Using Genetic Algorithms for Scheduling Data Flow Graphs

by
L.J.M. Cluitmans

EUT Report 92-E-266
ISBN 90-6144-266-4
December 1992
Using Genetic Algorithms for Scheduling Data Flow Graphs

by

L.J.M. Cluitmans

EUT Report 92-E-266
ISBN 90-6144-266-4

Eindhoven
december 1992
Cluitmans, L.J.M.

Using genetic algorithms for scheduling data flow graphs / L.J.M. Cluitmans. - Eindhoven : Eindhoven University of Technology, Faculty of Electrical Engineering. - Fig., tab. - (EUT report, ISSN 0167-9708 ; 92-E-266)
Met lit. opg.
ISBN 90-6144-266-4
NUGI 832
Trefw.: IC's en CAD / optimalisering (informatica) en algoritmen.
Abstract

A Silicon Compiler is a program (or set of programs) that translates an abstract behavioral description of a chip into a chip layout. The first main section of a Silicon Compilation system is called High Level Synthesis. It translates the behavioral description (given as a set of operations and their time precedences) into a controller description and a network of functional building blocks with their interconnections. Finding the number of functional building blocks to be used is called allocating. Assigning operations to functional building blocks is called binding. Assigning timeslots to each operation is called scheduling. In allocating, scheduling and binding, the target is usually to minimize the area of the used functional building blocks or to minimize the number of timeslots used. The problem of optimally performing these three tasks is often called a scheduling problem. A program that solves a scheduling problem is called a scheduler.

In this thesis a scheduler is described that uses a genetic algorithm to perform the optimization. Internally it uses a (resource constrained) list scheduler in the process of representing schedules.

The genetic scheduler is fast and produces good results. However, several features that would be nice for a scheduler (chaining, pipelining, taking into account register costs, etc.) are not yet implemented. Furthermore, the genetic part of the scheduler could not be tested very well because for many problems the initialization of the algorithm already gave the optimal solution. Larger benchmarks would be necessary to test the genetic part more thoroughly, but they weren't available at test-time.
## Contents

1 Introduction 1

2 High Level Synthesis 3
  2.1 High Level Synthesis 3
  2.2 Formal description of High Level Synthesis 7
  2.3 Scheduling 13
    2.3.1 ASAP and ALAP scheduling 13
    2.3.2 Some other schedulers 14
  2.4 List scheduling 14
    2.4.1 Resource constrained list scheduling 15
    2.4.2 Time constrained list scheduling 18
  2.5 Multicycling, chaining and pipelining 18

3 Genetic algorithms 21
  3.1 Optimization problems 21
  3.2 Genetic algorithms 22
    3.2.1 Introduction to genetic algorithms 22
    3.2.2 Basic definitions 23
    3.2.3 Generic genetic algorithm 25
    3.2.4 A procedure for using genetic algorithms 26
  3.3 An example: function maximization 26
    3.3.1 Problem description 26
    3.3.2 Fitting a genetic algorithm to the problem 27
    3.3.3 Running the genetic algorithm 28
  3.4 Choosing genotypes and operators 30
  3.5 Tuning the algorithm 31
    3.5.1 Numeric degrees of freedom 32
    3.5.2 Tuning the operators 33

4 Genetic algorithms and scheduling 37
  4.1 Resource constrained genetic scheduling 37
    4.1.1 Problem description 38
    4.1.2 Definition of phenotypes and cost function 38
    4.1.3 Definition of genotypes and translator 39
    4.1.4 Defining the operators 39
    4.1.5 Implementation and results 41
  4.2 Time constrained scheduling using genetic algorithms 42
## CONTENTS

4.2.1 Problem description ................................................. 42  
4.2.2 Definition of phenotypes and cost function .................. 43  
4.2.3 Definition of genotypes and translator ...................... 43  
4.2.4 Defining the operators ......................................... 43  
4.2.5 Results .......................................................... 44  
4.3 A non-greedy genetic scheduler ................................... 46  
4.4 Avoiding local minima .............................................. 47  
4.5 Extending the genetic scheduler .................................. 48  

5 Conclusions and suggestions ......................................... 51  

Bibliography ................................................................... 53  

A Tested graphs  
A.1 Graph f2 .............................................................. 57  
A.2 Graph sehwa .......................................................... 58  
A.3 Graph fdct ............................................................ 59  
A.4 Graph wdelf .......................................................... 60  
A.5 Graph mwdelf ........................................................ 61  

B Results of the time constrained genetic scheduler .............. 63  
B.1 Library 1 ................................................................. 64  
B.1.1 Graph f2 ............................................................ 64  
B.1.2 Graph sehwa ....................................................... 64  
B.1.3 Graph fdct ........................................................ 65  
B.1.4 Graph wdelf ....................................................... 65  
B.1.5 Graph mwdelf ..................................................... 66  
B.2 Library 2 ................................................................. 67  
B.2.1 Graph f2 ............................................................ 67  
B.2.2 Graph sehwa ....................................................... 67  
B.2.3 Graph fdct ........................................................ 68  
B.2.4 Graph wdelf ....................................................... 68  
B.2.5 Graph mwdelf ..................................................... 69  

C Comparison between schedulers ..................................... 71  
C.1 Library 1 ................................................................. 72  
C.1.1 f2 ................................................................. 72  
C.1.2 sehwa .............................................................. 72  
C.1.3 fdct ................................................................. 73  
C.1.4 wdelf .............................................................. 74  
C.1.5 mwdelf ............................................................ 74  
C.2 Library 2 ................................................................. 76  
C.2.1 f2 ................................................................. 76  
C.2.2 sehwa .............................................................. 76  
C.2.3 fdct ................................................................. 77  
C.2.4 wdelf .............................................................. 78  
C.2.5 mwdelf ............................................................ 78
Chapter 1

Introduction

At the design automation section of Eindhoven University of Technology software tools are being developed for the design of integrated circuits. One of these tools is a silicon compiler which translates an abstract behavioral specification of an chip into a chip layout.

![Diagram showing the design flow of the silicon compiler]

Figure 1.1: A Silicon Compiler overview

The design traject for the silicon compiler consists of three main sections (figure 1.1). These are High Level Synthesis, Logic Synthesis, and Layout Generation. The High Level Synthesis part transforms the functional specification into a description for a network of modules (like adders or multipliers) and a controller for that network. The Logic Synthesis transforms these to a network of gates. The Layout Generation finally creates a chip layout from the gate network. For a more thorough approach to these topics, see [Michel 92].
This thesis is about a problem in High Level Synthesis called scheduling problem. Chapter 2 defines this kind of problem more formally. Previous work performed in the design automation section of Eindhoven University of Technology ([Heijligers 91]) described a solution to this kind of problem. The time constrained list scheduler described there suffers from a number of parameters that must be tuned carefully before best results occur. My task was to find a way to tune these parameters automatically.

To do this, I used genetic algorithms. Genetic algorithms are described in chapter 3. The way I used genetic algorithms for the scheduling problem is described in chapter 4. This work resulted in a genetic scheduler loosely based on the time constrained list scheduler.

The work was done in the NEAT-group of the design automation section. In this group several tools for high level synthesis are being developed. In the NEAT system one data file format and common data structures are used in all tools. As the NEAT system is implemented in the programming language C++, all tools (also the genetic scheduler) are implemented in C++.
Chapter 2

High Level Synthesis

This chapter describes what High Level Synthesis is. In section 2.1 the concepts used in High Level Synthesis, especially data flow graphs will be described. Section 2.2 gives a formal definition of the concepts used. Section 2.3 describes some schedulers. In section 2.4 list scheduling is introduced. This chapter is not meant as a full description of the scheduling and allocating problems and previous solutions to them, but as a sufficient introduction to understand the problems that are going to be solved in chapter 4. Most of this chapter is based on the master thesis by M.J.M. Heijligers, [Heijligers 91]. For a more thorough description of High Level Synthesis you can look in [McFarland 90] or [Gajski 92].

2.1 High Level Synthesis

High Level Synthesis is the part of the chip design path that transforms a formal description of a chip to a network of lower level functional building blocks and a description for a controller. The building blocks can be adders, multipliers, ALUs, registers, multiplexers etc. The controller controls the data flow between those building blocks and can be modelled by a finite state machine. Figure 2.1 shows an overview of High Level Synthesis. High Level Synthesis tries to optimize the resulting network for maximal speed, minimal chip area and minimal power consumption. The designer usually gives constraints on some of these goals, and wants to optimize the other goals subject to these constraints.

When performing the High Level Synthesis the following steps are performed. First the algorithm that is to be implemented in an Integrated Circuit is translated into an ASCIS Data Flow Graph. This graph is given to the components of the High Level Synthesis in the form of an ASD-file. The abbreviation ASD stands for Architectural Synthesis Data. The data in the ASD-file is stored in a LISP-like language described in [Eijndhoven 91]. This ASD format is not only used as input for the High Level Synthesis but also as an intermediate and final format. Going from the beginning to the end of the High Level Synthesis more information is stored in it. At the beginning of the design path only the
operations to be performed and their relations (which operations should have completed before other operations can be performed) are stored in the ASD-file. This description is the description of the data flow graph. At the end the ASD-file also contains information about the building blocks to be used and their interconnections (called the network graph), as well as information about the controller (the control graph). In the computer itself, ASD data is kept as a NEAT-database. All High Level Synthesis tools at the design automation section of Eindhoven University of Technology use this NEAT-database as their internal representation.

An example of a data flow graph is shown in figure 2.2. The nodes are operations to be performed like addition, subtraction and multiplication. The edges can be viewed in two different ways. Firstly, they describe the order in which the operations should be performed. Secondly, they describe the data flow between the different nodes. In a data flow graph nodes that control the dataflow can also appear (to perform if-statements for instance). Apart from data inputs and outputs, they also have a control input that determines what way to direct the data. Other kinds of edges can also appear in data flow graphs. They don’t always transport data but they do always induce a precedence between the nodes.

As figure 2.1 shows, there are many operations that can be performed on the NEAT database. The operations that are shown in the left column are performed from top to bottom during the High Level Synthesis. In the following paragraphs these operations are described.
2.1. **HIGH LEVEL SYNTHESIS**

Figure 2.2: An example data flow graph (the ptseng algorithm).

The **optimizer** tries to perform some optimizations to the data flow graph. These optimizations can for example include dead code elimination, constant propagation, common subexpression elimination, inline expansion of procedures, tree height reduction and loop optimization.

The **selector** determines the kind of building blocks that are going to be used. The **module allocator** determines the number of building blocks of each type that are to be used in the graph.

The **scheduler** assigns the operations of the data flow graph to cycle steps (the fundamental sequencing unit in synchronous systems).

After the scheduling, the following steps take place: the **module binder** assigns the operations in the data flow graph to the building blocks. The **register generator** (consisting of **register allocator** and **register binder**) creates the necessary registers to hold data values. The **multiplexer generator** creates the multiplexers needed to switch the right data values to the right building blocks. The **controller generator** creates a controller that controls the registers and multiplexers, so they do their work at the right time. Finally, the **interconnector** determines the links between all building blocks, registers and multiplexers.

Several steps of the High Level Synthesis are interdependent. That is they need the result of each other to produce optimal results. This problem can be solved in two ways: one solution is to combine those interdependant steps into one program that solves the combined problem. Another way is to iterate and re-iterate over the steps to grow towards an optimal result.

The most important operation in High Level Synthesis for this thesis is **scheduling**. The
The results generated by High Level Synthesis heavily depend on what kind of hardware modules are available. It might for instance be possible to implement an addition in the data flow graph on a specialized *adder* module, but also on a more general *ALU* (Arithmetic and Logic Unit). That same ALU may also be able to implement a multiplication. The information about which modules are available and what operations they can implement is available in a database file called the *library*.

Let us take another look at the graph of figure 2.2. It shows the data flow graph of the *P5eng* algorithm created from the following piece of pseudo-C:

```pseudo-c
typedef struct { int i1,i2,i4,i6,i10; } inputset;
typedef struct { int o1,o2; } outputset;

outputset PTSENG(inputset in)
{
    outputset out;
    v1 = in.i1; /* Operation N-6 */
    v2 = in.i2; /* Operation N-5 */
    v4 = in.i4; /* Operation N-4 */
    v6 = in.i6; /* Operation N-3 */
    v10 = in.i10; /* Operation N-2 */
    v3 = v1 + v2; /* Operation N-7 */
    v5 = v3 - v4; /* Operation N-8 */
    v7 = v3 * v6; /* Operation N-9 */
    v8 = v3 + v5; /* Operation N-10 */
    v9 = v1 + v7; /* Operation N-11 */
    v11 = v5 / v10; /* Operation N-12 */
    v12 = v1;
    v1 = v8 & v11; /* Operation N-14 */
    v2 = v9 | v12; /* Operation N-15 */
    out.o1 = v1; /* Operation N-16 */
    out.o2 = v2; /* Operation N-17 */

    return(out);
}
```

Let us assume the library contains three different kinds of modules: a multiplier that can implement multiplication and division, an adder that can implement addition and subtraction and a logical unit that can implement AND, OR and XOR. Let us also assume the target is to perform the algorithm in a minimal number of cycles. First the module allocator announces
that two adders, two logical units and one multiplier are necessary. Next, the scheduler assigns cycle steps to all operations. The result of this can be seen in figure 2.3.

In figure 2.3 two columns of time indicators are shown. The left column is labeled timeslot and the numbers in it are the time intervals in which operations can be performed. The right column is labeled time and the numbers in it are the borders between the timeslots. In most parts of this thesis these times (not timeslots) are used to describe scheduling results. When timeslots are meant, this explicitly will be mentioned.

The module binder now assigns modules to the operations in the data flow graph. The multiplier (MUL1) performs operations N9 and N12. The first adder (ADD1) performs operations N8 and N10. The second adder (ADD2) performs operations N7 and N11. The first logic unit (LOG1) performs operation N14, the second logic unit (LOG2) performs operation N15. The register generator determines that 5 registers are necessary for this schedule. The multiplexer generator determines that 6 multiplexers are needed. Finally the interconnector makes the links between all modules, registers and multiplexers. The result is the network of figure 2.4. The blocks colored gray are registers and multiplexers.

2.2 Formal description of High Level Synthesis

Definition 2.1 (Data Flow Graph)
A data flow graph is a directed graph, represented by a pair \((V, E)\). Here \(V\) is a set of nodes and
Figure 2.4: The network for the ptseng example.

$E$ is set of edges. An edge $e_{ij} \in E$ is a pair $(v_i, v_j)$ where $v_i, v_j \in V$.

In a data flow graph the nodes are also called operations.

A data flow graph is often depicted graphically, where a node $v_i$ is shown as a circle and an edge $e_{ij}$ is shown as an arrow from node $v_i$ to node $v_j$. In the rest of this thesis it is assumed that data flow graphs are acyclic (there should be no loops in the graph).

**Definition 2.2 (Path)** Let $G = (V, E)$ be a graph.

A path from node $v_i$ to $v_j$, $v_i \neq v_j$ in $G$ is defined as a set of edges

$\text{path}(v_i, v_j) = \{e_{i_1}, e_{i_2}, \cdots, e_{i_{N-1}}\} \subseteq E$.

**Definition 2.3 (Transitive closure)**

Let $G = (V, E)$ be a graph.

The transitive closure of $G$ is defined as $G^* = (V, E^*)$ where

$E^* = \{(v_i, v_j) \in V^2 | \exists \text{path}(v_i, v_j) \text{ in } G\}$

**Definition 2.4 (Precedence Graph)**

Let $G = (V, E)$ be a data flow graph.

The precedence graph of $G$ is a graph $(V, \prec)$. Here $\prec$ is a relation on $V^2$ for which $v_i \prec v_j \iff \exists e_{ij}^* \in E^*$. 
To put this more verbose: when \( v_i \prec v_j \), the operation \( v_i \) must have completed before the operation \( v_j \) can start. The relation \( v_i \prec v_j \) is pronounced as \( v_i \) precedes \( v_j \). It can be shown that \((V, \prec)\) is isomorphic to \((V, E^*)\).

**Definition 2.5 (Successor; Predecessor)**
Let \( G = (V, E) \) be a data flow graph and \((V, \prec)\) be its precedence graph.

When \( v_i \prec v_j \), \( v_i \) is called a relative predecessor of \( v_j \). In that case \( v_j \) is called a relative successor of \( v_i \).

When there is an \( e_{ij} \in E \), \( v_i \) is called a direct predecessor of \( v_j \). In that case \( v_j \) is called a direct successor of \( v_i \).

Let \( V^* \) be the powerset of \( V \).
\( \text{pred}(v) : V \mapsto V^* \) is a function giving all direct predecessors of \( v \).
\( \text{succ}(v) : V \mapsto V^* \) is a function giving all direct successors of \( v \).
\( \text{pred}^*(v) : V \mapsto V^* \) is a function giving all relative predecessors of \( v \).
\( \text{succ}^*(v) : V \mapsto V^* \) is a function giving all relative successors of \( v \).

In this thesis successor (predecessor) is used as synonym for direct successor (predecessor).

**Definition 2.6 (Operation type)**
Let \((V, \prec)\) be a precedence graph
Let \( T \) be a set of operation types.
The type of an operation describes the semantics of that operation.
\( r : V \mapsto T \) is mapping from an operation to an operation type.

The type of an operation can be an operator (+, *, -, /, etc.), an input/output operation (input, output) or a control operation (branch, merge, etc.). With these operations all kinds of graphs with expressions, loops, repetitions, conditionals, procedures, arrays and other constructs can be created. For an overview of the types available, see [Eijndhoven 91].

**Definition 2.7 (Modules, module types)**
A module is a functional piece of hardware.
Each module has a module type associated with it.

A module type can for instance be an adder, a multiplier, or an ALU. A module type can often implement more than one operation type. An adder for instance, can usually implement both addition and subtraction.

**Definition 2.8 (Library, network module set)**
A network module set \( L \) is a set of modules.
A library \( \Lambda \) is a set of module types.
The library associated with a network module set is the set of module types occurring in that library.

**Definition 2.9 (Module type mapping)**
Let $L$ be a network module set and let $\Lambda$ be its associated library.

$\vartheta : L \leftrightarrow \Lambda$ is a function that returns the type of the given module. It is called a module type mapping.

$\vartheta^{-1} : \Lambda \leftrightarrow L^*$ is a function returning the set of modules in $L$ that have the given module type.

$\vartheta'^{-1} : \Lambda^* \leftrightarrow L^*$ is a function returning the set of modules in $L$ that have a type occurring in the given module type set.

**Definition 2.10 (relation between operations and modules)**

Let $T$ be an operation type set

Let $L$ be a network module set. Let $\Lambda$ be its associated library.

$\mu : T \rightarrow \Lambda^*$ is a mapping from operation types to module types. For a $t \in T$, $\mu(t)$ is the set of module types that can implement the operation type $t$.

$\bar{\mu} : T \rightarrow L^*$ is a mapping from operation types to modules. For a $t \in T$, $\bar{\mu}(t)$ is the set of modules that can implement the operation type $t$. $\bar{\mu} = \vartheta'^{-1} \circ \mu$.

$\mu^{-1} : \Lambda \rightarrow T^*$ is a mapping from module types to operation types. For a $\eta \in \Lambda$, $\mu^{-1}(\eta)$ is the set of operation types that can be implemented on the module type $\eta$.

$\bar{\mu}^{-1} : L \rightarrow T^*$ is a mapping from modules to operation types. For an $\eta \in L$, $\bar{\mu}^{-1}(\eta)$ is the set of operation types that can be implemented on the module $\eta$. $\bar{\mu}^{-1} = \mu^{-1} \circ \vartheta$

**Definition 2.11 (Properties of modules)**

Let $L$ be a network module set. Let $\Lambda$ be its associated library. For each element $\eta \in \Lambda$ the following properties are known:

$\delta : \Lambda \rightarrow \mathbb{R}$ is a function called delay function. It is the number of cycle steps needed before a valid output is generated on a module. The function $\delta$ always returns a nonnegative number.

$\bar{\delta} : L \rightarrow \mathbb{R}$ is defined as $\bar{\delta} = \delta \circ \vartheta$

$\bar{c} : \Lambda \rightarrow \mathbb{R}$ is a function called cost function. It is usually related to the chip area necessary to implement the module and to the power the module dissipates.

$\bar{c} : L \rightarrow \mathbb{R}$ is defined as $\bar{c} = \bar{c} \circ \vartheta$

In this thesis it is assumed that a delay is the same for all operations that can be performed on a module. Otherwise delay would be a function of both module type and operation type.

**Definition 2.12 (Binding)**

Let $L$ be a network module set

Let $G = (V, E)$ be a graph

$\xi : V \rightarrow L$ is a mapping from operations to modules. It is called the binding function. For any $v \in V$, $\xi(v) \in \bar{\mu}(\tau(v))$.

$\xi^{-1}(\eta) : L \rightarrow V^*$ returns the set of operations that are bound to the module $\eta \in L$.

As the delay of an operation is often needed, a shorthand is defined for it: $d' : V \rightarrow \mathbb{R} : d'(v) = \bar{\delta}(\xi(v))$. 
2.2 **FORMAL DESCRIPTION OF HIGH LEVEL SYNTHESIS**

It is the task of the module binder to define some mapping  $\xi$. However, for some types of schedulers a good mapping arises from the chosen schedule, and the scheduler and the module binder are often combined for that kind of scheduler.

**Definition 2.13 (Scheduling, Schedule, begin, end)**

Let $G$ be a data flow graph $(V, E)$

Let $d' : V \mapsto \mathbb{R}$ be a delay function

Let $D$ be a time range $D = 0, \ldots, t_{max}; t_{max} \in \mathbb{N}$

A **schedule function** is a function $\varphi : V \mapsto D^2$. It assigns a time interval $[t_{\text{begin}}, t_{\text{end}}]$ to each operation $v \in V$. The length of that interval should be equal to $d'(v)$, ergo $t_{\text{end}}(v) = t_{\text{begin}}(v) + d'(v)$.

The interval $[t_{\text{begin}}, t_{\text{begin}} + d'(v)]$ is called the **schedule of the operation** $v$.

The union $\Phi(G) = \bigcup_{v \in V} (v, \varphi(v))$ is called a **schedule of the graph** $G$.

$\text{begin}(v)$ (begin : $V \mapsto D$) returns the first element of $\varphi(v)$.

$\text{end}(v)$ (end : $V \mapsto D$) returns the last element of $\varphi(v)$.

$\#\Phi(G)$ is the length of the schedule $\Phi(G)$. $\#\Phi(G) = \max_{v \in V} : \text{end}(v)$

This definition of schedule does not take into account the precedence relation in the graph. To do this the concept of **feasibility** is introduced.

**Definition 2.14 (feasibility)**

Let $G$ be a data flow graph $(V, E)$. Associated with $G$ is its precedence graph $(V, \prec)$

A schedule $\Phi(G)$ is called **feasible** if and only if the following relation holds:

$\forall v_1, v_2 \in V | v_1 \prec v_2 : \text{end}(v_1) \leq \text{begin}(v_2)$. Or more specific: $\forall v_1, v_2 | v_1 \prec \text{suCC}(v_2) : \text{end}(v_1) \leq \text{begin}(v_2)$.

When it is known that a schedule is feasible, it still isn't known whether it can be performed on a certain amount of hardware. For this information the concepts **well allocated schedule** and **valid schedule** are introduced.

**Definition 2.15 (valid schedules)**

Let $L$ be a network module set and let $\Lambda$ be its associated library

Let $G$ be a data flow graph $(V, E)$

A schedule $\Phi(G)$ is called **well allocated** for a binding function $\tilde{\xi}$ if:

$\forall v \in L | v, v \in \tilde{\xi}^{-1}(\mu(v)) | \mu(v) \neq \mu(v_2) : \text{begin}(v_1) \geq \text{end}(v_2) \lor \text{end}(v_1) \leq \text{begin}(v_2)$.

A schedule is called **valid** for a binding function $\tilde{\xi}$ if it is both feasible and well allocated.

To put it with words: a schedule is well allocated if each module in a network module set is busy working at most at one operation at a time.

**Definition 2.16 (Absolute network module set; Fully used network module set)**

A Library $\Lambda$ is called **absolute** with regard to a data flow graph $(V, E)$ when every operation in $V$ can be implemented on some module type in $\Lambda$: $\forall v \in V : \mu(\tau(v)) \neq \emptyset$. 

A network module set is called absolute with regard to a data flow graph \((V, E)\) when its associated library is absolute with respect to that graph.

A network module set \(L\) is called fully used with regard to a data flow graph \((V, E)\) and a binding function \(\xi\) if it is absolute and \(\forall m \in L \exists v \in V : m \in \xi(v)\).

To put the definition of fully used network module set in words: a network module set is fully used with respect to a graph and a binding function if each operation in the graph is mapped by the binding function to a module in the network module set and each module in the set is used.

**Definition 2.17 (scheduling costs)**

Let \(G = (V, E)\) be a data flow graph

Let \(L\) be a network module set

Let \(\tilde{c} : L \rightarrow \mathbb{R}\) be a cost function

Let \(\tilde{\xi} : V \rightarrow L\) be a binding function

Let \(\Phi(G)\) be a schedule using network module set \(L\)

The cost of a network module set \(C_L(L)\) is the sum of the costs of the modules in it: \(C_L(L) = \sum_{m \in L} c(m)\).

The cost of a schedule \(C_{\Phi}(\Phi(G), \tilde{\xi})\) is the cost of the fully used network module set necessary to implement it using binding function \(\tilde{\xi}\).

**Definition 2.18 (scheduling problems)**

Given a data flow graph \(G = (V, E)\)

Given a library \(\Lambda\)

Given a cost function \(c : \Lambda \rightarrow \mathbb{R}\)

A time constrained scheduling problem is, given a cycle time set \(D = [0, t_{\text{max}}]\), trying to find a fully used network module set \(L\) whose modules have types in \(\Lambda\), a binding function \(\tilde{\xi} : V \rightarrow L\) and a valid schedule function \(\varphi : V \rightarrow D^2\) in such a way that the cost of the resulting schedule is minimal.

A resource constrained scheduling problem is, given a network module set \(L\), trying to find a time \(t_{\text{max}}\), a binding function \(\tilde{\xi} : V \rightarrow L\) and a valid schedule function \(\varphi : V \rightarrow D^2\) where \(D = [0, t_{\text{max}}]\) in such a way that \(t_{\text{max}}\) is minimal.

A mixed scheduling problem or general scheduling problem is, given a weighting function \(w : \mathbb{R}^2 \rightarrow \mathbb{R}\), trying to find a time \(t_{\text{max}}\), a fully used network module set \(L\), a binding function \(\tilde{\xi}\) and a valid schedule function \(\varphi : V \rightarrow D^2\) where \(D = [0, t_{\text{max}}]\) in such a way that \(w(t_{\text{max}}, C_L(L))\) is minimal.

In words, in a time constrained scheduling problem the question is to find a schedule with as few modules as possible for a given time limit, whereas in a resource constrained scheduling problem the question is to find an as fast as possible schedule for a given set of modules.
2.3 Scheduling

In the previous section three types of scheduling problems have been defined. All are NP-complete. A hint for the proof of this can be found in [Ullman 75]. A scheduler solving a time constrained scheduling problem is called a time constrained scheduler. A scheduler solving a resource constrained scheduling problem is called a resource constrained scheduler. To solve scheduling problems two approaches can be taken. One approach is to use an exact solution method. Integer Linear Programming methods have been proposed for this purpose. Because of the NP-nature of the problems these exact methods are inherently slow. The other approach is using heuristics or approximation methods. These are not guaranteed to give optimal solutions but can produce near-optimal results in reasonable time. In this section some of the methods that are used for solving scheduling problems are described. The descriptions are kept terse. In section 2.4 list schedulers are described more elaborately. Most heuristics used to solve scheduling problems are local in nature and may find a local optimum instead of a global optimum.

2.3.1 ASAP and ALAP scheduling

The simplest kinds of schedulers are the ASAP scheduler and the ALAP scheduler. The term ASAP stands for As Soon As Possible and the term ALAP stands for As Late As Possible. Those two schedulers are very alike. They are neither resource constrained nor time constrained but they behave like resource constrained schedulers in that they try to minimize the number of cycle steps needed. There is however no network module set given.

The ASAP scheduling algorithm can be defined recursively as:

\[
\text{asap} : V \mapsto \mathbb{R} : \quad \text{asap}(v) = \begin{cases} 
0 & \text{if pred}(v) = \emptyset \\
\max_{v_i \in \text{pred}(v)} \text{asap}(v_i) + d'(v_i) & \text{otherwise}
\end{cases}
\]

\[
\varphi(v) = [\text{asap}(v), \text{asap}(v) + d'(v)]
\]

The ALAP scheduling algorithm can be defined recursively as:

\[
\text{alap} : V \mapsto \mathbb{R} : \quad \text{alap}(v) = \begin{cases} 
\max & \text{if succ}(v) = \emptyset \\
\min_{v_i \in \text{succ}(v)} \text{alap}(v_i) - d'(v_i) & \text{otherwise}
\end{cases}
\]

\[
\varphi(v) := [\text{alap}(v) - d'(v), \text{alap}(v)]
\]

These two schedulers are very simple, and do not take into account the cost of the schedule. However, they are often used to get information about the graph, such as the minimal length of a schedule.
2.3.2 Some other schedulers

In this subsection classification of schedulers will be given. As most classifications, this is only a coarse classification. It is only meant to give some structure to the large number of schedulers available. The borders in the classification are soft, so it is well possible that a certain scheduler can be considered member of more than one class.

First of all, schedulers can be classified in those that guarantee optimal results and those that only try to give optimal results.

Because of the NP-complete nature of the problem, schedulers of the first class are slow, especially for larger graphs. An example of this class is the Integer Linear Programming (ILP) scheduler ([Hwang 91], [Gebotys 90]).

The second class can be subdivided in approximation schedulers and heuristic schedulers. Approximation schedulers guarantee a certain closeness to the optimal solution. A few examples of approximation schedulers are:

- Percolation scheduler ([Potasman 90])
- Simulated Annealing based schedulers ([Devadas 89] [Nestor 90])
- Neural Net based schedulers ([Hemani 90], [Philipsen 92])

The largest class is the class of heuristic schedulers. They have no worst case constant bound on the solution they give. To name a few:

- Critical Path schedulers ([Parker 86], [Thomas 90])
- Force Directed schedulers ([Paulin 89], [Stok 91], [Verhaegh 91])
- List schedulers; These can be further classified in two types:
  - Resource Constrained list schedulers ([Goossens 90], [Gutberlet 91])
  - Time Constrained list schedulers ([Heijligers 91], [Kumar 91])

List schedulers will be described in some detail in the next section.

2.4 List scheduling

A list scheduler is a resource constrained scheduler. However, extensions that make a list scheduler a time constrained scheduler are known ([Heijligers 91]) and are described in subsection 2.4.2.
2.4. LIST SCHEDULING

2.4.1 Resource constrained list scheduling

In this subsection the resource constrained list scheduler will be described. It will be used as a part of the genetic scheduler described in section 4.2.

In list scheduling an upper bound is chosen to restrict the number of operations that can be scheduled in one cycle step. This bound is usually related to the number of modules of each module type to be used, i.e. a resource constraint.

A list scheduler keeps track of a list of operations that can be scheduled. An operation can be scheduled if all its predecessors have been scheduled. This list is called the free list. Each time it is created it is sorted according to a priority function. The priority is a function $V : \mathbb{R} \rightarrow \mathbb{R}$. The free list is sorted for increasing values of the priority function. For each operation in the list a module on which it can be mapped is searched. If it can be found the operation is scheduled and removed from the free list. If there is no such module the operation is kept in the free list. When there are no more operations that can be scheduled at the current cycle time, the free list for the next cycle time is determined and the procedure is repeated.

In figure 2.5 this process is depicted for a simple graph. The available modules are one adder and one multiplier. Both have a delay of one cycle. The free list is initialized to contain the
operations that have no predecessors. In this example the free list is initialized to contain nodes N1 and N2, colored black in figure 2.5b. All nodes in the free list can be scheduled on modules. In the next timeslot (figure 2.5c), there are three nodes in the free list. The multiplication of node N4 can be scheduled on the multiplier. For the adder, a choice must be made between the two additions N3 and N5. Because all operations are checked for schedulability in the order they appear in the list, the one that appears first in the list will be scheduled and the other one is kept in the free list. Which one will be first depends on the priority function. Here node N3 is scheduled and N5 is kept in the free list. The process of creating the free list and scheduling the elements thereof is repeated until all nodes have been scheduled.

The result of the list scheduling can be influenced by the choice of the priority function used. If in the example above a priority function was used that would have given node N3 a lower priority than both N5 and N7, a schedule that needs one timeslot more would have resulted (it is left as an exercise for the reader to verify this). The art of implementing a list scheduling algorithm is in part the art of choosing the right priority function for the right graph.

Several priority functions are described in literature. To name some:

- **Global freedom.** This is the difference between the ASAP-time of a node and its ALAP-time.
- **Local freedom.** This is the difference between the current time and the ALAP time.
- **Number of successors.**
- **Freeing count.** This is the number of successors that will become member of the free list when the node would be scheduled.
- **Distance.** This is the difference between the time constraint $t_{\text{max}}$ and the ASAP-time of the node.
- **Combinations.** Often a weighted sum of other priority functions is used.

For more information on these priority functions, see [Heijligers 91]. The priority functions shown above can be classified in two classes. In static priority functions, such as global freedom, number of successors and distance, the value of the priority function can be determined before scheduling takes place. In dynamic priority functions, such as local freedom and freeing count, the value of the priority function cannot be computed before the node is considered for being scheduled. A general static priority function will be described in chapter 4.

Practice shows that some priority functions perform better on some kinds of graphs and that other priority functions perform better on other graphs. For a combined priority function
List schedulers have, as all other schedulers, some advantages and some disadvantages. List scheduling is very fast and can produce good results when given an appropriate priority function. However, the determination of such a priority function can be quite a trouble. For some kinds of graphs a list scheduler will never find the optimal solution. This effect (a result of the greedy nature of the list scheduler) will be explained in the next paragraph. The largest problem with resource constrained list schedulers however is that they are not time constrained. Most designers of ICs want to be able to give a time constraint to the scheduler. A solution for this problem is presented in subsection 2.4.2.

Figure 2.6: Greedyness of the scheduler prevents optimal solution.

The greedy nature of the list scheduler (i.e. when a node is checked for schedulability and it can be scheduled, it is scheduled) is the reason the graph of figure 2.6a will never be scheduled as fast as possible when the library consists of a one-cycle adder and a two-cycle multiplier. This does not depend on the priority function. In the first cycle the free list consists of the upper addition and the right multiplication. Both operations are scheduled on their respective modules. This has the effect that the left multiplication cannot be started in the next cycle, because the multiplier is still in use. All list schedulers will generate the schedule of figure 2.6b. When the right multiplication could somehow be delayed, the faster schedule of figure 2.6c could be the result.
2.4.2 Time constrained list scheduling

In practical situations time constrained schedulers are preferred to resource constrained schedulers. In [Heijligers 91] a method is described to make a time constrained variant of a list scheduler.

It works as follows. A network module set is iterated. The initial network module set is determined by an allocator. Initially a resource constrained list scheduler is run. If the desired time constraint is met, the resulting schedule is the answer the time constrained scheduler gives. If the resulting schedule needs more cycles than the time constraint allows, it is decided that there weren't enough modules in the network module set. The case that an other priority function might result in a schedule that meets the time constraint is not considered. The network module set is now extended (good heuristics for doing this have yet to be found). The resource constrained scheduler is now run again with the extended network module set. If the time constraint is met now, the resulting schedule is given as the answer. If the time constraint still isn't met, the network module set is extended again. This running of the resource constrained list scheduler and extending of the network module set is repeated until a schedule meeting the time constraint is found.

The time constrained list scheduler performs good, but some problems have to be solved yet.

- A good heuristic for finding an appropriate priority function has to be found.
- A good heuristic for doing the module extensions has to be found.
- Some way of finding an optimal solution to problems where the greedy list scheduler cannot find one has to be found.

Solutions to these problems that use genetic algorithms will be described in chapter 4.

2.5 Multicycling, chaining and pipelining

Three concepts are often used in scheduling practice that haven't been defined yet. They are depicted in figure 2.7.

The first one, multicycling, has not been defined yet, but was already used in figure 2.6. The term multicycling is used when modules that need more than one cycle to complete are used (figure 2.7a). For a list scheduler to handle multicycling, it not only has to search for presence of a suitable module but also has to keep track whether the modules are free or busy.
The second concept to be introduced in this subsection is **chaining**. Chaining applies only to **combinatorial** modules. A module is said to be combinatorial when it doesn't need a clock signal to do its work. Modules that are not combinatorial are called **sequential**. Combinatorial modules can have non-integer delays (delays are expressed in number of cycle step times). When two combinatorial modules with delays that add up to less than one cycle step time have to be performed after each other, they can be performed in one cycle step. This is called chaining (figure 2.7b).

The third concept is (module) **pipelining**. It only applies to sequential modules. Some sequential modules can start working on a new data value when the previous result hasn't appeared at the output yet. In effect the module can be busy working at multiple data items (at different internal stages of processing) at the same time. This is called module pipelining (figure 2.7c). The number of cycles that must be waited before a new data value can be applied to a module is called the **data introduction interval** of that module (sometimes abbreviated to DII). Pipelining can be used on modules where the DII is smaller than the delay.
Many usefull techniques used by mankind are based on studying nature. One of those techniques is called *genetic algorithms*, which applies the knowledge of evolution theory ([Darwin 1859]) to computer science. Though the base for genetic algorithms was laid in the sixties and seventies ([Holland 75]), they have only recently become popular, especially by the work of Goldberg ([Goldberg 89]). This chapter explains what genetic algorithms are, how they are used and what kind of problems they are used for.

### 3.1 Optimization problems

Genetic algorithms are used for solving optimization problems. One can view an optimization problem as a black box with several input parameters and one output. The problem is to find the *optimum* input parameters, that is the set of parameters that minimizes or maximizes the output. See definition 3.1.

**Definition 3.1 (Optimization Problem)**

Let \( \mathcal{P} \) be a set.

*Given a function \( c_P : \mathcal{P} \rightarrow \mathbb{R} \) (the cost function),*

A **maximization problem** is finding a \( p \in \mathcal{P} \) for which \( \forall q \in \mathcal{P} : c_P(p) \geq c_P(q) \).

A **minimization problem** is finding a \( p \in \mathcal{P} \) for which \( \forall q \in \mathcal{P} : c_P(p) \leq c_P(q) \).

An **optimization problem** is either a maximization problem or a minimization problem.

In this thesis \( \mathcal{P} \) denotes the set over which a cost function has to be optimized. Unless stated otherwise, any reference to optimization problems in this thesis is about maximization problems. This can be safely done because any minimization problem can be written as a maximization problem by choosing \( c_P, \text{to be maximized} = -c_P, \text{to be minimized} \), and hence no restrictions are made.
3.2 Genetic algorithms

In this section an informal outline of a genetic algorithm will be given first. Then the terminology used in this thesis will be defined formally. Finally, a formal definition of a genetic algorithm will be given.

3.2.1 Introduction to genetic algorithms

Genetic algorithms try to find the optimum of a given set \( P \) for a given cost function \( c_P \). In genetic algorithms elements of \( P \) are called phenotypes. Most of the operations performed in a genetic algorithm are not done on phenotypes but on encoded versions thereof called genotypes. The reason for the encoding will become clear later. For now, it is sufficient to know that each genotype \( g \) can be decoded to one phenotype \( p = T(g) \), that a cost function can be defined on the genotypes \( (c_P(g) = c_P(T(g))) \) and that the genotypes are the objects the genetic algorithm is operating on.

The idea behind genetic algorithms is quite simple. First a set of genotypes is created. Those genotypes are created randomly. This set is called the population. For each genotype in the population the cost function (also called fitness) is computed. A genotype is said to be fit when it has a higher cost than most other genotypes in the population (assuming we are trying to solve a maximization problem, not a minimization problem). In fact being fit is a stochastic property. Later on we need to select fit genotypes from the population. A genotype has a higher probability of being selected when there are less other genotypes in the population having a higher fitness.

A new population is now created based upon fit genotypes in the old population. The genotypes in the new population are created by making small changes to fit genotypes in the old population (mutating) or by combining parts of different fit genotypes to a new genotype (crossing). The fittest genotypes in the old population are copied into the new population too.

For most occurring kinds of phenotypes no reasonable way of defining crossing or mutating can be found. Genotypes however can be designed to behave well when crossed or mutated. This is the reason that genotypes are used in the population instead of the phenotypes.

By repeatedly creating new populations the average fitness of the genotypes will get better (at least it won't get worse) for each new population. The best genotype in the last generated population is the solution the genetic algorithm gives. The question is when to stop the creation of new populations. One answer is to stop when a known optimum is found. Often it is not possible to tell whether a solution is an optimum or not. In that case the algorithm...
is stopped when a given number of populations has been generated.

### 3.2.2 Basic definitions

The terminology used in the genetic algorithms field is not uniform in publications. It is also different from the terminology used in mathematical optimization theory. In this thesis the terminology defined in the following definitions is used.

**Definition 3.2 (Terminology1: phenotype, genotype, population)**

Given an optimization problem where a cost function \( c_P : P \rightarrow \mathbb{R} \) must be maximized.

The things to be optimized (the elements of \( P \)) are called **phenotypes**.

**Genotypes** are encoded versions of phenotypes.

The set of all possible genotypes is denoted by \( G \).

The set on which the genetic algorithm is working is called the **population**. It is a set of genotypes.

Often a genotype can be split in several parts, called **chromosomes**.

These chromosomes can sometimes be split in even smaller parts called **genes**.

The introduction of genotypes is necessary (instead of using the phenotypes directly as elements of the population) because operations that are going to be performed on the elements of the population cannot be defined reasonably on all occurring kinds of phenotypes. The terminology about genotypes, chromosomes and genes is not always used consistently in literature: sometimes genotypes or chromosomes are called genes.

**Definition 3.3 (Translator)**

The **translator** is the function that transforms a genotype into its corresponding phenotype:

\[ T : G \rightarrow P. \]

The translator does not have to be one-to-one (i.e. there may be more than one genotype corresponding to one phenotype). The translator should be onto however (i.e. for each phenotype, there should be a genotype that maps to it).

**Definition 3.4 (Cost functions)**

In the optimization problem a cost function \( c_P : P \rightarrow \mathbb{R} \) is defined.

The cost function can also be defined on the genotypes:

\[ c_G : G \rightarrow \mathbb{R} : c_G = c_P \circ T. \]

The result of the cost function is called the **fitness** of the phenotype \( p \) or of the genotype \( g \).

The terms phenotype and genotype might be a little misleading. They are not derived from the mathematical term type, but they are derived from biology. In biology a genotype is the
genetic information needed to create an individual, as incorporated in the DNA of a cell. The phenotype in biology is the individual itself as created from its genotype (and possibly also influenced by the environment, but that doesn't matter here). The translator $T$ as defined in definition 3.3 is the mathematical equivalent of the process of growing up in biology.

To make new genotypes out of old ones operators are used.

**Definition 3.5 (Operators)**

$(\mathbb{N}_+ \text{ is the set of positive integers})$

An operator is a mapping $o : \mathcal{G}^p \rightarrow \mathcal{G}^c$ (possibly nondeterministic), where $p, c \in \mathbb{N}_+$. The type of an operator $o : \mathcal{G}^p \rightarrow \mathcal{G}^c$ is the ordered pair $(p, c) \in \mathbb{N}_+^2$. The set of all operators of type $(p, c)$ is denoted by $\mathcal{O}_{(p,c)}$. The operator space is the set $\mathcal{O} = \bigcup_{(p,c)} \mathcal{O}_{(p,c)}$.

The inputs of an operator ($p$ elements of $\mathcal{G}$) are called parents. The outputs of an operator ($c$ elements of $\mathcal{G}$) are called children.

**Definition 3.6 (Mutators, Crossers)**

An operator of type $(1, 1)$ is called a unary operator or mutator. An operator of type $(2, 2)$ is called a binary operator or crosser. Sometimes operators of type $(2, 1)$ are also called crossers.

In association with operators some functions are defined that determine their properties.

**Definition 3.7 (Type, ParentDim, ChildDim)**

Type : $\mathcal{O} \rightarrow \mathbb{N}_+^2$ returns the type of an operator.
ParentDim : $\mathcal{O} \rightarrow \mathbb{N}_+$ returns the dimension $p$ of the input vector of an operator (the number of parents).
ChildDim : $\mathcal{O} \rightarrow \mathbb{N}_+$ returns the dimension $c$ of the output vector of an operator (the number of children).

In practical situations only mutators and crossers are used as operators. A mutator is an operator that produces a slightly modified version of its argument. A crosser is an operator that takes two parent genotypes and produces two children by combining information of the two parents. Usually a crosser chops the parents in pieces and recombines these pieces into new genotypes. Exactly how this is done depends on the structure of the genotypes. An example will be described in section 3.3. Another example is in the main research of this thesis as described in chapter 4.
3.2. GENETIC ALGORITHMS

3.2.3 Generic genetic algorithm

A more formal outline of a genetic algorithm can be given now. It differs from the previously given informal approach in the way fit elements are copied from one population to the next. In fact they aren't copied, but some of the unfit genotypes in the population are replaced with newly generated genotypes. Those children are first accumulated in an auxiliary set until at least \( L \) children have been generated (\( L \) is some predetermined number). The modified population becomes the new population. This can be done safely because the old population isn't needed any more.

Algorithm 3.1 (Generic genetic algorithm)

\( O \subseteq \mathcal{O} \); (the set of operators used)

\( s \in \mathbb{N}^+; \) (the size of the population)

\( L \in \mathbb{N}^+; \) \( L < s \); (the number of genotypes to replace in each generation)

\( \{ P, C, T_p \} \); Bags of \( G \); (Population, childset, parent set)

- Fill \( P \) with \( s \) randomly generated genotypes
- do (until stopping criterion reached)
  - set \( C = \emptyset \)
  - do (while \(|C| < L\))
    - Choose randomly an operator \( op \) from \( O \)
    - Fill the bag \( T_p \) with \( ParentDim(op) \) copies of genotypes randomly chosen from \( P \), where genotypes with high fitness have a greater probability to be chosen than genotypes with low fitness.
    - \( C = C \cup op(T_p) \)
  - od
  - Remove randomly \(|C|\) entries from \( P \), where genotypes with low evaluation have a greater probability to be removed than genotypes with a high evaluation.
    - \( P = P \cup C \)
- od

Figure 3.1: Relations between genotypes and phenotypes.

\[
\text{genotype} \xrightarrow{\text{Translator}} \text{phenotype} \xrightarrow{\text{Cost Function}} \text{fitness} \\
G \xrightarrow{T} P \xrightarrow{\text{fitness}} R \\
G \xrightarrow{\text{fitness}} R
\]
Some portions of algorithm 3.1 are not written down very formally. Especially the portions about 'randomly choosing' of operators or genotypes with high or low evaluations might need some extra explanation. This explanation is postponed until section 3.4.

3.2.4 A procedure for using genetic algorithms

When attacking an optimization problem with genetic algorithms several steps must be performed. In this thesis the following procedure is used:

1. Define what the phenotypes (the objects to be optimized) are.
2. Define the cost function $c_P$.
3. Choose an encoding. The translator and the layout of the genotypes are defined implicitly by choosing an encoding.
4. Define operators.
5. Implement and run the algorithm.

3.3 An example: function maximization

For a better understanding of what is going on in a genetic algorithm it is instructive to look at an example. In this section a simple example is described. It serves to clarify the material described so far and is not meant to provide a shocking new method to tackle the function maximization problem. In fact many conventional methods exist for this problem that perform better than the genetic approach.

3.3.1 Problem description

The problem in this example is to find the maximum of the function $f(x) = \frac{\sin 10x}{x+1}$. A plot of this function is shown in figure 3.2. The finding of the maximum value of this function can be done by conventional techniques of course, but I chose this example because it is simple, so the problem doesn't distract the reader from the algorithm. This example has the special feature that it is possible to visualize the evolution of the population. A drawback of this example however is that it might be less instructive in explaining why crossers are so usefull in genetic algorithms.
3.3. AN EXAMPLE: FUNCTION MAXIMIZATION

![Graph of the function \( f(p) = \frac{\sin 10p}{p^2 + 1} \).]

**Figure 3.2: A plot of the function \( f(p) = \frac{\sin 10p}{p^2 + 1} \).**

3.3.2 Fitting a genetic algorithm to the problem

First should be determined what genotypes and phenotypes are in this problem and what the translator and cost function are. As we are interested in finding the argument \( x \) for which \( f(x) \) is maximal the phenotype is chosen to be that argument (as a floating point number). The cost function is now the function to be optimized, \( c_P(x) = \frac{\sin 10x}{2^{31} + 1} \). Though it is possible to use a floating point number as a genotype, this is not done. It is impossible to devise a sensible crossing operator for a genotype that consists of only one part because a crosser works by combining different parts of two parents. Instead, a bit-stream encoding is used as genotype.

In a bit-stream encoding an array of bits is used as genotype. To encode a floating point number one can take the bits that represent the number on the computer on which the genetic algorithm is running for instance. Another way of coding the number is to take a sequence of bits representing an integer and transforming that integer to a floating point number. In our example we take a 32-bit signed integer (whose value is between \(-2^{31}\) to \(2^{31} - 1\)) as genotype. By dividing it by \(2^{31}\) and multiplying it by 10 we get a floating point phenotype in the range \(-10\) to nearly 10 (in fact, this is a fixed point representation). In other words, the translator \( T \) is defined by \( p = T(g) = \frac{10g}{2^{31}} \). The genotype can be considered as an integer or as an ordered set of bits. Both ways of seeing the genotype can be used to devise mutators. Only the latter can be used to develop crossers, because the former contains only
one part.

By choosing this genotype and translator, the range of phenotypes that can be represented is shrunk to an evenly spaced discrete subset of the range $[-10, 10) \in \mathbb{R}$. This means the translator is not onto. For this case there is no way of coding the phenotypes (in a for computers usable form) that is onto. This is true for any case where $P$ is an infinite set. In fact the set $P$ used for this problem is the evenly spaced discrete subset of the range $[-10, 10)$ with $2^{32}$ elements.

Now it is time to devise mutators and crossers. Initially one mutator and one crosser are described, both based on the genotypes viewed as ordered sets of bits. The mutator is simple: the child is initially identical to the parent. Then the mutating takes place: for each bit in the child there is a small probability that it is flipped. See figure 3.3.

The crosser used here isn't much more difficult. It is depicted in figure 3.4. It is a crosser of the crossover type. In a crossover type crosser each genotype is split in two parts, both at the same (randomly determined) position. The first part of the first parent is glued together with the second part of the second parent to form the first child. The first part of the second parent is glued together with the second part of the first parent to form the second child. In this operation bits that are equal in both parents will also be the same in the children. This property is usually a goal when designing crossers.

### 3.3.3 Running the genetic algorithm

Having defined a crosser and a mutator, we can run the genetic algorithm. Initially a population of 100 randomly generated genotypes is created. The value $L$ of algorithm 3.1 is set to 10. Thus for each generation, a set of 10 or 11 children is generated by repeatedly
3.3. AN EXAMPLE: FUNCTION MAXIMIZATION

Choosing fit parents and creating children by either mutating or crossing them. As the operators are chosen at random, the operators produce either 1 or 2 children. The creating of children stops after at least \( L \) children have been generated. The number of children generated will then be either 10 or 11.

Next, 10 or 11 unfit genotypes are removed from the original population and the children are merged back into the population. The generating of children and merging them back into the population is repeated until 1000 children have been made in total.

Figure 3.5: Results of running the example genetic algorithm.

Results for this algorithm can be seen in figure 3.5. In figure 3.5a development of the
population in time is shown. Each time when twenty children have been generated the population is plotted: a dot is shown for each phenotype. As the algorithm proceeds most phenotypes crowd near the optimum just on the right side of the y-axis. But mutations keep occurring every now and then, so other possible solutions are tried continuously. In figure 3.5b the error (difference between the best genotype in the population and the best genotype known) is shown on a logarithmic scale. In fact $\ln |(\text{best genotype in population}) - (\text{best genotype known})| + 1$ is plotted. As the final value of this error function is about 11 one can deduce that the final result still has 16 ($11/\ln 2 \approx 16$) erroneous bits. That is better than can be expected from generating 1000 genotypes at random, but many conventional algorithms will produce better results in the same running time. But again, this example is not meant to provide a useful way of attacking one-dimensional function optimization problems, but to provide a simple example of using a genetic algorithm.

3.4 Choosing genotypes and operators

In previous descriptions of algorithms sentences like randomly choose an operator and randomly choose a genotype, where genotypes with high fitness have a greater probability of being chosen appeared without making clear how these choices were made. For these choices the roulette wheel choosing algorithm as described in [Davis 91] is used. This algorithm will be explained by describing how operators are chosen.

Let's say there are four operators to choose from, called $op_1$, $op_2$, $op_3$ and $op_4$. These have priorities $\text{prio}(op_i)$ assigned to them. The probability an operator $op_i$ is chosen is $P(op_i) = \frac{\text{prio}(op_i)}{\sum_j \text{prio}(op_j)}$. The choice is made by computing the cumulative priorities $(\text{cum}(op_i) = \sum_{j=1}^i \text{prio}(op_j))$, picking a random (uniformly distributed) number $r$ in the range $[0, \sum_j \text{prio}(op_j))$ and finding the first operator for which holds that $r < \text{cum}(op_i)$. Let's say that $\text{prio}(op_1) = 100$, $\text{prio}(op_2) = 10$, $\text{prio}(op_3) = 50$ and $\text{prio}(op_4) = 40$. The cumulative values are then $\text{cum}(op_1) = 100, \text{cum}(op_2) = 110, \text{cum}(op_3) = 160$ and $\text{cum}(op_4) = 200$. The random number must be non-negative and less than 200, let's say $r = 142$. For this number operator $op_3$ is chosen. In all genetic algorithms described in this thesis the operator priorities were kept fixed during the run, so the cumulative value could be precomputed and the choosing of operators was fast. In the example of section 3.3 the priority of the crosser was set to 100 and the priority of the mutator was set to 30.

For choosing genotypes nearly the same trick is used. For genotypes the priorities are not constant during the run of the algorithm, but have to be computed from the fitnesses of the genotypes. The fitnesses of genotypes can be mapped in different ways to priorities of genotypes. A simple mapping for making the probability of choosing high fitness genotypes greater than the probability of choosing low fitness genotypes is making the priority of the
genotype equal to its fitness. There are two reasons this isn't a good idea. First, fitnesses may be negative and priorities should be nonnegative. Second, it is well possible that all fitnesses are all near some big number.

For instance, it is possible that all fitnesses are in the range 9000 to 11000. In this case there won't be much difference in the probability of the genotypes to be chosen. Fit genotypes don't have a real advantage over unfit genotypes. One way to circumvent this problem is subtracting the lowest occurring fitness and using the resulting values as priorities. This still gives problems when there are a few fitnesses that are much lower than others. When the fitnesses are in the range 9000 to 11000 except for a few fitnesses that are near 1000, the problem still exists. The most common solution to this problem is as follows: first sort all genotypes in descending order. Then assign a priority to each genotype according to its position in the ordering. For instance, when there are 100 genotypes in the population, the fittest genotype is assigned priority 99, the second fit genotype is assigned priority 98 and so on. The most unfit genotype is assigned priority 0. This is the way of choosing genotypes that was used in the example. It is easy to extend this idea: one may want to choose only from the best half of the genotypes. In this case the priorities of the worst half of the population are set to 0 and the priorities of the better half are set to a descending series of values. For instance the priority of the fittest genotype is set to 49, the priority of the second fit genotype is set to 48 and so on till the genotype halfway is reached, where the priority is 0. Another way to use this principle is choosing the genotypes to be removed before merging the new children back into the population. In this case the most unfit genotype is given the highest priority and fitter genotypes are given lower priorities. In the example the fittest 20% of the genotypes are given priority 0 for this purpose (so they will not be removed), the other genotypes are given increasing priorities, and the most unfit genotype is given the highest priority.

3.5 Tuning the algorithm

Several degrees of freedom have been introduced up till now. All of them can influence the performance of the algorithm when changed. The degrees of freedom described in this thesis are:

- The size of the population.
- The number of genotypes to be replaced in the population between the generations ($L$ in algorithm 3.1).
- The number of children to be generated before stopping the algorithm (when there is no other stopping criterion).
• The operators used.
• The priorities of the various operators.
• Parameter settings of the operators.
• The way priorities are assigned to genotypes in the population for choosing parents.
• The way priorities are assigned to genotypes in the population for being removed.
• The seed of the random number generator used.

The influence of choosing other operators is described in subsection 3.5.2. The other degrees of freedom are described in subsection 3.5.1.

3.5.1 Numeric degrees of freedom

The last item in the list hasn’t been discussed explicitly up till now. Random numbers are used at various places in the algorithm. They are used for initializing the population, they are used in choosing operators and genotypes and they are used in mutators and crossovers (see figures 3.3 and 3.4 for instance). Random numbers in computers are generated by a pseudo-random number generator, which is initialized by a value called the seed. When two random number generators are initialized with the same seed they produce the same ‘random’ sequence. A run of a genetic algorithm can be reproduced exactly by reinitializing the seed with its original value. Of course, taking another seed should not greatly change the behaviour of the algorithm using the random number generator. In practice it sometimes however does. That is the reason why the random number generator seed is shown in the list of degrees of freedom above.

Changing the other degrees of freedom also effects the performance of the algorithm. Setting the population size too small will cause a rapid decrease of variety in the population, so crossovers no longer work well. Solutions that are far off the real solution and solutions that correspond to local optima will be the result. Setting the population size too large will cause a slower convergence toward good solutions. More children have to be created before a good solution is found.

Setting the number of genotypes in the population to be replaced between generations (or more usual: the percentage of the population to be replaced) too large will cause potential useful genotypes in the population that themselves don’t have a good fitness to be thrown away before they had a chance to be used. The variety in the population will decrease very fast if no precautions are taken to prevent creating children that look all alike. If it is set too small, the computationally expensive operation of merging the children into the population.
will occur more often, degrading the performance of the algorithm. The number of children generated in a certain amount of time will be lower.

For problems where there is no stopping criterion, some limit on the number of children to be generated has to be specified. Obviously, when this is too small, no good solution will have been found when the algorithm stops. When it is too large, time is wasted trying to find a better solution than the best one found yet when there is none.

More about alternative operators will be said in subsection 3.5.2. Choosing different operator priorities influences the efficiency of the algorithm and the quality of the solution. When no mutators are used, only settings available in the initial population will be combined and variety in the population will rapidly decrease. When no crossers are used convergence of the fittest genotype toward the optimum will be much slower (as practice shows). Also bad properties of mutators, that are otherwise camouflaged by the crossers will become more apparent. When there are several different parts (genes) in the genotype that can evolve relatively independent of each other (as is not the case in the example but is the case in virtually every practical application), leaving out the crossers will disable the possibility of combining two genotypes in which each one has a good gene and a bad gene to a genotype with two good genes.

3.5.2 Tuning the operators

In the example only two quite simple operators were used. As an alternative crosser the crosser depicted in figure 3.6 can be used. The difference with the crossover crosser of figure 3.4 is that all bits in the mask are randomly chosen. For some problems the first crosser works better, for others the second and for yet others a combination (using two crossers) gives best results.

An other direction of choosing mutators is using the properties of the phenotype. In the case of the example one can add or subtract some value to or from the genotype viewed as integer. In figure 3.7 a mutator that adds a random power of two to the genotype is depicted. These operators usually give better results than the ones of section 3.3.2 but they are not as general.

A quite unique feature of genetic algorithms is that they easily can work together with other algorithms. Many other algorithms or adaptations can be used as crossers or mutators in genetic algorithms. This use of genetic algorithms as encapsulation of other algorithms is called hybridization. These hybrid algorithms often have a remarkable performance.

As an example of hybridization I incorporated an algorithm that finds the maximum of a function on a range (of discrete points) in the example. It is out of the scope of this discussion to describe the algorithm fully. As far as I know there is no description of it in literature.
However, one important property will be given: it only works when the function satisfies certain smoothness conditions on the range (the second order approximation must fit well). In other words: for the given function (figure 3.2) it only gives the maximum of a range when that range contains exactly one ridge and no valleys. To find the global maximum the range must be around the highest ridge (just right of the y-axis). When choosing a range (a begin-point and an end-point) at random the algorithm will not often succeed in finding the global optimum.

However, this algorithm can be described as a crosser (an operator of type (2, 1) to be exact) for the example genetic algorithm. One parent is taken as the beginning of the range, the other parent is taken as the end of the range, and the result the maximum finding algorithm produces is the produced child. Once the genetic algorithm will give the right range to this
crossover, and the exact solution is found then. In fact, for this problem, that will happen very fast and the error will drop to zero very soon. The results of running the hybrid algorithm when the hybrid crosser has only a very low probability is shown in figure 3.8. The genetic algorithm that only converges slowly, and the maximum finding algorithm for which a good initialization is not known are combined to a hybrid algorithm with high performance.

![Graphs showing variety in the gene pool and error over generations.](image)

Figure 3.8: Results of running the hybrid example genetic algorithm.
In this chapter the application of genetic algorithms to scheduling problems will be described. The method described in section 3.2.4 for tackling an optimization problem with genetic algorithms will be used. As the words genetic algorithm are used very much in this chapter, the abbreviation GA will be used.

4.1 Resource constrained genetic scheduling

Section 2.4 explained what a list scheduler is. One of the problems with a resource constrained list scheduler is to find a suitable priority function. As will be explained in this section, genetic algorithms can be used to test several priority functions and to return the best one. In this section a resource constrained genetic scheduler is described that tries to make an as fast as possible schedule, given some resource constraint. It is based on a resource constrained list scheduler.

As described in section 3.2.4, there are several steps to be done when designing a genetic algorithm.

1. Define what is to be optimized, i.e. what the phenotypes are.
2. Describe the problem as an optimization problem, using these phenotypes. This is equivalent to defining the cost function $c_p$.
3. Determine the encoding. This involves finding a suitable genotype and finding a corresponding translator function.
4. Define suitable operators.
5. Implement the operations defined in previous steps and run the resulting program.
4.1.1 Problem description

The scheduler will be given the following information:

- A graph \( G = (V, E) \)
- A library \( A \)
- A network module set \( L \)
- An operation to operation type mapping \( \tau \)
- A operation type to module type mapping \( \mu \)
- A delay function \( d \)

The scheduler will return:

- A binding function \( \xi \)
- A valid schedule of the graph \( \Phi \)

The target is

- Make \( \Phi \) to be as fast as possible

4.1.2 Definition of phenotypes and cost function

As we are trying to optimize a scheduler the phenotype will be a schedule of the graph. In fact we will be using a subset of all possible schedules of the graph. This subset is the set of valid schedules of that graph that use the given network module set and that can be generated by a (greedy) resource constrained list scheduler with static priority functions. This definition will make the translator to be defined in the next subsection to be onto on \( P \) (i.e. each phenotype can be coded by some genotype).

All optimization problems described in this thesis are maximization problems. In the current problem the number of cycles that a schedule lasts has to be minimized. A suitable cost function for the problem is \( c_{\Phi}(p) = -\#p \).
4.1.3 Definition of genotypes and translator

Schedules themselves are very inconvenient as genotypes. It is difficult to define mutators on them, let alone crossers. Maybe it is possible to use schedules as genotypes, but I have chosen to use an encoding for schedules that can be manipulated much easier as genotype. As genotype a general static priority function is used and a resource constrained list scheduler is used as translator.

Remember that a static priority function is a priority function that can determine the ordering of the operations before the scheduling takes place. Considering that fact it follows that the most general description of a static priority function is given by an ordering of the operations. In other words: any static priority function for a given graph can be described by a permutation of the nodes of that graph! Permutations of the nodes in a graph are used as genotype.

This choice for the genotypes (priority functions) and the translator (list scheduler) implies that not all possible schedules can be coded. That is the reason the phenotype set is not the set of all valid schedules, but only a subset thereof.

This way the translator is onto, but it is however far from one-to-one: there are many different genotypes that translate to the same phenotype. The operators will often create child genotypes that have the same phenotype as their parents (as practice shows). When one genotype of the population is in a local optimum, but no genotypes are in the global optimum, operators will tend to create child genotypes of this genotype and its offspring that have equal phenotypes. The population will easily become dominated by this relatively good performing kind of phenotype. When a population is dominated by one kind of phenotype crossers are no longer effective. They produce the same phenotype over and over again. Despite their intent, mutators alone usually don’t have the power to let the population escape from such collapsed state.

4.1.4 Defining the operators

Though defining operators is more difficult for permutations than it is for a bit-stream encoding, they can be defined quite easily. There are some important characteristics of operators. First, operators should never generate invalid genotypes. Second, a crosser should retain information that is available in both parents. For the bitstring operators the first characteristic says that the result of an operator must still be a valid bitstring of the same length as the parents. The second characteristic says that when, for instance, the first bit of both parents is a 1, then the first bit of both children must be 1 too.

In a permutation the order in which the elements appear is important. The first characteristic
says that the result of an operator working on a permutation must still be a permutation of the same elements. The second characteristic says that when two elements have an equal ordering in both parents, they also should have the same ordering in the children. When in a permutation of integers the element labeled '1' precedes the element labeled '2' in both parents then the element labeled '1' should precede the element labeled '2' in both children too.

![Figure 4.1: A permutation mutator.](image)

The simplest way of defining a mutator on a permutation is choosing two elements of the permutation and swapping them. In figure 4.1 this is depicted for a permutation of integers.

![Figure 4.2: A permutation crosser.](image)

A crosser for permutations is depicted in figure 4.2. It does not fully fulfill the second characteristic, but it does a good job in trying to do so. When the parents are equal, for instance, the children will be equal to the parents. But, in the figure the element labeled '4' precedes the element labeled '5' in both parents, but it is the other way around in both
4.1. RESOURCE CONSTRAINED GENETIC SCHEDULING

children.

The permutation crosser works as follows: the first parent is split by some random mask in two sets: a 'pseudo-permutation' where some of the elements have changed to open slots, and the set of elements that can fill the slots to make a valid permutation. This second set is sorted in the order its elements appear in the second parent. Then the second set is put back in the slots of the pseudo-permutation to give the first child. The same operation is performed on the second parent to give the second child. Two elements that have the same order in both parents are only guaranteed to retain their order in a child if they are both in the pseudo-permutation or are both in the second set for that child. This method of crossing permutations is described in [Davis 91].

4.1.5 Implementation and results

For the general genetic algorithm I wrote some classes in the programming language C++ ([Stroustrup 91]). They implement the population and its activities. The selection functions for what genotypes are fit and what genotypes are unfit are the same as the ones used in the function-maximization example of chapter 3.

The non-genetic scheduling stuff was already to great extent programmed in the NEAT system ([NEAT 92]). The NEAT system is a general software frame in C++ for high level synthesis tasks, developed at the design automation section of the department of electrical engineering of Eindhoven University of Technology.

The combination of those two parts included writing a resource constrained list scheduler and writing the genotype classes necessary for this problem. The resource constrained list scheduler can handle multicycling, but cannot handle chaining, (module) pipelining and the availability of more than one kind of module for one type of operation.

The results were quite unexpected at first sight. For many of the graphs tested with this algorithm (only simple testgraphs, no real benchmarks) the (known) optimal result was already available in the randomly generated initial population! If it wasn't, the optimization sometimes succeeded, but only hesitantly. This was the result of the cost function being far from continuous. Of course the cost function for scheduling problems is discrete, but when only the number of cycles in the resulting schedule is taken into account in the cost function, the set of possible cost function values is very small. This phenomenon is called a badly discretized problem. The most extreme example of a badly discretized optimization problem would be a problem where the cost function is binary. This would be impossible to optimize by other means than randomly guessing solutions.
CHAPTER 4. GENETIC ALGORITHMS AND SCHEDULING

4.2 Time constrained scheduling using genetic algorithms

To make the problem of the previous section more suitable for being solved by genetic algorithms there must be more information taken into account in the cost function. The easiest way of doing this is trying to optimize not only the length of the schedule but also the cost of the modules used. In fact we arrive here at a general scheduling problem (in the sense of definition 2.18 instead of a resource constrained one. Some way of weighting the two goals (minimizing module costs and minimizing schedule time) must be defined for this type of schedule problem. Here a weighting function that makes the scheduler nearly a time constrained one will be used: The problem is to find a schedule of a given graph with a given library that fulfills a given time constraint and that has a minimal cost. The resulting scheduler will be referred to as the time constrained genetic scheduler.

4.2.1 Problem description

This scheduler will be given the following information:

- A graph \( G = (V, E) \)
- A library \( \Lambda \)
- An initial guess for the network module set \( L_0 \)
- An operation to operation type mapping \( \tau \)
- A operation type to module type mapping \( \mu \)
- A delay function \( d \)
- A time constraint \( t_{\text{max}} \)
- A module type cost function \( c(m) \)

The scheduler will return:

- A network module set \( L \)
- A binding function \( \hat{\xi} \)
- A valid schedule of the graph \( \Phi \)

The target is

- Minimize the cost \( C_I(L) \) (while meeting the time constraint)
4.2.2 Definition of phenotypes and cost function

The phenotypes in this problem are schedules, together with their network module set. For the cost function some weighted combination of the length of the schedule and the cost of the modules could be used. Practice however shows that best results come from the following cost function:

\[
c_p(p) = \begin{cases} 
-(M \cdot \#p) & \text{if } \#p > t_{max} \\
-(M \cdot t_{max} + C_L(L(p))) & \text{otherwise}
\end{cases}
\]

Here is \( M \) a number larger than the maximal possible value of the module costs \( C(L(p)) \). This cost function doesn't take into account the module costs unless the time constraint is met.

4.2.3 Definition of genotypes and translator

The genotype now consists of two chromosomes: the first chromosome is again a permutation of the nodes in the graph. The second chromosome is an array that gives for each module type in the library the number of modules in the network module set.

The translator is the same resource constrained list scheduler as the previous one. Apart from scheduling, the translator should now also copy the module information from genotype to phenotype.

4.2.4 Defining the operators

Of course the operators described in subsection 4.1.4 will be used for the permutation part of the genotype. Additional operators must be defined for the array of module counts. A crosser can be defined nearly analogous to the bitstring crosser of figure 3.6. Only now not arrays of bits are crossed, but arrays of integers.

A mutator for the module chromosome can be defined easily too: choose an index and add or subtract 1 from the value of the module array at that index. Of course one should take care that the resulting value is at least 1, otherwise the resulting network module set would no longer be complete. The number of modules might be set safely to 0 when operations mapped to this module type can be mapped to some other module type, as long as the resulting network module set is complete. However, this type of library (where an operation can be mapped to more than one module type) cannot be handled at all by the scheduler. The mutator is depicted in figure 4.3.

As crosser for the full genotype the combination of the permutation and array crossers is
used. The permutation and array mutators are however used separately. So three operators are used.

4.2.5 Results

Most runs I tried with this time constrained scheduler used the following parameters: The population size was 100 genotypes. In each generation 10% of the population was replaced by children. The simulations were stopped when 2500 children had been generated. The operators had the following weights: the crosser had weight 100, the mutators both had weight 30. The permutations in the initial population were chosen completely at random. The module arrays in the initial population were set to be equal to the module set given by the preselection tool. This is a separate program within the NEAT system, created by A. Timmer ([Timmer 92]). This program returns an initial guess for the number of modules necessary to schedule a graph, given a library and a time constraint (so the preselection tool is in fact a kind of allocator). The results the preselection tool gives are guaranteed to be minimal, i.e. when a schedule using the given network module set and meeting the given time constraint can be found, no better schedule will be found and the simulation can be stopped safely. Because of this, the genetic algorithm can often be stopped much sooner than after generating 2500 children.

Using these parameters several runs were done on several graphs. The graphs tested are depicted in appendix A. For each graph two libraries were tested: both contained one multiplier and one adder. Library 1 has a multiplier with a delay of two cycles, and library 2
has a multiplier with delay of one cycle. The adders always have a one-cycle delay. For each graph and library the scheduler was run on a large time constraint range (from the as fast as possible time till the time where a network module set consisting of one module of each type was sufficient). Sometimes a run gives a different result when a different random seed is used. All runs were done with four different random seeds.

In appendix B the results are shown. For some runs the network module set given by the preselection tool can never result in a schedule of the given time constraint. In those cases the best attainable result is shown in the best known module set column. The tables list two runtimes for each run. The first runtime is the time necessary to find the first occurrence of the best solution. The second runtime is the time the algorithm did run in total. For each graph/time-constraint/library combination the number of runs (for four seeds) that found the optimal solution is shown.

In appendix C the results of the same scheduler are shown in comparison to some other schedulers. The schedulers are compared to each other based on the cost of the module set used. Not visible in the tables are the speeds of the schedulers. Other virtues than speed and minimal functional unit costs are not considered in this thesis at all, but some of the other schedulers might have been designed with additional or different goals.

The results are very encouraging. There are only a few cases where the optimal network module set wasn’t found for all four seeds. In an even fewer number of cases the optimal solution wasn’t found at all. For most of those cases the optimal solution could be found by changing the weights of the operators. There were only two cases where another scheduler (a neural network scheduler) could find a better solution than the genetic algorithm scheduler (both for graph FDCT, for library 1 at time constraint 14 and for library 2 at time constraint 7). However, there are a lot more than two cases where the genetic scheduler did perform better than the neural net scheduler. It is well possible that a bit more experimenting with the parameters of the genetic algorithm would solve those two cases too. When comparing available schedulers (as in appendix C), the genetic scheduler outperforms all other schedulers!

Because no exact timing results were known for the other schedulers, no table with comparisons of runtimes is available. However, communications with the implementors of the other algorithms of appendix C indicate that the genetic scheduler is not only a high quality scheduler, but a fast one too.

Quite remarkable is the fact that an optimal solution is often available in the initial (randomly generated) population. These cases can be recognized in the tables of appendix B as the cases where execution times are nearly zero. Apparently, simply generating several random genotypes is for some cases enough to find the optimal solution. However, this occurs less for the technically interesting cycle times near the as fast as possible time constraint. When
the initial guess given by the preselection tool is wrong, this effect isn't useful either. Firstly, because in the initial population the module sets are set equal to the result of the preselection tool so no optimal results will occur in that initial population. Secondly, because an optimal result cannot be recognized in this case, and the full 2500 children will be generated before an answer is returned.

The cases where multiple runs with different seeds were necessary to find an optimum are frequent enough to think a bit longer about this. The effect that different random seeds produce different results is mainly due to the fact that populations converge too fast. In other words, the population sometimes gets trapped in a local minimum. An attempt to solve this problem is given in section 4.4.

The difficult cases encountered did not result from the greedy nature of the algorithm. It seems the greedy nature of the scheduler is seldom a problem. However, I have searched for a way to 'de-greedyize' the algorithm. This will be described in the next section.

Though the results so far are very promising, the algorithm is not good enough to be used in an industrial environment. The reason for this is that some important features of schedulers are not implemented in it yet.

1. Libraries with different module types that can implement the same operation cannot be handled now. For instance, a library with adders, multipliers and ALUs cannot be managed by the resource constrained list scheduler described so far.

2. Additional costs resulting from a schedule are not taken into account. It would be quite easy, for instance, to give a good guess of the number of registers necessary to implement a schedule, and to incorporate the chip-area and consumed power of those registers in the cost of a schedule.

3. Chaining and module pipelining must be supported.

4. Loops in the graph should be allowed

At the moment of writing of this thesis, work is done on writing a scheduler that does include the above features. In section 4.5 more will be said about the way these extensions are implemented.

4.3 A non-greedy genetic scheduler

One problem of list schedulers is their greedy nature. A way to de-greedyize the list scheduling can be deducted from figure 2.6: the problem is solved when the node causing
4.4 Avoiding local minima

Several runs of the scheduler of section 4.2 did not produce the optimal solution because the population collapsed into a local optimum. The reason for this was described at the end of section 4.1.3. The problem is that a local optimum occurs in the population in an early stage of development. This local optimum soon dominates the population and no other (global) optima are found any more. The collapse might be prevented by forcing the genetic algorithm to maintain variety in the population.

Several ways to maintain variety have been proposed in literature (for instance [Deb 89]). Many of the proposed methods try to prevent crossing of 'similar' genotypes. Others try to prevent the addition to the population of newly created genotypes 'similar' to ones already

the problem (the right multiplier) is somehow delayed. It should not be put in the free list immediately after it becomes available for scheduling, but only a given number of cycles later. The problem, however, is that it is not known what nodes should be delayed to get a faster schedule. It usually isn't even known whether a faster schedule can result from de-greedyation attempts. The genetic algorithm can be used to try several delays for the nodes.

To implement this idea I extended the genotype with an array of delays, one for each node. Initially these delays were set to zero. The mutator and crosser for this chromosome are equivalent to those for the module array (subsection 4.2.4).

When running this modified genetic algorithm on a special graph where the de-greedyation should have effect, the optimal solution was found. However it was only found after a long period of trying different seeds and carefully tuning the operator priorities. When running this modified algorithm on the graphs that were used for the previous algorithm, a decrease of performance could be noted. Apparently the delays hindered the finding of better greedy schedules.

It can be concluded that this de-greedyized genetic scheduler does not perform very well. Given the fact that for most graphs no improvement is to be expected at all from the de-greedyation, it may be wiser not to use this scheduler. However, when register costs are taken into account in the cost function, the de-greedyation might have a more profound effect, and it may be useful in that case. This has not been tested yet. Before giving a final judgement about the de-greedyized scheduler, it should be pointed out that many improvements are possible in the implementation. For instance, there are a lot of nodes for which it easily can be seen that adding a delay will not improve anything. These nodes could be excluded from adding a delay.

4.4 Avoiding local minima

Several runs of the scheduler of section 4.2 did not produce the optimal solution because the population collapsed into a local optimum. The reason for this was described at the end of section 4.1.3. The problem is that a local optimum occurs in the population in an early stage of development. This local optimum soon dominates the population and no other (global) optima are found any more. The collapse might be prevented by forcing the genetic algorithm to maintain variety in the population.

Several ways to maintain variety have been proposed in literature (for instance [Deb 89]). Many of the proposed methods try to prevent crossing of 'similar' genotypes. Others try to prevent the addition to the population of newly created genotypes 'similar' to ones already
occurring in the population. Yet others increase the probability of removing genotypes that 'resemble' other genotypes in the population.

All of these methods must somehow compare two genotypes to decide whether they are similar or not. This comparison can be done in two ways: the genotypes can be compared directly or the phenotypes corresponding to them can be compared. For the coding used in the genetic scheduler these two ways are quite different. There is a very large number of genotypes corresponding to one phenotype. Practice shows that nearly all genotypes in the final generation of a genetic run are different. Practice shows also that nearly all phenotypes of the final generation of a genetic run are equal, that is, the population has collapsed. A comparison based on the genotypes would imply that nearly all genotypes are considered different, and no real collapse prevention would occur. A comparison based on the phenotypes would show that in the final population nearly all genotypes are equivalent. This indicates that a comparison of the phenotypes should be used when trying to prevent a collapse of the population. However, it is possible that two different genotypes that both result in the same phenotype give rise to different phenotypes when crossed. When choosing phenotypes to perform the comparison, one of those two is easily removed from the population, so that crossing will never occur. Though the target was to maintain a richer population, it will get poorer in another sense too.

I have not yet tried to use this kind of algorithms to prevent collapse of the population.

4.5 Extending the genetic scheduler

At the time of writing this thesis, work is done to extend the genetic scheduler in several ways. First of all, the list scheduler is extended to handle chaining and pipelining. It is also extended to be able to use multiple module types for one operation type. Furthermore the new list scheduler now generates a guess for the number of registers necessary to implement the schedule. This information can be used to extend the cost function.

In this new scheduler a list of modules is attached to each node. It is the list of modules in the network module set on which that operation can be implemented. When an operation is in the free list a free module must be searched to perform that operation. Previously all modules of the type belonging to that operation were searched. In the new scheduler the modules on which the operation can be scheduled as described in the list attached to the operation are searched. This has several advantages:

- Multiple module types for one operation can be used now.
- A priority can be attached to each module for each operation by changing the order
of the list. The scheduler can be given a hint that a certain addition \textit{PLUS1} can be performed best on a certain adder \textit{ADD1} by putting \textit{ADD1} at the first position of the list of modules for \textit{PLUS1}.

- It is possible to specifically disable the mapping of an operation to a specific module by removing that module from the list.

These lists of modules (permutations of a subset of all modules) can be put in the genotypes, so the genetic algorithm can search for good orderings of the modules.

As the new scheduler is only being built now, results are not yet available.
In the work leading to this thesis I have tried to use genetic algorithms to solve scheduling problems. The main result of this work is a time constrained genetic scheduler (described in section 4.2).

The results produced by that scheduler (see appendix B and appendix C) are very promising. The genetic scheduler is very fast and produces very good results.

However, for many of the test runs, the genetic part of the algorithm is in fact not used at all. This is because the way schedules are coded is in a sense 'too good'. When a bunch of codings is generated at random to initialize the genetic algorithm, an optimal solution is often already among them. The benchmarks used for testing the genetic scheduler were rather small because no big benchmarks were available during the tests. It can be expected that the genetic part of the scheduler is used more when the graphs to be scheduled are larger.

There are several interesting practical cases that cannot be handled by the current genetic scheduler. To name a few:

- In the current scheduler, an operation in the data flow graph to be scheduled must be uniquely mapped to a module type. The case where, for instance, an addition can be mapped to both an adder as well as an ALU can not be handled yet.
- Operations cannot be chained.
- Pipelined modules cannot be used.
- Register and interconnect costs are not taken into account.
- Loops in the data flow graph cannot be scheduled.

At this moment an extended scheduler is being constructed that handles the first four cases.
Sometimes the population collapses into a local minimum. Several methods are proposed in literature to prevent a collapse of the population, but I haven't tried to use any of them. Many crossings and mutations do not result in new phenotypes because of the large number of different genotypes that result in the same phenotype. This fact possibly promotes the population collapse. Preventing an early collapse of the population is an important topic for further work.

The resource constrained list scheduler used internally in the genetic scheduler is greedy. Some solutions to scheduling problems can be missed by the genetic scheduler due to this. An attempt to de-greedyize the scheduler did not give satisfactory results: though non-greedy results were found, the general performance of the algorithm was reduced. It is very well possible that attempts to de-greedyize the scheduler have a more positive effect when register costs are taken into account and when mutations or crossings that don't result in new solutions are prevented, but I haven't experimented with that.
Bibliography

[Darwin1859] Darwin Ch.  

[Davis 91] Davis L. (ed.)  
Handbook of Genetic Algorithms.  

[Deb 89]  
Deb K. and D.E. Goldberg  
An Investigation of Niche and Species Formation in Genetic Function Optimization.  
In: Genetic algorithms. Proc. 3rd Int. Conf., George Mason University,  

[Devadas 89] Devadas S. and A.R. Newton  
Algorithms for Hardware Allocation in Data Path Synthesis.  
no. 7, p. 768-781.

[Eijndhoven 91] van Eijndhoven J.T.J et al.  
The ASCIS Data Flow Graph: Semantics and Textual Format.  
EUT Report 91-E-251.

High Level Synthesis, Introduction to Chip an System Design.  

[Goldberg 89] Goldberg D.E.
Amsterdam: Addison-Wesley, 1989.

An Efficient Microcode Compiler for Application Specific DSP Processors.

CASCH - a Scheduling Algorithm for High Level Synthesis.

[Heijligers 91] Heijligers M.J.M.
Time Constrained Scheduling for High Level Synthesis.

[Hemani 90] Hemani A. and A. Postula
A neural net based self organizing scheduling algorithm.

[Holland 75] Holland J.H.
Adaptation in Natural and Artificial Systems.

A Formal Approach to the Scheduling Problem in High Level Synthesis.
A Novel Integrated Scheduling and Allocation Algorithm for Data Path Synthesis.  

The High-Level Synthesis of Digital Systems.  

The Synthesis Approach to Digital System Design.  

The NEAT system.  
(Not published). 1

[Nestor 90] Nestor J.A. and G. Krishnamoorthy  
SALSA: A new approach to scheduling with timing constraints.  

[Parker 86] Parker A.C. et al.  
MAHA: A Program for Data Path Synthesis.  

[Paulin 89] Paulin P.G. and J.P. Knight  
Force Directed Scheduling for the Behavioral Synthesis of ASICs.  

[Philipsen 92] Philipsen W.J.M.  
Data Flow Graph Scheduling using Potts Neurons.  
In: Proc. of the Workshop on Circuits, Systems and Signal Processing,  

1The authors can be contacted at:  
Eindhoven University of Technology, Design automation section, Building EH, Floor 7,  
P.O. Box 513,  
5613 MB Eindhoven, the Netherlands
*Percolation based Synthesis.*  

[Stok 91] Stok L.  

[Stroustrup 91] Stroustrup B.  
*The C++ Programming Language (second edition).*  


*Module Selection and Scheduling using Unrestricted Libraries.*  

[Ullman 75] Ullman J.D.  
*Np-complete scheduling problems.*  

*Improved Force Directed Scheduling.*  
In this appendix the graphs on which the genetic scheduler was tested are shown.

A.1 Graph f2
A.2 Graph sehwa

sehwa
A.3 Graph fdct
A.4 Graph $wdelf$
A.5 Graph mwdelf

mwdelf
Appendix B

Results of the time constrained genetic scheduler

In this appendix the results of the genetic scheduler as described in section 4.2 are shown. The scheduler was run on the graphs depicted in appendix A. These graphs were run for two libraries. Library 1 (results are in section B.1) consists of a multiplier with a delay of two cycles and an adder with a delay of one cycle. Library 2 (results are in section B.2) also consists of a multiplier and an adder, but the multiplier has a delay of one cycle.

The results for each library/graph combination are shown in a table. The heading line of each table shows the graph name and the delays of the modules used. The first column (labeled 'TC') shows the time constraint used. The second column (labeled 'Pres.') shows the guess for the numbers of multipliers and adders necessary to meet the time constraint according to the preselection tool (these numbers are known to the genetic scheduler). The third column (labeled 'Opti:') shows the best known number of multipliers and adders when the guess in the second column seems to be incorrect. The following four main columns (labeled 'Run1' through 'Run4') contain the test results for four different random seeds. Each of them contains three subcolumns. The first subcolumn (labeled 't_{f1d}') contains the CPU-time used up till the first occurrence of the best genotype. The second subcolumn (labeled 't_{run}') contains the total CPU-time of the run. Both times exclude the time to read the graph. The times are in seconds and were measured on a HP 9000s/750 computer. The third subcolumn (labeled '.') contains a dot ('.') when the run succeeded in finding the optimal solution, that is, the solution occurring in the third or second main column. The last main column (labeled '#') summarizes the number of runs (out of four) that succeeded.
### APPENDIX B. RESULTS OF THE TIME CONSTRAINED GENETIC SCHEDULER

#### B.1 Library 1

#### B.1.1 Graph f2

<table>
<thead>
<tr>
<th>Run</th>
<th>Graph</th>
<th>Opt.</th>
<th>Delay(Multiplier) = 2</th>
<th>Delay(Adder) = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Run 1</td>
<td>Run 2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### B.1.2 Graph sehwa

<table>
<thead>
<tr>
<th>Run</th>
<th>Graph</th>
<th>Opt.</th>
<th>Delay(Multiplier) = 2</th>
<th>Delay(Adder) = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Run 1</td>
<td>Run 2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.1.3 Graph fdct

B.1.4 Graph wdelf
### B.1.5 Graph mwdef

<table>
<thead>
<tr>
<th>Graph mwdef</th>
<th>Delay(Mulitplier)</th>
<th>Delay(Adder)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run 1</td>
<td>Run 2</td>
</tr>
<tr>
<td></td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
</tr>
<tr>
<td>0.04</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>0.07</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>0.08</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>0.09</td>
<td>0.09</td>
<td>0.09</td>
</tr>
<tr>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
</tr>
<tr>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
</tr>
<tr>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
</tr>
<tr>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
</tr>
</tbody>
</table>

**THE TIME CONSTRAINED GENETIC SCHEDULER**
### B.2 Library 2

#### B.2.1 Graph f2

<table>
<thead>
<tr>
<th>N</th>
<th>Pres.</th>
<th>Optm.</th>
<th>Run 1</th>
<th>f_in</th>
<th>Run 2</th>
<th>f_in</th>
<th>Run 3</th>
<th>f_in</th>
<th>Run 4</th>
<th>f_in</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>2</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
</tbody>
</table>

#### B.2.2 Graph sehwa

<table>
<thead>
<tr>
<th>N</th>
<th>Pres.</th>
<th>Optm.</th>
<th>Run 1</th>
<th>f_in</th>
<th>Run 2</th>
<th>f_in</th>
<th>Run 3</th>
<th>f_in</th>
<th>Run 4</th>
<th>f_in</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td></td>
<td>0.200</td>
<td>0.200</td>
<td></td>
<td>0.200</td>
<td></td>
<td>0.200</td>
<td></td>
<td>0.200</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>0.500</td>
<td>0.500</td>
<td></td>
<td>0.500</td>
<td></td>
<td>0.500</td>
<td></td>
<td>0.500</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>0.220</td>
<td>0.220</td>
<td></td>
<td>0.220</td>
<td></td>
<td>0.220</td>
<td></td>
<td>0.220</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
<td></td>
<td>0.000</td>
</tr>
</tbody>
</table>
### B.2.3 Graph fdct

<table>
<thead>
<tr>
<th>Graph fdct</th>
<th>Delay(Multiplier) = 1</th>
<th>Delay(Adder) = 1</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### B.2.4 Graph wdelf

<table>
<thead>
<tr>
<th>Graph wdelf</th>
<th>Delay(Multiplier) = 1</th>
<th>Delay(Adder) = 1</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### B.2.5 Graph mwdelf

#### Table: Graph mwdelf

<table>
<thead>
<tr>
<th>TC</th>
<th>Freq.</th>
<th>Opt.</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Delay(Multiply)</th>
<th>Delay(Add)</th>
</tr>
</thead>
<tbody>
<tr>
<td>55</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.099</td>
<td>0.010</td>
<td>0.013</td>
<td>0.000</td>
</tr>
<tr>
<td>51</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3.530</td>
<td>22.220</td>
<td>3.460</td>
<td>22.530</td>
<td>2.790</td>
</tr>
<tr>
<td>47</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.100</td>
<td>0.670</td>
<td>0.670</td>
<td>0.000</td>
</tr>
<tr>
<td>46</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.090</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
</tr>
<tr>
<td>45</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
<td>0.120</td>
<td>0.000</td>
</tr>
<tr>
<td>44</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
<td>0.090</td>
<td>0.000</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
<td>0.090</td>
<td>0.000</td>
</tr>
<tr>
<td>42</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.120</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
</tr>
<tr>
<td>41</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
<td>0.090</td>
<td>0.000</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.140</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.140</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.140</td>
<td>0.000</td>
<td>0.100</td>
<td>0.000</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.150</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.160</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.170</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.180</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.190</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.210</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.220</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.230</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.240</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0.000</td>
<td>0.250</td>
<td>0.000</td>
<td>0.110</td>
<td>0.000</td>
</tr>
</tbody>
</table>
APPENDIX B. RESULTS OF THE TIME CONSTRAINED GENETIC SCHEDULER
Appendix

Comparison between schedulers

This appendix compares the results of some schedulers. Two libraries are used:

- Section C.1, adder 1 clock cycle, multiplier 2 clock cycles, equal costs
- Section C.2, adder 1 clock cycle, multiplier 1 clock cycle, equal costs

The schedulers mentioned are:

- List: Time constrained list scheduler ([Heijligers 91])
- Ifds: Improved force directed scheduler ([Verhaegh 91])
- Neural: Neural net scheduler ([Philipsen 92])
- Gen: Genetic algorithm scheduler (Described in section 4.2)
- Presel: Preselection tool (which is in fact no scheduler) ([Timmer 92])

The tables should be read as follows. In the first column the time constraint can be found in which the graph is to be scheduled. In the following columns the minimal number of modules ever found by a scheduler or by hand is shown. A dot (\(\cdot\)) in the table means that a scheduler has found this result. A dot for the preselector means that the preselector found this amount of modules hence this must be the minimal number of modules from which a schedule can be made. The genetic algorithm was run with four different random seeds for each result. A dot for the genetic algorithm indicates that the genetic algorithm indicates that all four produced the optimal result. A star (\(\ast\)) indicates that not all four random seeds produced the desired result, but at least one did. A circle (\(\circ\)) indicates that the result wasn’t found at first but could be found easily by changing some parameters of the algorithm. The same notation has been used for the neural network scheduler.
### C.1 Library 1

#### C.1.1 f2

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

#### C.1.2 sehwa

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
### C.1.3 fdct

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>4</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>4</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>0</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>0</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
### C.1.4 wdelf

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>3</td>
<td>3</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>1</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
</tbody>
</table>

### C.1.5 mwdelf

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds(n.a.)</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>3</td>
<td>3</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>66</td>
<td>3</td>
<td>3</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>67</td>
<td>3</td>
<td>3</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>68</td>
<td>3</td>
<td>3</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>69</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>70</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>71</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>72</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>73</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>75</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>76</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>77</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
<tr>
<td>78</td>
<td>2</td>
<td>2</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
<td>⬤</td>
</tr>
</tbody>
</table>
### mwdelf, continued

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds(n.a.)</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>2</td>
<td>2</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>81</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>85</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>86</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>87</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>89</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>93</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>98</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>2</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>1</td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
C.2 Library 2

C.2.1 f2

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>lfs</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>4</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

C.2.2 sehwa

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>lfs</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
### C.2.3  fdct

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>tids</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>8</td>
<td>5</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>4</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
</tbody>
</table>
### C.2.4 wdelf

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>2</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>34</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>36</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

### C.2.5 mwdelf

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds(n.a.)</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>2</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>54</td>
<td>2</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>55</td>
<td>2</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>56</td>
<td>2</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>57</td>
<td>1</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>58</td>
<td>1</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>59</td>
<td>1</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
<td>3</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>61</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>62</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>63</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>64</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>65</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>66</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>67</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>68</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>69</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
**C.2. LIBRARY 2**

---

### mwdefl, continued

<table>
<thead>
<tr>
<th>time</th>
<th>mult</th>
<th>adders</th>
<th>list</th>
<th>ifds(n,a.)</th>
<th>neural</th>
<th>gen</th>
<th>presel</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>71</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>72</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>73</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>74</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>75</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>76</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>77</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>78</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>79</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>81</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>82</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>83</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>84</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>85</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>86</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>87</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>88</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>89</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>90</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>91</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>92</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>93</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>94</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>95</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>96</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>97</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>98</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>102</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>103</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>104</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>105</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>106</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>107</td>
<td>1</td>
<td>2</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>108</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>109</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>110</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>112</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>113</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>114</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>115</td>
<td>1</td>
<td>1</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>
Index

SET*(powerset) .................................. 9
SET* (transitive closure) .......................... 8
N (nonnegative integers) .......................... 11
N+ (positive integers) ............................ 24
≺ (precedence) .................................. 8
G (genotype set) .................................. 23
P (phenotype set) .................................. 21

collapse ........................................ 39
combinatorial module ............................ 19
control graph ..................................... 4
cost ................................................ 10
of module ........................................ 10
of network module set ............................ 12
of schedule ....................................... 12
cost function ..................................... 21, 23
croiress ........................................... 24
crossing .......................................... 22
cycle steps ........................................ 5

data flow graph .................................. 4, 7
data introduction interval ......................... 19
degreegreedyized genetic scheduler ............ 46
delay ............................................... 10
of module ......................................... 10
of operation ....................................... 10
DII (data introduction interval) ................ 19
direct predecessor .................................. 9
direct successor ................................... 9
dynamic priority function ......................... 16
edge ............................................... 8
encoding .......................................... 23
bit-stream ......................................... 27
permutation ....................................... 39
feasible ............................................ 11

A

absolute library .................................. 11
absolute network module set ..................... 12
ALAP ............................................... 13
ALAP scheduler ................................... 13
ASAP ............................................... 13
ASAP scheduler ................................... 13
ASClS data flow graph ........................... 3
ASD (Architectural Synthesis Data) ............. 3
associated library ................................ 9
as late as possible (ALAP) ...................... 13
as soon as possible (ASAP) ..................... 13

B

badly discretized problem ......................... 41
binding function .................................. 10
bit-stream encoding .............................. 27

C

chaining .......................................... 19
ChildDim (of operator) ......................... 24
children .......................................... 24
choosing (roulette wheel) ....................... 30
chromosome ...................................... 23

cost function ..................................... 21, 23
croiress ........................................... 24
crossing .......................................... 22
cycle steps ........................................ 5

D

data flow graph .................................. 4, 7
data introduction interval ......................... 19
degreegreedyized genetic scheduler ............ 46
delay ............................................... 10
of module ......................................... 10
of operation ....................................... 10
DII (data introduction interval) ................ 19
direct predecessor .................................. 9
direct successor ................................... 9
dynamic priority function ......................... 16
edge ............................................... 8
encoding .......................................... 23
bit-stream ......................................... 27
permutation ....................................... 39
feasible ............................................ 11
fit ........................................ 22
fitness ................................. 22, 23
free list .............................. 15
fully used network module set ....... 12

G
GA (genetic algorithm) .................. 37
gene ..................................... 23
genetic scheduler .........................
de-greedyized .......................... 46
resource constrained .................... 37
time constrained ........................ 42
genotype .............................. 22, 23
sorting ............................... 31
greedy list scheduler ................... 17

H
High Level Synthesis .................... 1, 3
HLS (High Level Synthesis) .......... 3
hybridization ........................ 33

L
Layout Generation ....................... 1
library ................................... 6, 9
absolutely ............................. 11
list scheduling ........................ 14
Logic Synthesis ........................

M
maximization problem ................... 21
minimization problem ................... 21
module ................................... 9
combinatorial ........................ 19
sequential ............................ 19
module allocator ........................ 5
module binder ........................ 5
module pipelining ..................... 19
module type .......................... 9
multicycling .......................... 18
mutate .................................. 22
mutator .................................. 24

N
NEAT database .......................... 4
network graph ......................... 4
network module set .................... 9
  absolute ............................. 12
  cost .................................. 12
  fully used ........................... 12
node .................................... 7

O
operation ............................... 8
operation type ........................ 9
operator .................................. 24
  binary (crosser) ....................... 24
  unary (mutator) ....................... 24
optimization problem .................. 21
optimizer .................................. 5

P
ParentDim (of operator) ............... 24
parents .................................. 24
path ..................................... 8
permutation ............................ 39
crosser .................................. 40
mutator .................................. 40
phenotype ............................. 22, 23
pipelining ............................. 19
population ............................ 22, 23
precedence graph ..................... 8
predecessor ............................ 9
direct ................................... 9
relative .................................. 9
priority .................................. 30
  of genotypes ......................... 31
  of operators ........................ 30
priority function ........................................ 15
  dynamic ........................................ 16
  static ........................................ 16

time ........................................ 7
  timeslot ..................................... 7

random number generator ......................... 32
  in operator ................................ 28, 29, 34, 40, 44

relative predecessor ................................. 9
relative successor ................................. 9

resource constrained genetic scheduler .......... 37

roulette wheel choosing .......................... 30

S

schedule ........................................... 11
  cost .......................................... 12
  of graph ................................... 11
  of operation ................................ 11
  valid ........................................ 11

scheduler .......................................... 5
  ALAP ....................................... 13
  ASAP ....................................... 13
  genetic (de-greedyized) ....................... 46
  genetic (resource constrained) ............. 37
  genetic (time constrained) .................. 42

schedule function ................................ 11

scheduling ....................................... 13

scheduling problem ................................
  general ..................................... 12
  mixed ....................................... 12
  resource constrained ........................ 12
  time constrained ............................ 12

seed ............................................. 32

selector ......................................... 5

sequential module ............................... 19

sorting (of genotypes) ........................... 31

static priority function ......................... 16

successor ........................................ 9
  direct ....................................... 9
  relative .................................... 9

T
terminology ...................................... 23

time ............................................ 7

time constrained genetic scheduler .......... 42

transitive closure ................................ 8

translator ....................................... 23

type ........................................... 24
  of module ................................... 9
  of operation ................................ 9

type (of operator) .............................. 24

V

valid schedule .................................. 11

W

well allocated .................................. 11
(242) Kwaspen, J.J.M. and H.C. Heyker, J.I.M. Demarteau, Th.S. van de Roer
MICROWAVE NOISE MEASUREMENTS ON DOUBLE BARRIER RESONANT TUNNELING DIODES.

PRECISION OF AN EXPERIMENTAL (5-10 MW) DISK MHD FACILITY AND PROSPECTS OF COMMERCIAL (1000 MW) MHD/STEAM SYSTEMS.

(244) Klopstra, Martin and Ton van den Boom, Ad Damen
A COMPARISON OF CLASSICAL AND MODERN CONTROLLER DESIGN: A case study.
EUT Report 90-E-244. 1990. ISBN 90-6144-244-3

(245) Berg, P.H.G. van de
ON THE ACCURACY OF RADIOWAVE PROPAGATION MEASUREMENTS: Olympus propagation experiment.

(246) Maagt, P.J.I. de
A SYNTHESIS METHOD FOR COMBINED OPTIMIZATION OF MULTIPLE ANTENNA PARAMETERS AND ANTENNA PATTERN STRUCTURE.

(247) Jóźwiak, L. and T. Sponsko-Kwaśnica
DECOMPOSITIONAL STATE ASSIGNMENT WITH REUSE OF STANDARD DESIGNS: Using counters as submachines and using the method of maximal adjacencies to select the state chains and the state codes.

(248) Hoeijmakers, M.J. and J.M. Vliegbovum
DERIVATION AND VERIFICATION OF A MODEL OF THE SYNCHRONOUS MACHINE WITH RECTIFIER WITH TWO DAMPER WINDINGS ON THE DIRECT AXIS.

(249) Zhu, Y.C. and A.C.P.M. Backx, P. Eykhoff
MULTIVARIABLE PROCESS IDENTIFICATION FOR ROBUST CONTROL.

(250) Pfeifferhöfer, F.M. and P.J.M. Cluitmans, H.M. Kuipers
EMDABS: Design and formal specification of a datamodel for a clinical research database system.

(251) Eilendhoven, J.T.J. van and G.G. de Jong, L. Stok
THE ASCII DATA FLOW GRAPH: Semantics and textual format.

(252) Chen, J. and P.J.I. de Maagt, W.H.A.J. Hermen

(253) Haan, S.W.H. de
A PWM CURRENT-SOURCE INVERTER FOR INTERCONNECTION BETWEEN A PHOTOVOLTAIC ARRAY AND THE UTILITY LINE.
(254) Velde, M. van de and P.J.M. Cluitmans
EEG ANALYSIS FOR MONITORING OF ANESTHETIC DEPTH.

(255) Smolders, A.B.
AN EFFICIENT METHOD FOR ANALYZING MICROSTRIP ANTENNAS WITH A DIELECTRIC COVER USING A
SPECTRAL DOMAIN MOMENT METHOD.

(256) Backx, A.C.P.M. and A.A.H. Daenen
IDENTIFICATION FOR THE CONTROL OF MIMO INDUSTRIAL PROCESSES.

(257) Maagt, P.J.I. de and H.G. ter Morsche, J.L.M. van den Broek
A SPATIAL RECONSTRUCTION TECHNIQUE APPLICABLE TO MICROWAVE RADIOMETRY

(258) Wloesbouwers, J.M.
DERIVATION OF A MODEL OF THE EXCITER OF A BRUSHLESS SYNCHRONOUS MACHINE.

(259) Orlov, V.B.
DEFECT MOTION AS THE ORIGIN OF THE 1/F CONDUCTANCE NOISE IN SOLIDS.

(260) Rooijackers, J.E.
ALGORITHMS FOR SPEECH CODING SYSTEMS BASED ON LINEAR PREDICTION.

(261) Boom, T.J.J. van den and A.A.H. Daenen, Martin Klompstra
IDENTIFICATION FOR ROBUST CONTROL USING AN H-infinity NORM.

(262) Sgrates, M. and W. van Etten
LASER LINELWIDTH MEASUREMENT IN THE PRESENCE OF RIN AND USING THE RECIRCULATING SELF
HETERODYNE METHOD.

(263) Smolders, A.B.
RIGOROUS ANALYSIS OF THICK MICROSTRIP ANTENNAS AND WIRED ANTENNAS EMBEDDED IN A SUBSTRATE.

(264) Freriks, L.W. and P.J.M. Cluitmans, M.J. van Gils
THE ADAPTIVE RESONANCE THEORY NETWORK: (Clustering-) behaviour in relation with brainstem
auditory evoked potential patterns.

MANUFACTURING AND CHARACTERIZATION OF SiAs/AlGaAs MULTIPLE QUANTUMWELL RIDGE WAVESHAPE
LASERS.

(266) Cluitmans, L.J.M.
USING GENETIC ALGORITHMS FOR SCHEDULING DATA FLOW GRAPHS.