [Lipp91,Vanh93]. Control-flow can also be modelled as a finite state machine, which in combination with logic synthesis or in combination with other special techniques can lead to dedicated gate-level implementations.

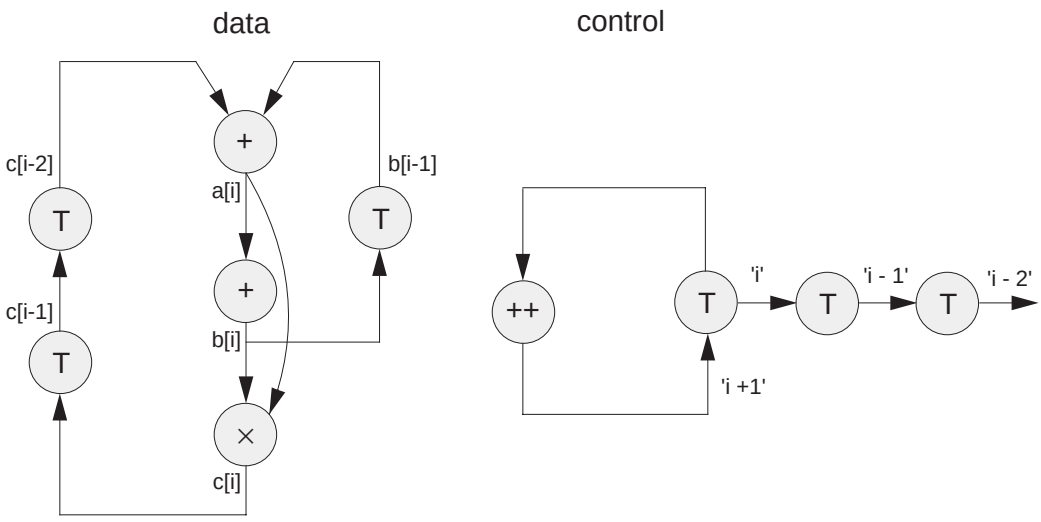


Figure 3.10 Calculation of index value with delay nodes.

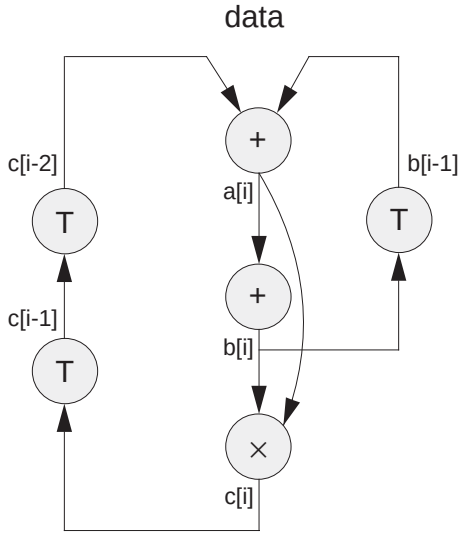


Figure 3.11 Abstraction of control.

As control can be synthesized by other synthesis tools, a scheduler must be capable to abstract from control. This can be achieved by taking into account the amount of time needed to generate control values, which on its turn are needed in the data-path, and model this execution time on the edges which represent retrieve and update of data from arrays. An example obtained from Figure 3.10 is shown in Figure 3.11.

So far, delay nodes have been used to describe order execution constraints with respect to a single loop iteration. An example of such a loop structure can be found in Figure 3.12, in which a simplified data-flow graph is shown (for simplicity array accessing hasn't been drawn explicitly).

In case of nested loops, the invocation distance of operations can be different with respect to each loop. Nested loop structures, in which more than one iterator is used, result in multi-dimensional index spaces. Let the period $p_i \in \mathbb{N}$ of a loop, using iterator i be the number of cycle steps such a loop execution takes. Because operations in multi-dimensional loops can be repeated with different periods with respect to each loop, a delay node needs to be annotated with a period vector $\boldsymbol{p}$ to be able to describe the multi-dimensional characteristic of the execution order constraints. In the example of Figure 3.12 the execution of f depends on two iterators i and j, and hence the period vector is given by (p_i , p_j). The distance between two successive executions of an operation is not necessarily constant. In [Verh92b] a stream model to describe data dependencies between operations based on period vectors is introduced. Determination of the execution order of two operations is modelled as an Integer Linear Programming (ILP) Problem. In general, such a problem can become rather difficult to solve efficiently, and in [Verh95] some special cases are described in detail.

In this thesis it is assumed that the distance between successive operation executions is constant over all loop indices.

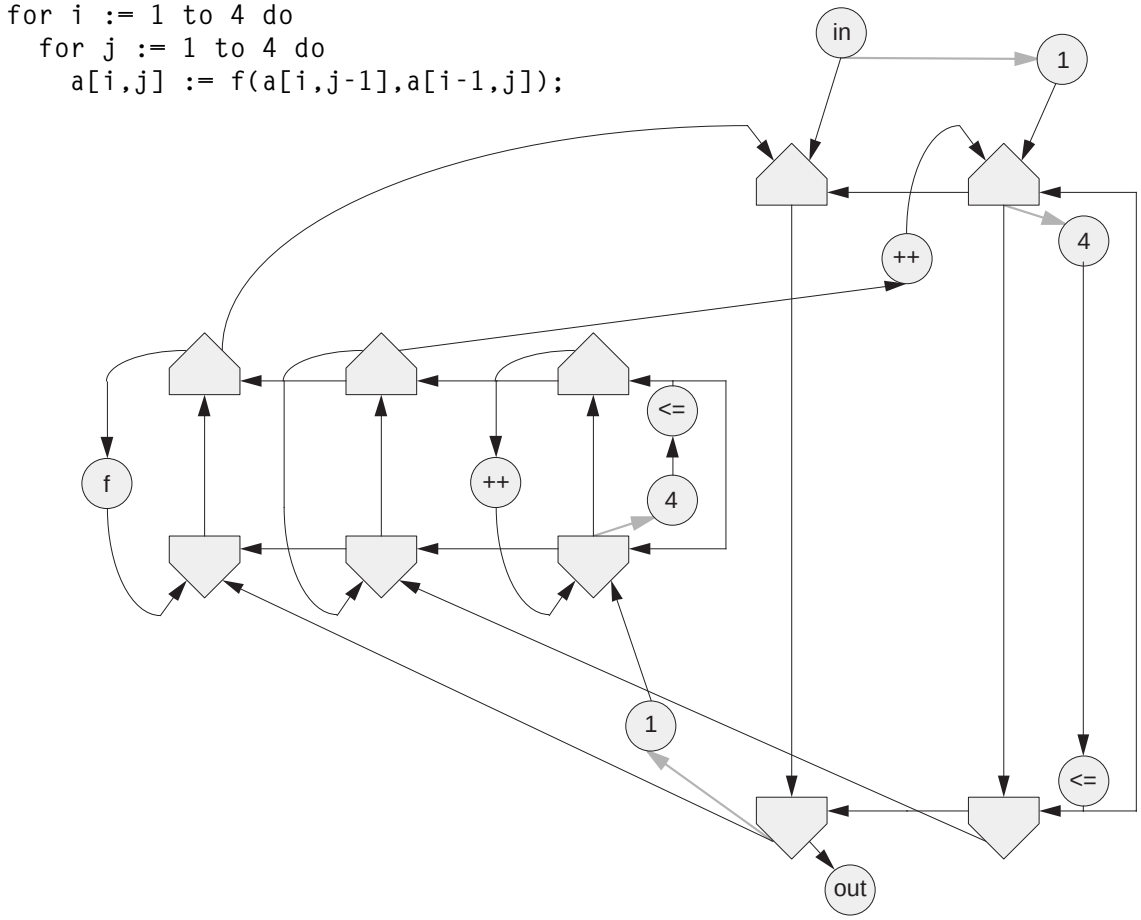


Figure 3.12 Example of multi-dimensional loop.

3.4.4 Time

When synthesizing digital circuits, a designer wants to be able to make a trade-off between the speed and the area of a circuit. This situation characterizes one of the main differences between high-level synthesis and ordinary software compilation techniques [Aho86], the importance of the notion of time. Throughput rate constraints imposed on for instance DSP applications are crucial, and must be satisfied in any case. This imposes severe requirements with respect to the quality of solutions produced by scheduling algorithms concerning time.

A common constraint in high-level synthesis is the so-called global time constraint $T_{max} \in \mathbb{N}$ (also called cycle step budget). Let (V, E) be a data-flow graph. A global time constraint T_{max} for a schedule φ implies $C_{max}(\varphi) \leq T_{max}$, in other words the completion time $C_{max}(\varphi)$ induced by schedule φ should not exceed time constraint T_{max}.

Besides a global time constraint, a local time constraint can be used. A local time constraint $t_c(u, v)$ between two arbitrary operations $u,v \in V$ denotes the maximal distance between these two operations, in other words $\varphi(v) \leq \varphi(u) + t_c(u, v)$.

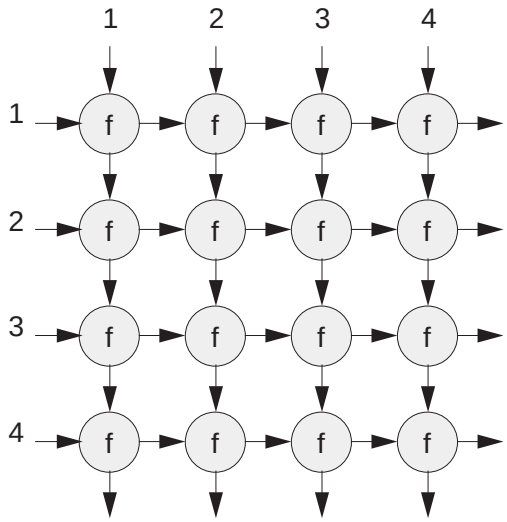


Figure 3.13 Unfolded loop of the example given in Figure 3.12.

In Chapter 4 it will be shown that a cycle in a dependence graph will impose local time constraints with respect to operations inside this cycle. It will also be shown how a combination of time constraints and dependence constraints influence the range of cycle steps in which operations can be scheduled. Some algorithms will be presented to determine and update these ranges, to be able to efficiently produce schedules satisfying their time constraints.

Time can also be used as a quantity to be optimized. A schedule objective, which can be found very often in high-level synthesis, is the completion time $C_{max}(\varphi)$ (see Definition 3.6) induced by schedule φ.

In some cases it is important to generate an implementation of a data-path in which data can be offered successively at a particular rate. This leads to the notion of the so-called throughput rate.

Definition 3.12 (Throughput rate *dii*). Let (V, E) be a data-flow graph. The throughput rate (also called data introduction interval) $dii(\varphi) \in \mathbb{N}$ of a schedule φ is defined as the distance between two consecutive invocations (often denoted by *dii*).

Just like with time constraints, the throughput rate can be used as a constraint or as an objective. The throughput rate of a schedule can be improved by the use of pipelining. In this situation operations from current iterations are scheduled concurrently with operations from preceding and/or successive loop iterations. An example of the difference of throughput rate of a pipelined and non-pipelined schedule can be found in Figure 3.14, which shows two different schedules for the example of Figure 3.7.

To be able to construct loop pipelined schedules, it is important to know the minimal and maximal distance between two operations. The result produced by the i^{th} multipli-

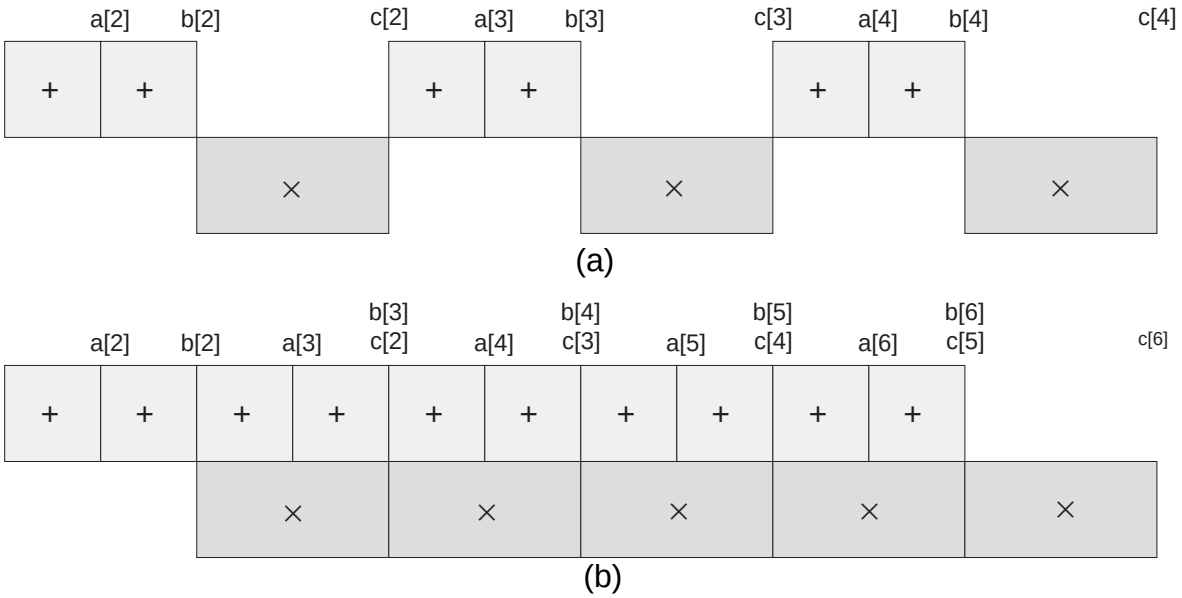


Figure 3.14 Non-pipelined (a) and pipelined (b) schedule.

cation of the example Figure 3.11 is needed in iteration $i + 2$ by the addition operation to generate $a[i+2]$. Depending on the execution rate of the loop, this imposes lower and upper bounds on the range of cycle steps available for scheduling operations, which will be discussed in more detail in Chapter 4.

3.4.5 Resources

Just like time, another important aspect in high-level synthesis are hardware resources, because each square millimetre saving in terms of chip area can lead to economical advantages.

A resource constraint imposes an upper bound on the resource allocation. Let (V , E) be a data-flow graph, and $RC(l)$ be an upper bound for module type $l \in ModType$. A resource constrained schedule φ implies that $\forall_{l \in ModType}\, RA(\varphi , l) \leq RC(l)$.

The most obvious resource bounds used in high-level synthesis is an upper bound on the number of functional units (also called modules). This imposes a restriction on the number of operations of a particular operation type that can be scheduled simultaneously. Only very little is known of schedulers which can cope with memory allocation constraints and interconnect allocation constraints during scheduling. Most methods reported try to optimize memory allocation or interconnect allocation during or after scheduling. In Chapter 4 a short overview of algorithms being capable of handling resource constraints will be given.

A minimal resource allocation $RA(\varphi)$ (see Definition 3.8) induced by schedule φ can also be used as an objective for scheduling.

3.5 Schedule problems

The key problems discussed in this thesis consist of two kinds of scheduling problems:

Definition 3.13 (Time constrained scheduling). Given are data-flow graph (V , E) and a time constraint T_{max}. Find a schedule φ such that $C_{max}(\varphi) \leq T_{max}$, inducing a minimal resource allocation RA(φ).

Definition 3.14 (Resource constrained scheduling). Given are a data-flow graph (V , E), and for each $l \in ModType$ a resource constraint $RC(l)$. Find a schedule φ such that $\forall_{l \in ModType}\, RA(\varphi , l) \leq RC(l)$, inducing a minimal completion time $C_{max}(\varphi)$.

In Chapter 4 and Chapter 5 it will be shown that time constraints and resource constraints are tightly related.

Another category of interesting scheduling problems deal with throughput rate. In Chapter 4 it will be shown that throughput constraints are a special case of time constraints, and therefore can be considered as a special case of time constrained scheduling problems.

3.6 Conclusions

In this chapter dependence constraints, time constraints, and resource constraints have been discussed, playing a leading role in high-level synthesis scheduling. The chapter is concluded with a formal introduction of two scheduling problems, time constrained and resource constrained scheduling, which will be the key problems discussed in the remainder of this thesis.

Chapter 4 Schedule Constraints

4.1 Introduction

In practical cases the high-level synthesis scheduling problem is subject to constraints, such as precedence constraints (derived from an algorithmic behaviour), resource constraints (derived from a network structure), and all kinds of time constraints (such as completion time and data arrival rate).

During the construction of a schedule, operations are assigned to cycle steps. Because of constraints, operations cannot be assigned to arbitrary cycle steps. The range of cycle steps available for operations to start their execution without violating any constraints, is called the feasible schedule range of operation u. Several methods to determine bounds on the feasible schedule ranges of operations will be presented in this chapter. These methods differ in accuracy and efficiency.

The feasible schedule range of operations can change during the construction of a schedule. If for instance, due to the schedule of a particular operation, all units of a resource type become occupied in cycle step c, the feasible schedule range of unscheduled operations needing this resource type for execution cannot include cycle step c in their schedule range any more.

Schedulers which allow operations to be assigned to cycle steps outside their feasible schedule range, may result in inefficient scheduling methods which need backtracking or repair algorithms to come up with a feasible schedule. To obtain more efficient scheduling methods, one can restrict the search space of a scheduler by only allowing operations to be scheduled within their feasible range. These feasible ranges should restrict the search space as good as possible without excluding all optimal solutions from the search space. The algorithms to determine feasible schedule ranges should be very efficient, to obtain overall efficient scheduling methods.

This chapter presents the influence of different kinds of constraints on the feasible schedule range. It gives an overview of existing methods which determine feasible schedule ranges, and presents some new algorithms to determine the feasible schedule range accurately and efficiently. Furthermore it presents a unified approach to treat all these constraints in a single model.

4.2 Distance matrix

Let (T, F) be a dependence graph derived from a data-flow graph (V, E), and let $u,v \in T$, such that $u \prec v$. The time where v can start its execution depends on the time u ends its execution, given by:

$$end(u) \leq begin(v)$$

Let $distance(u, v)$ denote the minimal distance between u and v, then:

$$\varphi(u) + distance(u, v) \leq \varphi(v)$$

Let dii be a shorthand notation for $dii(\varphi)$. The minimal distance $distance(u, v)$ between each pair of operations $u,v \in T$, with $(u, v) \in F$, is given by $\delta(u, v) - \lambda(u, v) \cdot dii$. The question is how the distance between two arbitrary operations from T can be determined.

A path p inside (T, F) is a sequence of operations $t_1, t_2, \ldots, t_r$, such that $(t_i, t_{i+1}) \in F$ for each $i = 1, 2, \ldots, r - 1$. For each relation (t_i, t_{i+1}) we have a corresponding distance relation:

$$\varphi(t_{i+1}) \geq \varphi(t_i) - \lambda(t_i, t_{i+1}) \cdot dii + \delta(t_i, t_{i+1})$$

Adding all distance relations of path p results in:

$$\varphi(t_r) \geq \varphi(t_1) - (\lambda(t_1, t_2) + \ldots + \lambda(t_{r-1}, t_r)) \cdot dii + \delta(t_1, t_2) + \ldots + \delta(t_{r-1}, t_r) \qquad (4.1)$$

which by summation of all lambdas and deltas gives:

$$\varphi(t_r) \geq \varphi(t_1) - \lambda'(t_1, t_r) \cdot dii + \delta'(t_1, t_r) \qquad (4.2)$$

which can be rewritten as:

$$\varphi(t_r) \geq \varphi(t_1) + distance(t_1, t_r) \qquad (4.3)$$

From equation (4.3) it can be derived that the time operation t_1 starts its execution restricts the time operation t_r may start its execution, in other words, it defines a minimal distance $distance(t_1, t_r)$ between the schedule time of operations t_1 and t_r. If during scheduling operation t_1 becomes scheduled in cycle step $\varphi(t_1)$, the minimal distance $distance(t_1, t_r)$ denotes that operation t_r can only be scheduled inside or later than cycle step $\varphi(t_1) + distance(t_1, t_r)$. On the other hand, if operation t_r becomes scheduled in cycle step $\varphi(t_r)$, operation t_1 can be scheduled inside or earlier than cycle step $\varphi(t_r) - distance(t_1, t_r)$. Hence the distance relations clearly restrict the feasible schedule range of operations with respect to the schedule of other operations. To derive feasible

schedule ranges during scheduling efficiently, the distance information between operations will be needed at any time.

In [Heem90] two algorithms are discussed to determine the distance between operations. In both cases the schedule time $\varphi(t_r)$ of a reference operation $t_r \in T$ is fixed, which by definition equals 0. The first method is based on Fourier-Motzkin elimination. By recursive elimination of variables in the set of distance inequalities, an $O(|T|^3)$ algorithm can be derived to determine feasible schedule ranges of operations. When the schedule time of another operation $t_i \in T$ is fixed, the schedule ranges need re-computation. In a worst case situation a complete Fourier-Motzkin elimination must be performed again. The second method is based on calculation of the all-pairs longest path, which can be calculated by the Floyd-Warshall algorithm (complexity $O(|T|^3)$) or Johnson's algorithm (complexity $O(|T| \cdot |F| + |T|^2 \cdot \log|T|)$) (for more details, see [Corm90]). Because $\varphi(t_r)$ equals 0, the distance between t_r and t_i is exactly the length of the longest path. If there is no path, the distance equals -∞. When an operation t_i is scheduled, an edge (t_r , t_i) with distance $[\varphi(t_i) , 0]$, and an edge (t_i , t_r) with distance $[-\varphi(t_i) , 0]$ are added, or, when these edges exist, the distances are updated to these values. The update of the feasible schedule ranges is performed by re-computing the longest paths by re-applying the Floyd-Warshall algorithm or the Johnson algorithm.

In [Heem92] a third method is derived from the second method, which applies a single-source longest path algorithm instead of an all-pairs longest path. The Bellman-Ford algorithm, with a complexity of $O(|T| \cdot |F|)$, is used twice to derive the distance from a reference operation t_r to all other operations and vice versa. When scheduling an operation, the Bellman-Ford algorithm is used again to update the distances between operations. When scheduling operations one by one, this will lead to a worst case complexity of at least $O(|T|^2 \cdot |F|)$. The complexity will increase significantly for scheduling methods which use many tentative movements of operations during scheduling. One such example is force directed scheduling [Paul89], in which $O(|T|^2 \cdot T_c)$ tentative movements are performed during scheduling, with $T_c \in \mathbb{N}$ being the overall time constraint.

In [Goos89] an iterative approach is suggested to calculate distances. It relies on the way a schedule is constructed, which is a list scheduling algorithm that assures scheduling operations in a topological order. In case of an intra-iteration dependency (u , v), scheduling an operation u will impose a lower bound on the schedule time $\varphi(v)$ of operation v, which is called a computable bound. In case of inter-iteration dependencies, a lower bound on $\varphi(v)$ cannot be computed explicitly using this topological method, and therefore backtracking is introduced. A weight function ω, which depends on dii, is used as follows:

$$\varphi(v) \geq \varphi(u) + \omega(dii)$$

The value of $\varphi(v)$ in a current list scheduling stage is calculated by using the $\varphi(u)$ value obtained by the previous list scheduling stage. From experiments it follows that the

choice of the initial value of $\varphi(u)$ is not important, and is set to 0. The method is however reported to be highly dependent on the value chosen for *dii*. Besides, a theoretical foundation why this method works correctly is missing. Finally, no results are given about the typical amount of iterations needed to make this backtracking algorithm converge.

As presented in [Heem90,Lam89], the minimal distance between two arbitrary operations $t_i,t_j \in T$ can be obtained by calculating the longest path distance from t_i to t_j, using the distance between successive operations as path weights in the following way. Let $T = \{t_0 , t_1 , ... , t_n\}$ be a set of operations. Let $D^{(k)}(i , j)$, with $0 \le k \le n$, be the longest path from operation t_i to t_j, consisting of operations $\{t_0 , t_1 , ... , t_k\}$. A recursive definition of $D^{(k)}(i , j)$ is given by:

$$D^{(k)}(i, j) = \begin{cases} \delta(t_i, t_j) - \lambda(t_i, t_j) \cdot dii & \text{if k = 0} \\ \text{MAX}(D^{(k-1)}(i, j), D^{(k-1)}(i, k) + D^{(k-1)}(k, j)) & \text{if k} \ge 1 \end{cases} \quad (4.4)$$

To determine the longest path distance $distance(t_i , t_j)$ between each pair of operations, several all-pairs longest path algorithms can be used [Corm90], such as the Floyd-Warshall algorithm ($O(|T|^3)$) or the Johnson algorithm ($O(|T|^2 \cdot \log|T| + |T| \cdot |F|)$). The resulting matrix $D^{(|T|-1)}$ is called a distance matrix, and for each $t_i,t_j \in T$, $distance(t_i , t_j) = D^{(|T|-1)}(i , j)$. In [Lam89] symbolic expressions are used in the distance matrix to be able to recalculate the longest paths very quickly if the value of the data introduction interval *dii* is changed.

When an operation $u \in T$ is scheduled, a lower bound (*asap*) and upper bound (*alap*) of the feasible schedule range of any unscheduled operation $v \in T$ can be recalculated by using the following assignments:

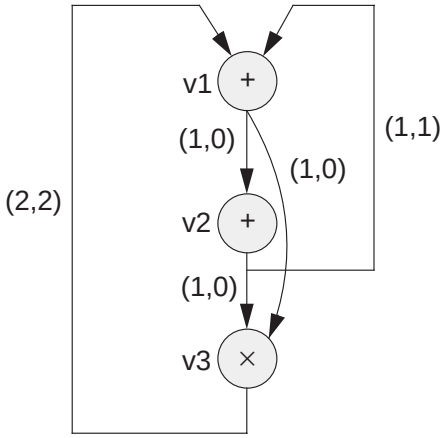
$$asap(v) := \text{MAX}(asap(v) , \varphi(u) + distance(u , v)) \quad (4.5)$$

$$alap(v) := \text{MIN}(alap(v) , \varphi(u) + \delta(u) - distance(v , u)) \quad (4.6)$$

In contrast to the method presented in [Heem90,Heem92], the worst case complexity of updating the feasible schedule ranges using equations (4.5) and (4.6) is $O(|T|)$.

Take for instance the dependence graph of Figure 3.11. The corresponding distance graph (limited to operations v_1, v_2, and v_3), the corresponding initial distance matrix, and a distance matrix after applying an all-pairs longest-path calculation can be found in Figure 4.1. It is assumed that an addition requires 1 cycle step, and a multiplication requires 2 cycle steps to complete execution.

Let *dii* = 3 cycle steps. In that case the distance matrix $D^{(2)}$ equals:



$$D^{(0)} = \begin{bmatrix} 0 & 1 & 1 \\ 1-\text{dii} & 0 & 1 \\ 2-2\cdot\text{dii} & -\infty & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 1 & 2 \\ 1-\text{dii} & 0 & 1 \\ 2-2\cdot\text{dii} & 3-2\cdot\text{dii} & 0 \end{bmatrix}$$

Figure 4.1 Distance graph and distance matrix.

$$D^{(2)} = \begin{bmatrix} 0 & 1 & 2 \\ -2 & 0 & 1 \\ -4 & -3 & 0 \end{bmatrix}$$

Suppose operation v_2 is scheduled in cycle step 2, in other words $\varphi(v_2) = 2$. The feasible schedule range of operation v_1 according to equations (4.5) and (4.6) becomes:

$$\varphi(v_1) \geq \varphi(v_2) + \mathit{distance}(v_2 , v_1) = 2 - 2 = 0$$

$$\varphi(v_1) \leq \varphi(v_2) - \mathit{distance}(v_1 , v_2) = 2 - 1 = 1$$

Hence the feasible schedule range where v_1 can start its execution is [0 , 1]. Using the same method results in schedule range [3 , 5] for operation v_3.

In case a dependence graph doesn't contain any cycles, a simplified all-pairs longest path algorithm for DAGs [Mesm95] can be used to calculate and update the distance information between operations. The algorithm has a worst case complexity of $O(|T| \cdot |F|)$. Updating schedule ranges is done in the same manner as described before, and has a worst case complexity of $O(|T|)$.

4.3 Process invocation constraints

Let t_1 , t_2 , ... , t_r be a path in a dependence graph (T , F). If a path t_1 , t_2 , ... , t_r is extended with edge (t_r , t_1), it becomes a so-called cycle. Operations in the current iteration of a cycle use data generated in previous iterations of the cycle (denoted by delay nodes). Therefore the process can only start a new iteration if this data from previous iterations has been produced. This restriction introduces a lower bound dii_{min} on the distance between successive process invocations $dii(\varphi)$ for each possible schedule φ. In this section several methods to determine the lower bound distance between successive

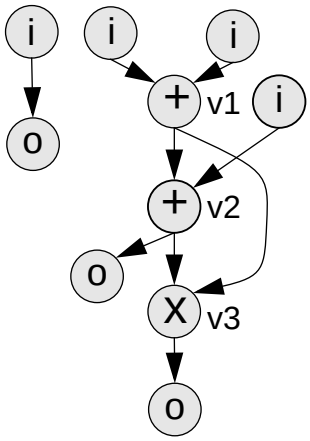


Figure 4.2 Basic block.

process invocations and their impact on the feasible schedule range of operations will be discussed.

4.3.1 Basic blocks

One way to determine a lower bound value for *dii* is to consider the execution of a single process iteration. To obtain a description of a single process iteration, delay nodes can be replaced by a pair of input-output nodes. This results in an acyclic description called a basic block (see Figure 4.2, which represents one single execution of the process given in Figure 3.11).

When considering a basic block, the minimal distance between successive process invocations equals the critical path (which is the longest possible distance from an input node to an output node) of the basic block. The single process invocation method ignores the fact that operations belonging to different iterations can be executed concurrently, and in general will result in a poor quality lower bound on the invocation distance.

Using the basic block method, the minimal process invocation distance dii_{min} of the example in Figure 4.2 equals 4 cycle steps (assuming that an addition requires 1 cycle step, and a multiplication requires 2 cycle steps).

4.3.2 Multiple process invocations

When only one single iteration is considered during the determination of dii_{min}, it is assumed that pipeline stages are emptied at the iteration boundaries of the process (denoted by the place of the delay nodes in the dependence graph). By considering multiple process invocations, hidden concurrency within processes will be unravelled. In other words, operations of iteration i+1 can be scheduled before all operations of iteration i have finished execution (also known as loop pipelining). Unfolding a process by factor k results in a dependence graph, which holds k iterations of the original process.

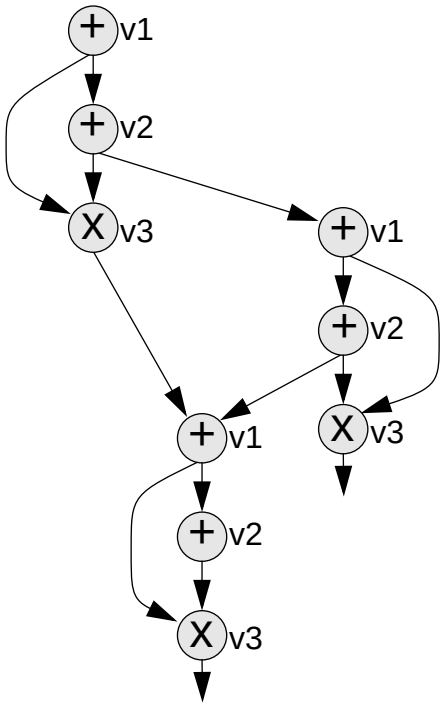


Figure 4.3 Multiple process invocations.

If, in case of the example of Figure 3.11, three process invocations are considered, the critical path length of these 3 process invocations equals 8 cycle steps (see Figure 4.3), which distributed among 3 process iterations leads to an average lower bound of $\lceil 8 / 3 \rceil = 3$ cycle steps. If k process iterations, $k > 1$, are considered, the operations of the critical path of Figure 3.11 will be distributed among the following cycle budget:

$$\left\lceil \frac{2 + 2 \cdot k}{k} \right\rceil = \left\lceil 2 + \frac{2}{k} \right\rceil = 3$$

Hence, using this method the minimal distance between successive process invocations will always lead to 3 cycle steps.

In [Parh91] the concept of perfect-rate programs is introduced. A perfect rate program is a program in which each cycle in a dependence graph contains at most one delay element. It is claimed that perfect rate programs have the property that they can always be scheduled with optimal throughput rate, requiring no retiming or unfolding transformations. Perfect rate programs can be obtained by unfolding a process the least common multiple of delay elements of each cycle in the corresponding dependence graph. A big disadvantage of the unfolding method presented in [Parh91] is that the unfolding factor may grow exponentially in the number of loops. Hence unfolding also complicates the scheduling problem significantly, because the complexity of scheduling in general grows exponentially with the number of operations to be scheduled.

One advantage mentioned in [Parh91] is that the schedules obtained are fully static (for each iteration each operation is bound to the same resource) with respect to the unfolded process, but in fact many results are cyclo-static with respect to a single proc-

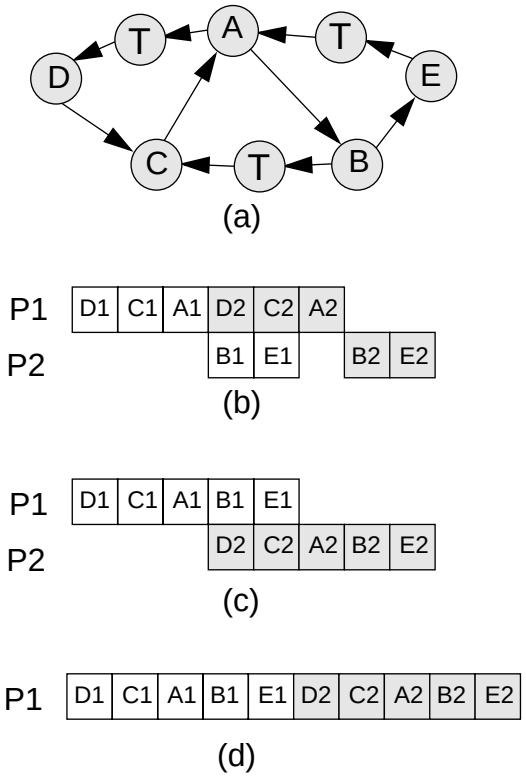


Figure 4.4 Perfect-rate program example.

ess iteration (characterised by a resource displacement for the same operation in different iterations [Schw85]). Whether schedules are fully static depends on the scheduling technique. If for instance greedy scheduling techniques such as list scheduling are used, there is no guarantee that fully-static schedules will result. In Figure 4.4(a) a perfect-rate program from [Parh91] can be found. In Figure 4.4(b) a fully static periodic schedule using 2 functional units with an invocation distance of 3 cycle steps can be found. The schedule shows how two successive schedules of a single perfect-rate process must be combined to result in a fully-static schedule with a minimal distance between process invocations. The way in which fully static and overlapped schedules for a perfect-rate process must be constructed, requires some additional analysis and scheduling technique. In Figure 4.4(c) a greedy schedule for the same example can be found, which is cyclo-static. In Figure 4.4(d) a fully-static schedule resulting from a greedy scheduler can be found. Hence the multiple process invocation method requires some extra analysis and scheduling techniques to obtain fully static schedules with a tight bound on the minimal distance between process invocations [Wang95].

4.3.3 Loop folding and retiming

Another way to determine a lower bound on the distance between successive process invocations can be found in [Goos89], in which a control-flow transformation called loop folding is presented. Loop folding introduces partial overlaps between the execution of successive process invocations in such a way that the critical path length of the process is shortened. This is achieved by transforming the index expressions in the algorithmic behavioural description. Take for instance the algorithmic behaviour of

```
a[B] := b[B-1] + c[C-2];
b[B] := a[B] + input[B];
for i := B to E - 1 do   { B < E }
  a[i+1] := b[i] + c[i-1];       (1)
  b[i+1] := a[i+1] + input[i+1]; (2)
  c[i] := a[i] * b[i];           (3)
endfor;
c[E] := a[E] * b[E];
```

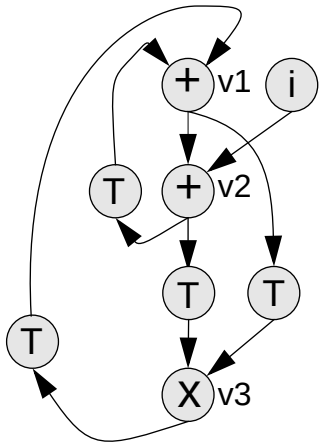


Figure 4.5 Process after retiming.

Figure 4.5, which is a re-write of the algorithmic behaviour which can be found in Figure 3.7. As opposed to the original algorithm in Figure 3.11, there are no intra-iteration dependencies between (1) and (3), and between (2) and (3), and hence the critical path of the process is decreased from 4 to 2 cycle steps.

There are many similarities between the concept of loop folding and retiming [Leis91]. Comparing Figure 3.11 and Figure 4.5 shows that the delay nodes have been moved to other places, but the functional behaviour is equivalent. When a single iteration of the retimed process is considered, a critical path delay of 2 cycle steps can be found.

The disadvantage of retiming (and hence also of loop folding) is that it may place delay nodes in such a way that it might exclude all optimal schedule solutions, and hence reduces the design space of a scheduler inadequately (see Section 5.11 for more details). The second problem of retiming is that it can't handle multi-cycle operations properly (see Figure 4.6). No matter how the delay nodes are shifted, it will still result in a critical path delay of 4 cycle steps, assuming that a multiplication requires 2 cycle steps. A cyclo-static overlapped schedule results in a minimal distance of 3 cycle steps between two process invocations.

4.3.4 Distance relations

The lower bound constraint on the distance between successive process invocations can be derived directly from the distance relations in the following way. For each cycle $t_1 , t_2 , ... , t_r$ the corresponding distance relations are:

$$\varphi(t_2) \geq \varphi(t_1) + \delta(t_1 , t_2) + \lambda(t_1 , t_2) \cdot dii$$

$$\varphi(t_3) \geq \varphi(t_2) + \delta(t_2 , t_3) + \lambda(t_2 , t_3) \cdot dii$$

...

$$\varphi(t_1) \geq \varphi(t_r) + \delta(t_r , t_1) + \lambda(t_r , t_1) \cdot dii$$

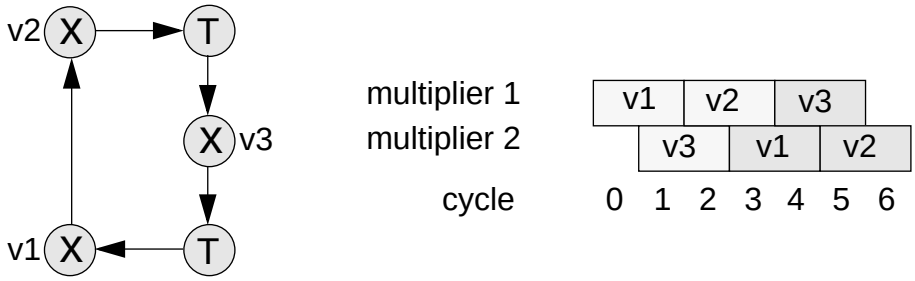


Figure 4.6 Cyclo-static schedule, which cannot be generated by retiming.

Adding all the distance relations of this cycle results in the following equation.

$$(\lambda(t_1 , t_2) + \lambda(t_2 , t_3) + ... + \lambda(t_r , t_1)) \cdot dii \geq \delta(t_1 , t_2) + \delta(t_2 , t_3) + ... + \delta(t_r , t_1)$$

From this equation a lower bound constraint on the process invocations distance $dii(c) \in \mathbb{N}$ for cycle $c \in (T , F)$ can be derived:

$$dii(c) \geq \frac{\delta(t_1, t_2) + \delta(t_2, t_3) + \ldots + \delta(t_r, t_1)}{\lambda(t_1, t_2) + \lambda(t_2, t_3) + \ldots + \lambda(t_r, t_1)} \qquad (4.7)$$

The lower bound constraint on the process invocations distance is determined by taking the maximum value found for the minimum value of *dii* of each cycle:

$$dii_{min} = \mathrm{MAX}_{c \in T} \; dii(c) \qquad (4.8)$$

For Figure 3.11 this method would lead to $dii_{min} = 2$. In the next section a new algorithm will be described, which efficiently calculates the minimal process invocations distance using distance relations.

4.3.5 An algorithm to determine the minimal invocation distance

From Section 4.3 we know that the minimal data introduction interval of a process can be found by dividing the total intra-iteration distances by the total inter-iteration distances for each cycle in the process (see equation (4.7)). The overall minimal data introduction interval can be found by calculating the minimal data introduction interval of each cycle in a process separately, and take the maximum value of these calculations as the minimal process invocations distance for the whole process. Enumerating each cycle of a process can be very complicated [Tarj73]. Therefore, a new method for determining the process invocations distance has been developed.

Let $c = t_1 , t_2 , ... , t_r , t_1$ be a cycle. The current execution of c uses data which is produced in previous executions of c, denoted by inter-iteration dependencies. The total inter-iteration distance $\Lambda(c) = \lambda(t_1 , t_2) + \lambda(t_2 , t_3) + ... + \lambda(t_r , t_1)$ of cycle c multiplied with *dii* denotes the number of cycle steps data may use to traverse through cycle c. The

total intra-iteration distance $\Delta(c) = \delta(t_1 , t_2) + \delta(t_2 , t_3) + ... + \delta(t_r , t_1)$ of cycle c denotes the total amount of cycle steps needed by all operations to process data which traverse the cycle. The slack of cycle c is defined as:

$$slack(c) = (\lambda(t_1 , t_2) + \lambda(t_2 , t_3) + ... + \lambda(t_r , t_1)) \cdot dii - (\delta(t_1 , t_2) + \delta(t_2 , t_3) + ... + \delta(t_r , t_1)) \quad (4.9)$$

For each cycle c, we have $slack(c) \geq 0$, because if $slack(c) < 0$ data will be consumed which hasn't been produced yet. If $slack(c) = 0$, then data produced in a previous process execution is immediately consumed. Hence according to equation (4.7), the resulting dii is the smallest dii possible for cycle c. Hence, the minimal invocation distance dii_{min} is defined as the value of dii with:

1. $\forall_{c \in (T , \prec)} slack(c) \geq 0$, and

2. $\exists_{c \in (T , \prec)} slack(c) = 0$

Hence, searching for a cycle c with the least amount of slack will provide a way to determine dii_{min}. Adding all distance relations of cycle c results in the so-called cycle weight $cw(c)$ of cycle c.

$$cw(c) = \delta(t_1 , t_2) + \delta(t_2 , t_3) + ... + \delta(t_r , t_1) - (\lambda(t_1 , t_2) + \lambda(t_2 , t_3) + ... + \lambda(t_r , t_1)) \cdot dii \quad (4.10)$$

or, in short hand notation,

$$cw(c) = \Delta(c) - \Lambda(c) \cdot dii \quad (4.11)$$

From (4.9) and (4.10) follows that $slack(c) = -cw(c)$.

In each cycle of a dependence graph (T , F), delay nodes are used to describe the inter-iteration boundaries of the corresponding process. Let t_d be a delay node in cycle c. Adding the distance relations from delay node t_d to the same delay node t_d will return the cycle weight $cw(c)$ of c. Let t_p be the predecessor of delay node t_d. In that case:

$cw(c)$

$= \{\text{def. } cw(c)\}$

$\delta(t_d , t_1) + \delta(t_1 , t_2) + ... + \delta(t_p , t_d) - (\lambda(t_d , t_1) + \lambda(t_1 , t_2) + ... + \lambda(t_p , t_d)) \cdot dii$

$= \{\text{def. } distance\}$

$$distance(t_d , t_p) + \delta(t_p , t_d) - \lambda(t_p , t_d) \cdot dii \qquad (4.12)$$

The problem is how to calculate the value for $distance(t_d , t_p)$. If $dii \leq dii_{min}$, then $\exists_{c \in (T , <)}\ slack(c) \leq 0$, hence $\exists_{c \in (T , <)}\ cw(c) \geq 0$, and an all-pairs longest-path does not exist. Let dii_{temp} be defined as the summation of all delays $\delta(t)$ of all operations $t \in T$ of the dependence graph (T , F) plus one:

$$dii_{temp} = \sum_{t \in T} \delta(t) + 1$$

In that case:

$\forall_{c \in (T , <)}\ dii_{temp} > \delta(t_d , t_1) + \delta(t_1 , t_2) + ... + \delta(t_p , t_d)$, hence

$\forall_{c \in (T , <)}\ slack(c) > 0$, hence

$\forall_{c \in (T , <)}\ cw(c) < 0$

Thus by taking $dii = dii_{temp}$, a feasible distance matrix can be calculated, and for each cycle the cycle weight can be determined in constant time by calculating equation (4.12) for each delay node of the graph.

Because $\forall_{c \in (T , <)}\ dii_{temp} > \delta(t_d , t_1) + \delta(t_1 , t_2) + ... + \delta(t_p , t_d)$, or $\forall_{c \in (T , <)}\ dii_{temp} > \Delta(c)$, using equation (4.11) we can derive:

1. $\forall_{c \in (T , <)}\ cw(c)$ div $dii_{temp} = -\Lambda(c)$, and

2. $\forall_{c \in (T , <)}\ cw(c)$ mod $dii_{temp} = \Delta(c)$

Let D be the set of delay nodes. Let $pred(t)$ be the predecessor of delay node t. Using equation (4.7) and (4.8) we can calculate dii_{min} by the Algorithm 4.1.

The complexity of the algorithm is determined by the complexity of calculating the distance matrix, which in case of the Johnson algorithm is $O(|T|^2 \cdot \log|T| + |T| \cdot |F|)$. Because in practical cases the number of input edges for each operation is 2 or less, the complexity will be $O(|T|^2 \cdot \log|T|)$.

In [Lam89,Goos89] list scheduling is applied to a basic block of a process to obtain an upper bound dii_{single} on the distance between two iterations. A lower bound is chosen equal to 0. Within the range specified by $[0 , dii_{single}]$, a binary search method is used to search for the lowest dii in which a schedule is possible. In case of [Lam89], a symbolic distance matrix is used in such a way that updating the distance matrix for another dii can be done within $O(|T|^2)$ instead of re-applying the all-pairs longest-path algorithm each time. Hence the worst case complexity is $O(|T| \cdot |F| + |T|^2 \cdot \log|T| + |T|^2 \cdot \log(dii_{single}))$.

Algorithm 4.1 (Calculate dii_{min}).

```
dii = 0; diimin = 0;
forall t ∈ T
  dii = dii + delay(t);
  diimin = MAX(delay(t),diimin);
endfor;
D = APLP(dii,(T,F));                          // all-pairs longest-path
forall t ∈ DelayNodes
  cw = distance(t,pred(t)) + δ(pred(t),t) - λ(pred(t),t) * dii;
  lambda = cw div dii;
  delta = cw mod dii;
  diimin = MAX(diimin,⌈delta / lambda⌉);
endfor;
```

In [Gere92] an $O(|D|^4 + |D| \cdot |F|)$ algorithm is presented, with D the set of delay nodes of a dependence graph. The method is based on construction of a longest path matrix L with size $|D| \times |D|$, in which $L(i\ ,\ j)$ denotes the length of the longest path from delay element d_i to delay element d_j. Let L^k be a $|D| \times |D|$ matrix, in which $L^k(i\ ,\ j)$ denotes the longest path distance between delay element d_i and delay element d_j which passes through exactly k -1 delay elements. Matrix L is obtained by longest-path matrix multiplication [Corm90] using the following recursive rule:

$$L^{k+1} = L^1 \cdot L^k$$

Matrix L^1 can be obtained by calculation of the longest path from each delay node to all other delay nodes. The time complexity to compute L^1 is $O(|D| \cdot |F|)$ [Heem90]. Calculation of $L^{|D|}$ requires $|D|$ matrix multiplications, which results in a complexity of $O(|D|^4)$. Hence the total complexity of this algorithm is $O(|D|^4 + |D| \cdot |\prec|)$.

Let the pairwise distances between delay nodes be represented by a graph $G(D\ ,\ E_D)$. Application of the cycle-mean algorithm from [Karp78] gives the lower bound of the distance between process invocations in $O(|D| \cdot |E_D|)$, which has the lowest worst-case complexity of all methods known [Ito94].

Though the complexity of Algorithm 4.1 is in general higher than the complexity of the methods presented in [Heem92] and [Karp78], the application of Algorithm 4.1 in combination with the (obligatory) calculation of a distance matrix for other values of the process invocation distance is very efficient in practice.

4.4 Time constraints

The second type of constraints which play an important role in high-level synthesis are time constraints. A time constraint $t_c(u\ ,\ v)$ between two operations $u,v \in T$ denotes the maximal distance between the start cycle step of operation u and operation v, in other words

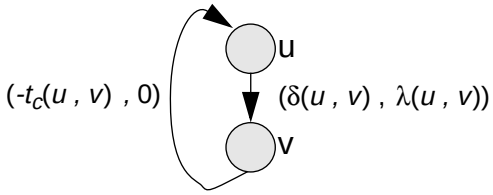


Figure 4.7 Time constraint in distance graph.

$$\varphi(v) \leq \varphi(u) + t_c(u , v)$$

or

$$\varphi(u) \geq \varphi(v) - t_c(u , v) \qquad (4.13)$$

Equation (4.13) has a similar form as equation (3.3), and can be depicted in a distance graph in a similar way by adding an edge (v , u) with a tuple (-$t_c(u , v)$, 0) (see Figure 4.7).

A time constraint may introduce a cycle c_{tc} in a dependence graph, which on its turn introduces some limitations with respect to the values of the time constraint. Let *distance*(u ,v) be the longest path distance from u to v. Just like in Section 4.3.5, a cycle weight can be defined as:

$$cw(c_{tc}) = distance(u , v) - t_c(u , v) \qquad (4.14)$$

An all-pairs longest-distance algorithm only produces feasible results if no positive weight cycle exists. This means that $t_c(u , v) \geq distance(u , v)$, in other words, a time constraint between u and v cannot constrain in such a way that v starts its execution and consumes data before u has finished its execution and produced this data.

The most common time constraint found in high-level synthesis is the global time constraint $T_{max} \in \mathbb{N}$ (also called schedule length, makespan or latency). A global time constraint denotes how many cycles it takes to process all input data into output data. For each input operation $i \in T$ and output operation $o \in T$, the following relation is true:

$$\varphi(o) \leq \varphi(i) + T_{max}$$

This relation results in a distance graph in which an edge is added between every input and output operation. A global time constraint can be represented more efficiently by slightly modifying a distance graph. Two dummy operations s and t, respectively called source and sink, are added to T. For each input operation $i \in T$, an edge (s , i) is added, labelled with a tuple (0 , 0). For each output operation $o \in T$, an edge (o , t) is added,

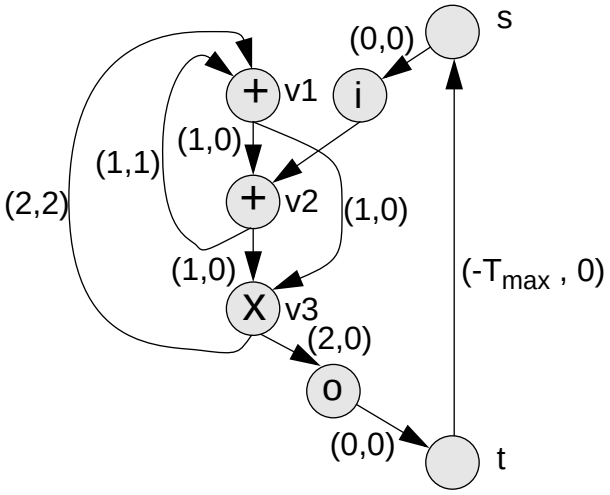


Figure 4.8 Distance graph with time constraint.

also labelled with a tuple (0 , 0). Finally, an edge (*t* , *s*) is added with a tuple (-T_{max} , 0) (see also Figure 4.8).

Care must be taken not to introduce positive weight cycles, hence *distance*(*s* , *t*) - T_{max} ≥ 0, or T_{max} ≥ *distance*(*s* , *t*). This implies that the global time constraint should be equal to or larger than the critical path from *s* to *t* inside the dependence graph.

One of the advantages of adding an edge (*t* , *s*) to a dependence graph is that one strongly connected component consisting of all operations in the process is obtained. By selecting operation *s* as a reference operation (in other words φ(*s*) = 0), the feasible schedule ranges of operations obtained by an all-pairs longest-path algorithm and equations (4.5) and (4.6) are always relative to the inputs of the process.

When an operation $u \in T$ is scheduled, the feasible schedule range of any unscheduled operation $v \in T$ can be recalculated using equations (4.5) and (4.6), which has a worst case complexity of $O(|T|)$.

A partially defined schedule is a schedule φ in which for some operations $u \in T$ the schedule φ(*u*) has been determined. Suppose that the start cycle step of operation $u \in T$ has been pre-determined, in other words $\varphi(u) = c_u$. In that case $\varphi(u) - \varphi(s) = c_u$, which can be modelled by the following inequalities:

$\varphi(u) - \varphi(s) \leq c_u$, hence $\varphi(s) \geq \varphi(u) - c_u$

and

$\varphi(u) - \varphi(s) \geq c_u$, hence $\varphi(u) \geq \varphi(s) + c_u$

These relations can be modelled in a distance graph as can be found in Figure 4.9.

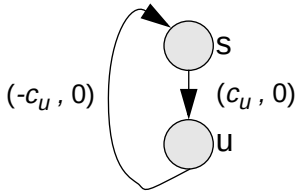


Figure 4.9 Modelling pre-scheduled operations.

Checking whether time constraints or pre-scheduled operations result into infeasible schedule ranges of operations, can be done by checking whether an all-pairs longest-path algorithm results in a distance matrix D for which $\exists_{i \in [0\ ,\ |T|\ -\ 1]}\ D(i\ ,\ i) \neq 0$ [Corm90]. To see if a single time constraint or pre-scheduled operation is the cause of infeasibility, an all-pairs longest-path can be performed without this time constraint or pre-scheduled operation. In case a combination of conflicting time constraints and pre-scheduled operations cause infeasibility, it is very difficult to find unambiguously which constraint(s) is (are) causing the trouble.

4.5 Resource constraints

A resource constraint imposes an upper bound on the resource allocation that can be used during scheduling. The most obvious resource constraint used in high-level synthesis is an upper bound on the number of functional units (also called modules). This restricts the number of operations, requiring the same operation type, that can be scheduled simultaneously. In the field of high-level synthesis, scheduling methods such as list scheduling [Thom90] have been used quite successfully to deal with constraints on the number of functional units.

Only very little is known of schedulers which can cope with memory allocation constraints and interconnect allocation constraints during scheduling. One method, called cut-reduction, is presented in [Depu93]. Cut-reduction adds edges to a dependence graph to lower the number of possible simultaneously data transfers, but due to the application of branch and bound algorithms the run-time efficiency becomes bad for large examples. To be able to handle large size instances, a hierarchical scheduling method called clustering is introduced. Clustering is a method which schedules parts of a process hierarchically, in order to obtain a new smaller process. Cut-reduction is applied to this smaller process, which compared to the original problem is more run-time efficient. This basic-block like scheduling method might reduce the search space in such a way that it may exclude the optimal solution.

Most other methods used in high-level synthesis try to optimize memory allocation or interconnect allocation during or after scheduling. Methods optimizing memory allocation during scheduling can be found in [Paul89] or [Verh91], which try to balance the use of registers over time by using forces. In [Hwan91] the sum of lifetimes is minimized during ILP scheduling. Finally, heuristic techniques are used in [Romp92] to schedule production and consumption of values as close as possible.

Similar observations can be made for interconnect allocation. Whereas there are many articles reporting interconnect optimization (see for instance [Weng91]), only a few methods deal with constraints directly (for example [Woer94], [Hart92]). Also, many methods only use the number of multiplexers and buses resulting from a schedule, and don't deal with layout specific information (placement and routing). Some initial work on this topic can be found in [Timm95b], [Jaco95], [Jang93], and [Weng91].

4.6 The relation between time and resource constraints

In [Timm93] the close relationship between resource constraints and time constraints is explained. The essence is that for a time constrained scheduling problem an accurate lower bound resource allocation can be found efficiently, by a technique called module selection. On the other hand, for a resource constrained scheduling problem an accurate lower bound estimation for the completion time can be found by a technique called module execution interval analysis. In both cases the initial constraints can be extended such that both a time constraint and a resource constraint result. Hence in both cases the original scheduling problem is transformed into a feasibility scheduling problem.

The approach in [Timm93a] tries to find an accurate lower bound estimation of functional area by investigating the structure of a data-flow graph. It is based upon a relaxation of the dependence constraints, in other words, the method guarantees that each operation can be scheduled within its initial schedule range. This is achieved by generating MILP constraints, which try to enforce the selection of sufficient module capacity to perform all operations within their initial schedule range. The effect of the schedule of an operation upon the feasible schedule range of other operations is disregarded, hence the resulting constraints might not lead to a feasible schedule. The number of integer variables in the method depends on the size of the module type library used. If only simple libraries are used (in other words, each operation type can only be implemented by a single module type), the module selection problem can be solved efficiently using a polynomial time algorithm based on the methodology presented in [Timm93].

To find a lower bound on the global completion time, the initial feasible schedule intervals of operations can be reduced under the influence of resource constraints in such a way that it doesn't limit the solution space [Timm93b]. The method, called execution interval analysis, globally works as follows.

Let $m \in L$ be a module type. Let $T_L(m)$ be a list of operations sorted by increasing asap-value, for which for each operation $t \in T$, we have $t \in T_L(m) \Leftrightarrow \xi(t) = m$. Let $T_L(m\ ,\ i)$ be the i^{th} operation from $T_L(m)$. Let $K(m)$ be the number of modules of type $m \in L$ given by the resource constraint. The module schedule range *MEI* is the range in which some module of type m must execute some operation. Module schedule ranges can be calculated using Algorithm 4.2. The end of each module range can be determined in a similar way.

Algorithm 4.2 (Module Execution Intervals for module type *m*).

```
for i = 1 to K(m) do
  start(MEI(i)) = asap(T_L(m,i));
endfor;
for i = K(m) + 1 to |T_L(m)| do
  start(MEI(i)) = MAX{⌈asap(T_L(m,i))⌉,
                      start(MEI(i - K(m))) + d(T_L(m,i),m)}
endfor;
```

In the next phase, a bipartite graph $G(m)$ is constructed. The vertices of the bipartite graph consist of the operations from T_L and the set $R(m) = \{MEI(i) \mid 1 \leq i \leq |T_L(m)|\}$. There is an edge (a , b), $a \in T_L$ and $b \in R(m)$, if and only if the feasible schedule range of a has an overlap with the module schedule range b, and which is at least as large as the corresponding execution delay. For each feasible schedule, a corresponding complete matching exists. Edges which can never be part of a complete matching, can be deleted from $G(m)$ without excluding any feasible schedule from the search space. Deletion of edges can be used to tighten the feasible schedule range of operations incident to these edges.

During determination of the module ranges, the number of cycle steps in which a module can start its execution can be a negative number. In this case a complete matching does not exist, and no feasible schedule is possible within the time constraint, and hence the estimation on the lower bound completion time or resource allocation must be increased.

The total complexity of module execution interval analysis is $O(|T|^2 + |T| \cdot |L|)$. The number of bipartite graphs equals $O(|L|)$.

In case a resource constraint is imposed together with a time constraint, the method presented in [Timm93b] can be used to determine whether the constraint set might introduce an infeasible schedule. In that case either the resource constraint or the time constraint needs to be adjusted.

By using estimation techniques, the constraints of the original scheduling problem can be used as a performance measure and vice versa. Some possible scheduling strategies are given by the scheduling template in Figure 4.10.

A time constrained scheduling problem can for instance be solved by a time constrained scheduling method directly (which may use a lower bound resource allocation estimation to tighten the schedule constraints). It can also be solved by using a resource constrained scheduler which tries to minimize the completion time of a schedule, by use of a lower bound resource allocation estimation. If the completion time exceeds the original time constraints, supplementary resources need to be allocated to come up with a feasible schedule. Finally, a feasibility scheduler can be applied which tries to satisfy both the time constraint and resource constraint directly. If the method fails to do so,

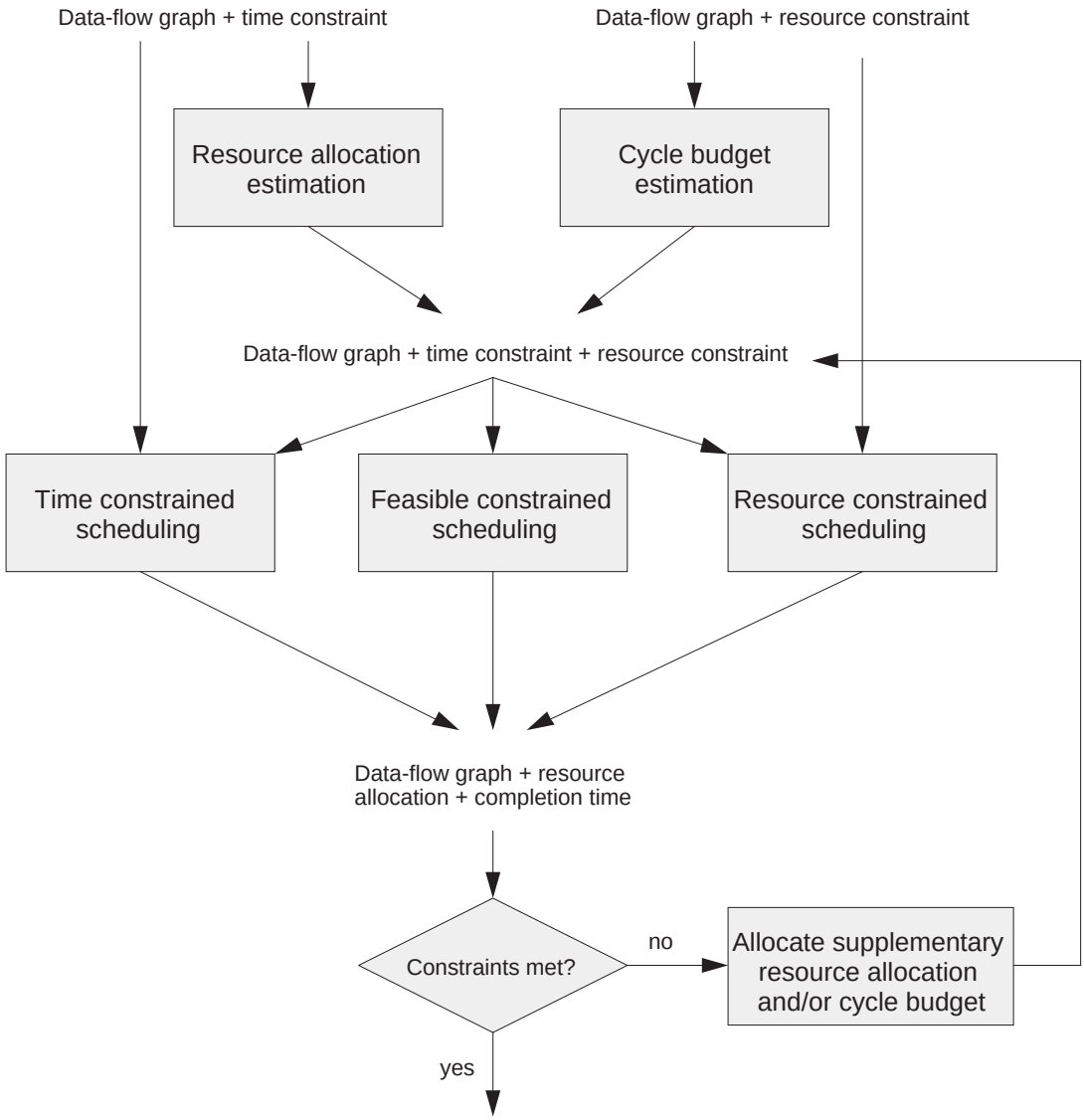


Figure 4.10 Scheduling template.

supplementary resources need to be allocated to come up with a feasible schedule. A similar strategy can be applied for resource constrained scheduling problems.

4.7 Conclusions

In this chapter several methods have been presented to determine the feasible schedule range of operations. The central idea is to use distance relations from a distance graph, to determine the feasible schedule range of operations with respect to other operations. Time constraints can be incorporated very easily by adding distance relations to a distance graph obtained from a data-flow graph. All-pairs longest-path algorithms are used to construct a distance matrix from a distance graph, which is used to determine and to update feasible schedule ranges of operations very efficiently. Finally, a short introduction shows how resource constraints can be used to tighten the feasible schedule ranges of operations, by using module execution interval analysis.

The use of all-pairs longest-path algorithms results in a new efficient algorithm to determine the minimal process invocation distance. In contrast to methods such as folding, unfolding, and retiming, this new method doesn't need to transform the original process description. Besides, the loss of the optimum solution from the search space is avoided.

All of this has resulted in a uniform model, in which dependence relations, time constraints, and resource constraints are used to decrease the size of the schedule range of operations, without excluding any optimal solution. Additionally, estimation techniques result in a schedule scheme in which constraints can be exchanged, resulting in different scheduling strategies. In the following two chapters a new scheduling strategy to deal with both resource constrained and time constrained scheduling problems will be presented.

Chapter 5 Constructive Scheduling

5.1 Introduction

There are many different ways to solve a scheduling problem under given constraints. In this chapter a classification of scheduling *methods* will be presented, to gain some insight in the applicability of certain scheduling algorithms in particular situations.

A new constructive scheduling algorithm will be presented, which given a permutation of operations, will construct a resource constrained schedule in a topological manner. This new algorithm will serve as a scheduling engine for a search procedure based on genetic algorithms, presented in the next chapter.

This chapter concludes with a discussion about how permutations of operations can be used to solve (loop) pipelined scheduling problems.

5.2 High-level synthesis scheduling complexity

Let (F , c) be an instance of a combinatorial optimization problem. Most real-life synthesis problems will have instances with a very large number of candidate solutions $|F|$, hence listing all candidate solutions and calculating the value of cost function c for each solution, will in general lead to inefficient use of computer resources (in other words memory and CPU-time). The mathematical background presented in [Gare79] provides us a way to classify optimization problems. This has resulted in the notion of NP-hard problems, for which no polynomial time algorithms have been found so-far to solve each instance to optimality (and are believed not to exist). In general, solving a NP-hard problem requires an exponential amount of CPU-time.

A straightforward classification of high-level synthesis scheduling problems would be a classification that tells whether a scheduling problem belongs to the class of NP-hard problems. Typical classifications such as in [Blaz94,Lens85,Coff76,Gonz77] classify scheduling problems based on properties such as dependence relations (independent, tree, forest, DAG, graph), number of processors (single, parallel), kind of processors (identical, uniform, unrelated), processing modes (one-by-one, one-to-many [Timm93a], flow-shop, open-shop, job-shop), execution mode (pre-emptive, non-preemptive), and more. Most of these issues don’t play a leading role inside high-level synthesis, because the behavioural description in general leads to a process which is a graph, the number of processors is larger than one, the processing mode is one-to-one (i.e. an operation can be implemented on one module type) or one-to-many

(i.e. an operation can be implemented in various module types), and the execution mode is non-preemptive.

In [McFa90] a classification of scheduling problems based upon time-constraints and resource-constraints is made. In a time constrained scheduling problem a global time constraint is imposed, and the aim is to find a schedule inducing a minimal resource allocation. In a resource constrained scheduling problem a resource constraint is imposed, and the aim is to minimize the global completion time induced by the schedule. These high-level synthesis scheduling problems are proven to be NP-hard (see [Verh91] for time constrained scheduling, and [Heem90] for resource constrained scheduling). As shown in Section 4.6, time constraints and resource constraints are tightly inter-related. It is therefore questionable whether a "general" classification of high-level scheduling problems should be based on such criteria. A more useful classification for high-level synthesis problems would be a classification based on cyclic versus acyclic data-flow graphs. Take for instance register allocation, which is equal to finding a minimal colouring of an interval graph [Golu80], and can be solved very efficiently for the acyclic case by the left edge algorithm [Kurd87]. This becomes an NP-hard problem in case the process becomes cyclic, which is equal to finding a minimal colouring of an cyclic arc graph [Golu80]. Another example is the all-pairs longest-path problem. In the previous chapter an all-pairs longest-path algorithm for cyclic graphs has been presented, which has a worst-case complexity of $O(|T| \cdot |F| + |T|^2 \cdot \log|T|)$, whereas in [Mesm95] an algorithm is presented which only takes $O(|T| \cdot |F|)$ for acyclic graphs.

Only a very small set of problems found in high-level synthesis are proven to be polynomially solvable problems. Some examples are the ASAP/ALAP scheduling problem [DeWi85], the interval graph colouring problem [Kurd87], and the retiming problem [Leis91]. A classification of typical high-level synthesis scheduling problems based on complexity issues doesn't provide any useful insights. In this chapter an alternative classification based on constructive scheduling *methods* will be presented, which is based on the way a schedule is constructed from a permutation of operations. It will be shown that for some classes of schedulers, certain decisions might impose severe constraints to operations during scheduling, implying equally severe effects on the optimality or feasibility of the solution.

5.3 Optimality

In theory the goal of a scheduling algorithm is to find an optimal schedule, in other words it should return the best feasible schedule possible with respect to the performance measures. As one of the main objectives of using algorithms to solve optimization problems is to solve them efficiently, an algorithm which always returns an optimal solution for an NP-hard problem is regarded as inefficient. Therefore some trade-offs between CPU-time and the accuracy of the solution must be considered.

A straightforward classification would be to classify high-level synthesis scheduling algorithms based on the accuracy of the solution found:

1. Exact algorithms. These kind of algorithms always find the optimal solution to the scheduling problem imposed. Examples of such algorithms can be found in [Hwan91,Gebo92,Lee89]. The main disadvantage of applying these methods to real-world scheduling problems is that they might end up in performing an exhaustive search, which results in the use of excessive amounts of computer resources (CPU-time or memory). The current status of these algorithms is that they can only be applied successfully to instances with a small input size, and their use in practical situations is questionable.

2. Approximation algorithms. These kind of algorithms find a solution, for which the difference in cost with respect to the cost of an optimal solution is independent of the input size of the problem.

3. Heuristics. These kind of algorithms always find a solution. There are no guarantees given about the quality of the solution generated. Many kinds of heuristic high-level synthesis schedulers have been reported, such as list scheduling [Girc84], force directed scheduling [Paul89], and critical path scheduling [Park86]. The main advantage is that, in general, these algorithms generate solutions relatively fast compared to approximation or exact algorithms.

In the previous chapter some methods, which derive supplementary constraints from existing scheduling constraints, have been presented to reduce the set of candidate solutions of the scheduling problem without excluding an optimal solution. This increases the possibility that heuristics find a good quality solution, or the chance that approximation algorithms and exact algorithms need less computer resources.

5.4 Construction of schedules

A classification based on the way schedules are created, is the following:

1. Constructive. The first class of schedulers, called constructive schedulers, schedule operations one by one. The different constructive scheduling techniques can be distinguished by the order in which and the cycle step where they schedule operations. The intermediate solutions produced during the construction of the schedule consist of partially specified schedules. More about constructive scheduling can be found from Section 5.6 to Section 5.11.

2. Iterating. The second class of schedulers, called iterating schedulers, take an existing schedule as their input, and try to improve the schedule by altering the schedule. Examples of these type of schedulers are percolation scheduling [Pota90], move-scheduling methods based on annealing such as in [Nest90,Deva89], and

multiple-exchange pair selection as in [Park91]. The intermediate solutions consist of fully specified schedules.

The search method, which is used to find a schedule, can be roughly divided in greedy search techniques, local search techniques, and enumeration techniques.

1. Greedy search techniques. Greedy algorithms always make a choice that looks best at a specific moment. It makes such a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Greedy algorithms work well for some problems, such as for instance for the ASAP/ALAP scheduling problem [DeWi85], the colouring of interval graphs [Kurd87], and the retiming problem [Leis91], for which greedy algorithms exist which always return optimal solutions. In the case greedy algorithms are not guaranteed to return optimal solutions, this is because a trade-off between the quality of the solution obtained and the amount CPU-time required to generate a solution is made. Examples are for instance list scheduling [Girc84], force directed scheduling [Paul89], and critical path scheduling [Park86].

2. Local search techniques. Let (F , c) be an instance of a combinatorial optimization problem. Local search is based on the existence of a so-called neighbourhood space N: $F \rightarrow 2^F$, with $N(t)$ a set of solutions, which in some sense are 'close' to $t \in F$. Element $j \in N(t)$ is called a neighbour of t. Let $c(t)$ be the cost of t. A local minimum s with respect to the neighbourhood space $N(t)$ is defined as $s \in N$(t) with $\forall_{x \in N(t)}\, c(s) \leq c(x)$ (see [Papa82] for more details). Examples of scheduling techniques based on local search can be found in [Nest90,Deva89] (which use simulated annealing), and [Park91] (which uses the k-change neighbourhood as defined in [Lin73]).

3. Enumeration techniques. Enumerative algorithms enumerate all feasible solutions, and return the best solution found. Most real-life problems will have instances with a very large number of feasible solutions $|F|$, hence enumerating all feasible solutions and evaluating the cost function c for each feasible solution will lead to inefficient use of computer resources. Explicit enumeration techniques, such as branch and bound or dynamic programming, can be used to reduce as drastically as possible the set of solutions that need to be enumerated. For scheduling examples see for instance [Park86a], [Davio79], and [Fabe94]. Implicit enumeration techniques solve a set of equations, derived from an IP-formulation of the problem, by application of algorithms such as the simplex algorithm (in combination with a branch and bound algorithm to obtain integer solutions). Some examples of enumerative scheduling techniques are IP scheduling [Lee89, Hwan91,Gebo92], and non-linear programming gradient methods [Shin89]. CPU-time remains one of the biggest concerns based on methods using enumerating techniques.

5.5 Search space versus candidate solutions

A scheduling problem is an instance of a combinatorial optimization problem (F , c), with F the set of candidate solutions (feasible schedules), and c the cost function. In Section 3.5 some common high-level synthesis scheduling problems have been presented, such as time constrained scheduling, resource constrained scheduling, and time and resource constrained scheduling.

Let $F_{opt} \subseteq F$ be the set of optimal solutions from F. In Chapter 4 some algorithms have been presented which decrease the feasible schedule range of operations, based on precedence, time, throughput, and resource constraints. These algorithms decrease the size of the set of candidate solutions F in such a way that no optimal solution from F_{opt} is excluded, resulting in a new set of candidate solutions F' with $|F'_{opt}| / |F'| \geq |F_{opt}| / |F|$.

In the remaining sections of this chapter, attention will be paid to constructive scheduling techniques. The main goal is to gain insight in the applicability of these scheduling techniques in combination with certain scheduling constraints.

The task of a constructive scheduling algorithm, denoted by `schedule`, is to generate a feasible solution $f \in F$. The domain of a scheduling algorithm is called the search space S, hence `schedule`: $S \rightarrow F$. Constructive scheduling techniques are characterized by the fact that they schedule operations one by one. Hence, in case of a constructive scheduling algorithm, an element $s \in S$ should determine the order in which and the cycle step where operations are scheduled.

The remaining sections of this chapter will focus on the order in which operations are scheduled, which leads to a classification of constructive scheduling methods.The main goal of this classification is to gain insight in the applicability of these scheduling techniques in combination with particular scheduling constraints. In case of scheduling acyclic graphs, the classification will show a clear advantage for a particular scheduling technique, called constructive topological scheduling, which will be presented in Section 5.6. In Section 5.10 the consequence for creating pipelined schedules will be shown. In Section 5.11 it will be shown that constructing schedules for cyclic graphs is a more complex problem than constructing schedules for acyclic graphs.

5.6 Permutation scheduling

Constructive schedulers assign operations to cycle steps one by one. The resulting schedule depends on (1) the order in which operations are scheduled, and (2) the cycle steps where operations are scheduled.

The order in which operations of a data-flow graph (V , E) are scheduled can be determined by a permutation Π consisting of the operations of V. When the order in which operations are scheduled by a scheduling algorithm is equal to the order specified by

the permutation, the scheduling algorithm is called a strict permutation scheduling algorithm (see Section 5.7 for more details). If the order in which operations are scheduled is determined by the permutation in combination with the partial order induced by the data-flow graph, the scheduling algorithm is called a topological permutation scheduling algorithm (see Section 5.8 for more details).[1]

The cycle step selection procedure determines for each operation a cycle step in which it is scheduled. The cycle step selection procedure is driven by the permutation, in other words, it accepts operations in a particular order, and for each operation separately it determines the cycle step in which it should be scheduled. The quality of a schedule depends on the way the cycle step selection procedure responds to Π. This means that there should be a close relation between the order, given by Π, and the selection strategy, depending on the constraints and goals imposed on the scheduling problem. Schedule specific information can be used to predict in which cycle steps particular operations should be scheduled. When dealing with constructive scheduling methods, schedule decisions can be derived from partial schedule information, schedule constraints, and schedule goals. This implies that the selection procedure has a local scope, and hence the effects of a particular selection strategy are difficult to foresee. In case of NP-hard scheduling problems, no selection strategy is known which in general will lead to global optimal solutions.

The determination of the order of operations in a permutation is performed by a so-called permutation generator. In some cases the order of operations of the permutation depends on partial schedule information, which for instance is the case with force directed scheduling (see Section 5.7.2) or module execution interval analysis (see Section 5.7.4). In such a case the generation of the permutation can be integrated inside the scheduling construction algorithm to obtain a more efficient scheduling algorithm. In other cases the order of operations inside the permutation is specified before the schedule is constructed (such as for instance the priority list used in combination with list scheduling - see Section 5.8.3). In both cases the order of scheduling operations is characterized by a permutation Π. In the remaining sections of this chapter an overview will be given of how the order of scheduling affects the schedule range of operations, and how the order of scheduling may restrict the possibility that a sequence of local schedule decisions may result in a feasible or optimal schedule.

5.7 Strict permutation scheduling

Constructive schedulers assign operations to cycle steps one by one. Obviously the way a constructive schedule is generated depends on the order in which operations are scheduled. This order of operations can be represented by a permutation Π of operations.

1. Permutation scheduling as defined in this thesis should not be confused with permutation scheduling as defined for flow-shop scheduling [Pine95].

In this section the relation between permutation Π and the schedule that is generated from Π will be investigated. In strict permutation scheduling Π specifies the exact order in which operations are scheduled. Let Π(*i*) denote the i^{th} operation in permutation Π. A general method to generate strict permutation schedules can be found in Algorithm 5.1.

Algorithm 5.1 (Permutation scheduling template).

```
for i = 0 to |Π|-1 do
  v = Π(i);                      // select operation in order of Π
  φ(v) = Select(T_min,T_max);    // select cycle step
endfor;
```

For each operation *v*, selected according to the order specified by Π, the procedure `Select` determines the cycle step in which *v* is scheduled. The selection strategy of procedure `Select` may depend on the properties of operation *v*, characterized by the schedule constraints, the schedule goals imposed, and the partial schedule results achieved so far. To obtain feasible schedules, the selection procedure can only select a cycle step from the schedule range of operations, as described in Chapter 4. If operations are scheduled outside this range of cycles, infeasible schedules result. Equations (4.5) and (4.6) imply that the order in which operations have been scheduled may affect the schedule range of unscheduled operations, and may introduce new constraints during scheduling, as will be explained in the remaining part of this section. Therefore, the range of cycles in which an operation can be scheduled will always be explicitly shown inside the argument list of procedure `Select`.

The search space of Algorithm 5.1 is composed of $\Pi \times T_{max}$, in which T_{max} is an upper bound on the schedule range of operations. The main problem of Algorithm 5.1 is that the design space contains many infeasible solutions, and searching for a schedule using Algorithm 5.1 could result in the evaluation of many infeasible solutions, which will be the main topic of this section.

5.7.1 Precedence constraint satisfaction

When precedence constraints are imposed (which, of course, is always the case), operations should be scheduled according to the order specified by Π, but in such a way that precedence constraints are always satisfied. This means that the cycle step in which an operation *v* can start its schedule, is bounded by the schedule of predecessor operations (denoted by the asap-value) and successor operations (denoted by the alap-value). Hence, scheduling a particular operation may impose (time) constraints on other operations, and may force these operations to be scheduled implicitly. This phenomenon invalidates the position of these operations in Π, in other words, operations are scheduled before they are considered for scheduling according to the order specified by Π.

To guarantee that the schedule ranges of operations are feasible at any time, the schedule ranges of unscheduled operations need to be updated each time after scheduling a particular operation (see also Section 4.2). In that case, operations can always be scheduled in their schedule range, as shown in Algorithm 5.2.

Algorithm 5.2 (Precedence constrained permutation scheduling).

```
for i = 0 to |Π|-1 do
  v = Π(i);                          // select operation
  φ(v) = Select[asap(v),alap(v)];    // select cycle step from feasible range
  update schedule ranges;
endfor;
```

The selection strategy mainly depends on the performance measures to be optimized. If for instance the completion time of the schedule must be optimized, the procedure `Select` will always return the earliest cycle step possible, which equals the asap-value of operations. In that case Algorithm 5.2 results in an ordinary asap-schedule algorithm. In high-level synthesis strategies, asap-scheduling is often used to determine the schedule ranges of operations, serving as an initialisation for other scheduling algorithms.

If the resource allocation of a schedule must be optimized, the procedure `Select` should avoid the allocation of unnecessary resources. Operations are scheduled according to the order induced by the permutation, but as has been explained before, other operations may become scheduled before they are considered according to the order specified in the permutation. In general this means that there should be a relation between the permutation Π and the `Select` procedure for such an algorithm to become successful. It is however very difficult to foresee the global effect on the final resource allocation of scheduling a particular operation locally (i.e. whether a local decision results in a global optimal solution). In Section 5.8 a different precedence constrained method is presented, which will be able to generate a minimal resource allocation more efficiently.

5.7.2 Time constraint satisfaction

If both precedence constraints and (global) time constraints are imposed, additional upper bounds can be introduced, denoted by the alap-value of an operation. This is identical to the situation in which operations obtain an upper-bound constraint with precedence constrained scheduling as discussed before, and hence Algorithm 5.2 is applicable in this case too.

Optimizing the completion time is just as trivial as in the case of precedence constrained scheduling, and also results in an asap-scheduler.

A more useful application of time constrained scheduling is in combination with an attempt to minimize the resource allocation induced by the resulting schedule. Just like with precedence constrained scheduling, care must be taken not to schedule operations in such a way that they fix the schedule of other operations, inducing an unnecessary allocation of resources. Again it can be concluded that this means that there should be a relation between permutation Π and the `Select` procedure for such an algorithm to become successful.

A well-known high-level synthesis scheduling method, called critical path scheduling [Park86], is a method which has a greedy selection strategy with respect to both resources and time. The validation given for critical path scheduling is based on the assumption that operations on the critical path have less freedom to be allocated on a certain hardware module, and should therefore be considered for scheduling first. First, functional units are allocated and bound to operations on the critical path in a first-come first-serve way. If possible, hardware is re-used to prevent allocation of superfluous hardware. Then, off-critical path operations v are assigned to hardware, based on the mobility $m(v) = alap(v) - asap(v) - \delta(v)$ of operation v. The off-critical path operation v with the smallest mobility $m(v)$ is chosen for scheduling in the first feasible cycle step where it can be scheduled without resource conflicts. The idea behind this choice is that deferring the operation with the smallest mobility, has the largest probability of increasing the length of the critical path. If there is not enough hardware to schedule a particular operation, it is scheduled in the first cycle step from its schedule range, and additional resources are allocated. If necessary, additional cycle steps are added, depending on the constraints, and the procedure is repeated again. The method doesn't explicitly specifies a permutation Π before scheduling, but derives the permutation during scheduling.

A method called force-directed scheduling, which has a more global scope with respect to selecting a cycle step to schedule an operation, is reported in [Paul89]. Force-directed scheduling tries to balance the operations in such a way that the resource utilization is distributed equally over the available cycle steps. A statistical measure of the resource utilization of partial schedules is obtained by assuming that the probability an operation is scheduled somewhere inside its feasible schedule range, is uniformly distributed inside its interval. A probabilistic distribution function can be defined as the summation of these probabilities, and gives statistical information about the concurrency of a particular module type of a partial schedule. Scheduling a particular operation in a cycle step may have an impact on the schedule range of other operations, and hence may change the value of the probabilistic distribution functions. Force directed scheduling tries to equalize the value of the probabilistic distribution function for each cycle step. This is achieved by investigating the effect of attempted cycle step assignments of operations in their feasible schedule range on the probabilistic distribution function, which together with the module area induced, results in the so-called force. The force directed scheduling algorithm is given in Algorithm 5.3.

Algorithm 5.3 (Force Directed Scheduling).

```
i = 0;
calculate Schedule Ranges;
calculate Distribution Functions;
while (unscheduled operations) do
  Calculate Forces;
  v, t = Select Operation and Cycle Step with Lowest Force;
                                    // t ∈ [asap(v),alap(v)]
  Π(i++) = v;                       // Dynamic determination of permutation
  φ(v) = t;                        // Cycle step with Lowest Force;
  update Schedule Ranges;
  update Distribution Functions;
endwhile;
```

In force-directed scheduling, statistical measures are used to derive the order and cycle step in which operations are scheduled. While a permutation Π hasn't been specified explicitly, it can be derived in a straightforward way. The algorithm has an $O(|T|^3 \cdot {T_c}^2)$ complexity, in which $|T|$ denotes the number of operations to be scheduled, and T_c denotes the number of cycle steps available for scheduling.

In [Verh91] some improvements on force directed scheduling are presented to improve the effectiveness of the method, without affecting its time complexity. The first modification is called gradual time-frame reduction. Instead of assigning an operation to a cycle step immediately, the feasible schedule range of an operation is reduced by one cycle step. The underlying idea is that the probabilistic distribution functions become a better estimate of the final distribution functions of the resulting resource allocation. The improved method is not a permutation scheduler, as operations may be regarded for scheduling more than once. Another improvement, called global spring constants, is used to emphasize the effect of changing the probabilistic distribution functions in situations where they are near the maximum distribution values found so far. In [Verh92] a complexity reduction is presented, based on a more efficient way of calculating forces, reducing the complexity from $O(|T|^3 \cdot {T_c}^2)$ to $O(|T|^2 \cdot {T_c}^2)$. It is based on an incremental calculation of the change in distribution functions, based on gradual time-frame reduction.

In Chapter 6 a new time constrained permutation scheduling method will be presented, which tries to minimize the resource allocation induced by the schedule, and in which the search for a permutation Π is controlled by the use of genetic algorithms.

5.7.3 Resource constraint satisfaction

If both precedence constraints and resource constraints are imposed, the number of operations in each cycle step, requiring the same resource type, is restricted. This restriction can be fulfilled by an additional procedure, called `selectCycles` (see Algorithm 5.4). If inside the feasible schedule range of an operation no cycle steps can be found in which a resource is free, no feasible schedule can be constructed from permutation Π.

Just like the other permutation scheduling methods discussed before, the schedule of an operation during the execution of Algorithm 5.4 can cause the feasible schedule range of other operations to become restricted. Hence the method is not only driven by a resource constraint, but also by time constraints, induced by precedence constraints.

In Section 5.8 a proof will be given that there exists at least one permutation Π, for which Algorithm 5.4 returns the optimal solution.

Algorithm 5.4 (Resource constrained permutation scheduling).

```
for i = 0 to |Π|-1 do
  v = Π(i);                                  // select operation
  C = selectCycles(v,asap(v),alap(v));       // determine cycle steps in which
                                             // resources able to implement
                                             // v are free
  if (C == ∅) then return("infeasible schedule");
  φ(v) = Select(C);                          // select cycle step from C
  update schedule ranges;
  update resource usage;
endfor;
```

5.7.4 Time and resource constraint satisfaction

Schedulers which try to schedule operations satisfying precedence, time, and resource constraints are called feasibility schedulers. Because of the time constraint, the schedule of operations is upper bounded by an alap-value. This is identical to the situation where operations obtain an upper-bound constraint during resource constrained scheduling as discussed before, and hence Algorithm 5.4 can also be used for feasibility scheduling.

A method which can cope with both resource constraints and time constraints efficiently, is reported in [Timm95]. It is based on a bipartite graph matching formulation called MEI analysis, already mentioned in Section 4.6. The scheduling problem is translated into finding a permutation Π of operations in such a way that each operation is adjacent to at most one MEI in the bipartite graph. A permutation Π which represents a feasible schedule implies a bijection between operation schedule ranges and MEIs, and consequently defines a complete matching. A branch-and-bound approach is used to find a correct permutation Π. It uses a greedy strategy to obtain a sparse search tree by first investigating operations adjacent to the module execution interval with the smallest number of operations adjacent. If there are no such MEIs, then the MEI with the smallest end cycle step is selected.

5.8 Topological permutation scheduling

The main disadvantage of strict permutation scheduling as presented in the previous section, is that scheduling a particular operation may constrain other operations, caused by the dominance of the order specified by the permutation. Let Π be a permutation of

the operations of V. Let $v \in V$ be an operation which is currently scheduled. Let $\Pi^{-1}(v)$ denote the position of v in Π. Let $v \prec_\Pi u \Leftrightarrow \Pi^{-1}(v) < \Pi^{-1}(u)$. Let SUC($v$, Π) be the set of successor operations of v in Π, given by SUC(v , Π) = $\{u \in V \mid v \prec_\Pi u\}$. If scheduling operation v implies that operation $u \in$ SUC(v , Π) will be scheduled in such a way that it induces a non-optimal or infeasible schedule, then the coordinate of Π pointing to u will lose its influence on the schedule of u.

One way to prevent constraining the schedule range of operations of complete paths, caused by scheduling an operation, is to schedule in a topologically sorted manner. The topological order is specified by the partial order induced by the dependence relations inside an acyclic process. A permutation Π can be used to obtain a total order $\prec_C$ from a partial order, as specified by the following equation:

$$u \prec_C v \Leftrightarrow ((u \prec v) \vee (\neg(v \prec u) \wedge (u \prec_\Pi v)) \qquad (5.1)$$

Hence Π only enforces a schedule order if the partial order of the process itself doesn't make any requirement about the execution order of operations.

The class of topological permutation schedulers is a subset of the class of strict permutation schedulers. At the end of this section it will be proven that restricting scheduling to topological permutation scheduling doesn't exclude the optimal solution from the search space, hence there exists at least one permutation for which topological permutation scheduling returns an optimal solution.

Algorithm 5.5 (Topological permutation scheduling template).

```
repeat
  for i = 0 to |Π|-1 do
    v = Π(i);                                  // select operation
    if (unscheduled(v) ∧ scheduledpreds(v))    // in a topological way
      φ(v) = Select[T_min,T_max];              // select cycle step
    endif;
  endfor;
until all operations scheduled;
```

An easy way to construct $\prec_C$, given $\prec$ and Π, is shown by Algorithm 5.5. The operations of V are visited in order of the permutation Π, to search for the first unscheduled operation $v \in V$ (`unscheduled (v)`) for which each predecessor operation has been scheduled (`scheduledpreds(v)`).

Let $|\Pi|$ denote the length of the permutation. The worst case complexity of Algorithm 5.5 is determined by the complexity of procedure `Select` and by the complexity of searching for an unscheduled operation in permutation Π for which each predecessor operation has been scheduled. In the worst case the unscheduled operation is situated at the end of the permutation, resulting in a worst case complexity of $O(|\Pi|^2)$. This result has to be extended with the worst case complexity of procedure `Select`. In case of strict permutation scheduling algorithm the search for an operation to be scheduled is $O(1)$,

hence the total worst case complexity of Algorithm 5.1 is $O(|\Pi|)$ extended with the worst case complexity of procedure `Select`.

A more efficient search for unscheduled operations can be obtained by making use of a heap data structure [Corm90]. Each operation contains a field called *key*, indicating its position in Π. The value of the key can be computed by one linear scan among Π ($\forall_{\,i \in [0\,,\,|\Pi|\,-\,1]}$ *key*($\Pi(i)$) := i), yielding a $O(|\Pi|)$ algorithm. While visiting each operation, the number of initially unscheduled predecessor operations can be stored in a field called *indegree*. Visiting each predecessor operation in a process (T , F) has a worst case complexity $O(|T| + |F|)$. In practical cases each operation will have at most 2 incoming edges, thus this complexity can be reduced to $O(|T|)$, which in case $|\Pi| = |T|$ equals $O(|\Pi|)$. When an operation $u \in T$ is scheduled, *indegree*(v) is decreased by 1 for each successor operation $v \in T$, with $u \prec v$. If *indegree*(v) becomes 0 for an operation $v \in T$, operation v is stored inside a heap. Adding an element to a heap, while preserving the heap property, has a complexity $O(\log n)$, in which n is the size of the heap. During the execution of Algorithm 5.5, at most $|\Pi|$ operations have to be stored simultaneously inside the heap, hence the worst case complexity of building a heap is $O(\log |\Pi|)$. Extracting an element with minimal (or maximal) key from the heap also has a worst case complexity of $O(\log |\Pi|)$. Searching for such an element must be performed exactly $|\Pi|$ times during the run of Algorithm 5.5, hence the worst case complexity of Algorithm 5.5 is $O(|\Pi| \cdot \log |\Pi|)$, which has to be extended with the worst case complexity of procedure `Select`. See Algorithm 5.6 for the complete algorithm.

Algorithm 5.6 (Using a heap structure).

```
// Initialize heap structure
HEAP = ∅;                                  // Empty heap at start
for i = 0 to |Π|-1 do
  key(Π(i)) = i;
  indegree(Π(i)) = |pred(Π(i))|;           // number of direct predecessors
  if (indegree(Π(i)) == 0) then
      add(Π(i),HEAP);
  endif;
endfor;
// Start topological scheduling
for i = 0 to |Π| - 1 do
  v = ExtractMIN(HEAP);                    // select operation
  φ(v) = Select[0,T_max];                  // select cycle step
   // update HEAP structure
  for all u ∈ suc(v) do                    // for each direct successor,
    if (indegree(u)-- == 0) then           // if all predecessors have
      add(u,HEAP);                         // been scheduled, add to HEAP
    endif;
  endfor;
endfor;
```

For the sake of simplicity, the functionality in Algorithm 5.6 regarding the manipulation of the heap structure will not be explicitly mentioned in successive algorithms about topological scheduling. This results in the template given by Algorithm 5.7, in

which a function call `GetFirstFree` is used to denote the extraction of the unscheduled operation from the heap with the smallest key value.

Algorithm 5.7 (Simplified topological permutation scheduling).

```
for i = 0 to |Π|-1 do
  v = GetFirstFree(Π);            // select operation
  φ(v) = Select[T_min,T_max];     // select cycle step
endfor;
```

5.8.1 Precedence constraint satisfaction

By scheduling operations within their feasible schedule range, and by updating the feasible schedule range of unscheduled operations, topological permutation scheduling will always satisfy the precedence constraints. Topological scheduling will never affect the upper bound of the schedule range of operations, hence in comparison to strict permutation scheduling, it will never produce infeasible schedules with respect to the precedence constraints. See Algorithm 5.8 for a description of precedence constrained topological permutation scheduling.

Algorithm 5.8 (Precedence constrained topological scheduling).

```
for i = 0 to |Π|-1 do
  v = GetFirstFree(Π);            // select operation
  φ(v) = Select[asap(v),∞];       // select from feasible range
  update schedule ranges;         // only influences asap value
endfor;
```

Because the scheduling technique is topological, no explicit update of schedule ranges is needed. The lower bound *asap*(*v*) of an operation *v* can be determined by:

$$asap(v) = \mathrm{MAX}_{\,u \in T \mid u \prec v}\,(\varphi(u) + \delta(u\,,\,v))$$

The asap value needs to be determined only once for each operation, leading to a worst case complexity of $O(|F|)$, which in case each operation has at most 2 input edges equals $O(|T|)$, with $|T|$ the number of operations to be scheduled. Hence the total complexity of the algorithm is $O(|T| \,.\, \log |T|)$, to be extended with the complexity resulting from the `Select` procedure. The result is given in Algorithm 5.9, in which T_{min} denotes the time the first operation starts its execution.

Algorithm 5.9 (Precedence constrained topological permutation scheduling 2).

```
for i = 0 to |Π|-1 do
  v = GetFirstFree(Π);            // select operation
  temp_asap = T_min;              // start time of schedule
  for all u ∈ pred(v) do
    temp_asap = MAX(temp_asap,φ(u) + δ(u,v));
  endfor;
  asap(v) = temp_asap;
  φ(v) = Select[asap(v),∞];     // select from feasible range
endfor;
```

For sake of simplicity, the derivation of the asap values, as shown in Algorithm 5.9, will not be explicitly mentioned in successive algorithms about topological scheduling. Instead, to obtain the asap-value *asap*(*v*) of an operation $v \in T$, the function `getAsap(v)` will be used as an abbreviation, resulting in Algorithm 5.10.

Algorithm 5.10 (Precedence constrained topological permutation scheduling 3).

```
for i = 0 to |Π|-1 do
  v = GetFirstFree(Π);              // select operation
  φ(v) = Select[getAsap(v),∞];      // select from feasible range
endfor;
```

When the completion time of the schedule must be optimized, the schedule of an operation can be assigned to the first cycle step of its feasible schedule range, resulting in an asap-scheduling algorithm.

When the resource allocation must be optimized, only one resource will be allocated for each module type. The procedure `Select` should defer operations such that no additional resources need to be allocated. This can be achieved by administrating the resource usage of the schedule so-far, and defer operations until the corresponding resource is free. This is an important advantage with respect to Algorithm 5.2, which because of its non-topological way of scheduling may obstruct operations to be deferred, and might therefore introduce the allocation of supplementary resources, and hence may miss out on the optimal solution. The topological way of scheduling results into an efficient algorithm which will always return the optimal solution.

5.8.2 Time constraint satisfaction

When besides precedence constraints time constraints are imposed, in contrast to Algorithm 5.8, upper bounds are needed to reflect the feasible schedule range of operations, denoted by their alap-value.

Algorithm 5.11 (Time constrained topological permutation scheduling).

```
for i = 0 to |Π|-1 do
  v = GetFirstFree(Π);                 // select operation
  φ(v) = Select[getAsap(v),alap(v)];   // select from feasible range
endfor;
```

If during topological scheduling operations are scheduled somewhere at the end of their feasible schedule range, the feasible schedule range of successor operations will also decrease. In case of optimizing the resource allocation this might lead to the same situation as with non-topological based permutation scheduling, in which the feasible schedule range of operations can be decreased such that the optimum drops of the search space. This implies that parts of the permutation become insignificant for scheduling. Therefore it is important that the `Select` procedure in Algorithm 5.11 tries to prevent to schedule operations unnecessary in the later region of their feasible schedule range.

Because the alap value doesn't change during scheduling, there is no need to update the schedule range of operations due to the schedule assignment of a single operation.

5.8.3 Resource constraint satisfaction

If both precedence constraints and resource constraints are imposed, the number of operations which, are scheduled in the same cycle step and require the same resource type, is restricted. This means that for an operation $v \in T$ a cycle step $c \geq asap(v)$ must be chosen, with a free resource of type $\xi(v)$. This choice is performed by a procedure which is called `satisfyResConstr` in Algorithm 5.12. For each module type $l \in ModType$, with $\exists_{v \in T}\ \xi(v) = l$, an array implementation of a doubly linked list can be used to be able to access the cycle steps $c \geq asap(v)$ in which modules are free to implement operation $v \in T$.

The main difference between Algorithm 5.12 and Algorithm 5.4 is the fact that the feasible schedule range of operations will never be bounded from above, and hence a cycle step in which an operation can be scheduled without introducing resource conflicts can always be found. Hence in contrast to Algorithm 5.4, the schedules constructed by Algorithm 5.12 are always feasible, and the search effort can be oriented towards finding a good quality solution instead of finding a feasible solution.

Algorithm 5.12 (Resource constrained topological permutation scheduling).

```
for i = 0 to |Π|-1 do
  v = GetFirstFree(Π);                        // select operation
  φ(v) = satisfyResConstr(v,getAsap(v),∞);    // determine cycle step
  update resource usage;
endfor;
```

The most useful application for this kind of scheduling algorithms, is to optimize the completion time of the resulting schedule. An example of such a scheduling method, which is very common in high-level synthesis, is list scheduling, originally published by [Hu61]. The name list scheduling originates from the fact that in the original algorithm a list of operations is used to keep track of all operations for which all predecessor operations have been scheduled.

Algorithm 5.13 `list_schedule`: $\Pi \rightarrow \varphi$

```
cycle = 0;
repeat
  // Visit operations in order of permutation
  for i = 0 to |Π|-1 do
    v = Π(i);
    // Check whether v can be scheduled in the current cycle step
    if (unscheduled(v) ∧ scheduledpreds(v) ∧ ResourceFree(v, cycle))
      φ(v) = cycle;
  endfor;
  cycle++;              // proceed schedule in successive cycle step
until all operations are scheduled;
```

In Algorithm 5.13 a template for the general list scheduling algorithm can be found. An operation is allowed to be scheduled in the current cycle step if it has no unscheduled predecessors, and a resource is available for the execution of this operation in the current cycle step. Consecutively, cycle steps are selected, and according to the order specified by the permutation, unscheduled operations are searched for, which are allowed to be scheduled in the current cycle step. This procedure is repeated until all operations have been scheduled.

The quality of list scheduling depends on the permutation (in case of list scheduling also called a priority function) used. In [Girc84] the urgency of an operation is used to define a permutation. The urgency of an operation is defined as the minimum number of cycle steps required between the operation and any enclosing timing constraint (in other words the alap-value of the operation). In [Pang87] the mobility is used to define a permutation. The mobility $m(v)$ of an operation v is defined as $m(v) = alap(v) - asap(v) - \delta(v)$. Operations with zero mobility are situated on the critical path, and are selected for scheduling first. To be able to distinguish between operations with the same mobility, the operation with the highest number of successors is chosen first. In [Thom90], several priority functions are used. Operations which are not affected by the primary priority function are passed to the secondary priority function, and on its turn operations which are not affected by the secondary priority function are passed to the tertiary priority function. The first priority function determines whether delaying an operation causes it to be scheduled behind its alap value. The second priority function concerns resource constraints. By assigning operations of a critical path first, a better idea of the resource utilization can be achieved. Also the total number of successors of an operation can serve as a priority measure, which detects operations that can be a bottleneck when they are deferred. The third priority function tries to maximize the resource utilization by checking which operations become ready for scheduling in the next cycle step.

In [Heij91] an overview of several list scheduling algorithms is published. The first scheduler uses the global freedom as a priority function. Initially, the global freedom of an operation equals the mobility of that operation. If an operation is deferred in time, the global mobility will be decreased by one. The idea is that operations which are not situated on the critical path, and are deferred many times, will gain more priority to be scheduled. The second scheduler is based upon the number of direct successors of an operation. The idea is that operations with many successors, which are deferred to successive cycle steps, cause all their successors to be deferred too, and might increment the completion time of the schedule. The third scheduler, uses the alap of an operation as a priority function. The fourth scheduler uses the distance $T_{max} - asap(v)$ of operation v as a priority function. The idea is that operations which are situated far from the time constraint T_{max} can be moved more easily without increasing the completion time than operations which are close to the time constraint T_{max}. The fifth scheduler uses a weighted priority function, in which the alap value of an operation is used as the main priority function. If the alap of two operations is equal, the number of successors is used to distinguish between these operations, and if still no difference can be made, the

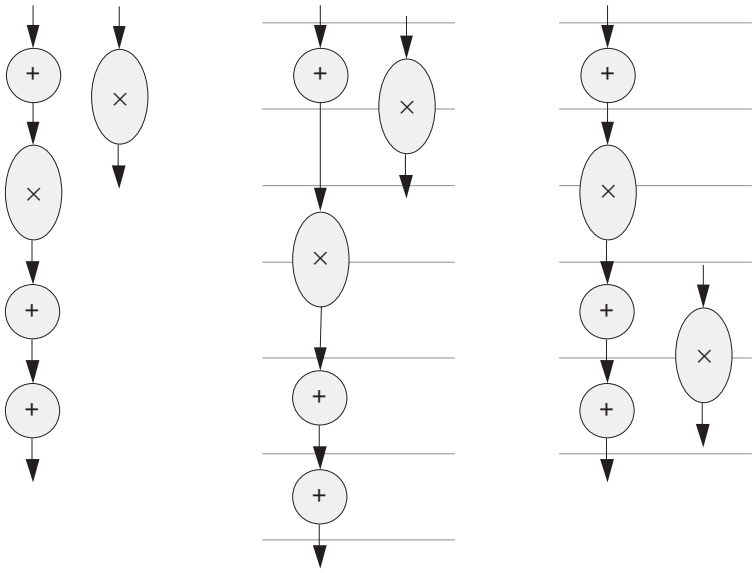


Figure 5.1 Partial data-flow graph, list schedule, and optimal schedule.

distance is used as priority function. The results in [Heij91] show that the quality of the solution heavily depends on the priority function used, but in a non-obvious way.

In [Potk89] and [Paul89] global measures are offered to define a priority function. In [Potk89], both a local priority function and a global priority function are presented. They are both defined as the ratio of the available resource allocation of a particular resource type divided by the required resource allocation of the same type. The local priority only looks at this ratio in the current cycle step. The global priority looks at all unscheduled operations. Operations are scheduled in such a way that these ratios are kept as large as possible. In [Paul89], forces are used to see the effect of an attempted schedule in the current cycle step. The operation which results in the best force will be selected and scheduled.

An important disadvantage of list scheduling is that it may miss out the optimal solution regardless which priority function is used. Suppose that a multiplication requires 2 cycle steps to execute on a multiplier, and an addition requires 1 cycle step to execute on an adder. The list schedule of the process shown in Figure 5.1(a) will lead to a completion time of 6 cycle steps (Figure 5.1(b)), independent of the priority function used. The optimal schedule is shown in Figure 5.1(c), and takes 5 cycle steps. In [Grah76] it is shown that an increment in the number of resources, a reduction of the delay of operations or weakening the precedence constraints also may lead to an increment in the completion time of a solution produced by a list scheduler.

Algorithm 5.14 `construct_schedule`: $\Pi \rightarrow \varphi$

```
cost = 0;
for i= 0 to |Π|-1 do
  v = GetFirstFree(Π);                          // select operation
  φ(v) = firstFreeResource(v,getAsap(v),∞);     // determine first cycle step
                                                // in which resource is free
  update resource usage;
endfor;
```

A new alternative way of performing resource constrained scheduling in high-level synthesis is by using Algorithm 5.14. The algorithm schedules each operation from permutation Π by repeatedly searching for the first unscheduled operation $v \in T$ from Π for which each predecessor has been scheduled. The selected operation v is attempted to be scheduled in the earliest cycle step from its feasible range. When all resources are occupied at this cycle step, the function `firstFreeResource(v,getAsap(v),∞)` searches for the first cycle step $c \geq asap(v)$ in which a resource is available to implement v. After an operation v is scheduled, the resource requirements due to scheduling v are administrated. The cost $C_{max}(\varphi)$ of schedule φ is defined by the last cycle step in which an operation ends its execution.

A proof will be given that there exists at least one permutation Π for which the topologically sorted schedule constructor results in an optimal schedule.

Let $\varphi^{-1}(t)$ denote the set of operations that are scheduled in cycle step t. The following algorithm constructs a permutation out of a schedule:

Algorithm 5.15 `construct_permutation`: $\varphi \rightarrow \Pi$

```
i = 0;
for t = t_begin to t_end do
  foreach v ∈ φ^-1(t) do
    Π(i++) = v;
  endfor;
endfor;
```

Theorem 5.1: There exists a permutation Π for which `construct_schedule(`Π`)` returns an optimal schedule.

proof: Let φ_{opt} be an optimal (and hence feasible) schedule. Let Π be given by:

Π = `construct_permutation(`φ_{opt}`)`

Then according to algorithm `construct_permutation`, Π can be written as:

Π

= {*def.* `construct_permutation`}

$\Pi(0) \oplus \Pi(1) \oplus ... \oplus \Pi(|\Pi| - 1)$

= {*def.* `construct_permutation`}

$\varphi_{opt}^{-1}(t_{begin}) \oplus \varphi_{opt}^{-1}(t_{begin} + 1) \oplus ... \oplus \varphi_{opt}^{-1}(t_{end})$

In which $\oplus$ denotes the concatenation symbol. The concatenation $P = S \oplus T$ of two sets S and T is defined as a concatenation of a sequence containing all elements of S in an arbitrary order plus a sequence containing all elements of T in an arbitrary order.

For all $u,v \in \varphi_{opt}^{-1}(i)$, $i \in [t_{begin} , t_{end}]$, there are no precedence constraints or resource conflicts, because otherwise φ_{opt} would be an infeasible schedule.

Let φ = `construct_schedule`(Π). We first prove by induction that:

$$\forall_{t \in [t_{begin}, t_{end}]} \forall_{v \in \varphi_{opt}^{-1}(t)} \varphi(v) \leq t$$

hence,

$$\forall_{v \in T} \varphi(v) \leq \varphi_{opt}(v)$$

First, let $\Pi = \varphi_{opt}^{-1}(t_{begin})$. From the definition of φ_{opt} we know that there are no resource or precedence conflicts between any operations of Π, and hence `construct_schedule` will schedule all operations of Π in cycle step t_{begin}. Thus:

$$(\forall_{v \in \varphi_{opt}^{-1}(t_{begin})} \varphi(v) = t_{begin}) \Rightarrow (\forall_{v \in \varphi_{opt}^{-1}(t_{begin})} \varphi(v) \leq t_{begin})$$

Let the induction hypothesis be true for $t \in [t_{begin} , t_{begin} + n]$. Thus,

Π

=

$\varphi^{-1}(t_{begin}) \oplus \varphi^{-1}(t_{begin} + 1) ... \varphi^{-1}(t_{begin} + n)$

and hence $\forall_{v \in \Pi} \varphi(v) \leq t_{begin} + n$.

Let $\Pi' = \varphi^{-1}(t_{begin}) \oplus \varphi^{-1}(t_{begin} + 1) \oplus ... \oplus \varphi^{-1}(t_{begin} + n + 1) = \Pi \oplus \Pi''$.

Because in the original schedule φ_{opt} all operations from $\varphi_{opt}^{-1}(t_{begin} + n + 1)$ could be scheduled without constraint violation in cycle step $t_{begin} + n + 1$, and from the induc-

tion hypothesis we know that no operations from Π are scheduled in cycle steps larger than $t_{begin} + n$, the operations from Π'' can be scheduled without constraint violation inside cycle step $t_{begin} + n + 1$ or smaller. Hence $\forall_{v \in \Pi \oplus \Pi''} \varphi(v) \leq t_{begin} + n + 1$, which proves the induction hypothesis.

So if Π = `construct_permutation`(φ_{opt}), and φ = `construct_schedule`(Π), then $C_{max}(\varphi) \leq C_{max}(\varphi_{opt})$. Because φ_{opt} is an optimal solution, we know that $C_{max}(\varphi) \not< C_{max}(\varphi_{opt})$, which ends the proof.

Observe that the solution space of list scheduling, in which the order of operations in a permutation Π is restricted to non-decreasing asap values of its operations, is a subset of the solution space of Algorithm 5.14. The complexity to build a schedule from a permutation by Algorithm 5.14 is $O(|\Pi| \cdot \log|\Pi|)$.

5.8.4 Time and resource constraint satisfaction

Time constraints and resource constraints impose a restriction with respect to the set of operations that can be scheduled simultaneously in the same cycle step. If inside the feasible schedule range of an operation no cycle steps can be found in which a resource is free, no feasible schedule can be found for permutation Π.

Care must be taken not to schedule operations such that they fix the schedule of other operations in such a way that infeasible schedules results. This means that there should be a close interaction between the permutation Π and the `Select` procedure for such an algorithm to become successful.

Algorithm 5.16 (Feasible constrained topological permutation scheduling).

```
for i = 0 to |Π|-1 do
  v = GetFirstFree(Π);                          // select operation
  φ(v) = satisfyResConstr(v,getAsap(v),alap(v));    // determine cycle step
  if (φ(v) == ∅) then return("infeasible schedule");
  update resource usage;
endfor;
```

To determine the (earliest) cycle step in which an operation must be scheduled, Algorithm 5.16 can be extended with the MEI method presented in [Timm93] (see also Section 4.6). The MEI analysis may prevent operations from being scheduled in cycle steps for which there doesn't exist a corresponding matching (and hence no feasible schedule) in the bipartite matching graph. The success of applying such a strategy has been shown in [Timm95].

5.9 Permutation statistics

In the preceding sections some algorithms have been presented which use permutations to generate a schedule. A permutation can be considered as an encoding of a schedule.

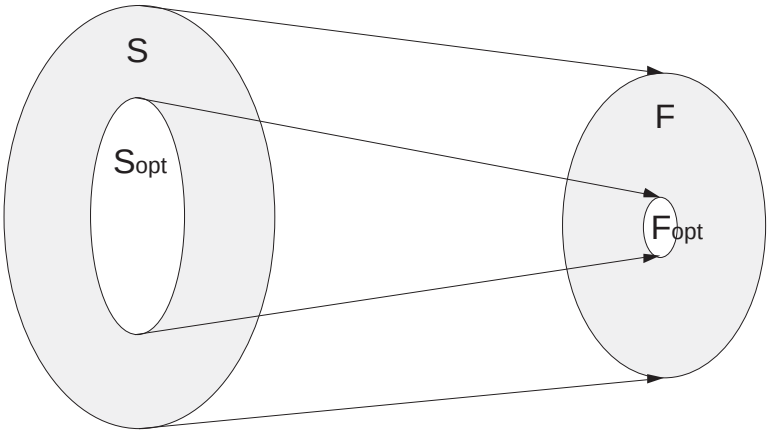


Figure 5.2 Search space S versus solution space F.

Consider a time constrained scheduling problem, consisting of a data-flow graph *DFG* (precedence constraints) and a time constraint T_{max}, and aiming at a schedule inducing a minimal resource allocation. With respect to this combination of constraints, a set of feasible schedules F can be created (also called solution space), containing a subset $F_{opt} \subseteq F$ consisting of solutions which are optimal with respect to the resource allocation induced (also called optimal solution space).

The expression $|F_{opt}| / |F|$ denotes the relative amount of optimal solutions with respect to the total size of the solution space. This ratio is of importance when probabilistic search methods such as genetic algorithms (see Chapter 6) are applied to the scheduling problem. The higher this ratio, the higher the probability that an optimal solution is encountered.

For the schedule constructors presented in this chapter, the size of the search space S is determined by the number of possible orders of a permutation Π in combination with the possible cycle step assignment of the `Select` procedure. The set S_{opt} consists of all optimal schedule solutions obtained after applying a particular schedule constructor to particular permutations. The ratio $|S_{opt}| / |S|$ denotes the relative amount of permutations resulting in an optimal schedule solution. For a relation between S and F, see Figure 5.2.

In Table 5.1 some results for the size of F, F_{opt}, S, and S_{opt} for some specific scheduling problems can be found. Because counting the total number of feasible schedules can be a quite cumbersome task for large schedule examples, only relative small schedule examples shown in Figure 5.3 have been used to obtain the results presented in Table 5.1. The schedule constructor used to obtain the results in column ‘permutation search space size’ is based on Algorithm 5.4, with the exception that if an operation cannot be scheduled inside its feasible range (hence $C = \emptyset$), the resource allocation is increased. Each operation is selected to be scheduled inside the earliest possible cycle step in which a resource is free to execute the operation. The results of the column ‘topological permutation search space size’ have been obtained by Algorithm 5.14.

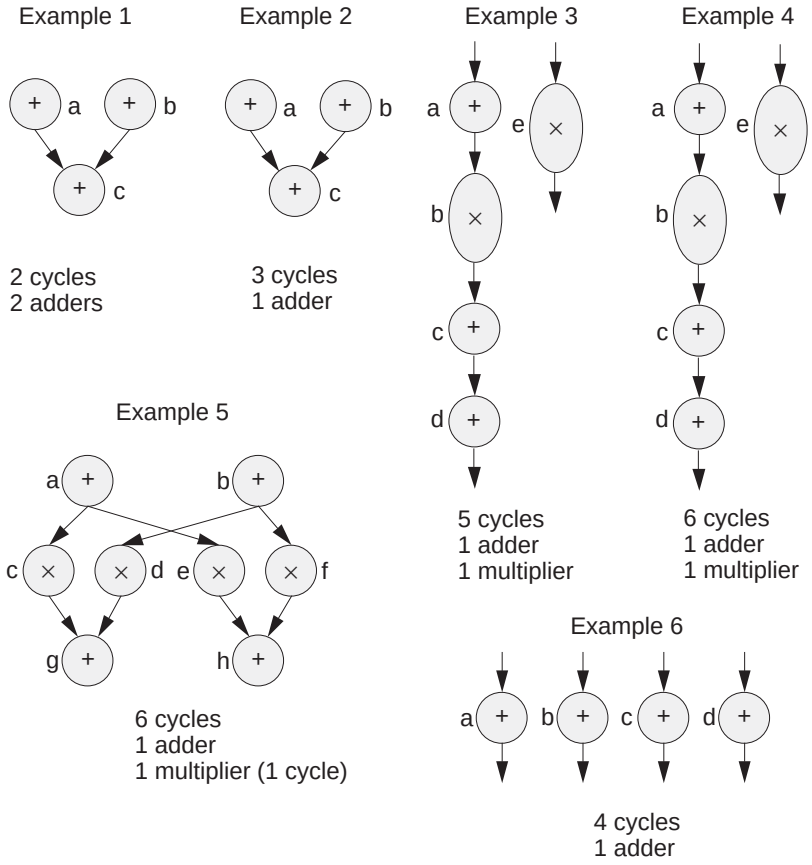


Figure 5.3 Some relative small schedule examples.

From all examples in Table 5.1, it can be concluded that the ratio between the number of optimal solutions and the total number of feasible solutions is the largest in case topological permutation scheduling techniques are used. It is expected that probabilistic search methods such as genetic algorithms will be more efficient when using topological permutation scheduling techniques. This observation will be confirmed in Chapter 6, where some empirical results obtained for larger scheduling examples will be presented.

Table 5.1 Permutation statics of examples of Figure 5.3.

Example	solution space size		permutation search space size		topological permutation search space size	
	$\|F\|$ / $\|F_{opt}\|$	ratio	$\|S\|$ / $\|S_{opt}\|$	ratio	$\|S\|$ / $\|S_{opt}\|$	ratio
1	1	100%	6	100%	6	100%
	1		6		6	
2	5	40%	6	33%	6	100%
	2		2		6	

Example	solution space size		permutation search space size		topological permutation search space size	
	$\|F\|$	ratio	$\|S\|$	ratio	$\|S\|$	ratio
	$\|F_{opt}\|$		$\|S_{opt}\|$		$\|S_{opt}\|$	
3	4	25%	120	50%	120	50%
	1		60		60	
4	25	40%	120	100%	120	100%
	10		120		120	
5	2350	1.7%	40320	5.9%	40320	100%
	40		2384		40320	
6	256	9.4%	24	100%	24	100%
	24		24		24	

5.10 Permutation scheduling and pipelining

During the pipelined execution of a data-flow graph $G = (V , E)$, a new execution of G is started before G has finished its previous execution(s). This implies that operations involved with data related to different executions of G are executing in parallel.

Data-flow graph G can be partitioned in so called pipeline stages. In a pipelined schedule, operations assigned to the same pipeline stage are concerned with the execution of data from the same invocation. Let a (non-pipelined) schedule of G be given by φ, which assigns a cycle step to each operation $v \in V$. Let the distance between successive executions (called invocation distance) of G be given by data introduction interval dii. In that case the pipelined cycle step assignment of G is given by $\varphi_{dii}(v)$, defined by $\varphi_{dii}(v) = \varphi(v) \bmod dii$. The pipeline stage assignment $\sigma(v)$ of operation $v \in V$ is defined by $\sigma(v) = \varphi(v) \operatorname{div} dii$. The schedule of such an operation is given by a tuple $(\varphi_{dii}(v) , \sigma(v))$. A simplified example of a pipelined execution of a data-flow graph can be found in Figure 5.4.

Assume that for each operation $v \in V$, $\delta(v) = 1$. To explicitly model the concurrency of the pipelined execution of operations in a data-flow graph $G = (V , E)$, it can be thought to be folded among the pipelined cycle step budget $\{0, 1, ... , dii - 1\}$ as shown in Figure 5.4, resulting in a pipelined data-flow graph $G_{dii} = (V , E_{dii})$. The operations of G can be partitioned into n subsets, with $V = V_0 \cup V_1 \cup ... \cup V_{n-1}$, and $i \in \{0, 1, ... , n - 1\}$, in which V_i represents the set of operations situated in pipeline stage i (in other words $\forall_{v \in Vi}\, \sigma(v) = i$). The partitioning of G is accomplished by cutting each edge (u , v), with $\sigma(v) - \sigma(u) = k$, k times. The set $E_{dii} \subset E$ consists of edges of G, excluding the edges (u , v) with $u \in V_i$, $v \in V_j$, and $0 \le i < j < n$.

The concept of pipeline stages is similar to the concept of inter-iteration (see Section 3.4). The first pipeline stage processes data from the current process invocation, while

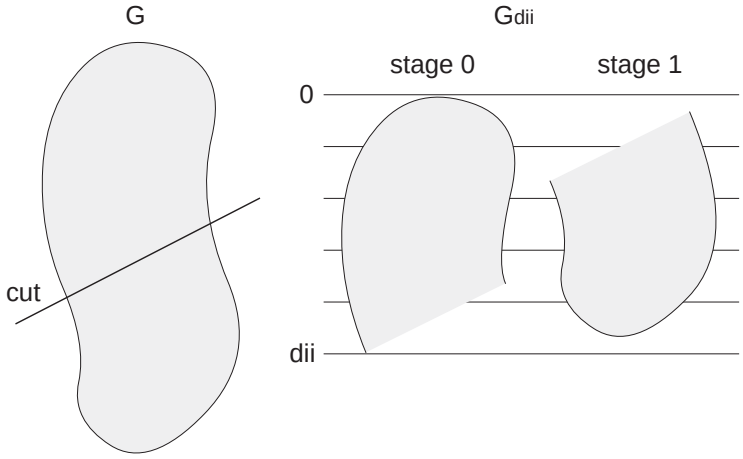


Figure 5.4 Simplified pipelined schedule example.

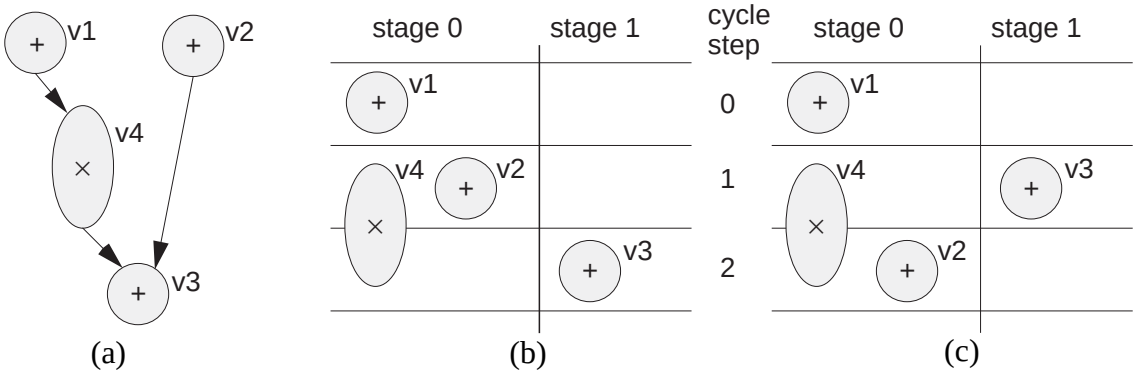


Figure 5.5 Pipelined schedule example.

successive pipeline stages are processing data from previous process invocation. Just like the case with loop structures, pipeline boundaries can be explicitly modelled by the use of delay nodes. If $\sigma(v) - \sigma(u) = k$, edge (u, v) should be replaced by a sequence of k delay nodes connected by a sequence of edges.

One of the main questions in this section is whether permutation scheduling can be used to construct (optimal) pipelined schedules. For this section we assume a pipelined resource constrained scheduling problem (in other words given a data-flow graph, a resource constraint, and a data introduction interval, find a schedule with minimal completion time).

Assume that a data introduction interval of 3 cycle steps is given, together with a resource constraint of 1 adder (requiring 1 cycle step), 1 multiplier (requiring 2 cycle steps), and the data-flow graph as given in Figure 5.5(a). In case of topological permutation scheduling (as in Algorithm 5.14), operation v_1 and v_2 will always be scheduled before operation v_3. This causes the adder to be occupied inside the first two (folded) cycle steps, and hence operation v_3 will have to be scheduled in cycle step 5 (which is equal to folded cycle step 2, see Figure 5.5(b)). In Figure 5.5(c) an example of a pipe-

lined schedule is shown, in which operation v_3 is scheduled in cycle step 4, inducing a shorter length schedule. This schedule can never be obtained by using topological permutation scheduling for the data-flow graph of Figure 5.5(a) directly. The problem originates from the fact that topological scheduling assumes that operations which are on head of a path are scheduled earlier in time than operations which are in the tail of a path, and therefore will never lead to resource conflicts. If pipelining is applied, and hence resource usage is folded, this situation is no longer true. Because operations in pipeline stage x, $x \in \mathbb{N}$, are always scheduled before operations in pipeline stage $x + 1$, all optimal schedules for a particular example can be excluded from the search space.

Another question is whether an optimal pipelined resource constrained schedule can be constructed by using a non-topological permutation scheduling technique derived from Algorithm 5.14 (in other words, operations are scheduled in their asap value, but without violating resource constraints).

A counter example is given in Figure 5.6, assuming a data introduction interval of 3 cycle steps, together with a resource constraint of 1 adder (requiring 1 cycle step), and 2 multipliers (requiring 1 cycle step). The optimal schedule with the smallest completion time is given in Figure 5.6(b). In this schedule, operation v_6 is scheduled one folded cycle step earlier than operation v_4. This implies that operation v_6 should be scheduled before operation v_4. Let's assume that operation v_1, v_2, and v_3 have been scheduled in pipeline stage 1. If operation v_6 is scheduled in its earliest possible cycle step, before operation v_4 is scheduled, it will be assigned to the last cycle step of pipeline stage 2, forcing operation v_4 to be scheduled inside the first cycle step of pipeline stage 2, inducing an infeasible resource allocation of 2 adders. If operation v_5 is scheduled before operation v_4, it will be scheduled in cycle step 1 of stage 1, resulting in an infeasible schedule because operation v_4 is forced to be scheduled inside cycle step 0 of stage 1, inducing a resource allocation of 2 adders. To obtain a feasible schedule, operation v_4 should be scheduled before operation v_6, resulting in the schedule of Figure 5.6(c). Hence there exists no permutation in combination with a cycle step assignment as given in Algorithm 5.14, leading to an optimal schedule.

The problem originates from the fact that scheduling an operation inside its earliest possible cycle step can move this operation towards a pipeline stage in which it imposes time constraints towards other operations such that non-optimal or infeasible solutions are created because the resource utilization is folded. The conclusion is that in case of pipelined resource constrained scheduling, the `Select` procedure as presented in Section 5.6 should not only consider scheduling operations within their earliest possible cycle step. The general question is what kind of `Select` procedure is needed, and whether a complicated `Select` strategy must be applied to all operations of a data-flow graph.

It will now be shown that the proof as given on page 77 is not applicable, because it makes use of the fact that some operations can be safely scheduled in earlier cycle steps, which in a folded cycle step range might imply that an operation is scheduled

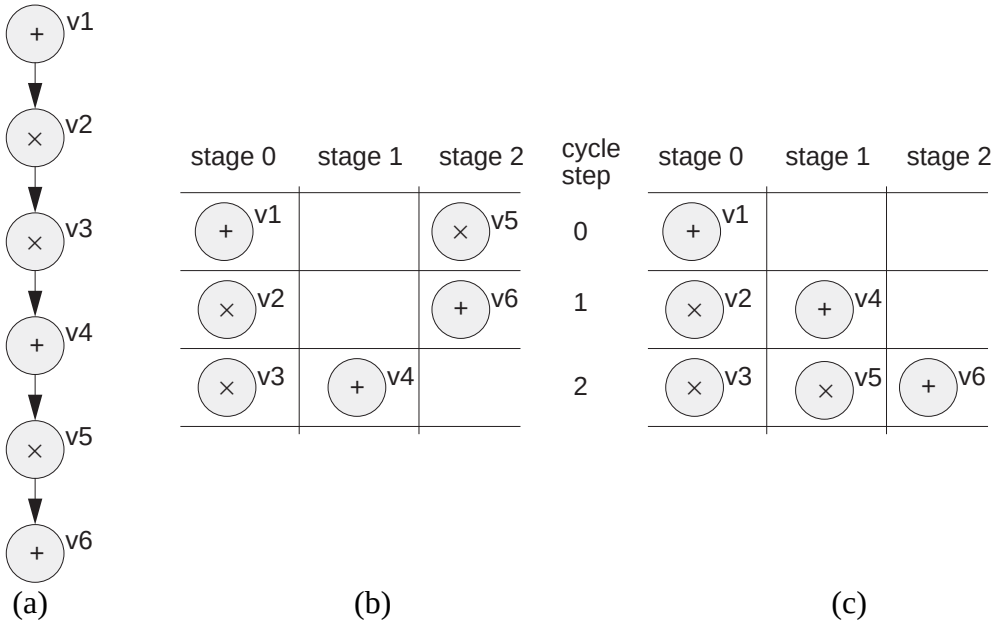


Figure 5.6 Pipelined schedule example (2).

'later'. Let φ_{opt} be an optimal pipelined schedule of data-flow graph $G = (V, E)$. Let $\Pi =$ `construct_permutation`(φ_{opt}), and let $\varphi =$ `construct_schedule`(Π) (see Algorithm 5.14 and Algorithm 5.15). The proof on page 77 makes use of the fact that `construct_schedule` constructs a schedule φ with $\forall_{v \in V} \varphi(v) \leq \varphi_{opt}(v)$. Applying `construct_permutation` to the optimal schedule φ_{opt} as shown in Figure 5.6(b) would induce a permutation $\Pi = v_1\ v_2\ v_3\ v_4\ v_5\ v_6$. Scheduling Π using Algorithm 5.14 results in the schedule as given in Figure 5.6(c). In this schedule $\varphi(v_6) > \varphi_{opt}(v_6)$, and hence φ is a non-optimal schedule, which on its turn shows that the proof of page 77 is not applicable.

If a permutation Π_{dii} based on folded cycle steps is constructed, still a non-optimal schedule results. Let $\Pi_{dii} =$ `construct_permutation`$(\varphi_{opt} \bmod dii)$, and let $\varphi =$ `construct_schedule`(Π_{dii}). The example of Figure 5.6 shows that $\Pi_{dii} = v_1\ v_5\ v_2\ v_6\ v_3\ v_4$, which results in an infeasible schedule inducing a resource allocation of 2 adders because `construct_schedule`(Π) will schedule operation v_1 in cycle step 0 and operation v_5 in cycle step 4, and hence will schedule operation v_4 in cycle step 3.

From Figure 5.7 it can be concluded that if instead of G, scheduling is applied to the pipelined graph G_{dii}, then according to the proof given on page 77 there exists a permutation Π_{dii} consisting of operations from V for which Algorithm 5.14 results in an optimal schedule. Thus if G is cut at the right places, and scheduling is applied to G_{dii}, then there exists a permutation of operations which results in an optimal schedule. Hence the problem to be solved is given data introduction interval dii, find the places where to cut G to obtain a pipelined data-flow graph G_{dii}, for which a permutation exists, resulting in an optimal schedule. In other words, determine the pipeline stage assignment $\sigma(v)$ for all operations $v \in V$.

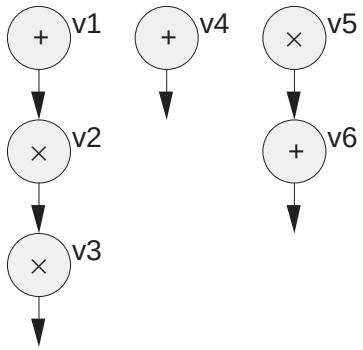


Figure 5.7 Pipelined data-flow graph derived from data-flow graph of Figure 5.6.

Suppose V is partitioned in (at most) n pipeline stages. Assume that the schedule range of an operation $v \in V$ is given by $[asap(v) , alap(v)]$. A feasible schedule of operation v requires that $\varphi(v) \in [asap(v) , alap(v)]$. Assigning v to pipeline stage $\sigma(v) = k$, with $0 \leq k \leq n$, and $\forall_{c \in [k \cdot dii , (k+1) \cdot dii]}\ c \notin [asap(v) , alap(v)]$, can never lead to a feasible schedule, and hence should be avoided. Furthermore, if the placement of pipeline stages between an operation and all its successor (or predecessor) operations is restricted to one stage, an optimal schedule can still be created. Let $\sigma(v) = k$, with $0 \leq k \leq n$, such that $\exists_{c \in [k \cdot dii , (k+1) \cdot dii]}\ c \in [asap(v) , alap(v)]$. Let $v \in V$, and $\forall_{u \in V \mid (u , v) \in E}\ \sigma(v) - \sigma(u) \geq 2$. Let φ_{opt} be an optimal schedule. No difference of the cost of $\varphi_{opt}(v)$ can be detected when the pipeline stage assignment $\sigma(v)$ of operation v is lowered in such a way that $\exists_{u \in V \mid (u , v) \in E}\ \sigma(v) - \sigma(u) = 1$. Hence, an upper bound on the pipeline stage of operation v which contains an optimal schedule is given by $\sigma(v) \leq \mathrm{MAX}_{u \in V \mid (u , v) \in E}\ \sigma(u) + 1$.

In Algorithm 5.17 a permutation Π of operations is used to determine an assignment of pipeline stages to all operations of a data-flow graph. Let `selectStage` be a procedure which selects for each $v \in V$ the smallest $k(v) \in \mathbb{N}$, with $0 \leq k(v) \leq n$, and $k(v) \cdot dii \in [asap(v) , alap(v)]$ (in other words the interval can be cut by a pipeline boundary). If no such k exists, then $k(v) = 0$.

Algorithm 5.17 (Pipeline stage assignment).

```
for i = 0 to |Π|-1 do
  v = Π(i);
  k = SelectStage(asap(v),alap(v),dii);
  if (k ≠ 0) then
    asap(v) = k * dii;
    update Schedule Ranges;
  endif;
endfor;
```

By using Algorithm 5.17, each possible pipeline assignment of G can be constructed, with the limitation that $\sigma(v) \leq \mathrm{MAX}_{u \in V \mid (u , v) \in E}\ \sigma(u) + 1$ (because procedure `selectStage` selects the smallest k with $k \cdot dii \in [asap(v) , alap(v)]$).

The pipeline stage assignment can be performed in a topologically sorted manner, without imposing restrictions to number of pipelined data-flow graphs. Let G_{dii} represent an

arbitrary pipelined data-flow graph, with $\forall_{v \in V} \sigma(v) \leq \text{MAX}_{u \in V \mid (u, v) \in E} \sigma(u) + 1$. Let permutation Π be such that:

1. the first operations of Π consist of the operations of G_{dii} with in-degree 0 (in other words without predecessor operations).

2. Let these operations be sorted topologically with respect to the precedence relations of data-flow graph G.

In that case Algorithm 5.17 will cut G in a topological manner. First the input operations $v \in V$ with $asap(v) = 0$ are visited, because by rule (1) and (2) they are located at the head of the permutation. Because $k = 0$ for these operations, no action will be performed. Then the operations $v \in V$, with $\sigma(v) = 1$, and which have no predecessors in G_{dii}, are visited. For these operations, $k = 1$, and the `update Schedule Ranges` will update the schedule range of successor (predecessor) operations $u \in V$ in such a way that $\sigma(u) \geq 1$ ($\sigma(u) < 1$). This procedure is repeated for increasing values of k, no operation $v \in V$ will be found with $k \cdot dii \in [asap(v), alap(v)]$, and hence the algorithm terminates with no further action, because all operations have implicitly been assigned to a pipeline stage.

The idea of finding a good cut inside a data-flow graph is similar to the concept of retiming (see Section 4.3.3 and Section 5.11). When retiming is applied to a data-flow graph, it can assign pipeline stages in such a way that it excludes the optimal schedule from the solution space. An example of such a case (containing loop structures) is given in Figure 5.9, in which two different retimings are given for the same data introduction interval *dii*, but for which the optimal schedule induces different resource allocations. Retiming G in such a way that the resulting graph G_{dii} contains the optimal schedule is proven to be an NP-hard problem [Potk91]. In [Potk91] heuristics are used which try to balance the resource allocation over the cycle steps available.

The retiming algorithms described in Algorithm 5.17 can be combined with Algorithm 5.14 to find a pipelined schedule by using permutations. One can use a combination of two permutations (one for retiming, one for scheduling), for which it is known that there exists a combination of these two permutations leading to an optimal pipelined schedule. A simplified method can be constructed which uses one permutation for both algorithms. No proof or counter example is known concerning the optimality of such a strategy. Nevertheless empirical results presented in Chapter 6 show that such a strategy, in combination with a genetic search, produces optimal results in all cases tested, which validates the use of such a strategy.

So far, it has been assumed that for each operation $v \in V$ the execution delay $\delta(v) = 1$. In case multi-cycled operations are allowed, an arbitrary cut of a graph cannot be described by a retiming. An example is shown in Figure 5.8, in which the multiplication operation is distributed among two pipeline stages. It also causes an increment of 1

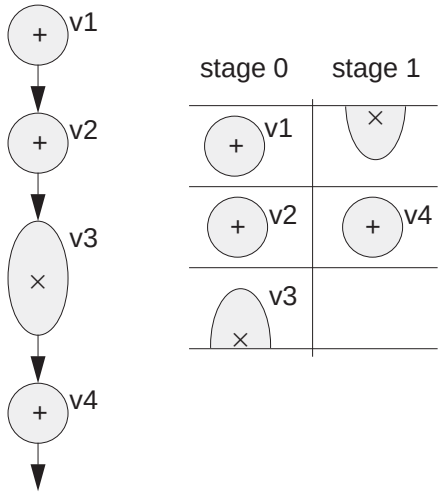


Figure 5.8 Pipelining and multi-cycling.

cycle step of the lower bound of operation v_4 in pipeline stage 1, which has to be accounted for when scheduling is applied to such a data-flow graph.

An operation *v,* for which the operation execution delay $\delta(v)$ equals *c* cycle steps, can be cut at *c* - 1 places. One can think of extending the permutation with cycle step assignments for multi-cycled operations to account for situations as depicted in Figure 5.8. In the results shown in Chapter 6 multi-cycled operations aren't explicitly accounted for during the assignment of pipeline stages (in other words the pipeline boundary is always assigned as close as possible to the asap-value of such an operation).

5.11 Permutation scheduling and cyclic data-flow graphs

Because a cycle contains at least one delay node, cyclic data-flow graphs are always related to the concept of pipelining. There are different approaches how to cope with the concept of delay nodes, which closely relate to the methods with respect to invocation distance constraints mentioned in Section 4.3.

In general, most schedule methods reported in the literature dealing with cyclic structures transform these cyclic structures into acyclic structures, and apply scheduling to these acyclic structures. In the next sections a short overview of the advantages and disadvantages of each method will be discussed.

5.11.1 Single iteration model

A single iteration of a cyclic structure can be obtained by splitting delay nodes, as has been shown in Section 4.3.1 on page 44. Because inter-iteration dependencies are discarded completely, the acyclic structure doesn't hold any information about the concurrency among different iterations. Depending on the schedule constraints this restriction might induce non-optimal or even infeasible schedules, regardless of the scheduling method that is used.

5.11.2 Multiple iteration model

Unfolding a graph *n* times exposes the inter-iteration concurrency between *n* iterations (see also Section 4.3.2, page 44). During loop unfolding inter-iteration dependencies are transformed into intra-iteration dependencies, and hence scheduling the unfolded graph using constructive scheduling techniques implicitly makes use of loop pipelining. The main problem of multiple iteration models is the increment in the number of operations to be scheduled. Unfolding may also lead to an increment in controller size, especially with nested loops.

5.11.3 Loop Winding, Loop Folding, Retiming

By reorganisation of the location of pipeline stages, both single iteration and multiple iteration methods might yield more successful results. Pipeline stage boundaries are represented by delay nodes, and hence reorganization of pipeline stages can be accomplished by transforming the graph. Changing the location of pipeline stages by transforming a graph is called loop winding, loop folding or retiming.

There are mainly two methods of loop winding. One method transforms the graph before scheduling. The other method transforms the graph during scheduling.

Loop winding before scheduling
In [Girc87] the first step consists of unwinding all cycle structures in a data flow graph. The acyclic graph that is obtained by this step is partitioned using the data introduction interval and wound in parallel. This achieves a functional pipeline, because operations from several iterations of the loop may be executed in parallel (depending on the results of the winding). Disadvantages of the method are that unwinding may increment the input size of the problem considerably, or isn't possible because the number of loop executions is data dependent. Secondly, the winding process fixes the location of pipeline stages, which may exclude the optimal loop-pipelined schedule from the search space.

In [Leis91] retiming is used to change the position of pipeline stages in logical circuits. In [Fran94] the main idea of retiming to change the location of pipeline boundaries inside ASCIS data-flow graphs has been implemented. It is based on the Bellman-Ford all-pairs longest-path algorithm, resulting in an $O(|V|^3 \cdot \log|V|)$ algorithm, with $|V|$ the number of operations inside a data-flow graph.

The main disadvantage of graph transformations before scheduling is that it may place delay nodes at the wrong places, such that it reduces the design space of a scheduler in such a way that it might exclude the optimal throughput-constrained schedule (which is the schedule with the lowest resource allocation). In Figure 5.9 two examples in which this problem becomes obvious can be found. In both examples the minimal distance between process invocations is 3 cycle steps (assuming unity delay for each operator).

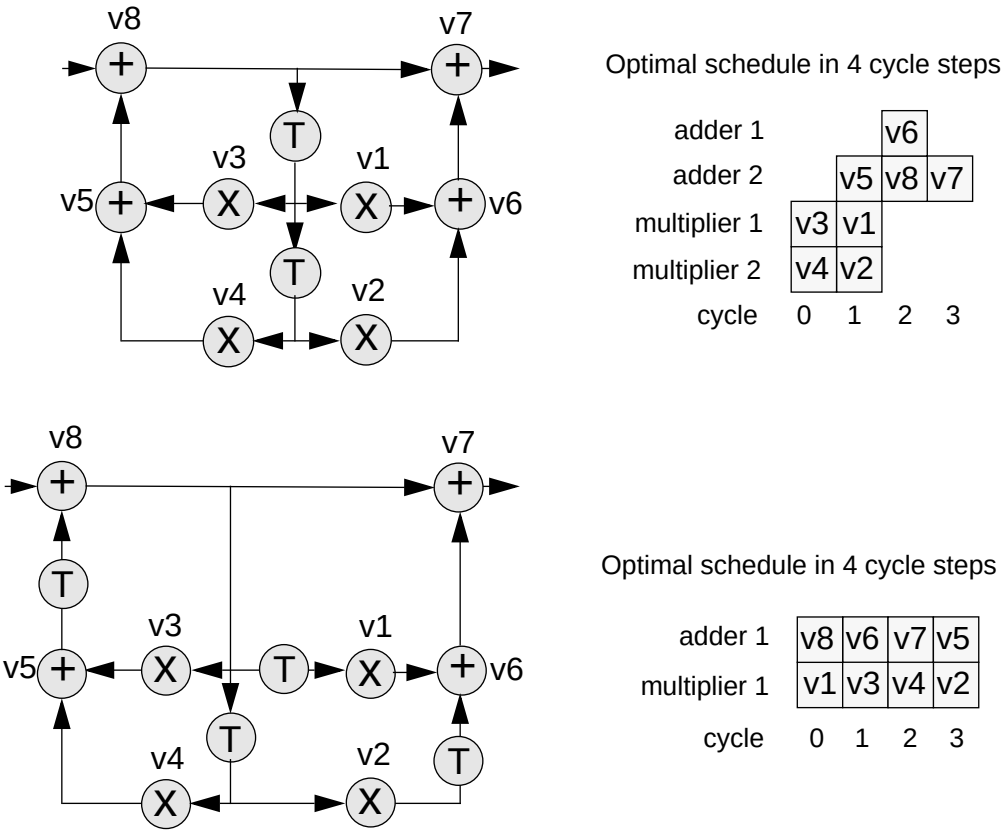


Figure 5.9 Different retimings and their optimal schedules.

In the second example one multiplier and one adder less are needed to obtain an optimal schedule, assuming 4 cycle steps are available for scheduling.

In [Potk91] an algorithm is presented to apply retiming in such a way that it balances the resource utilization (the ratio of the number of cycle steps a resource is exploited over the total number of available cycle steps). Because solving this retiming problem optimally has been proven to be an NP-hard problem, heuristics are used, which try to balance the resource allocation over the available cycle steps. Heuristic measures, such as the mobility of an operation, the probability of resource sharing, and critical path length are used as an object function for retiming. Nevertheless, the method cannot guarantee that it may exclude the optimal schedule solution from the search space of a scheduler.

Loop winding during scheduling

In [Goos89,Hwan91a,Lee92,Chao93,Wang93], loop winding is integrated into the scheduling procedure itself. Depending on partial schedule results, operations are moved to previous or successive pipeline stages in such a way that a more efficient resource utilization or a smaller data introduction interval become possible.

Despite the fact that these transformation algorithms in combination with scheduling might lead to better results, the main disadvantage of these methods is that the movement of operations to other pipeline stages is performed using heuristics, and also

depends on the kind of scheduler used. This means that because of their greed, these methods may reduce the search space in such a way that the optimal solution is excluded from the search space.

5.11.4 Cyclic scheduling

Based on the calculation of the schedule ranges of operations, constructive algorithms as reported in Section 5.6 and 5.8 can be used to produce pipelined schedules. A permutation of operation determines the order in which operations are scheduled, and the `Select` procedure determines the cycle step in which these operations are scheduled. The precedence constraints, time constraints, and throughput constraints of the loop structure are accounted for by updating the schedule ranges of operations.

When topological scheduling techniques are applied to cyclic structures directly, it searches for operations for which all predecessors have been scheduled. Assuming that delay nodes contain initial tokens, the initial set of unscheduled operations consist of the set of successor operations of each delay node and the input nodes. The search space of scheduling in such a case is equal to the single iteration model reported in Section 5.11.1, severely restricting the search space for scheduling.

The method presented in Section 5.10 for creating pipelined schedules can also be applied to cyclic loop structures. A permutation is used to determine both the pipeline stage assignment and the schedule of operations of the data-flow graph. Such a strategy allows operations besides successor operations of delay nodes to be scheduled at the top of pipeline stages, but according to Algorithm 5.17 is restricted to those operations for which deferring to successive pipeline stages is sensible. Although optimal results are not guaranteed, empirical results presented in Section 6.9 show that the approach produces optimal solutions in all cases tested, which validates the use of such a strategy.

5.12 Conclusions

In this chapter a classification of constructive high-level synthesis scheduling methods is presented, based on permutations of operations. This classification shows some advantages of constructing a schedule in a topologically sorted way, because in many cases it prevents the creation of infeasible solutions, and the search effort can be oriented towards finding good quality solutions instead of finding feasible solutions.

A topological schedule constructor, assigning operations to the first cycle step in which no resource conflicts arise, has been proven to contain the optimal schedule solution inside its search space. By application of special data structures, such as heaps and array implementations of doubly linked lists, the complexity to construct a schedule from a permutation in a topological manner equals $O(|\Pi| \cdot \log|\Pi|)$, where $|\Pi|$ denotes the length of the permutation.

Furthermore it has been shown that the constructive creation of pipelined schedules or cyclic schedules is a more complicated task to perform. A new strategy to incorporate retiming into the scheduling process has been presented. Despite the fact that optimal results are not proven to be part of the search space, it provides a way to search for good quality schedules in a more global way when compared to strategies which perform retiming before scheduling is started.

The methods presented in this chapter can be considered as an engine to translate a permutation of operations into a schedule. The search for an optimal schedule is defined as finding a permutation for which the scheduling method will return an optimal solution. Simple heuristic strategies, some of which have been presented in this chapter, often fail to find good quality solutions. In the next chapter a probabilistic search method, called genetic algorithms, will be used to search for permutations, resulting in better quality schedules.

Chapter 6 Genetic Algorithms and Scheduling

6.1 Introduction

In the previous chapter various algorithms have been examined to construct schedules from permutations. A classification of constructive schedulers has been made, emphasizing the feasibility and greed of the algorithms constructing the schedule. The question about how to find permutations which result in optimal or near-optimal schedules hasn't been explored yet, and will be the main topic of this chapter.

In general, exhaustive search takes too much computation time and heuristics provide poor quality results. The choice for genetic algorithms is based on the fact that they have been successfully applied to many combinatorial optimization problems (for an overview see [Mich92]), and are assumed worth considering for solving the scheduling problem.

This chapter first presents a short introduction about genetic algorithms. Then a statistical analysis about the convergence of genetic algorithms is presented. After that, a new encoding of the resource constrained scheduling problem is presented, accompanied with some benchmark results. Subsequently the genetic algorithm is extended with the possibility to allocate extra resources, resulting in a time constrained scheduler. Finally, results are presented with respect to scheduling cyclic data-flow graphs.

6.2 Introduction to genetic algorithms

Genetic algorithms [Holl75] are probabilistic search algorithms which are inspired on the principle of "survival of the fittest", derived from the theory of evolution described by Charles Darwin in *The Origin of Species*. Genetic algorithms maintain a collection of potential solutions, which evolve according to a measure reflecting the quality of solutions.

The evolution process of a genetic algorithm works on an encoding of the search space, represented by a chromosome.

Definition 6.1 (Chromosome χ). Let A be an alphabet (in other words a set of symbols). A chromosome χ is a string of symbols from alphabet A. The number of symbols of χ is called the length of χ, denoted by $|\chi|$. The set A^l, with $l \in \mathbb{N}$, consists of all possible chromosomes χ with $|\chi| = l$. $\chi(i)$ denotes the i^{th} symbol, with $0 \leq i < l$ of chromosome χ.

Definition 6.2 (Encoding *Enc*, decoding *Dec*). Let (*F, c*) be a search problem. Let A^l be a set of chromosomes. The onto[1] function $Dec: A^l \rightarrow F$ is called a decoding. The function $Enc: F \rightarrow A^l$ is called an encoding. The encoding $Enc(f)$ of an element $f \in F$ is defined as an element of $\{\chi \in A^l \mid Dec(\chi) = f\}$. Hence, for each element $f \in F$, one or more encodings $\chi \in A^l$ exist. If for all $f \in F$ the set of possible encodings consists of exactly one element, the encoding is called one-to-one.

Classical genetic algorithms as described in [Holl75] use bit-strings as encodings, in other words alphabet $A = \{0 , 1\}$. In other publications alternative encodings are proposed, consisting of arbitrary symbols (for example natural numbers, or nodes of a graph). Also the length of the chromosomes might not be a constant. In [Koza92], chromosome representations are even extended to graph structures.

To make the process of evolution possible, a distinction between “more fit” and “less fit” chromosomes is needed. This is accomplished by assigning a fitness value to each chromosome, which is associated to the cost $c(f)$ of a candidate solution $f \in F$ of a combinatorial optimization problem (F , c).

Definition 6.3 (Fitness *s*, scaling function Σ). Let (F , c) be an instance of a combinatorial optimization problem. Let A^l be a set of chromosomes, and let $Dec: A^l \rightarrow F$ be an onto function. The fitness $s: A^l \rightarrow \mathbb{R}$ is a function, with $s(\chi)$ the fitness (or score) of chromosome $\chi \in A^l$. Fitness *s* is related to cost function *c* by use of a scaling function $\Sigma: \mathbb{R} \rightarrow \mathbb{R}$, given by $s(\chi) = \Sigma(c(Dec(\chi)))$.

In many cases the scaling function equals the identity function, and hence the fitness value equals the cost of the original combinatorial optimization problem. Alternative scaling functions and their effect on the evolution of genetic algorithms will be explained in Section 6.4.

During the run of a genetic algorithm, it keeps track of a collection of chromosomes, called a population.

Definition 6.4 (Population *P*, population size $|P|$, individuals). A population *P* is a bag (also called collection), the elements of which are taken from the set of chromosomes A^l. The elements of *P* are called individuals. The size of the population *P*, denoted by $|P|$ is called the population size of *P*.

In a genetic algorithm the initial population P_0 is created by randomly selecting $|P_0|$ individuals from the set of chromosomes A^l. A genetic algorithm iteratively tries to improve the average fitness of a population by the construction of new populations, using selection and recombination mechanisms.

1. each element of *F* is the image under *Dec* of some element of A^l

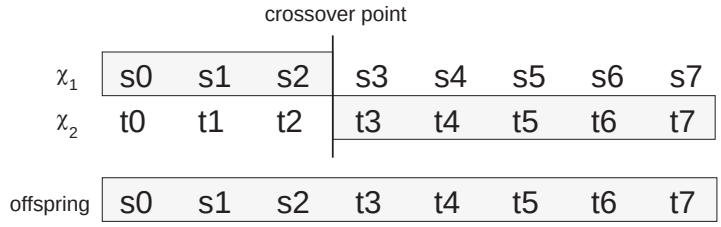


Figure 6.1 Example of single point crossover.

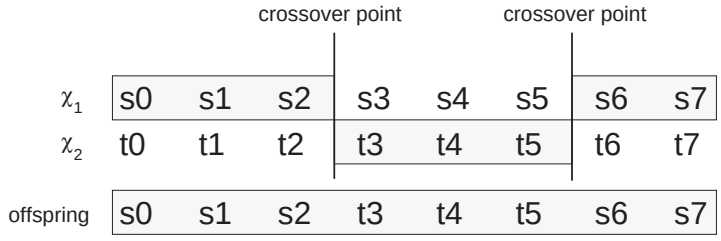


Figure 6.2 Example of 2-point crossover.

Recombination of individuals is performed by so-called operators. An operator accepts a set of chromosomes (sometimes called parents), and constructs new chromosomes (called offsprings or children) by copying information from the parents.

Definition 6.5 (Operator O). An operator O is a mapping O: $(A^l)^m \rightarrow (A^l)^n$, with $n,m \in \mathbb{N}$. It accepts m chromosomes (also called parents), and, using a particular mechanism, generates n chromosomes (called children or offsprings).

Many different operators for genetic algorithms have been published. The most popular operators are called crossover and mutation. Mutation takes one chromosome, changes its contents, and returns the modified chromosome as a result. Crossover takes two chromosomes $\chi_1,\chi_2 \in A^l$, exchanges information between these chromosomes to create new chromosomes, and returns one or two chromosomes as a result. Without any loss of generality this thesis assumes that only one offspring is created by crossover. There are many different types of crossover operators, some of which are the following:

- Single point crossover. A crossover-point $k \in \mathbb{N}$ is randomly chosen in the interval $[0 , l - 2]$, and the offspring is defined as (see also Figure 6.1):

$$\chi(i) = \begin{cases} \chi_1(i) \text{ if } i \in [0, k] \\ \chi_2(i) \text{ if } i \in [k+1, l-1] \end{cases}$$

- n-point crossover. Instead of 1 cross-over point, $n \in \mathbb{N}$ crossover points are chosen, with $n \leq l$. A child is constructed by copying symbols, starting from the first parent, and changing to the other parent each time a crossover point is encountered (see also Figure 6.2).

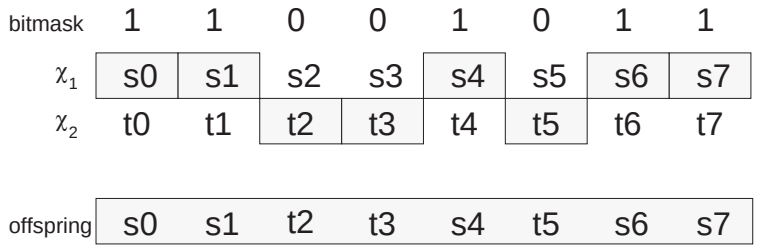


Figure 6.3 Example of uniform crossover.

- Uniform crossover. For each position a bitmask string (a string with the same length as the parents, consisting of '0's and '1's) determines whether a symbol of χ_1 or χ_2 is copied to the same position of the offspring χ. If the value of the bitmask at position i equals 1, then $\chi(i) = \chi_1(i)$, else $\chi(i) = \chi_2(i)$. The values of the bitmask are generated randomly. The probability that the value of the bitmask at a particular position equals 1 is given by the so-called bit-mask probability, denoted by p_{UC} (see also Figure 6.3).

In Section 6.4 more details about operators will be presented.

The selection of parents from a population is performed such that better (above average) individuals have a higher probability to be selected than other individuals. The selection process is a stochastic process, based on the fitness of the individuals of a population.

Definition 6.6 (Selection probability *sel*). Let P be a population, and let $\chi \in P$. The selection probability is a function $sel: P \rightarrow [0\ ,\ 1] \subset \mathbb{R}$, with $sel(\chi)$ the probability that individual χ is chosen from the population as a parent for a particular operator.

A well-known way of performing selection is by using so-called roulette wheel selection (also called proportionate selection), in which for a chromosome χ the selection probability *sel* is defined as follows:

$$sel(\chi) = \frac{s(\chi)}{\sum_{x \in P} s(x)} \qquad (6.1)$$

A template for a genetic algorithm can be found in Algorithm 6.1.

Definition 6.7 (Generation). The population at the i^{th} iteration of a genetic algorithm, denoted by P_i, is called the i^{th} generation of population P.

Algorithm 6.1 (Genetic Algorithm Template).

```
i = 0;                                          // generation count
P_i = 'bag of random individuals';              // initialize population
while !('stop criterion is met') do
  i = i + 1;
  while (|P_i| < |P_i-1|) do
    Operator = Select(Operators);               // Select an operator
    Parents = Select(P_i-1,Operator);           // Select sufficient
                                                // individuals for operator
    Children = Apply(Operator,Parents);         // Create new individuals
    P_i = P_i ∪ Children;                       // Add to current population
  endwhile;
endwhile;
return best solution found;                     // Return result
```

Definition 6.8 (Average score of population). Let P_i denote the i^{th} generation of a genetic algorithm, and let s be a fitness function. The average score of population $s(P_i)$ is defined by:

$$s(P_i) = \frac{1}{|P_i|} \cdot \sum_{\chi \in P_i} s(\chi)$$

In [Holl75] a theorem called the *schema theorem* is presented to give a possible explanation about how a genetic algorithm works.

Definition 6.9 (Schema, defining length δ, order *o*). Let A be an alphabet consisting of symbols, excluding don't care symbol '*'. A schema is a string consisting of symbols from the alphabet $A \cup \{*\}$. Let $H \in (A \cup \{*\})^l$. Let δ: $(A \cup \{*\})^l \rightarrow \mathbb{N}$ be the defining length of a schema, with $\delta(H) = j - i$, with $i = \text{MIN}(k \in [0 , l - 1] \mid H(k) \in A)$, i.e the first symbol element of alphabet A in H, and $j = \text{MAX}(k \in [0 , l - 1] \mid H(k) \in A)$, i.e. the last symbol element of alphabet A in H. Let o: $(A \cup \{*\})^l \rightarrow \mathbb{N}$ be the order of a schema, with $o(H) = \#(k \in [0 , l - 1] \mid H(k) \in A)$, in other words the number of symbols in H element of alphabet A.

Definition 6.10 (Contain). Let A be an alphabet, with '*' $\notin A$. A chromosome $\chi \in A^l$, with $l \in \mathbb{N}$, is said to contain schema $H \in (A \cup \{*\})^l$, denoted by $\chi \in H$, if and only if:

$$\forall_{i \in [0 , 1 , \ldots , l - 1]} (\chi(i) = H(i) \vee H(i) = \text{'*'})$$

in other words χ can be obtained from H by substituting symbols from A for the don't care '*' in H. Each chromosome of length l contains 2^l schemas.

Definition 6.11 (Average score of schema). Let $H \in (A \cup \{*\})^l$ be a schema. The average fitness of the individuals in a population P containing schema H is defined as:

$$s(H,P) = \frac{1}{|\{\chi \in P | \chi \in H\}|} \cdot \sum_{\chi \in P \wedge \chi \in H} s(\chi)$$

Let χ be a chromosome which contains a schema H. If χ is selected for recombination, the resulting offspring might or might not contain schema H. If a schema is destroyed during recombination, it is said to be disrupted. The probability that an operator disrupts a schema is denoted by p_{dis}.

Let $N(H, P_i)$ denote the number of individuals $\chi \in P_i$, containing schema H. Let $E[N(H, P_i)]$ denote the expected number of individuals containing schema H in population P_i. Assuming that individuals are selected proportionally to their fitness, the following relation can be derived, called the schema theory [Gold89]:

$$E[N(H, P_{i+1})] = E[N(H, P_i)] \cdot \frac{s(H, P_i)}{s(P_i)} \cdot (1 - p_{dis}) \qquad (6.2)$$

Equation (6.2) shows that short low order, above average schema receive exponentially increasing trials in subsequent generations. A genetic algorithm builds new individuals by juxtaposition of building blocks. The building block hypothesis, as presented in [Gold89], claims that juxtaposition of building blocks results in the construction of better individuals. It is concluded that a combinatorial optimization problem should be encoded in such a way that the building blocks are not misleading to the genetic search. In such a case genetic algorithms are assumed to have a good chance of finding good quality solutions.

Despite the successful application of genetic algorithms to many optimization problems, the underlying theory presented in [Holl75,Gold89] doesn't guarantee a certain degree of performance. This can considered to be the main drawback of the use of genetic algorithms. Because of the lack of an underlying theory, little is known about *how* to efficiently apply genetic algorithms to solve combinatorial optimization problems (for example how should operators look like, which selection scheme should be applied, what kind of encoding should be used, how should the stop criterion look like, and how many individuals should a population consist of). Most publications report rather arbitrary choices with respect to the implementation of a genetic algorithm, mainly guided by some empirical results achieved. In Section 6.4 an analytical relation between the statistics of two subsequent populations will be presented, providing some knowledge about how to apply a genetic algorithm to the scheduling problems defined in Chapter 3 as efficient as possible.

6.3 Genetic Algorithms and combinatorial optimization

In the context of combinatorial optimization, genetic algorithms can be considered as probabilistic search algorithms, which try to find an optimal solution. Genetic algo-

rithms are abstracted from problem specific details, and therefore are not limited to a restricted set of problems, and hence fall into the class of so-called general purpose search algorithms. They closely follow the concept of local search strategies, in other words, they search for successive improvements by examination of so-called neighbourhood solutions [Papa82] (see page 62 for more details).

In Algorithm 6.2 a general local search algorithm template as presented in [Papa82] can be found. It starts with a (randomly generated) initial solution $i_{initial} \in F$, and searches for a better solution in a so-called neighbourhood structure. If such a solution exists, it replaces the current solution and the algorithm is repeated using the new solution. The algorithm terminates if no improvements can be obtained.

Algorithm 6.2 (Local search algorithm).

```
i = i_initial;
while (∃_{j ∈ N(i)} c(j) < c(i)) do
  j = Select(N(i));                    // Select a solution j ∈ N(i)
  if (c(j) < c(i)) then
     i = j;
od;
```

A disadvantage of the local search algorithm as presented in Algorithm 6.2 is that the quality of the solution obtained usually depends on the initial solution. An advantage of genetic algorithms is that they are less sensitive to the initial solution, by working on a population of solutions instead of on single solutions. Another important difference between Algorithm 6.2 and genetic algorithms is that subsequent populations are not created based on cost improvements, but stochastic mechanisms are used to select and construct new solutions.

There is no easy mechanism known for a genetic algorithm to determine whether it did find a local or global optimum. If a genetic algorithm is stuck in a local optimum and continues processing, the usage of time and resources can considered to be very inefficient. From this perspective it should be avoided that a genetic algorithm converges too quickly, and might get trapped in a local optimum. If there is no convergence whatsoever, the underlying mechanisms of a genetic algorithm as suggested in the building block hypothesis are missing. Hence there should be a proper balance of convergence, which can be viewed from the perspective of the so-called exploration-exploitation trade-off.

If the correlation of the scores between two subsequent populations is low, the genetic algorithm is called explorative. If the correlation of the scores between two subsequent populations is high, the genetic algorithm is called exploitative. A random search algorithm is highly explorative, while the local search algorithm as given in Algorithm 6.2 can be highly exploitative, depending on the neighbourhood structure. The key mechanisms to control exploration and exploitation in a genetic algorithm are selection and recombination, as will be explained in more detail in Section 6.4.

Many empirical results have been published about how to control the exploration-exploitation trade-off in genetic algorithms (for example by use of alternative selection strategies, operators, cost scaling, parameter tuning, and many more). Because of the lack of theoretical analysis, these results are difficult to generalize, and therefore it is difficult to predict how to apply genetic algorithms to particular combinatorial optimization problems efficiently. In the following sections, theoretical results will be presented which give an indication about how to use genetic algorithms for the scheduling problem.

6.4 Recombination and disruption

A closer look to the crossover mechanisms as presented on page 95 show that they have a different effect with respect to the disruption of schemas. With single point crossover, the probability of disruption increases if the defining length of schemas is higher. Hence, in a genetic algorithm using single point crossover, schemas with a low defining length have higher probability of survival than schemas with a higher defining length. This property of such a crossover strategy is called positional bias. The building block hypothesis doesn't consider the effects of positional bias on the defining length of schemas. A genetic algorithm based on crossover techniques with positional bias, will have a bad convergence if schemas with high defining lengths represent solutions with the highest fitness.

The probability of disruption caused by uniform crossover does not depend on the defining length of a schema, and hence eliminates encoding effects leading to positional bias. Furthermore, the disruptiveness of uniform crossover can be controlled by a single parameter p_{UC} (see also page 96), which leaves the question how to determine p_{UC} for a particular situation.

Let the order $o(H)$ of schema H be equal to k, denoted by H_k. Let $s(H_k , P_i) = a \,.\, E_i[s]$, in which $a \in \mathbb{R}$, with $E_i[s]$ denotes the expected average score of population P_i (see also equation (6.4)). In [Mesm95] it is derived that for uniform crossover $(1 - p_{dis}) = p_{UC}^k$, with p_{UC} the bit-mask probability for uniform crossover, hence equation (6.2) can be rewritten as:

$$E[N(H_k, P_{i+1})] = E[N(H_k, P_i)] \cdot a \cdot p_{UC}^k \qquad (6.3)$$

From this equation it can be concluded that the probability of survival of H_k decreases exponentially with the order k of schema H_k. For a schema H_k to survive, the average fitness of H_k must increase exponentially in k. This shows that uniform crossover is very disruptive, the amount of disruptiveness depending on probability p_{UC}.

In genetic algorithms, disruption of schemas is associated with exploration because the correlation between the cost of parents and offspring is considered to be low. The schema theorem from [Holl75] guarantees that above average schemas grow exponen-

tially in subsequent populations when they are not disrupted in the crossover process. This result has lead researchers to use exploitative genetic algorithms, in order to preserve schemas, and enhance the convergence. Nevertheless experimental results in [Sysw89] show that uniform crossover outperforms single point crossover, and hence exploration seems more desirable. The following section will analyse the phenomenon in more detail.

6.5 Evolution statistics

To obtain an idea about the convergence of a genetic algorithm, it is interesting to derive a statistical relation between the average score of successive populations P_i and P_{i+1}, with $i \in \mathbb{N}$. For this purpose the distribution (or relative frequency) of fitness values in a population is considered in more detail in this section.

Definition 6.12 (Distribution *f*). Let P_i be a population, let S_i be a range of scores in P_i, given by tuple $(\text{MIN}_{c \in Pi}\, s(\chi)\,,\, \text{MAX}_{c \in Pi}\, s(\chi))$. The distribution f_i: $S \rightarrow \mathbb{R}$ is given by: $f_i(s) = \left| \{\chi \in P_i \mid s(\chi) = s\} \right| / \left| P_i \right|$

Some characteristics of a distribution can be summarized by the moment of a distribution. The m^{th} order moment of distribution $f_i(s)$ is defined as:

$$E_i[s^m] = \sum_{s \in S_i} f_i(s) \cdot s^m \quad \text{, also written as} \quad E_i[s^m] = \sum_{s} f_i(s) \cdot s^m$$

The first order moment, also known as the *mean* or *expectation*, is given by:

$$E_i[s] = \sum_{s} f_i(s) \cdot s \tag{6.4}$$

The relation between the expected score and the average score of population P_i is given by:

$E_i[s]$

= {def. (6.4)}

$$\sum_{s \in S_i} f_i(s) \cdot s$$

= {def. distribution}

$$\sum_{s \in S_i} \frac{\left|\{\chi \in P_i \left| s(\chi) = s\right\}\right|}{\left|P_i\right|} \cdot s$$

= {calculus}

$$\frac{1}{|P_i|} \cdot \sum_{s \in S_i} \left| \{\chi \in P_i | s(\chi) = s\} \right| \cdot s$$

= {summation over all individuals separately}

$$\frac{1}{|P_i|} \cdot \sum_{\chi \in P_i} s(\chi)$$

Hence the expected value $E_i[s]$ equals the average score $s(P_i)$ of population P_i.

Let $p_i(s(\chi) = s)$ (shorthand notation $p_i(s)$) be the probability that an offspring χ in population P_i is created with fitness $s(\chi) = s$. The distribution $f_i(s)$ of the fitness values s in population P_i is then given by:

$$f_i(s) = \begin{cases} p_i(s) & \text{if } s = s(\chi) \wedge \chi \in P_i \\ 0 & \text{else} \end{cases} \qquad (6.5)$$

Let's assume that all members of P_{i+1} are offsprings created by using crossover applied on parents x , y, which are selected from P_i. Let $p(x , y)$ represent the probability that individuals x and y are selected from population P_i for crossover, and let $p(s(cross(x , y)) = s \mid x,y)$ represent the probability that crossover generates an offspring $\chi = cross(x , y)$ with fitness $s(\chi) = s$. Then:

$$p_{i+1}(s) = \sum_{x, y \in P_i} p(x, y) \cdot p(s(cross(x, y)) = s | x, y) \qquad (6.6)$$

According to [Mesm95] the first order moment of population P_{i+1} is then given by:

$E_{i+1}[s]$

= {equation (6.6) and (6.5)}

$$\sum_{s} \sum_{x, y \in P_i} p(x, y) \cdot p(s(cross(x, y)) = s | x, y) \cdot s$$

= {calculus}

$$\sum_{x,\,y \in P_i} p(x,y) \cdot \sum_{s} p(s(cross(x,y)) = s \mid x,y) \cdot s$$

= {express explicitly in summation over all chromosomes}

$$\sum_{x,\,y \in P_i} p(x,y) \cdot \sum_{s} \sum_{c \in A^l \mid s(c) = s} p(cross(x,y) = c \mid x,y) \cdot s$$

= {summation over s}

$$\sum_{x,\,y \in P_i} p(x,y) \cdot \sum_{c \in A^l} p(cross(x,y) = c \mid x,y) \cdot s(c) \qquad (6.7)$$

Assume that proportionate selection is used, in other words

$$p(x) = \frac{s(x)}{\sum_{c \in P} s(c)}$$

Let $S_i = \sum_{c \in P_i} s(c)$. Then (6.7) can be rewritten as:

$$\frac{1}{S_i^2} \cdot \sum_{x,\,y \in P_i} s(x) \cdot s(y) \cdot \sum_{c \in A^l} p(cross(x,y) = c \mid x,y) \cdot s(c) \qquad (6.8)$$

In the following step of this analysis, it is assumed that the fitness of each offspring generated equals the average fitness of the parents used (in other words $s(c) = 1/2 \cdot (s(x) + s(y))$). Although this assumption will not be true for a specific pair of parents applied to a specific crossing, the assumption only needs to be valid on the average of all chosen parents. Hence, in contrast to the general belief, it is assumed that crossover doesn't improve the fitness of parents by exchange of schemas. It is only assumed that, on average, crossover doesn't produce below average individuals. By using this assumption, (6.8) can be rewritten as:

$$\frac{1}{2 \cdot S_i^2} \cdot \sum_{x,\,y \in P_i} s(x) \cdot s(y) \cdot (s(x) + s(y)) \cdot \sum_{c \in A^l} p(cross(x,y) = c \mid x,y)$$

$$= \{ \quad \sum_{c \in A^l} p(cross(x,y) = c | x,y) = 1 \quad \}$$

$$\frac{1}{2 \cdot S_i^2} \cdot \sum_{x,y \in P_i} s^2(x) \cdot s(y) + s(x) \cdot s^2(y)$$

= {symmetry}

$$\frac{1}{S_i^2} \cdot \sum_{x \in P_i} s^2(x) \cdot s(y)$$

= {calculus}

$$\frac{1}{S_i^2} \cdot \sum_{x \in P_i} s^2(x) \cdot \sum_{y \in P_i} s(y)$$

= {def. S_i}

$$\frac{1}{S_i} \cdot \sum_{x \in P_i} s^2(x)$$

= {def. S_i}

$$\frac{1}{E_i[s] \cdot |P_i|} \cdot \sum_{x \in P_i} s^2(x)$$

= {def. 2^{nd} order moment, $|P_i|$ and proportionate selection}

$$\frac{E_i[s^2]}{E_i[s]}$$

= {def. $E_i[s^2] = E_i^2[s] + var_i[s]$}

$$E_i[s] + \frac{var_i[s]}{E_i[s]}$$

Hence the progress in fitness between two succeeding populations P_i and P_{i+1} is proportional to the variance in the population and inversely proportional to the average score, described by the following expression:

$$E_{i+1}[s] - E_i[s] = \frac{var_i[s]}{E_i[s]} \qquad (6.9)$$

From this relation, some conclusions can be drawn (endorsed by empirical results published about genetic algorithms):

- Because the expected average score $E_i[s]$ increases for increasing generations i, the factor $var_i[s] / E_i[s]$ will decrease for increasing i. This behaviour will lead to populations of which the score will become more homogeneous, which reduces $var_i[s]$, again leading to a decreasing factor $var_i[s] / E_i[s]$ for increasing i. This means that for increasing generations the increase in expected score will decline, leading to convergence.

- The increment of the expected score of population P_{i+1} is inversely proportional to the average score $E_i[s]$ of population P_i. If $E_i[s]$ is very high with respect to $var_i[s]$, then the increment in score is expected to be rather poor.

  A possible way to obtain low average scores, without messing up with the search towards better solutions, is to apply cost scaling (see also Definition 6.3). Let (F, c) be an instance of a combinatorial optimization problem. Let χ be an element of the encoding of F. Let $\Sigma(\chi)$ be a scaling function, given by the identity function, in other words $s(\chi) = c(Dec(\chi))$, and let $E_i[s]$ be the average score of population P_i, using Σ as scaling function for s. Let $c_{min} = \mathrm{MIN}_{f \in F}(c(f))$ be the minimal cost of an instance of a combinatorial optimization problem (F, c). Let $\Sigma'(\chi) = \Sigma(\chi) - \beta$, with $0 < \beta < c_{min}$ (hence if $c(f) > 0$ for all $f \in F$, then $\Sigma'(Enc(f)) > 0$), and let $E_i'[s]$ be the average score of population P_i, using Σ' as scaling function for s'. In that case $E_i'[s] = E_i[s] - \beta$ and $var_i'[s] = var_i[s]$, and hence $var_i'[s] / E_i'[s]$ is larger than $var_i[s] / E_i[s]$, resulting in a larger expected score for the next generation. This agrees with empirical studies done on the effect of cost scaling to improve the convergence behaviour of genetic algorithms.

- The increment of the expected score of population P_{i+1} is proportional to the variance $var_i[s]$ in the current population P_i. The variance depends on the kind of crossover mechanism used, because proportionate selection tends to diminish the variance of successive populations, which results in a bad long-term effect with respect to the

average score. Hence a crossover operator should be designed such that it maximizes the variance of the next generation instead of just maximizing the average score.

A high variance in score doesn't necessarily imply the need for large stochastic variations in search (i.e. a search which closely approaches randomness). If optimal solutions are characterized by a schema consisting of low order building blocks, this imposes the need for much disruptiveness. If optimal solutions are characterized by a schema consisting of high order building blocks, these can only be obtained and maintained in a population when disruption is not too severe. It is assumed that uniform crossover is the most disruptive crossover mechanism known. The amount of disruptiveness can be controlled by a single bitmask parameter p_{UC} as has been shown in Section 6.4. The question is how p_{UC} should be adapted with respect to the statistics of the population.

In [Mesm95] the linear all one problem is used to derive some statistical analysis on how to adapt p_{UC} with respect to the statistics of the population.The linear all one problem is a problem in which a chromosome containing many 'ones' has higher fitness than chromosomes containing less 'ones'. The optimal solution is the chromosome consisting of only 'ones'. The linear all one problem is characterized by low order schema consisting of strings containing all ones, and the building block hypothesis is clearly applicable to this problem. It is proven that for this problem the optimum bitmask parameter p_{UC} equals 0.5, and hence it is independent of the population statistics. From this result it can be concluded that uniform crossover leads to exactly the correct amount of variation if it is set to its most disruptive behaviour. It is likely that this conclusion holds in general for all problems which are characterized by low order schema. Problems which are not characterized by low order schema have a higher risk to get trapped in local optima. For these kinds of problems it is expected that even more exploration is needed, in other words for these problems the amount of disruption that can be achieved by uniform crossover should be set to its maximum value, hence $p_{UC} = 0.5$.

- If the population size is very small, the variance among individuals in successive populations will be small too.

From these observations it can be concluded that the trade-off between exploration and exploitation mainly depends on the kind crossover mechanism used. The selection mechanism of the genetic algorithm (exploitation) can be considered as the process which increases the average score, but will lead to a decrease of the value of the variance. The selection pressure should be balanced with disruptive crossover (exploration) to achieve an acceptable amount of variance. Analysis performed on the linear all one problem shows that uniform crossover using maximum disruptiveness is expected to give best performance.

Table 6.1 lists the results for the application of a genetic algorithm to a particular scheduling problem (topological scheduling using the fast discrete cosine transform example

with a time constraint of 14 cycle steps; for details see Sections 6.6.7 and 6.7), using different rates of recombination and different population sizes. The results shown in this table are typical for numerous of examples that have been tested.

A random generator used by a genetic algorithm can be initialized by a seed value, shown in the first column of the table. A random generator which is initialized by the same seed value will produce the same sequence of random numbers. For different seed values a random generator will in general produce different sequences of random numbers. Hence for each different seed-value, the initial population will in general contain a different set of individuals. The number of generations needed to reach an optimal solution is shown in the following five columns, given the population size (100 and 50) and the amount of offsprings created by different operators. In the second column (100/100UC) the results are given in case uniform crossover is used to create all offsprings. In the last row the (ceiling of the) average number of generations needed to generate an optimal solution is given. The third column (100/100SP) shows the results when single point crossover is used to generate all offsprings. The maximum number of generations equals 100, and the table shows that, using this bound, the optimal solution is not found in many cases. Comparison of single point crossover and uniform crossover shows that a genetic algorithm using uniform crossover produces optimal solutions much faster. The fourth column (100/classical) shows the results when using a 'classical' genetic algorithm, in which operators like copy (just copy the selected individual to the new population without changing its contents), mutate (select two positions, and swap the corresponding elements), invert (select two position, and mirror the elements between these two positions), and uniform crossover are used. The rate at which these operators are applied is determined empirically, and has resulted in a stochastic distribution of copy/mutate/invert/cross given by 50%/4%/6%/40%. Although the results are better than those of single point crossover, on average twice as many generations are needed as compared to a strategy in which uniform crossover creates all its offsprings (see the second table column labelled 100/100UC). Almost similar results are obtained when using a 50%/50% distribution for copy and uniform crossover (see the table column labelled 100/50UC), making the use of specialized operators such as mutation and inversion debatable. The sixth column shows the results obtained when the population size is decreased from 100 to 50 (see the fifth column labelled 50/50UC). The column shows that in some cases the optimal solution isn't found within 100 generations. Comparing column 6 to column 2 and 5 shows that a population size of 50 is too small for this problem. The last column shows the results obtained when individuals are created randomly. After the creation of 10.000 'individuals' the algorithm is terminated. An 'x' in the table denotes that no optimal result has been found, while an 'o' in the table means that an optimal result has been found.

The average number of generations needed to generate an optimal solution is the smallest in case uniform crossover is used to generate all offsprings, and comparisons with other strategies show that this strategy is very fruitful indeed.

Table 6.1 Number of generations needed to find an optimal solution.

seed value	population size / cross rate					
	100/100UC	100/100SP	100/classical	100/50UC	50/50UC	random search
1	10	42	8	3	12	x
2	2	7	14	18	12	x
3	14	> 100	23	11	11	x
4	10	> 100	14	27	>100	x
5	6	19	27	18	5	x
6	11	> 100	13	18	13	x
7	5	7	11	20	7	x
8	8	10	12	14	28	x
9	10	10	18	20	5	o
10	7	> 100	12	4	17	x
100	7	> 100	22	22	5	x
123	6	> 100	13	33	24	o
145	4	> 100	5	9	> 100	x
167	15	> 100	20	24	5	x
190	11	14	19	23	> 100	x
200	7	9	14	8	1	x
1001	4	> 100	29	9	55	x
1300	13	> 100	21	25	18	x
2344	6	> 100	7	17	11	x
5689	8	> 100	19	15	11	x
9453	11	> 100	17	5	14	x
‘average’	8	> 67	16	17	> 26	-

This leaves us with the problem how to encode a search problem, in specific the time constrained scheduling problem.

6.6 Scheduling encodings

In this section the relation between an encoding of a schedule and the way a schedule is constructed will be investigated. The section starts with very straightforward encodings, some disadvantages of these encodings are pointed out, and suggestions are used to overcome these disadvantages resulting in new and better encodings.

6.6.1 Classic bit-vector encoding

In classical genetic algorithms the encoding alphabet $A = \{0 , 1\}$. In that case, chromosomes consist of bit vectors. Although such an encoding may work fine for some kind of problems, they may introduce efficiency problems for other kind of search problems.

An encoding of a schedule must describe how operations are assigned to cycle steps. Let $V = \{v_0 , v_1 , ... , v_{n-1}\}$ be the set of operations to be scheduled, and let $T = \{0 , 1 , ... , T_{max} - 1\}$ be the range of cycle steps available for operations to be scheduled.

To encode operations from V, at least $\lceil \log_2 n \rceil$ digits are needed. This means that at least $2^{\lceil \log 2\, n \rceil} - n$ strings represent infeasible operations. While running the genetic algorithm, the construction of these infeasible strings needs to be avoided, implying restrictions with respect to the construction of offsprings, and hence on the initialization of the population and on the operators involved. It is difficult to predict the effects on the quality of the search under these kind of restrictions. Furthermore, an encoding of a schedule contains each operation exactly once, and extra care is needed to maintain this property for offsprings created by application of operators to individuals. A similar analysis can be given for the encoding of the range of cycle steps. One can think of a genetic algorithm in which infeasible encodings are accepted, but a genetic algorithm spending large amounts of computation time generating and evaluating infeasible encodings can hardly considered to be efficient, and should therefore be avoided.

A feasible encoding can be achieved by using the elements from V and T directly. This still leaves many encodings to be possible, some of which will be discussed in the following subsections.

6.6.2 Cycle assignment encoding

Another straightforward encoding is a sequence τ consisting of n elements from T, in which $\tau(i)$ denotes the schedule time $\varphi(v_i)$ for each $v_i \in T$. The disadvantage of such an encoding is that it includes infeasible schedules with respect to precedence constraints ($u \prec$ v and $\varphi(u) + distance(u , v) > \varphi(v)$). The encoding shows similar problems as with the binary encoding, in other words, preventing encodings of infeasible schedules is difficult to achieve during the run of the genetic algorithm. A genetic algorithm based on such an encoding would result in an inefficient search strategy.

6.6.3 Absolute displacement encoding

In [Wehn91] an encoding is presented which assigns to each operation $v \in V$ an absolute displacement $d_a(v) \in \mathbb{N}$. The schedule $\varphi(v)$ of operation v is determined by:

$$\varphi(v) = asap(v) + d_a(v)$$

The asap and alap values of other operations are updated after scheduling a particular operation. The advantage of encoding a schedule in terms of absolute displacements instead of encoding a schedule directly in terms of elements of T is the fact that each encoding represents a feasible schedule with respect to the precedence constraints. A closer look at this encoding shows that the displacement of operations in the critical path has a large impact on the completion time of the schedule. In [Wehn91] special routines are presented to construct an initial population exhibiting schedules. For each

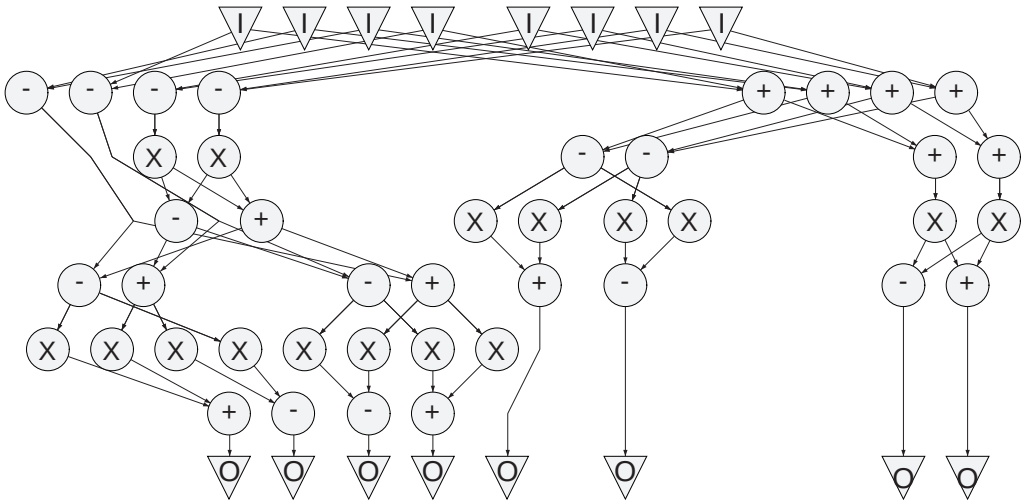


Figure 6.4 Fast Discrete Cosine Transform Filter Example.

path p in the data-flow graph, a particular amount of cycles, called the global displacement $\Delta_a \in \mathbb{N}$, are distributed among the operations of p in the following way:

$$\sum_{v \in p} d(v) \leq \Delta_a$$

An improvement of the quality of the results is reported, however no attention has been paid to adapt operators such that distribution of the global displacement is preserved during the run of the genetic algorithm. This is confirmed by our own experiments, in which a time constrained method derived from the method originally presented in [Wehn91] has been tested (see [Jaco94] for details). Originally, the method does not address the problem of meeting constraints, but searches for a trade-off between the resource allocation and the completion time. In the time constrained method, experiments show that only a few offsprings represent feasible schedules with respect to the time constraint. Hence the genetic algorithm spends a lot of time in creating and evaluating infeasible solutions, and hence in the original algorithm of [Wehn91] explores significantly more schedules with a large completion time than with a short completion time. The use of penalty functions to favour feasible solutions is questionable, because the search might be trapped in local optima. The method has been implemented and tested, and some results for different time constraints can be found in Table 6.2 and Table 6.3 In these tables the resource allocation is specified by the number of adders and multipliers, assuming that a multiplier requires 2 cycle steps for a multiplication and an adder requires 1 cycle step for an addition. An ‘X’ in the tables means that no feasible schedule with respect to the time constraint could be found. The first example, shown in Figure 6.4, is the fast discrete cosine transform taken from [Mall90], and the second example shown in Figure 6.5, is the wave digital filter taken from [DeWi85].

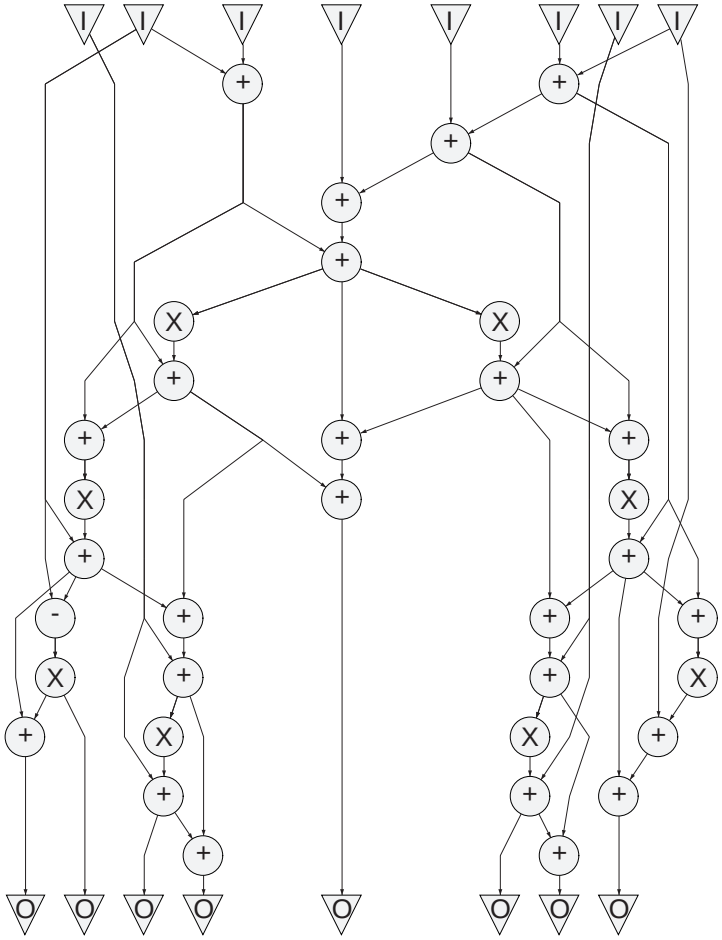


Figure 6.5 Wave Digital Filter Example.

From these tables it is clear that the method based on absolute displacements fails to find feasible solutions in many cases.

Table 6.2 Encoding results for Wave Digital Filter.

Time constraint	Optimal		Absolute displacement		Relative displacement	
cycles	# mult	# add	# mult	# add	# mult	# add
17	3	3	3	3	3	3
18	2	2	3	3	2	3
19	2	2	2	3	2	3
20	2	2	2	2	2	2
21	1	2	X	X	2	2
22	1	2	X	X	1	2
23	1	2	X	X	2	2
24	1	2	X	X	1	2
25	1	2	X	X	1	2
26	1	2	X	X	1	2
27	1	2	X	X	1	2
28	1	1	X	X	1	1

Table 6.3 Encoding results for Fast Cosine Transform Filter.

Time constraint	Optimal		Absolute displacement		Relative displacement	
cycles	# mult	# add	# mult	# add	# mult	# add
8	8	4	8	4	8	5
9	8	4	9	4	8	4
10	5	4	X	X	5	4
11	4	3	5	6	5	4
12	4	3	X	X	4	4
13	4	2	X	X	4	4
14	3	2	X	X	4	3
15	3	2	X	X	3	4
16	3	2	X	X	3	3
17	3	2	X	X	3	4
18	2	2	X	X	3	3
19	2	2	X	X	3	3
20	2	2	X	X	3	2
21	2	2	X	X	3	3
22	2	2	X	X	3	2
23	2	2	X	X	3	2
24	2	2	X	X	2	3
25	2	2	X	X	2	2
26	2	1	X	X	2	3
27	2	1	X	X	3	2
28	2	1	X	X	2	3
29	2	1	X	X	2	3
30	2	1	X	X	2	3
31	2	1	X	X	2	2
32	2	1	X	X	2	2
33	2	1	X	X	2	2
34	1	1	X	X	2	2

6.6.4 Relative displacement encoding

One way to prevent the creation of infeasible schedules with respect to precedence constraints and time constraints (specified by T_{max}) is to use an encoding based on relative displacements. In this encoding, to each operation $v \in V$ a relative displacement value $d_r(v) \in [0 , 1]$ is assigned, which contains at least T_{max} finite numbers. Operations are selected in a particular (fixed) order, and the schedule $\varphi(v)$ of operation v is determined by:

$$\varphi(v) = asap(v) + \lceil d_r(v) \,.\, (alap(v) - asap(v) - \delta(v)) \rceil$$

If the asap and alap values of operations are updated after scheduling an operation (see equation (4.5) and (4.6)), feasibility with respect to both precedence constraints and time constraints is guaranteed while constructing the schedule.

In Table 6.2 and Table 6.3 some schedule results based on the relative displacement encoding can be found. The results are rather disappointing, because in many cases non-optimal solutions are found. An extension of the encoding, in which the order operations are scheduled is exchanged (encoded by a permutation Π of operations), has been incorporated into the schedule encoding. This hasn't resulted in any substantial change in the quality of the solutions generated.

A possible explanation for the failure of the algorithm is the lack of so-called problem specific knowledge during decoding chromosomes. The method for example doesn't prevents 2 additions being scheduled simultaneously, even if both operations have large schedule ranges. Strategies to increase the performance of the schedule, by rescheduling operations after decoding, might incidentally produce better solutions, but it is hard to predict whether the optimal solution can be reached at all with such a strategy.

6.6.5 Permutation encoding

To characterize the problem of the relative displacement encoding more clearly, suppose that a part of a data-flow graph has been scheduled. Suppose that the relative displacement decoding decides to place an operation in parallel to another operation, increasing the resource allocation induced by the new partial schedule. At this specific moment it is unknown whether the partial schedule, inducing an increment of the resource allocation, is part of an optimal solution. To get more information about the optimal solution, the final resource allocation ought to be known, but this is the result searched for!

This brings us back to Section 4.6, in which the close relationship between time constraints and resource constraints has been presented. If for a time constrained (resource constrained) scheduling problem a lower bound estimate of the minimal resource allocation (completion time) is known, this lower bound can be used to decide about local schedule choices. Another important aspect of such a lower bound is that when a solution has been found meeting this bound, the solution is an optimal solution by definition, hence it provides the genetic algorithm with a very accurate stop criterion.

The decision whether operations should be deferred in time, depends on the fact whether resources are available in a particular cycle step. From the proof given in Section 5.8 it is known that an operation $v \in V$ can be assigned to the first cycle step $c \geq asap(v)$ where an appropriate resource is available, without excluding the optimal solution from the search space. Hence the use of an encoding using displacements seems to be unnecessary in this case. Rather, a schedule can be encoded by a permutation Π of operations from V (see Algorithm 5.14).

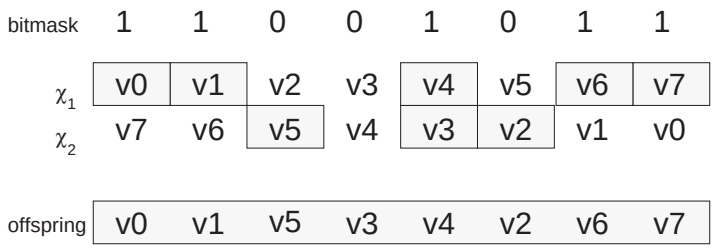


Figure 6.6 Uniform crossover for permutations

Operators should be able to construct new permutations from existing permutations. In [Star91] an overview of crossover operators dealing with permutations is given. In [Sysw91] an uniform crossover operator for permutations is given. Using a bitmask, it selects several positions in one parent, and copies the operations at these position to the same position of the offspring. Operations that haven't been copied yet are copied in an order-preserving way from the other parent, filling the empty positions of the offspring (see Figure 6.6 for an example).

Algorithm 5.4 on page 69 can be used to decode a permutation which tries to satisfy both time constraints and resource constraints. Given a permutation, this algorithm might be aborted, because no resource is available in the range of cycle steps in which an operation can possibly be scheduled. Instead of aborting the scheduling procedure, one can try to schedule as many operations as possible, and keep track of the number of unscheduled operations (Algorithm 6.3). The number of unscheduled operations can serve as a cost function. If a schedule with no unscheduled operations has been found, an optimal solution has been found, and the scheduling procedure can be stopped.

Algorithm 6.3 (Feasible constrained permutation scheduling).

```
u = |Π|;                                    // # unscheduled operations
for i = 0 to |Π| - 1 do
  v = Π(i);                                 // select operation
  C = selectCycles(v,asap(v),alap(v));      // determine cycle steps in which
                                            // resource for v are free
  if (C <> ∅) then φ(v) = MIN(C);           // select cycle step from C
                   u = u - 1;
                   update schedule ranges;
                   update resource usage;
  endif;
endfor;
```

One point of concern is whether the encoding provides enough information for the genetic search. When an operation is scheduled such that it fixes other operations, a large part of the permutation is not investigated. This means that the order information of these operations is of no value to the schedule produced so-far, and the emphasis lies on the first elements of the permutation. It is difficult to see how the exchange of non-investigated parts of the permutation can lead to successful crossover mechanisms with a high variance in fitness for the succeeding population.

Another point of concern is the fact that the lower bound resource allocation might be estimated wrongly (in other words, too few resources are available, and hence a feasible schedule does not exist). In that case the resource allocation needs to be increased. More details about how to increase the lower bound resource allocation during scheduling can be found in Section 6.7.

6.6.6 Permutation encoding and list scheduling techniques

To avoid the creation of infeasible schedules, list scheduling techniques as described in Section 5.8 can be used. The resource allocation is performed less greedy when compared to the method in the previous section, and when relaxing the time constraint the genetic algorithm is given the opportunity to explore the information contained by the whole permutation.

A possible way of performing constructive scheduling topologically is by using the permutation as a priority list for a list scheduler [Heij95a]. The completion time of the schedule can be used to determine the fitness of a chromosome. The genetic algorithm will search for a priority function, which in combination with the list scheduler results in the smallest completion time. The results of an implementation based on this mechanism can be found in Table 6.4 and Table 6.5.

Table 6.4 Encoding results for Wave Digital Filter.

Resource constraint		Optimal	Genetic List	Ordinary List	Topological
# mult	# add	cycles	cycles	cycles	cycles
3	3	17	17	17	17
2	2	18	18	19	18
1	2	21	21	21	21
1	1	28	28	28	28

Table 6.5 Encoding results for Fast Discrete Cosine Transform.

Resource constraint		Optimal	Genetic List	Ordinary List	Topological
# mult	# add	cycles	cycles	cycles	cycles
8	4	8	8	8	8
5	4	10	10	10	10
4	3	11	11	13	11
4	2	13	13	15	13
3	2	14	14	17	14
2	2	18	18	21	18
2	1	26	26	27	26
1	1	34	34	40	34

In all cases the genetic algorithm based on a list scheduler finds the optimal results. Comparison with an ordinary list scheduler, using the critical path as a priority function, shows that the genetic search improves the quality of the results obtained. The run times of the scheduler are between 0.1 and 20 seconds for each entry of the tables using an HP9000/735 computer.

In [Ahma95,Dhod95] permutations are encoded by assigning integer numbers to operations. Operations can be converted into a permutation by sorting the operations using the integer numbers as sorting key. It is claimed in these articles that such an encoding is closer towards the principles of classical genetic algorithms, but there is no clear explanation about the advantages of such an encoding. The methods use roulette wheel selection in combination with single-point crossover, and mutation at a very low rate. The first population is initialized with solutions created by some well-known heuristics, and individuals are derived from these solutions by applying special kind of mutations. It is unclear how these methodologies affect the variance and average score of the population, and whether the selection pressure leads to locally optimal results. Only a few results are published, which are given in Table 6.6 and Table 6.7.

Table 6.6 Comparison Wave Digital Filter.

Resource constraint			Optimal	[Dhod95]	Ordinary List	Genetic List
# mult	# add	pipelined mult.	cycles	cycles	cycles	cycles
3	3	-	17	17	17	17
2	2	-	18	19	19	18
1	2	-	21	21	21	21
-	3	2	17	17	17	17
-	3	1	18	18	19	18
-	2	1	19	19	19	19

Table 6.7 Comparison Fast Discrete Cosine Transform.

Resource constraint		[Dhod95]		Genetic List		Ordinary List	
# mult	# add	cycles	registers	cycles	registers	cycles	registers
3	2	18	13	14	10	17	12
2	2	-	-	18	13	21	11

Comparisons show that the completion time and the register allocation (see Section 6.8 for details about register allocation) resulting from the methods presented in [Ahma95,Dhod95] are significantly larger than the genetic list scheduling technique presented in this section when using 3 multiplier (requiring 2 cycle steps) and 2 adders (requiring 1 cycle step). A more tight resource constraint of 2 multipliers and 2 adders leads to similar results for the genetic list scheduling approach presented in this section.

The results of [Dhod95] hardly improve the results obtained by an ordinary list scheduler.

In [Ahma95b] list scheduling is used in combination with a strategy called simulated evolution [Koza92]. Within simulated evolution, mutation is the dominant operator, and the goal is to introduce variations in the solution. The encoding is similar to the one presented in [Ahma95]. It is not clear from the article what the advantages are of using simulated evolution instead of genetic algorithms (no such comparisons can be found). Furthermore, no high-level synthesis benchmark results have been reported.

6.6.7 Permutation encoding and topological scheduling techniques

Despite the good performance, the disadvantage of list scheduling is that it has been proven that there exist examples for which it misses out on the optimal solution, independent of the permutation used (see also Figure 5.1). In Algorithm 5.14 a constructive scheduler has been presented, for which at least one permutation Π exists, resulting in an optimal solution [Heij95b]. Compared to the methods presented in Section 6.6.4 and 6.6.5, it gradually constructs schedules, and hence prevents the greedy allocation of resources. The results achieved using this encoding can be found in Table 6.4 and Table 6.5. In all cases the genetic algorithm based on a topological scheduler finds the optimal results. The running times of the scheduler are between 0.1 and 10 seconds for each entry of the tables using an HP9000/735 computer.

6.7 Supplementary resource allocation

In some cases, lower bound resource allocation estimation finds a lower bound for which no feasible schedule exists, in other words the resource allocation induces a completion time for which each resource constrained schedule exceeds the time constraint specified initially. Consequently, the method must allow the allocation of extra hardware. When exact methods like IP scheduling [Hwan91,Gebo92] are used, the danger exists that they will perform an exhaustive search, because they cannot detect that a combination of constraints is infeasible, and hence large run times might result. Heuristic iterative methods allocate supplementary resources depending on the scheduling results achieved. In [Kuma91] a list scheduling strategy is proposed, in which operations are detected which cannot be scheduled within their schedule range, and extra resources are allocated immediately to make the schedule of these operations possible. In [Heij91] a similar strategy is followed, but the decisions about the resource type to be increased are based on statistics obtained by complete schedules instead of partial schedules, and the whole schedule process is restarted using a new resource allocation to balance the resource usage more equally over the whole schedule. The disadvantage of iterative schemes is that they heavily depend on the initial resource allocation and the scheduling method.

The supplementary resource allocation can also be integrated in the scheduling method as follows. Let *ModType* be a set of module types. Let $RA_{min}(l) \in \mathbb{N}$ represent a lower

bound resource allocation of module type $l \in ModType$. Let $RA_{max}(l)$ represent an estimated upper bound resource allocation of a module type $l \in ModType$.

The supplementary resource allocation $RA_{sup}(l) \in [0 , RA_{max}(l) - RA_{min}(l)]$ of module type $l \in ModType$ denotes the number of supplementary resources admitted for scheduling, in other words, the resource constraint for each module type $l \in ModType$ equals $RA_{min}(l) + RA_{sup}(l)$.

A lower bound resource allocation RA_{min} can be estimated using the method from [Timm93] as presented in Section 4.6. In [Potk89] an asap-scheduling algorithm is used to determine RA_{max}. Although this method will result in an upper bound for the total resource allocation cost, it will not result in a correct upper bound for each module type separately. In Figure 6.7, an example can be found, in which an asap schedule induces a resource allocation of two multipliers and one adder (an addition requires one cycle step to execute on an adder, and a multiplication requires two cycle steps to execute on a multiplier), whereas the optimal schedule within 6 cycle steps induces a resource allocations of two adders and one multiplier.

The upper bound resource allocation RA_{max} can be modelled as a min-flow max-cut problem on a so called comparability graph [Golu80]. An undirected graph (V , E) is a comparability graph if there exists an orientation of (V , E), specified by a directed graph (V , F), satisfying:

1. $F \cap F^{-1} = \emptyset$

2. $F \cup F^{-1} = E$

3. $(u , v) \in F \wedge (v , w) \in F \Rightarrow (u , w) \in F$

In which F^{-1} denotes the reversal of F, given by $F^{-1} = \{(v , u) \mid (u , v) \in F\}$.

Let $(X , \prec)$ be a partially ordered set. Let (X , F) be a graph for which $\forall_{u,v \in X \mid u \prec v} (u , v) \in F$, and let (X , E) be a graph for which $X = V$, and for each $\forall_{u,v \in X \mid u \prec v} \{u , v\} \in E$ (in which $\{u , v\}$ denotes an undirected edge between u and v). Because $(X , \prec)$ is a partial order, (X , F) is an orientation of (X , E), and hence (X , E) is a comparability graph. Let $E_A = \{\{u , v\} \mid A \subseteq X \wedge u,v \in A \wedge \{u , v\} \in E\}$. A clique is a subset $A \subseteq X$ such that (A , E_A) induces a complete subgraph (in other words $\forall_{a,b \in A} \{a , b\} \in E_A$). A clique cover is a partition of A in $A_1 , A_2 , ... , A_k$ such that for each $i \in \{1 , 2 , ... , k\}$, A_i is a clique. A stable set is a subset $A \subseteq X$ of which no two vertices are adjacent (in other words $\forall_{a,b \in A} \{a , b\} \notin E$). A maximum stable set is a stable set of maximum cardinality.

Let $(Y , \prec)$ be a partially ordered set. Y is called a chain (or linearly ordered subset) if each distinct pair $a,b \in Y$ is comparable, in other words, either $a \prec b$ or $b \prec a$. Y is

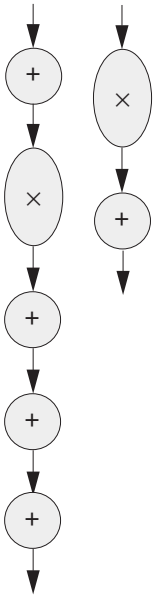


Figure 6.7 Example of partial data-flow graph for upper bound determination.

called an anti-chain if each distinct pair $a,b \in Y$ are incomparable, in other words $a \prec b$ and $b \prec a$. Let $(X , \prec)$ be a partially ordered set. There exists a partition $X = C_1 \cup C_2 \cup ... \cup C_n$, in which C_i, with $i \in \{1 , 2 , ... , n\}$, is a chain, and n is called the width of $(X , \prec)$, which is equal to the smallest clique cover of (X , E) (see [Trot92] for a proof). Because a comparability graph G is a perfect graph (see [Golu80] for details), the size of the smallest possible clique cover $k(G)$ equals the number of vertices in a maximum stable set $\alpha(G)$ of G. In [Golu77] an algorithm is given to find a maximum stable set, by transforming (X , F) into a network flow problem as follows:

1. Add two vertices s and t to X.

2. Add for each input node $i \in X$ an edge (s , i) to F, and for each output node $o \in X$ an edge (o , t) to F.

3. Split each node $x \in X$ into two nodes x_0 and x_1, and add an edge (x_0 , x_1) to F with a low capacity flow of 1.

4. In the resulting network graph, initialize a compatible integer values flow. This can be achieved by for each edge e in the network graph, increasing the flow on a path from s to t containing edge e.

5. A minimum flow from s to t can be found by searching for reducible paths (paths for which the flow on each edge is larger than the low capacity flow), and reduce the flow on that path. If no reducible paths can be found, the algorithm can be stopped. The resulting flow from s tot t will equal the cardinality of the maximum stable set of (X , F) [Golu77].

The data-flow edges E_{DFG} of an acyclic data-flow graph (V_{DFG} , E_{DFG}) induce a strict partial order on V_{DFG}. Let (V_{DFG} , E) be the comparability graph induced by (V_{DFG} , $\prec$). The minimum number of paths that partition V_{DFG} equals the maximum number of resources needed to implement the operations in those paths, which on its turn is given by the number of vertices in a maximum stable set of (V_{DFG} , E). To be able to obtain an upper bound resource allocation for each module type $l \in ModType$ separately, the lower bound capacity flow in the corresponding network flow graph is set to 1 only for those operations $v \in V$, with the operation type mapping $\xi(v) = l$, and is set to 0 for all the other operations. An alternative method presented in [Boer94] generates a reduced network flow graph, consisting of operations and precedence relations restricted to a particular module type.

An extension to the min-flow max-cut method presented above consists of the incorporation of time constraint information. Let v_1 and v_2 be two operations in a data-flow graph *DFG*, and let v_1 and v_2 be elements of a maximum stable set of the corresponding comparability graph G_{DFG}, induced by the partial order of the *DFG* (in other words, there is no flow of data between v_1 and v_2, and they can be executed concurrently). Let $alap(v_1)$ and $alap(v_2)$ be the upper bound of the schedule range of operations v_1 and v_2, caused by a time constraint T_{max}. If $alap(v_1) < asap(v_2)$ or $alap(v_2) < asap(v_1)$, then operation v_1 and v_2 can never be executed concurrently in a feasible schedule, and hence should not contribute to an upper bound resource allocation extracted from the topology of the data-flow graph. By adding an edge (v_1 , v_2) in case $alap(v_1) < asap(v_2)$ or an edge (v_2 , v_1) in case $alap(v_2) < asap(v_1)$ to data-flow graph *DFG*, the comparability graph G_{DFG} induced by the new partial order will contain an edge $\{v_1 , v_2\}$, and hence v_1 and v_2 can never be member of the maximum stable set simultaneously. The addition of edges implies that for small time constraints the upper bound resource estimations will be more accurate than with large time constraints. Some results confirming this behaviour can be found in Table 6.8.

Table 6.8 Upper bound resource allocations.

Example DFG	T_{max}	# mult	# add	# sub
Wave Digital Filter	17	4	5	1
	34	4	5	1
Fast Discrete Cosine Transform	8	14	5	5
	16	14	6	7

The resource allocation available for scheduling is determined by the lower bound estimation RA_{min}, extended with a supplementary resource allocation. The supplementary resource allocation is encoded as follows. For each $l \in ModType$, $RA_{max}(l)$ - $RA_{min}(l)$ positions are allocated in a string. A binary value at such a position denotes whether a supplementary resource is available (1) or not available (0) for scheduling.

The permutation encoding used in Section 6.6.7 is extended by the supplementary resource allocation string. Uniform crossover is performed separately on both the per-

mutation and the supplementary resource allocation string. The fitness of an individual is determined by a combination of the completion time and the resource allocation. A small penalty on the completion time is used to favour individuals representing schedules that are within their original time constraint. Let T_{max} be the original time constraint, let RA_{max} represent the cost of the maximal resource allocation, let $C_{max}(\varphi)$ represent the completion time of the schedule φ, and let $RA(\varphi)$ represent the resource allocation used for scheduling. The scale function Σ is given by:

$$\Sigma = \Phi \cdot (C_{max}(\varphi) - (T_{max} - 1)) + RA(\varphi)$$

with $\Phi \geq RA_{max}$. Some results obtained for the wave digital filter and the fast cosine transform filter are given in Table 6.9 and Table 6.10 ($\Phi = 2 \cdot RA_{max}$).

Table 6.9 Results for Wave Digital Filter.

Time constraint	Optimal		Topological		ifds	
cycles	# mult	# add	# mult	# add	# mult	# add
17	3	3	3	3	3	3
18	2	2	2	2	2	3
19 - 20	2	2	2	2	2	2
21 - 27	1	2	1	2	1	2
28	1	1	1	1	1	1

Table 6.10 Results for Fast Cosine Transform Filter.

Time constraint	Optimal		Topological		ifds	
cycles	# mult	# add	# mult	# add	# mult	# add
8 - 9	8	4	8	4	8	4
10	5	4	5	4	5	4
11	4	3	4	3	4	4
12	4	3	4	3	4	3
13	4	2	4	2	4	3
14 - 17	3	2	3	2	3	3
18	2	2	2	2	3	2
19 - 25	2	2	2	2	2	2
26 - 31	2	1	2	1	2	2
32	2	1	2	1	2	1
33	2	1	2	1	2	2
34	1	1	1	1	2	1

Comparison with the results obtained by state of the art heuristic schedulers, such as improved force directed scheduling (denoted by ifds) [Verh91,Beso94] shows that the genetic topological strategy finds better results in many cases. Some execution times

reported in Table 6.11 show that the genetic scheduling approach is faster, especially for large time constraints (both methods are implemented using the NEAT system on a HP9000/735).

Table 6.11 Run times in seconds for Fast Discrete Cosine Transform Filter.

Time constraint	genetic	ifds
10	1.30	17.7
20	1.30	101.5
30	1.34	203.0
40	1.40	327.6

In Table 6.12 and Table 6.13 the schedule results of some larger examples are shown. The example used to produce the results in Table 6.12, is a four times unfolded Wave Digital Filter (152 operations). In all cases, optimal results are found. Run times are 30 seconds or less for each example tested.

Table 6.12 Results for unfolded Wave Digital Filter.

Time constraint	Topological	
cycles	# mult	# add
65 - 68	3	3
69 - 80	2	2
81 - 111	2	1
112	1	1

The example used to obtain the results of Table 6.13 is an artificial example constructed from the fast discrete cosine transform filter shown in Figure 6.4. It has been ‘unfolded’ four times, and this unfolded graph has been duplicated twice to generate an example which contains a lot of concurrency and symmetry, making it rather difficult to schedule. It contains 582 operations. In all cases, optimal results are found. Run-times are 120 seconds or less for each example tested.

Table 6.13 Results for unfolded Fast Cosine Transform Filter.

Time constraint	Topological	
cycles	# mult	# add
36	24	12
37	24	8
38	15	8
39	14	8
40	13	8
44	12	7
47	11	7

Table 6.13 Results for unfolded Fast Cosine Transform Filter.

Time constraint	Topological	
cycles	# mult	# add
49	11	6
51	10	6
56	9	6
59	9	5
63	8	5
72	7	5
74	7	4
83	6	4
98	5	3
122	4	3
147	4	2
162	3	2
242	2	2
294	2	1
482	1	1

6.8 Extensions

By using lower bound estimations in combination with a special schedule construction mechanism, specific knowledge is incorporated into the genetic algorithm to improve its efficiency. By using problem specific information, genetic algorithms are moved away from their general characteristics towards the class of tailored algorithms. Nevertheless, there is still the possibility to optimize other parameters by making use of the general characteristics of the genetic algorithm. One example is to extend the cost of a schedule with the cost of the register allocation induced by the schedule. The register allocation induced by a schedule can be determined efficiently by using the left-edge algorithm from [Kurd87]. Assuming that each input value of the data-flow graph needs to be stored immediately at the first cycle step, the total area including register costs can be found in Table 6.14. In this table it is assumed that the area of a multiplier equals 100, the area of an adder equals 10, and the area of a register also equals 10, which have been chosen close to the ratio of module areas produced by module generators used in the NEAT system [Thee93]. The results show that the genetic algorithm including register allocation costs finds schedules requiring less registers. The results are comparable with the results achieved with improved force directed scheduling extended with register costs, as described in [Paul89].

It should be noted that by extending a cost function, register costs are optimized in a general way. This means that no strategies tailored to optimize register costs are used by this method. Therefore it is expected that for memory intensive applications this general

approach will not provide an efficient way to optimize the overall register costs, and methods tailored to this problem will be needed as discussed in Section 4.5.

In [Dhod95,Wehn91] the scheduling and allocation problem is defined in terms of finding a trade-off between speed (completion time) and area (resource allocation). They use a general approach by application of a weighted cost functions to find solutions to the resource allocation problem and completion time simultaneously. Disappointing results presented in Table 6.6, Table 6.7, Table 6.2, and Table 6.3 show that these general approaches fail to find good results.

Table 6.14 Scheduling results for Fast Discrete Cosine Transform Filter.

Time constraint	genetic without reg. cost	genetic with reg. cost	ifds with reg. cost
8	960	960	940
13	540	520	530
18	350	340	420
23	340	320	440
28	340	320	330
33	340	320	320

6.9 Scheduling cyclic data-flow graphs

The method presented in Section 5.11 can be used to create schedules for cyclic data-flow graphs from permutations. Just like with scheduling acyclic graphs, genetic search strategy can be applied to search for a permutation which leads to good quality solutions.

Table 6.15 Schedule results of Cyclic Wave Digital Filter.

Resource constraint			Throughput constraint	Section 5.11	[Radi96]
# mult	# add	pipelined mult.		Latency	Latency
3	3	-	16	14	18
2	3	-	16	17	18
2	2	-	17	15	19
1	2	-	19	21	21
1	1	-	28	28	-
-	3	2	16	14	18
-	2	2	17	15	19
-	3	1	16	17	18
-	2	1	17	17	19
-	1	1	28	28	-

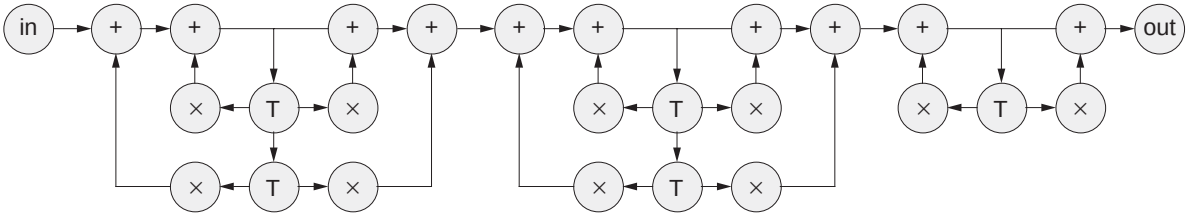


Figure 6.8 Fifth order PCM voiceband-filter.

The first example tested is the cyclic wave digital filter of Figure 1.2 (see Table 6.15). The results are compared to the results reported in [Radi96], which use a OBDD-based representation for scheduling. Although it is claimed in [Radi96] that their results are the best and optimal results published so-far, the genetic approach presented in Section 5.11 finds better results in all cases tested. An example of an optimal schedule having a completion time of 14 cycle steps, using a throughput rate of 16 cycle steps and a resource constraint of 3 multiplier and 3 adders, is shown in Figure 6.9.

In case retiming is used in combination with scheduling techniques for acyclic data-flow graphs as presented in Section 6.6.7 [Fran94], the results presented in Table 6.16 have been found. In all cases the genetic strategy in which retiming and scheduling are integrated into one method finds better solutions.

Table 6.16 Schedule results of Cyclic Wave Digital Filter (2).

Resource constraint		Section 5.11		[Fran94]	
# mult	# add	Throughput	Latency	Throughput	Latency
3	4	16	14	16	16
3	3	16	14	-	-
2	3	16	17	17	17
2	2	17	15	18	18
1	2	19	21	20	20
1	1	28	28	28	28

The second example is the fifth order PCM voiceband-filter example from [Goos89b]. The results are given in Table 6.17.

Table 6.17 Schedule results of fifth order PCM voiceband-filter.

Resource constraint		Throughput	Latency
# mult	# add	cycles	cycles
5	3	4	6
4	3	5	6
4	2	6	6
3	2	7	6

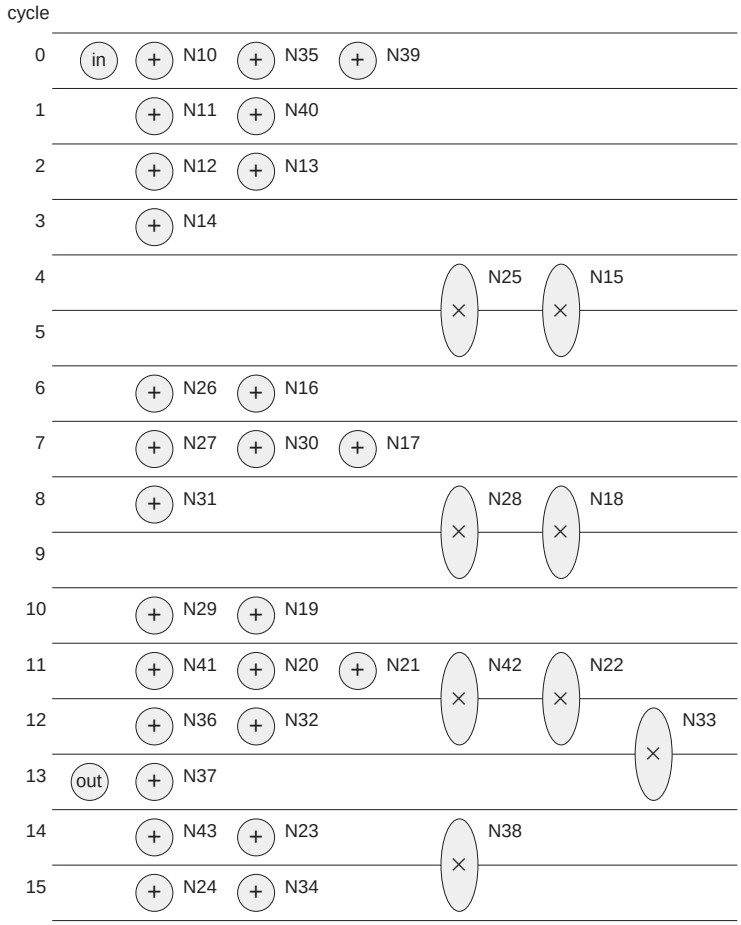


Figure 6.9 Schedule of cyclic wave digital filter.

Table 6.17 Schedule results of fifth order PCM voiceband-filter.

Resource constraint		Throughput	Latency
# mult	# add	cycles	cycles
2	2	10	6
2	1	10	9
1	1	20	11

The last example is from [Chao93], and is called 2-cascaded biquad filter, which is derived from the fifth-order PCM voice-band filter (see Figure 6.8) by excluding the last first order section. The algorithm in [Chao93] starts with a given schedule, and moves operations to different pipeline stages, in order to obtain a smaller data introduction interval. No extra analysis to minimize the completion time are reported in [Chao93], in other words the completion time equals the shortest path in the data-flow graph, given by the data introduction interval. The latency found by the genetic search

using the method reported in 5.11 are therefore equal or better than the results found in [Chao93].

Table 6.18 Schedule results of fifth order PCM voiceband-filter.

Resource constraint			Section 5.11		[Chao93]	
# mult	# add	pipel mult	Throughput	Latency	Throughput	Latency
4	2	-	4	4	4	4
3	2	-	6	4	6	6
2	2	-	8	4	-	-
2	1	-	8	7	8	8
1	1	-	16	7	16	16
-	2	2	4	4	4	4
-	2	1	8	6	8	8
-	1	1	8	7	8	8

6.10 Exhaustive search

The topological scheduling technique has also been incorporated and tested in an exhaustive search, using branch and bound techniques.

Let Π be a permutation. A new permutation Π' can be obtained by changing the order of the elements in permutation Π. Some pairs of elements can be identified for which it makes no sense to exchange their order in a permutation. Operations $v_i \in V$ which have data-flow in common with an operation $v_j \in V$ (in other words for which $v_i \prec v_j \vee v_j \prec v_i$), will never be scheduled concurrently. When using topological scheduling techniques, exchange of these operations will lead to the same schedule. Operations which don't have overlapping execution intervals can never be scheduled concurrently in a feasible schedule. Although changing the order of a permutation might lead to a change in the order of operations which do have overlapping execution intervals, the process of exchange can be restricted to those operations which have cycle steps in common in their schedule ranges.

Let $l \in ModType$. Let $u \in V$ be an operation with operation mapping $\xi(u) = l$, which has overlapping execution intervals with n - 1 other operations $v \in V$ with $\xi(v) = l$. If n is smaller than the number of resources available for scheduling, according to the proof on page 77, these operations never have to be deferred in time. If such an operation becomes available for scheduling, it can be scheduled immediately without introducing any resource conflicts. Assuming that these operations will always be scheduled immediately after all their predecessors have been scheduled (and hence these operations can never block the schedule of successor operations because of their position in a permutations), the exchange of order of these operations with any other operation in a permutation is useless.

All of these observations result in a reduction of the number of permutations to be investigated in an exhaustive search (depending on the size of the time constraint and the resource constraint given or derived). An exhaustive search based on these observations has been implemented and tested. Despite the reduction techniques used, the resulting execution times were outrageous, and many runs had to be cancelled after days of running times for almost each example tried out.

6.11 Conclusions

In this chapter constructive scheduling methods have been combined with genetic algorithms to search for a suitable order to schedule operations of a data-flow graph.

First, a statistical analysis about the fitness values of successive populations of genetic algorithms has been performed, to gain some insight in how to obtain a genetic algorithm with maximum progress. The results have been observed in terms of exploration versus exploitation, and the effects of applying uniform crossover to create all offsprings have been discussed and tested with positive results.

It has been shown that applying a genetic search strategy without the incorporation of specific knowledge regarding scheduling, gives poor results. For a genetic approach to be successful, the problem should be approached from the problem characteristic point of view, and not only from a genetic algorithm point of view. In this section much attention has been paid to avoid the creation of infeasible solutions (by using a permutation encoding instead of bit-strings), to avoid the creation of non-optimal solutions (by using lower bound estimations, and by building schedules in a topological way to prevent the greedy allocation of resources), and by deriving accurate stop criteria (also by making use of lower bound estimations).

The method is extended with an encoding capable of allocating supplementary resources during scheduling. This makes the scheduling method very suitable for high-level synthesis strategies based on lower bound estimation techniques.

Experiments and comparisons show high quality results and fast run-times, the combination of which outperforms results produced by other heuristic scheduling methods. Although optimal results are not guaranteed by the method, optimal results are found in all cases tested.

Chapter 7 Conclusions and future work

7.1 Conclusions

In this thesis a solution approach to the high-level scheduling and allocation problem is presented. Solutions are constructed using topological scheduling techniques, guided by a permutation of operations, and genetic algorithms are used to search for good quality solutions with acceptable run time.

The principles of genetic algorithms have been analysed statistically, providing some new insights in how to efficiently apply genetic algorithms to the scheduling and allocation problem. From the analysis it can be concluded that the variance of a population should be kept as high as possible to obtain efficient convergence, which can be achieved by generating *all* offsprings using uniform crossover. Some empirical results have been presented to support these observations.

Furthermore, the relation between the encoding of the high-level synthesis scheduling problem and the search principles of genetic algorithms has been analysed in detail. The conclusions from these investigations can be summarized as the need to prevent the creation of infeasible encodings (by using permutations of operations instead of bit-strings), the need to prevent the creation of infeasible solutions (decoding an individual always results in a solution satisfying the schedule constraints), and the need to prevent the creation of non-optimal solutions as much as possible (by preventing a greedy construction of a schedule, and by using lower bound resource allocation estimations). Without these extra additions, the genetic search fails to come up with acceptable solutions.

These observations have resulted in permutations of operations to be the key mechanism for constructing schedules. It is proven that there exists at least one permutation for which the construction of schedules in a topologically ordered manner, combined with an as early as possible cycle step selection strategy (satisfying both dependence and resource constraints), results in an optimal schedule (in other words with the smallest completion time). Another important observation is that the ratio of optimal solutions versus the total number of solutions in the resulting search space is larger than the ratio resulting from other constructive scheduling algorithms or from the solution space. This increases the chance that probabilistic search methods like genetic algorithms find an optimal solution more quickly.

The topological permutation scheduling method used in the genetic algorithm is resource constrained. Lower bound resource allocation estimations can be used to translate time constrained scheduling problems into resource constrained scheduling problems. To be able to find feasible schedules with respect to the original time constraint, a possibility to allocate additional resources should be integrated. A new method is presented in which additional resource allocation are encoded in combination with a permutation, and genetic search is applied to search for good quality solutions. Results and comparisons show that optimal solutions have been found with acceptable run-times in all cases that have been tested.

Furthermore it is shown that constructive scheduling of loop structures is a more difficult task than scheduling acyclic structures. A topological permutation based schedule constructor is presented, in which loop pipelining or retiming is integrated in the permutation encoding. Genetic algorithms are used to search for permutations resulting in good quality solutions, and results show that optimal solutions have been found in all cases that have been tested.

Efficient algorithms are presented to update the schedule range of operations, to maintain feasibility with respect to throughput rate and time constraints at any time during the construction of a schedule. These algorithms are based on all-pairs longest-path algorithms and a distance matrix. Also, a new efficient algorithm is presented to calculate the minimal throughput rate given a data-flow graph containing loop structures.

Finally, an object-oriented synthesis system called NEAT is presented, which provides a software platform for interacting high-level synthesis tools. Various synthesis strategies have been implemented using NEAT, without having to bother about the order and way in which tools are applied.

7.2 Future work

7.2.1 Conditionals

One point of concern is the scheduling of conditional structures. Resource conflicts induced by parallelism is not directly visible from the structure of the graph, and extra analysis is needed to investigate whether operations can be scheduled concurrently to obtain efficient schedules without inducing superfluous resource allocations and/or completion times.

Operations enclosed by basic blocks have limited resource sharing capabilities. Graph transformations known as code motion (in which operations are transferred from one basic block to another) are often used to exploit resource sharing capabilities across basic blocks (see [Rim95] for an overview).

Code motion, state assignment, and scheduling of basic blocks are interdependent tasks, and de-coupling these tasks and solving them separately by heuristic methods may lead to non-optimal results. A way is needed to solve these problems simultaneously in such a way that the optimal solution is still part of the search space.

In [Sant96] the scheduling problem and code motion problem are stated as one single problem. The main idea is to bind operations to the basic block in which or before which they have to be scheduled, depending on the control-selection strategy chosen for the data-path (for example pre-execution, control selection or data selection). By extending the topological constructive scheduler of Algorithm 5.14 with this basic block information, it can be decided whether operations can be moved to other basic blocks *during* scheduling, which establishes a close interaction between resource constraints imposed, schedule results obtained so-far, and code motions. It is proven that using this method, the set of possible code motions is such that it doesn't exclude the optimal solution from the search space with respect to a class of optimization criteria based on the execution lengths of paths. This proof is based on the fact that a schedule is constructed in a topological way, and just like in Chapter 5 the scheduling problem can be stated as a search problem in terms of a permutation. Code motions resulting in worse solutions are prevented, hence a pruning technique is embedded in the method to reduce the size of the search space. A genetic search strategy is applied to the resulting search problem, and the results obtained are comparable or better than other results published in literature.

7.2.2 Module execution interval analysis

To increase the quality of the results of topological scheduling even more, module execution interval analysis can be integrated. In [Timm95] it is shown that this strategy proves to be very successful. Nevertheless, the way a permutation of operations is searched for is not topologically oriented, and only an intuitive explanation is given how to guide the search with particular heuristics. More research is needed to get some unambiguous statistics about different search strategies and their efficiency.

Literature

[Ahma95] I. Ahmad, M.K. Dhodhi, C.Y.R. Chen, Integrated scheduling, allocation and module selection for design-space exploration in high-level synthesis, *IEE Proc. Comput. Digit. Tech.*, vol. 142, no. 1, January 1995.

[Ahma95b] I. Ahmad, M.K. Dhodhi and K.A. Saleh, An Evolution-Based Technique for Local Microcode Compaction, *Proceedings of the IFIP International Conference on Very Large Scale Integration*, pp. 729-734, 1995.

[Aho86] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers, principles, techniques and tools*, Addison Wesley, 1986.

[Arts91] H.M.A.M. Arts, J.T.J. van Eijndhoven and L. Stok, Flexible Block-Multiplier Generation, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 106-109, 1991.

[Arts92] H.M.A.M. Arts, M.J.M. Heijligers, H.A. Hilderink, W.J.M. Philipsen and A.H. Timmer, *The Neat Reference Manual*, Software Manual, Eindhoven University of Technology, 1992.

[Bane93] U. Banerjee, *Loop transformations for restructuring compilers: the foundations*, Kluwer Academic, 1993.

[Beso94] P.W.P.M. van Besouw, *Improved Force Directed Scheduling*, Training Report, Eindhoven University of Technology, 1994.

[Black88] R.L. Blackburn, D.E. Thomas and P.M. Koenig, CORAL II: Linking Behavior and Structure in an IC Design System, *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 529-535, 1988.

[Blaz94] J. Blazewicz, K.H. Ecker, G. Schmidt, J. Weglarz, *Scheduling in Computer and Manufacturing Systems*, Second Revised Edition, Springer Verlag, Berlin, 1994.

[Boer94] J. de Boer, *Bepaling bovengrens hoeveelheid functionele modules gebaseerd op minimum flow*, Training Report (Dutch), Eindhoven University of Technology, 1994.

[Camp91] R. Camposano, Path-Based Scheduling for Synthesis, *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 1, pp. 85-93, January 1991.

[Chan92] A.P. Chandrakasan, M. Potkonjak, J. Rabaey and R.W. Brodersen, HYPER-LP: A System for Power Minimization Using Architectural Transformations, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 300-303, 1992.

[Chao93] L.F. Chao and A. LaPaugh, Rotation Scheduling: A Loop Pipelining Algorithm, *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pp. 566-572, 1993.

[Coff76] E.F. Coffman Jr, *Computer and Job Shop Scheduling Theory,* John Wiley & Sons, New York, 1976.

[Corm90] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to algorithms*, The MIT Press, McGraw-Hill, 1990.

[Davio79] M. Davio and A. Thayse, Algorithms for Minimal-Length Schedules, *Philips Journal of Research*, no. 34, pp. 26-47, 1979.

[DeMi88] G. De Micheli and D.C. Ku, HERCULES - A System for High-Level Synthesis, *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 483-488, 1988.

[Depu93] F. Depuydt, *Register optimization and scheduling for real-time digital signal processing architectures*, Ph.D. thesis, Katholieke Universiteit Leuven, 1993.

[Deva89] S. Devadas and A.R. Newton, Algorithms for Hardware Allocation in Data Path Synthesis, *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 7, July 1989.

[DeWi85] P. DeWilde, E. Deprettere and R. Nouta, Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms, in: *VLSI and Modern Signal Processing*, ed. S.Y. Kung, H.J. Whitehouse and T. Kailath, Prentice-Hall, Englewood Cliffs, pp.258-264, 1985.

[Dhod95] M.K. Dhodhi, F.H. Hielscher, R.H. Storer and J. Bhasker, "Datapath Synthesis Using a Problem-Space Genetic Algorithm", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, August 1995.

[Eijn91] J.T.J. van Eijndhoven, G.G. de Jong and L. Stok, *The ASCIS Data Flow Graph: Semantics and Textual Format,* EUT Report 91-E-251, Eindhoven University of Technology, 1991.

[Eijn92] J.T.J van Eijndhoven and L. Stok, A Data Flow Graph Exchange Standard, *Proc. of the European Conference on Design Automation (EDAC)*, pp. 193-199, 1992.

[Elli90] M.A. Ellis and B. Stroustrup, *The annotated C++ reference manual*, Addison-Wesley, 1990.

[Fabe94] H. Faber, *Branch-and-Bound Scheduling using Execution Interval Analysis*, Master's thesis, Eindhoven University of Technology, 1994.

[Fleu93] H. Fleurkens, Interactive Systems Design in ESCAPE, *Proceedings of the IEEE International Workshop on Rapid System Prototyping*, pp. 108-113, 1993.

[Fran94] F. Fransen, *Retiming voor Dataflow Grafen*, Training Report, Eindhoven University of Technology, October, 1994.

[Garey79] M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979.

[Gebo92] C.H. Gebotys and M.I. Elmasry, *Optimal VLSI architectural synthesis: area, performance and testability*, Kluwer Academic Publisher, 1992.

[Gere92] S.H. Gerez, S.M. Heemstra de Groot and O.E. Herrmann, A polynomial-time algorithm for computation of the iteration-period bound in recursive data-flow graphs, *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 39, no. 1, pp. 49-52, January 1992.

[Girc84] E.F. Girczyc and J.P. Knight, An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling, *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pp. 726-731, 1984.

[Girc87] E.F. Girczyc, Loop Winding - A Data Flow Approach to Functional Pipelining, *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 382-385, 1987.

[Gold89] D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.

[Golu77] M.C. Golumbic, The complexity of comparability graph recognition and coloring, *Computing 18*, pp. 199-208, 1977.

[Golu80] M.C. Golumbic, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, 1980.

[Gonz77] M.J. Gonzalez Jr., Deterministic Processor Scheduling, *Computing Surveys*, vol. 9, no. 3, September 1977.

[Goos89] G. Goossens, J. Vandewalle and H. De Man, Loop optimization in register-transfer scheduling for DSP-systems, *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 826-831, 1989.

[Goos89b] G. Goossens, *Optimisation techniques for automated synthesis of application-specific signal-processing architectures*, Ph.D. Thesis, Katholieke Universiteit Leuven, 1989.

[Grah76] R.L. Graham, Bounds on the performance of scheduling algorithms, in: *Computer and Job Shop Scheduling* Theory, ed. J.L. Bruno, E.F. Coffman Jr. and R.L. Graham et. al., pp. 165-227, John Wiley & Sons, 1976.

[Hart92] R. Hartmann, Combined Scheduling and Data Routing for Programmable ASIC Systems, *Proceedings of the European Conference on Design Automation*, pp. 486-490, 1992.

[Heem90] S.M. Heemstra de Groot, *Scheduling Techniques for Iterative Data-flow Graphs*, Ph.D. thesis, University of Twente, 1990.

[Heem92] S.M. Heemstra de Groot, S.H. Gerez and O.E. Herrmann, Range-chart-guided iterative data-flow-graph scheduling, *IEEE Trans-*

actions on Circuits and Systems, I: Fundamental theory and applications, vol. 39, no. 5, pp. 351-364, May 1992.

[Heij91] M.J.M. Heijligers, *Time Constrained Scheduling for High Level Synthesis*, Master's Thesis, Eindhoven University of Technology, May, 1991.

[Heij94] M.J.M. Heijligers, H.A. Hilderink, A.H. Timmer and J.A.G. Jess, NEAT: an Object Oriented High-Level Synthesis Interface, *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp 1.233-1.236, 1994.

[Heij95a] M.J.M. Heijligers, L.J.M. Cluitmans and J.A.G. Jess, High-Level Synthesis Scheduling and Allocation using Genetic Algorithms, *Proceedings of the Asia and South Pacific Design Automation Conference*, pp. 61-66, 1995.

[Heij95b] M.J.M. Heijligers and J.A.G. Jess, High-Level Synthesis Scheduling and Allocation using Genetic Algorithms based on Constructive Topological Scheduling Techniques, *Proceedings of the International Conference on Evolutionary Computation*, pp. 56-61, 1995.

[Hild93] H.A. Hilderink and J.A.G. Jess, ROM-based Multi Thread Controller, *IFIP Workshop on Logic and Architecture Synthesis*, pp. 231-241, 1993.

[Hild94] H.A. Hilderink, NESCIO: An Interactive High Level Synthesis Framework, *Proceedings of the Workshop on Circuits, Systems and Signal Processing*, pp. 119-123, 1994.

[Hilf85] P.N. Hilfinger, A High-Level Language and Silicon Compiler for Digital Signal Processing, *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 213-216, 1985.

[Hill81] F.J. Hill and G.R. Peterson, *Introduction to Switching Theory & Logical Design*, Third Edition, John Wiley & Sons, 1981.

[Holl75] J.H. Holland, *Adaption in Natural and Artificial Systems*, MIT Press, 1975.

[Hout94] J.G.M. van Houtert, *Tree Height Reduction in High-Level Synthesis*, Thesis of Practical Work, Eindhoven University of Technology, 1994.

[Hu61] T.C. Hu, Parallel Sequencing and Assembly Line Problems, *Operations Research*, no. 9, pp. 841-848, 1961.

[Hwan91] C.T. Hwang, J.H. Lee and Y.C. Hsu, A formal approach to the scheduling problem in high-level synthesis, *IEEE Transaction on Computer-Aided Design*, vol. 10, no. 4, pp. 464-475, April 1991.

[Hwan91a] C.T. Hwang, Y.C. Hsu and Y.L. Lin, Scheduling for Functional Pipelining and Loop Winding, *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 764-769, 1991.

[IEEE88] IEEE standard 1076-1987, *IEEE Standard VHDL Language Reference Manual*, New York: Institute of Electrical and Electronics Engineers, 1988.

[Ito94] K. Ito and K.K. Parhi, Determining the iteration bounds of single-rate and multi-rate data-flow graphs, *IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 163-168, 1994.

[Jaco94] E.T.A.F. Jacobs, *Using Genetic Algorithms for Time Constrained Scheduling*, Training Report, Eindhoven University of Technology, 1994.

[Jaco95] E.T.A.F. Jacobs, *High-Level Synthesis Interconnect Minimization*, Master Thesis, Eindhoven University of Technology, 1995.

[Jang93] H.J. Jang and B.M. Pangrle, GB: A New Grid-Based Binding Approach for High-Level Synthesis, *Proceedings of the 6th International Conference on VLSI Design*, pp. 180-185, 1993.

[Karp78] R.M. Karp, A characterization of the minimum cycle mean in a digraph, *Discrete Mathematics*, 23, pp. 309-311, 1978.

[Kost95] R. Koster, *A loop representation for scheduling*, Training Report, Eindhoven University of Technology, 1995.

[Koza92] J.R. Koza, *Genetic Programming; on the Programming of Computers by Means of Natural Selection*, MIT Press, 1992.

[Kuma91] A. Kumar, A. Kumar and M. Balakrishnan, A Novel Integrated Scheduling and Allocation Algorithm for Data Path Synthesis, *International Symposium on VLSI Design*, pp. 212-218, 1991.

[Kurd87] F.J. Kurdahi and A.C. Parker, REAL: A Program for REgister ALlocation, *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pp. 210-215, 1987.

[Lam89] M.S. Lam, *A Systolic Array Optimizing Compiler*, Kluwer Academic Publishers, 1989.

[Lann91] D. Lanneer, G. Goossens, F. Catthoor, M. Pauwels and H. De Man, An Object-Oriented Framework Supporting the full High-Level Synthesis Trajectory, in: D. Borrione and R. Waxman (ed.), *Computer Hardware Description Languages and their Applications*, Elsevier Science Publisher B.V., pp. 301-320, 1991.

[Lee89] J.H. Lee, Y.C. Hsu and Y.L. Lin, A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 20-23, 1989.

[Lee92] T.F. Lee, A.C.H. Wu, Y.L. Lin, A New Algorithm for Pipelining Loop Execution, *Proceedings of the Synthesis And SImulation Meeting and International Interchange (SASIMI)*, pp. 198-207, 1992.

[Leis91] C.E. Leiserson and J.B. Saxe, Retiming Synchronous Circuitry, *Algorithmica*, no. 6, pp. 5-35, 1991.

[Lens85] J.K. Lenstra and A.H.G. Rinnooy Kan, Sequencing and Scheduling, in: *Combinatorial optimization: annotated bibliographies*, ed. M. O'hEigeartaigh, J.K. Lenstra and A.H.G. Rinnooy Kan, John-Wiley & Sons, Chisester, 1985.

[Lin73] S. Lin and B.W. Kernighan, An effective heuristic algorithm for the travelling salesman problem, *Operations Research*, vol. 21, pp. 498-516, 1973.

[Lipp91] P.E.R. Lippens, J.L. van Meerbergen, A. van der Werf, W.F.J. Verhaegh and B.T. McSweeney, Memory Synthesis for High Speed DSP Applications, *Proceedings of the IEEE Custom Integrated Circuits Conference*, pp. 11.7.1-11.7.4, 1991.

[Mall90] D.J. Mallon and P.B. Denyer, A New Approach To Pipeline Optimisation, *Proceedings of the European Conference on Design Automation*, pp. 83-88, 1990.

[McFa90] M.C. McFarland, A.C. Parker and R. Camposano, The High-Level Synthesis of Digital Systems, *Proceedings of the IEEE*, 78(2), pp. 301-318, February 1990.

[Mesm95] B. Mesman, *Genetic Algorithms for Scheduling Purposes*, Master Thesis, Eindhoven University of Technology, 1995.

[Mich92] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 1992.

[Nest90] J.A. Nestor and G. Krishnamoorthy, SALSA: A New Approach to Scheduling with Timing Constraints, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 262-265, 1990.

[Pang87] B.M. Pangrle and D.D. Gajski, Slicer: A State Synthesizer for Intelligent Silicon Compilation, *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pp. 42-45, 1987.

[Pang91] B.M. Pangrle, F.D. Brewer, D.A. Lobo and A. Seawright, Relevant issues in high-level connectivity synthesis, *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 607-610, 1991.

[Papa82] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, Englewood Cliffs, 1982.

[Parh91] K.K. Parhi and D.G. Messerschmitt, Static Rate-Optimal Scheduling of Iterative Data-Flow Programs via Optimum Unfolding, *IEEE Transactions on Computers*, vol. 40, no. 2, February 1991.

[Park85] N. Park and A.C. Parker, Synthesis of Optimal Clocking Schemes, *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pp. 489-495, 1985.

[Park86] A.C. Parker, J.T. Pizarro and M. Mlinar, MAHA: A program for data path synthesis, *Proceedings of the 23th ACM/IEEE Design Automation Conference*, pp. 461-466, 1986.

[Park86a] N. Park and A.C. Parker, SEHWA: A Program for Synthesis of Pipelines, *Proceedings of the 23th ACM/IEEE Design Automation Conference*, pp. 454-460, 1986.

[Park91] I.C. Park and C.M. Kyung, Fast and Near Optimal Scheduling in Automatic Data Path Synthesis, *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 680-685, 1991.

[Paul89] P.G. Paulin and J.P. Knight, Force-Directed Scheduling for the Behavioral Synthesis of ASIC's, *IEEE Transaction on Computer-Aided Design*, vol. 8, no. 6, pp. 661-679, June 1989.

[Pine95] M. Pinedo, *Scheduling Theory, Algorithms, and Systems*, Prentice Hall, 1995.

[Pota90] R. Potasman, J. Lis, A. Nicolau and D. Gajski, Percolation Based Synthesis, *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 444-449, 1990.

[Potk89] M. Potkonjak and J. Rabaey, A scheduling and resource allocation algorithm for hierarchical signal flow graphs, *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 7-12, 1989.

[Potk91] M. Potkonjak and J. Rabaey, Optimizing Resource Utilization using Transformations, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 88-91, 1991.

[Radi96] I. Radivojevic and F. Brewer, A New Symbolic Technique for Control-Dependent Scheduling, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 1, January 1996.

[Rama92] C. Ramachandran and F.J. Kurdahi, Combined Topological and Functionality Based Delay Estimation Using a Layout-Driven Approach for High Level Applications, *Proceedings of the European Design Automation Conference*, pp. 72-78, 1992.

[Rim92] M. Rim, R. Jain and R. De Leone, Optimal Allocation and Binding in High-Level Synthesis, *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 120-123, 1992.

[Rim95] M. Rim, Y. Fann and R. Jain, "Global Scheduling with Code-Motions for High-Level Synthesis Applications", *IEEE Transactions on VLSI Systems*, vol. 3, no. 3, pp. 379-392, September 1995.

[Romp92] K. Van Rompaey, I. Bolsens and H. De Man, Just in time scheduling, *Proc. of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 295-300, October 1992.

[Rund93] E.A. Rundensteiner, Design Tool Integration Using Object-Oriented Database Views, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 104-107, 1993.

[Sant96] L.C.V. dos Santos, M.J.M. Heijligers, C.A.J. van Eijk, J.T.J. van Eijndhoven and J.A.G. Jess, A constructive Method for Exploiting Code Motions, To appear in: *International Symposium on System Synthesis*, 1996.

[Schw85] D.A Schwartz and T.P. Barnwell, Cyclo-static Multiprocessor Scheduling on the Optimal Realization on Shift-Invariant Flow Graphs, *IEEE*

International Conference on Acoustics, Speech and Signal Processing, pp. 1384-1387, 1985.

[Shin89] H. Shin and N.S. Woo, A Cost Function Based Optimization Technique for Scheduling in Data Path Synthesis, *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pp. 424-427, 1989.

[Star91] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley and C. Withley, A comparison of Genetic Sequencing Operators, *Proceedings of the 4th International Conference on Genetic Algorithms*, pp. 69-76, 1991.

[Stok91] L. Stok, *Architectural Synthesis and Optimization of Digital Systems*, Ph.D. thesis, Eindhoven University of Technology, 1991.

[Sysw89] G. Syswerda, Uniform crossover in genetic algorithms, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pp. 2-9, 1989.

[Tarj73] R. Tarjan, Enumeration of the Elementary Circuits of a Directed Graph, *SIAM J. Computing*, pp. 211-216, June, 1971.

[Thee93] J.F.M. Theeuwen, Module generators and their integration in an architectural synthesis system, *IFIP Workshop on Logic and Architecture Synthesis*, pp. 401-410, December 1993.

[Thom90] D.E. Thomas, E.D. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan and R.L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer Academic Publisher, 1990.

[Thom91] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, Boston, 1991.

[Timm93] A.H. Timmer, M.J.M.Heijligers and J.A.G. Jess, Fast System-Level Area-Delay Curve Prediction, *Proceedings of the APCHDLSA*, pp. 198-207, 1993.

[Timm93a] A.H. Timmer, M.J.M. Heijligers, L. Stok and J.A.G. Jess, Module Selection and Scheduling using Unrestricted Libraries, *Proceedings of the EDAC/EuroASIC Conference*, pp. 547-551, 1993.

[Timm93b] A.H. Timmer and J.A.G. Jess, Execution Interval Analysis under Resource Constraints, *Digest of the technical papers of the ICCAD*, pp. 454-459, 1993.

[Timm95] A.H. Timmer and J.A.G. Jess, Exact Scheduling Strategies based on Bipartite Graph Matching, *Proceedings of the European Design & Test Conference*, pp. 42-47, 1995.

[Timm95b] A.H. Timmer, M.T.J. Strik, J.L. van Meerbergen and J.A.G. Jess, Conflict Modelling and Instruction Scheduling in Code Generation for In-House DSP Cores, *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, 1995.

[Trot92] W.T. Trotter, Combinatorics and partially ordered sets: dimension theory, Johns Hopkins University Press, London, 1992.

[Vanh93] J. Vanhoof, K. Van Rompaey, I. Bolsen, G. Goossens and H. De Man, *High-Level Synthesis for Real-Time Digital Signal Processing*, Kluwer Academic Publisher, 1993.

[Veen85] A.H. Veen, *The Misconstrued Semicolon*, Ph.D. Thesis, Eindhoven University of Technology, 1985.

[Verh91] W.F.J. Verhaegh, E.H.L. Aarts, J.H.M. Korst and P.E.R. Lippens, Improved Force-Directed Scheduling. In: *Proceedings of the European Design Automation Conference (EDAC)*, pp. 430-435, 1991.

[Verh92] W.F.J. Verhaegh, P.E.R. Lippens, E.H.L. Aarts, J.H.M. Korst, A. van der Werf and J.L. Van Meerbergen, Efficiency Improvements for Force-Directed Scheduling, *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 286-291, 1992.

[Verh92b] W.F.J. Verhaegh, P.E.R. Lippens, E.H.L. Aarts, J.H.M. Korst, J.L. van Meerbergen and A. van der Werf, Modelling Periodicity by PHIDEO Streams, *Proceedings of the Sixth International Workshop on High-Level Synthesis*, pp. 256-266, 1992.

[Verh95] W.F.J. Verhaegh, *Multidimensional Periodic Scheduling*, Ph.D. Thesis, Philips Electronics N.V., 1995.

[Walk92] R.A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publisher, 1992.

[Wang93] C.Y. Wang and K.K. Parhi, Loop List Scheduler for DSP Algorithms Under Resource Constraints, *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 1662-1665, 1993.

[Wang95] C.Y. Wang and K.K. Parhi, High-Level DSP Synthesis Using Concurrent Transformations, Scheduling and Allocation, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 3, March 1995.

[Weng91] J.P. Weng and A.C. Parker, 3D Scheduling: High-Level Synthesis with Floorplanning, *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pp. 668-673, 1991.

[Wehn91] N. Wehn, M. Glesner and M. Held, A Novel Scheduling and Allocation Approach for Datapath Synthesis based on Genetic Paradigms, *IFIP Working Conference on Logic and Architecture Synthesis*, pp. 47-56, 1991.

[Werf91] A. van der Werf, B. McSweeney, J. van Meerbergen, P. Lippens and W. Verhaegh, Hierarchical Retiming Including Pipelining, in: *VLSI 91*, ed. A. Halaas and P.B. Denyer, pp. 451-460, Elseviers Science Publisher, 1991.

[Woer94] H. van Woerkom, *Interconnect Constraints tijdens het Schedulen*, Thesis of Practical Work, Eindhoven University of Technology, 1994.

[Zima90] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press, 1990.

Biography

Marc Heijligers was born on May 25th, 1967 in Eindhoven, the Netherlands.

In Eindhoven he received his diplomas HAVO in 1984 and VWO in 1986 at the van der Putt Lyceum. Subsequently, he studied Information Technology at the Eindhoven University of Technology, where he graduated in May 1991 on a Master Thesis entitled “Time Constrained Scheduling for High-Level Synthesis”. In June 1991 he started working on a doctorate under the supervision of prof.Dr.-Ing. J.A.G. Jess at the Design Automation Section of the Department of Electrical Engineering of the Eindhoven University of Technology.

Since April 1996, Marc Heijligers has been working at the Philips Research Laboratories in Eindhoven, the Netherlands.

Stellingen

behorende bij het proefschrift van Marc Heijligers

1. Het in een topologische volgorde construeren van een schedule van een data-flow reduceert de kans op de generatie van niet geldige oplossingen. [Dit proefschrift]

2. Een formaat of standaard ten behoeve van synthese van digitale schakelingen moet op zijn minst de synthese problematiek duidelijk kunnen representeren en kunnen anticiperen op bepaalde voor de hand liggende oplossingen. Indien het daaraan niet voldoet, dan zullen synthese tools gebruik makende van zo'n formaat in zijn algemeenheid geen goede oplossingen kunnen creëren.

3. Omdat menig artikel over genetische algoritmen de lezer probeert te overtuigen met behulp van argumenten gebaseerd op analogieën uit de evolutieleer, loopt deze wetenschap groot gevaar zijn geloofwaardigheid te verliezen.

4. Aangezien de eigenschappen van een object zich door meer dingen laten bepalen dan alleen het type van het object, zou het predikaat 'type-georiënteerde programmeertaal' in plaats van 'object-georiënteerde programmeertaal' minder valse verwachtingen opwekken omtrent de toepasbaarheid van de in C++ object-georiënteerde geboden mogelijkheden.

5. Het idee van christelijke politieke partijen om de evolutieleer uit het middelbaar onderwijs te schrappen, en het impliciet prefereren van een mogelijke interpretatie van een bijbelse tekst boven de tot nu toe verkregen wetenschappelijke resultaten, veronderstelt dat men onwetendheid verkiest boven algemene ontwikkeling, een gedachte die een obstakel vormt voor de progressie van de wetenschap in het algemeen.

6. Een startend minister zou net zoals een AIO om dezelfde redenen evenredig op zijn salaris gekort moeten worden.

7. Het bekritiseren van de eufonische eigenschappen van een buizenversterker op basis van meet-technische gegevens zoals harmonische vervorming, dempingsfaktor en bandbreedte, getuigt van wetenschappelijke bekrompenheid.

8. Het aanpassen van de regeling van verkeerslichten ter stimulatie van het gebruik van het openbaar vervoer heeft een negatieve invloed op de hoeveelheid uitgestoten uitlaatgassen.

9. Een authentieke uitvoering van een muziekstuk is een farce als men bedenkt dat menig componist beschouwd wordt als een slecht vertolker van eigen werk.

10. Het is onjuist om het begrip persvrijheid te vertalen in het recht hebben op informatie.

11. Het verbieden van het kopen van produkten uit het buitenland in combinatie met het afdwingen van een verkoop adviesprijs aan de detailhandel is per definitie een prijsafspraak, en dus bij de wet verboden.

12. Het gevaar van statistiek voor de volksgezondheid blijkt uit de recente advertenties van de fabrikant Philip Morris, waarin deze met selectief cijfermateriaal aan probeert te tonen dat het drinken van een glas water een grotere kans op kanker zou geven dan het passief meeroken van tabakswaar.

13. Een ster in het vermenigvuldigen, daar zou een wiskundige geen punt van mogen maken!

14. Over smaak valt juist wel degelijk te twisten.