UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Algorithms and tools for automatic generation of DSP hardware structures

Pedro Filipe Zeferino Tomás
(*Licenciado*)

Dissertação para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: Doutor Leonel Augusto Pires Seabra de Sousa

Membros do jurí

Presidente: Doutor Leonel Augusto Pires Seabra de Sousa
Vogais: Doutor João Manuel Paiva Cardoso
Doutor Ana Teresa Correia de Freitas

Janeiro de 2006

Abstract

The increased complexity of Digital Signal Processing (DSP) algorithms demands for the development of more complex and more efficient hardware structures. The work presented herein describes the core components for the development of a tool capable of automatic generation of efficient hardware structures, therefore facilitating developers work. It comprises algorithms and techniques for *i)* balancing the paths in a graph, *ii)* scheduling of operations to functional units, *iii)* allocating registers and *iv)* generating the VHDL code. Results show that the developed techniques are capable of generating the hardware structure of typical DSP algorithms represented in data-flow graphs with over 2,000 nodes in around 200 ms, scaling to 80,000 nodes in about 214 s. Within the developed techniques, solving the scheduling problem is one of the most complex tasks: it is a NP-complete problem and directly influences the number of functional units and registers required. Therefore, experimental analysis was made on scheduling algorithms for time-constrained problems. Results show that simple list-based algorithms are more efficient in large problems than more complex algorithms: they run faster and tend to require less functional units.

Keywords

High-level synthesis, digital signal processing, scheduling, retiming, hardware generation, register allocation.

Resumo

O aumento da complexidade dos algoritmos de Processamento Digital de Sinal (PDS) exige o desenvolvimento de arquitecturas mais complexas e mais eficientes. O trabalho aqui apresentado descreve as principais componentes para o desenvolvimento de uma ferramenta capaz de gerar automaticamente arquitecturas eficientes, facilitando assim o trabalho dos engenheiros. Esta ferramenta é constituída por algoritmos e técnicas capazes de *i)* equilibrar os caminhos de um grafo, *ii)* escalonar as operações nas unidades funcionais, *iii)* alocar os registos e *iv)* gerar o código VHDL. Os resultados apresentados mostram que a ferramenta desenvolvida é capaz de gerar o código VHDL de estruturas típicas de PDS com mais de 2.000 nós em cerca de 200 ms, escalando para 8.000 nós em 214 s. Das técnicas desenvolvidas, o escalonamento das operações é uma das mais complexas: é um problema NP-completo e influencia directamente o número de unidades funcionais e registos necessários. Assim, foi feita uma análise experimental de várias técnicas de escalonamento para problemas restritos temporalmente. Os resultados mostram que, para problemas complexos, algoritmos simples baseadas em listas são mais eficientes que algoritmos complexos: têm menor tempo de execução e requerem menos unidades funcionais.

Palavras Chave

Sintese de arquitecturas, processamento digital de sinal, escalonamento, retemporização, geração de hardware, alocação de registos.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Objectives 3

1.3 Main Contributions 4

1.4 Dissertation Outline 5

2 Background on High Level Synthesis **7**

2.1 Loop Bound and Iteration Bound 9

2.2 Retiming 11

2.3 Unfolding 13

2.4 Folding 15

2.4.1 Scheduling of nodes to execution units 18

3 Algorithms for High-Level Synthesis **21**

3.1 Critical Path Analysis 23

3.2 Retiming 27

3.3 Scheduling 32

3.3.1 Force-Directed Schedulers 34

3.3.2 Adapted Static-Based List Scheduler 37

3.3.3 Time-Constrained List-Based Scheduler 39

3.4 Register Allocation 43

3.5 Conclusions 47

4 Tool Architecture and Implementation **49**

4.1 Describing the DFG 52

4.2 Data structures 56

4.3 Programming the developed tools 64

4.3.1 Speeding up execution 65

4.3.2 Data path binding: VHDL code generation 67

4.4 Conclusions 68

5 Experimental Results **71**
5.1 Evaluating Benchmarks 72
5.1.1 Artificial Neural Network 72
5.1.2 Fast Fourier Transform 73
5.1.3 Random Networks 75
5.1.4 Typical Scheduling Benchmarks 76
5.2 Performance of the Algorithms 77
5.2.1 Reading the input file 78
5.2.2 Critical Path Analysis 80
5.2.3 Retiming 80
5.2.4 Time Constrained List Based Scheduling algorithms 82
5.2.5 Comparative analysis of the scheduling algorithms 85
5.2.6 Register allocation 89
5.2.7 Describing the architecture in VHDL code 91
5.3 Comparative evaluation with optimal scheduling 92
5.4 Improving scheduling results 93
5.5 Conclusions 94

6 Conclusions **97**
6.1 Future Work 100

A Proofs of High-Level Synthesis properties **105**
A.1 Retiming 106
A.2 Unfolding 107
A.3 Folding 109

B Additional experimental results **111**
B.1 FFT benchmark 112

List of Figures

1.1 Main components of the ATAS project. 3

2.1 Sample algorithm: C-code. 8
2.2 Sample algorithm: graph translation. 10
2.3 Sample data-flow graph. 11
2.4 Retiming of a first order IIR filter. 12
2.5 2-unfolding of a first order IIR filter. 15
2.6 Folding of first order IIR filter. 17

3.1 Result of the Critical path analysis on a sample DFG. 24
3.2 Example DFG. 28
3.3 Main computing phases of the TCLBS algorithm. 41
3.4 DFG a first order IIR filter and its scheduling values. 44
3.5 Circular lifetime chart of the second order IIR filter. 45
3.6 Register allocation of the second order IIR filter. 47

4.1 Architecture of the ATAS tool. 51
4.2 DFG of the 5th order elliptical wave filter. 53
4.3 Global declarations for the 5th order elliptic wave filter. 54
4.4 Declaration of a functional unit type: multiplier. 54
4.5 Declaration of the nodes to describe the 5th order elliptical wave filter. 55
4.6 Excerpt of the 5th order elliptical wave filter connection declaration. . 56
4.7 Hierarchy of the implemented data structures. 57
4.8 Structure of the data type which stores the architecture. 58
4.9 Structure of the data type to store node type relative information. . . 59
4.10 Structure of the data type to store node relative information. 61
4.11 Structure of the data type which defines the edges. 62
4.12 Structure of the data type which store variable relative information. . 63
4.13 Structure of the data type which stores variable definitions. 63
4.14 Memory allocation technique. 66

4.15 Structure of the generated VHDL code. 68

5.1 The structure of the Artificial Neural Networks. 73
5.2 Structure of a Fast Fourier Transform. 74
5.3 Probability distribution of edge weights. 76
5.4 Performance of the algorithm to read the DFG (benchmark A1). . . . 79
5.5 Performance of the algorithm to analyse the critical path (benchmark A1). 80
5.6 Performance of the retiming algorithm. 81
5.7 Performance of the TCLBS algorithms (benchmark A3). 83
5.8 Example DFG. 84
5.9 Computational time of the TCLBS algorithms. 85
5.10 Performance of the scheduling algorithms (Benchmark A3). 86
5.11 Performance of the scheduling algorithms (Benchmark R1). 87
5.12 Performance of the FDS algorithm by varying the mobility reduction factor. 88
5.13 Performance of the scheduling algorithms (Benchmark A4). 89
5.14 Performance of the scheduling algorithms (Benchmark R2). 90
5.15 Results on register allocation after applying different scheduling algorithms. 91
5.16 Time to perform register allocation 92
5.17 Time to generate the VHDL code. 93
5.18 Performance of a ASLBS+TCLBS combined scheduling (benchmark R1). 94

B.1 Performance of the scheduling algorithms (Benchmark F1). 112
B.2 Performance of the scheduling algorithms (Benchmark F2). 113
B.3 Results on register allocation after applying different scheduling algorithms. 114

List of Tables

2.1 Schedule times for folding of first order IIR filter 16

3.1 Complexity of the TCLBS algorithm in the different scheduling options. 43
3.2 Scheduling results, in c-steps, of the second order IIR filter. 44

5.1 Benchmarks used to evaluate the computational time. 78
5.2 Description of the Personal Computer used to assess the performance of the ATAS application algorithms. 79
5.3 Heuristic comparison for the EWF and 1-D DCT benchmarks. 96

List of Algorithms

2.1 Unfolding 14
2.2 Folding 16
2.3 Translating the scheduling problem into a ILP problem 19
3.1 Critical path analysis 25
3.2 Compute shortest path and maximum delay from node U to all nodes V 31
3.3 Force-Directed Scheduling 36
3.4 Adapted Static List-Based Scheduling 38
3.5 Time-Constrained List-Based Scheduler 40
3.6 Life-time analysis algorithm 45
3.7 Forward-backward register allocation algorithm 46

List of Acronyms

ANN : Artificial Neural Network

ASAP : As Soon As Possible

ASLBS : Adapted Static List Based Scheduler

ALAP : As Late As Possible

ATAS : Algorithms and Tools for Automatic hardware Synthesis

BD : Block Diagram

c-step : control-step

CP : Critical Path

DCT : Discrete Cosine Transform

DFG : Data-Flow Graph

DSP : Digital Signal Processing

EWF : Elliptical Wave Filter

FDS : Force Directed-Scheduler

FFT : Fast Fourier Transform

HDL : Hardware Description Language

HLS : High-Level Synthesis

IIR : Infinite Impulse Response

ILP : Integer Linear Programming

LBS : List-Based Scheduler

PAPI : Performance Application Programming Interface

PB : Processing Block

RTL : Register-Transfer Level

SLBS : Static List-Based Scheduler

TCLBS : Time-Constrained List-Based Scheduler

VHDL : Very High Speed Integrated Circuit (VHSIC) Hardware Description Language

XML : eXtensible Markup Language

1

Introduction

Contents

1.1 Motivation 2
1.2 Objectives 3
1.3 Main Contributions 4
1.4 Dissertation Outline 5

1.1 Motivation

As the complexity of Digital Signal Processing (DSP) algorithms grows, so does the complexity for developing space-efficient architectures. Moreover, due to the increased complexity of these problems, designers tend to initially developed their algorithms in high-level languages. This allows one to concentrate on the details of the mathematic equations rather than on the implementation characteristics. The most important aspect during the initial development stage is getting the algorithm to work properly, not how fast it runs, how much circuit area is required for its implementation, or how much power it will consume. However, for hardware architectures, after the initial stage, the algorithm has to be completely re-written into a Hardware Description Language (HDL), so that it can be integrated as a product. By adopting this two stage development of the product, one can increase the development speed, thus decreasing the *time-to-market*. However the second stage can be tedious and painful: the whole code has to be transformed and needs to be re-written into another language. Also, during this stage it is likely for errors to be encountered: typos or simply incorrect translations of the original algorithm into HDL code introduces bugs in the hardware that may take long to be found.

The solution for the problem is the development of a tool capable to automate the process of translating the algorithm, written in a high-level language such as MATLAB, C/C++ or Java, into a HDL [1–5]. This will simplify the developers work, while allowing for the development of efficient architectures. The tool to perform this task should therefore be able to identify and translate the operations described in the high level language and to produce the equivalent graph of the DSP algorithm. Several difficulties are known to exist in performing this task, such as the identification of loops and dependencies, manipulation of pointers and dynamic memory or identification of recursivity. After executing this tasks other operations need to be performed in order to obtain a Register-Transfer Level (RTL) description of the DSP algorithm, such as balancing the weight of computational paths in the graph (retiming), scheduling of operations and the allocation of registers to store initial, intermediate and/or final variables of the algorithm. In this sense, we firstly need to define the constraints for the generated architecture, such as circuit area, power consumption, time to perform each iteration or all of the above. After setting the initial constraints, the problem will be to minimise the others while being able to fulfil the set of constraints.

Two main problems can be identified when trying to minimise area, power or iteration time. Firstly describing the constraints in a mathematical equation that

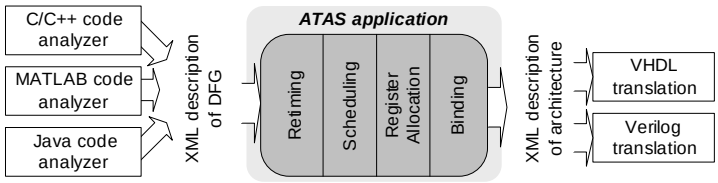


Figure 1.1: Main components of the ATAS project.

allows for the development of an algorithm capable of reaching optimal solutions is not a easy nor trivial task. Secondly, even considering a precisely described model, obtaining an optimal solution is shown to be NP-complete (in fact the scheduling of nodes to functional units alone is a NP-complete problem [6]).

Nevertheless, if one would be able to create a tool capable of automatic translation of a high-level language into a RTL description, a next step could be taken: designing an automatic software/hardware co-design tool. In this case, through the identification of the complex and computational intensive structures of the Data-Flow Graph (DFG), one could ally the benefits of sequential processors with the advantages of parallelism provided by dedicated hardware processors.

1.2 Objectives

With the purpose of creating a tool capable to automatically create efficient hardware structures, a research project was initiated [7] to develop the Algorithms and Tools for Automatic hardware Synthesis (ATAS) application. As shown in Figure 1.1, it consists on developing a tool capable of translating a high-level description of a DSP algorithm into an RTL description, such as Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) or Verilog. The goal of this project is as follows: *i)* development of a tool capable of translating a high-level description into a graph like format (currently, work is being made for C language); *ii)* creation of a second tool, hereafter identified as the ATAS application, capable of performing all graph transformations required to produce the datapath and its corresponding controller; *iii)* description of the datapath in a RTL like format. The final goal of the project is to create several tools in a modular way, in order to be able to easily integrate new functions and algorithms. The master thesis herein presented is focused on the development of the second task, which is the core of the project: the High-Level Synthesis (HLS) of any algorithm described by a graph. In this sense, the ATAS application input is the DFG of the algorithm and the output is a description of the dedicated processor by using VHDL. Since the development of the ATAS application started before the creation of the project, the eXtensible

Markup Language (XML) [8] description is not yet implemented. Instead, the DFGs are written in a dedicated input language and the VHDL code is generated in the last stage of the application.

1.3 Main Contributions

The developed application is focused on generating efficient architectures for large[1] time-constrained problems. This goal is in fact rather limitative: when the number of computations grows too large, the use of optimal algorithms is no longer feasible since the time to reach a solution is too large. Although techniques for all the shady blocks in Figure 1.1 have been developed and implemented, the research has been mainly focused on algorithms for retiming and scheduling of nodes. Performing retiming can be written as a Integer Linear Programming (ILP) problem. However, the number of constraints tends to be very large, thus leading to a long execution time. In this sense, the ATAS tool implements constraint pruning techniques which, by reducing the number of constraints in over 16,000 times, allow a decrease in the execution time of over 99%. Scheduling algorithms, the second focus of research, are typically developed by using small benchmarks of up to 100 operations. Thus, the research in this field of study is usually made on the development of an algorithm capable of performing scheduling fast, while obtaining optimal or close to optimal solutions [9–12]. However, other sub-optimal schedulers, such as the Force Directed-Scheduler (FDS) [13, 14], have been investigated. Nonetheless, the development of this algorithm was made for small problems. Consequently, as will be shown with experimental results, it fails to retrieve the general properties of large graphs resulting in a very inefficient scheduling. Therefore, is thus mandatory to develop new time-constrained scheduling algorithms. Experimental results show that the proposed algorithms are 10 to 10,000 faster and require more than 20% less resources for the synthesis of DSP problems.

Developing fast algorithms is however insufficient in order to synthesise algorithms with over 10,000 iterations: efficient implementations are also very important. As a result, the implementation of the ATAS tool was carried out taking into consideration two main aspects: *i)* the speed of execution and *ii)* the ease to add new features and new algorithms in the future. The first aspect has lead to the development and implementation of a new memory management system which, by reducing the number of calls to system functions, allows for a reduction of the execution time of up to 66%. Furthermore, in order to speed up the task of reading

[1] When the total number of computations is beyond 10,000 per iteration

and validating a DFG, a dynamic hashtable was implemented thus allowing for a reduction in the execution time of this task of up to 93%.

1.4 Dissertation Outline

The work presented in this dissertation is organised in four chapters. In Chapter 2 an introduction to HLS algorithms is presented. This chapter, starts by presenting the basic concepts of creating the graph representation of an algorithm. Following, some of the most important HLS techniques are also presented, namely for: *i)* computing the iteration bound, therefore defining the minimum iteration time, *ii)* retiming, which allows to obtain a better balance of the computational paths, *iii)* unfolding, that expands the DFG such to increase the level of parallelism, and finally *iv)* folding, which allows to optimise the use of functional units and therefore circuit area required. In Chapter 3, the algorithms for high-level synthesis are presented, namely the ones concerning critical path analysis, retiming, scheduling and register allocation. The complexity analysis for each of the presented algorithms is also made. In Chapter 4 the implementation aspects of the ATAS tool are presented, namely the used data structures and the most relevant characteristics responsible for the speed-up in the execution of the application. Within this chapter, the language used for inputing DSP algorithms is described and the structure of the generated VHDL code is presented. Experimental results are presented in Chapter 5 as well as the benchmarks used for the evaluating the performance of the algorithms. In what concerns scheduling, results are also presented for comparing the performance of the proposed algorithms with those of typical time-constrained schedulers, i.e. the FDS. This analysis considers: *i)* the number of functional units allocated, *ii)* the number of registers required and *iii)* the time required for the execution of the scheduling algorithm. Finally, in Chapter 6 the main conclusions are drawn and future work is outlined.

[1]The iteration time represents the time necessary to perform 1 iteration of the algorithm.

2

Background on High Level Synthesis

Contents

2.1 Loop Bound and Iteration Bound 9
2.2 Retiming 11
2.3 Unfolding 13
2.4 Folding 15

The first step into the translation of an algorithm into a RTL description is to build a DFG that represents the algorithm. This transformation consists on creating a level of abstraction which allows to easily apply several transformations to the algorithm, such as: *a) retiming*, allowing for redistributing the weight of the edges in the graph (see section 2.2); *b) unfolding*, that by unrolling the iterations of the algorithm, allows to explore its parallelism (see section 2.3); or *c) folding*, which by mapping different operations into the same functional units reduces the circuit area (see section 2.4). This transformation can be summarised as follows:

1. draw a node U for each operation in the algorithm;

2. for each variable, draw an edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$ from the node U that generates its value to each node V that uses it;

3. set the weight of each node U ($\mathtt{t}_{\mathsf{U}}$) as the computational time required for performing the operation; for hardware implementation purposes the computational weight of a node is measured by the number of clock cycles, or control-steps (c-steps), that the functional unit takes to execute the operation;

4. set the weight of each edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$, ($w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$), as the difference of iterations between the generation of the variable by node U and the consumption of the variable by node V.

```
while (1) {
   x=inputValue();
   /* shift-register */
   a[2]=a[1];
   a[1]=a[0];
   a[0]=x;

   /* processing */
   for (i=0;i<3;i++)
      c[i]=a[i]*b[i];
   y=c[2]+c[0]+2*c[1];

   /* shift-register */
   b[2]=b[1];
   b[1]=b[0];
   b[0]=y;
   outputValue(y);
}
```

Figure 2.1: Sample algorithm: C-code.

For instance, let us consider the sample C-code in Figure 2.1. As it is shown, the code is executed an infinite number of iterations. The functions `inputValue()` and `outputValue()` are used to get a new value for processing and to output the result, respectively. The algorithm is composed by two shift-register blocks (one for the input value and another for the output value) and a processing block. Based on this, one can easily construct the Block Diagram (BD) shown in Figure 2.2(a). To translate the BD into a DFG it can be noted that each shift-register element introduces a one-level inter-cycle dependency. Thus, the assignment of variables $c[0]$, $c[1]$ and $c[2]$ at iteration n, $c[0]\{n\}$, $c[1]\{n\}$ and $c[2]\{n\}$ respectively, can be written as

$$\begin{cases} c[0]\{n\} = \quad x\{n\} \quad \times y\{n-1\} \\ c[1]\{n\} = x\{n-1\} \times y\{n-2\} \\ c[2]\{n\} = x\{n-2\} \times y\{n-3\} \end{cases}$$

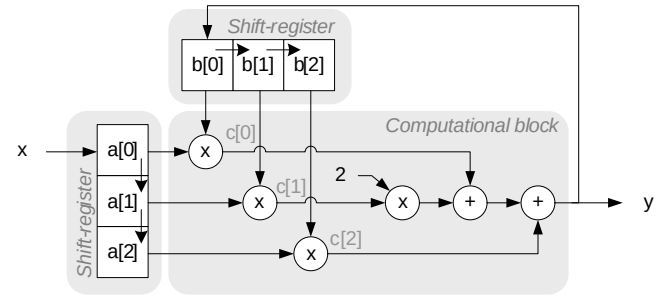
Describing the shift-register in a DFG is made by edges with weights equal to the inter-cycle dependency, as shown in Figure 2.2(b): the multiplications are identified by m1, m2, m3 and m4 and the additions by a1 and a2.

The weight of an edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$, $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$, between two nodes U and V defines the level of dependency between the execution of both nodes. Similarly so does the weight of a path P, w_{P}. In the example of Figure 2.2, consider the paths $\mathsf{P1} = \mathsf{m4} \rightarrow \mathsf{a1} \rightarrow \mathsf{a2}$ and $\mathsf{P2} = \mathsf{m4} \rightarrow \mathsf{a1} \rightarrow \mathsf{a2} \rightarrow \mathsf{m3}$. The path P1 as a weight, i.e., the sum of weight of the edges in the path, equal to zero. Thus a2 has an intra-cycle dependency to node m4: computing a2 at iteration n depends on the result of operation m4 at iteration n. On the other hand, the weight of the path P2 is 3, so computing node m3 depends on the value produced by m4 at iteration $n-3$.

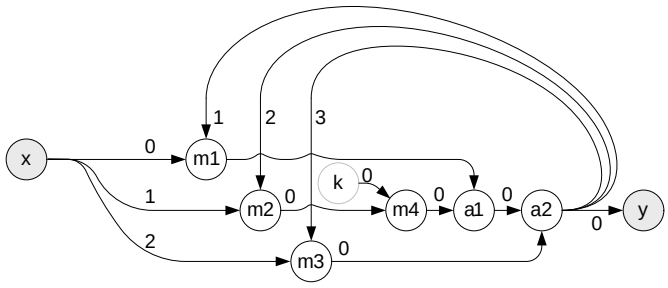
Intra-cycle dependencies define the order by which operations must be performed. If a node as a zero weight path leading to it, than it can only be executed after the execution of all the nodes in that path. Due to this restriction, a zero weight path is usually refereed to as a Critical Path (CP). The CPs of the example C-code can be identified in Figure 2.2(c) as the edges with thicker lines. In this figure, and for the remaining designs presented in this dissertation, if an edge does not have its weight expressed in the graph, than it is assumed as zero.

2.1 Loop Bound and Iteration Bound

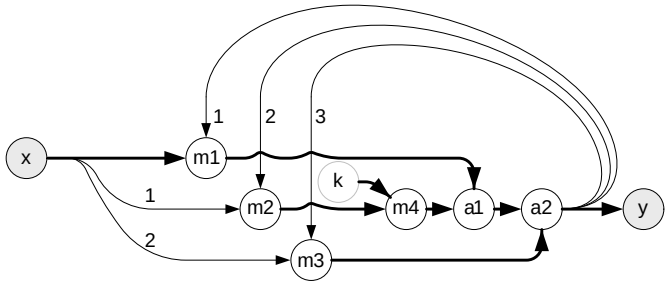
The *Loop Bound* of a cycle P (LB_{P}) defines the minimum time required to execute a cycle. It can be computed as the sum of weights of the nodes in the cycle,



(a) Block diagram.



(b) Data-flow graph.



(c) Potential critical paths of the data-flow graph.

Figure 2.2: Sample algorithm: graph translation.

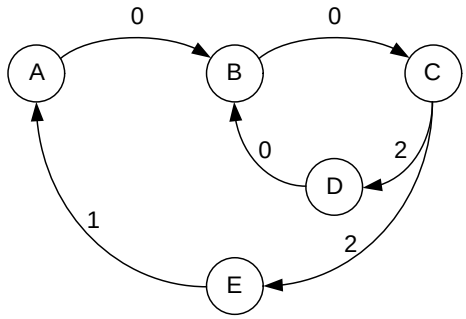
t_{P}, divided by the sum of the weights of the edges in that cycle, w_{P}:

$$LB_{\mathsf{P}} = \frac{t_{\mathsf{P}}}{w_{\mathsf{P}}} \tag{2.1}$$

For instance, looking at the DFG in Figure 2.3 two loops can be identified: $\mathsf{L}_1 = \mathsf{B} \rightarrow \mathsf{C} \rightarrow \mathsf{D} \rightarrow \mathsf{B}$ and $\mathsf{L}_2 = \mathsf{B} \rightarrow \mathsf{C} \rightarrow \mathsf{E} \rightarrow \mathsf{A} \rightarrow \mathsf{B}$. Assuming that the computational time of a node X is $\mathtt{t}_{\mathsf{X}}$ and the weight of the edge connecting node X to node Y is $w_{\langle \mathsf{X} \rightarrow \mathsf{Y} \rangle}$, the loop bound of the cycles L_1 and L_2 can be computed as:

$$LB_{\mathsf{L}_1} = \frac{\mathtt{t}_{\mathsf{B}} + \mathtt{t}_{\mathsf{C}} + \mathtt{t}_{\mathsf{D}}}{w_{\langle \mathsf{B} \rightarrow \mathsf{C} \rangle} + w_{\langle \mathsf{C} \rightarrow \mathsf{D} \rangle} + w_{\langle \mathsf{D} \rightarrow \mathsf{B} \rangle}} = 3$$

$$LB_{\mathsf{L}_2} = \frac{\mathtt{t}_{\mathsf{B}} + \mathtt{t}_{\mathsf{C}} + \mathtt{t}_{\mathsf{E}} + \mathtt{t}_{\mathsf{A}}}{w_{\langle \mathsf{B} \rightarrow \mathsf{C} \rangle} + w_{\langle \mathsf{C} \rightarrow \mathsf{E} \rangle} + w_{\langle \mathsf{E} \rightarrow \mathsf{A} \rangle} + w_{\langle \mathsf{A} \rightarrow \mathsf{B} \rangle}} = 2$$



	A	B	C	D	E
Computational time (c-steps)	2	1	1	4	2

Figure 2.3: Sample data-flow graph.

The *iteration bound* (T_∞) is equal to the maximum loop bound of all the cycles in the graph, and represents the minimum number of periods required to execute one iteration of the algorithm represented by the DFG. In the example of Figure 2.3, the maximum loop bound is l_1 with value 3. Therefore the iteration bound is $T_\infty = 3$.

Note that there is no guarantee that the value of the iteration bound is an integer value. Instead it is likely for it to be a rational number. In this case, the minimum number of cycles to compute the algorithm will be the next integer (rounding up) of the iteration bound (unless unfolding techniques are applied, see Section 2.3).

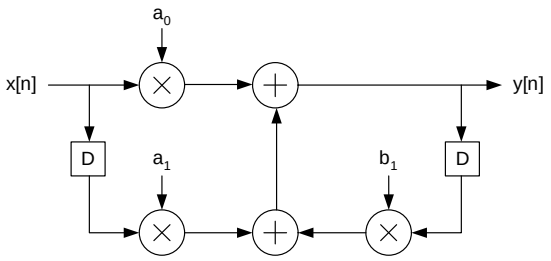
2.2 Retiming

Retiming is a transformation technique which allows to change the positions of the delay elements in a circuit. This technique is mostly used for: *i)* reducing the critical path(s), therefore decreasing the overall clock period of the circuit; *ii)* decrease the number of registers required; *iii)* lowering the power consumption.

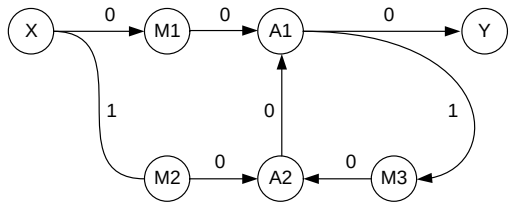
The retiming of a node X by a value r_X can be interpreted as increasing all input edges of node X by a value of r_X and by decreasing all output edges by the same value. Expanding this concept, it is possible to retime an entire circuit by assigning that each node in the DFG has a different retiming value. In this case, after retiming, an edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$ will increase its weight from $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$ to $w^r_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$, as shown in (2.2).

$$w^r_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} = w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} + r_\mathsf{V} - r_\mathsf{U} \tag{2.2}$$

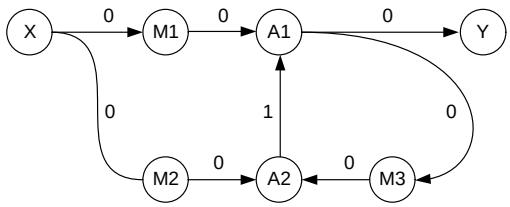
The concept of retiming is illustrated in Figure 2.4, where several diagrams of



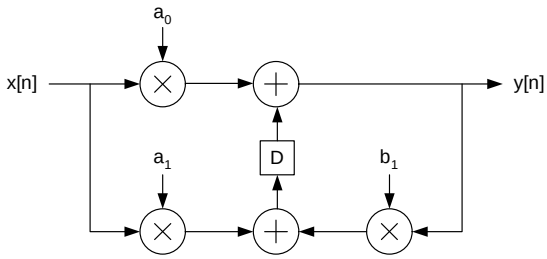
(a) Block diagram.



(b) Data-flow diagram.



(c) Data-flow diagram of retimed circuit.



(d) Block diagram of retimed circuit.

Node	X	Y	A1	A2	M1	M2	M3
Retiming	0	0	0	-1	0	-1	-1

(e) Retiming values

Figure 2.4: Retiming of a first order IIR filter.

the same first order Infinite Impulse Response (IIR) filter are shown: Figure 2.4(a) presents the original block diagram and Figures 2.4(b) and 2.4(c) show the original and retimed DFG by applying the retiming values in Figure 2.4(e). The block diagram of the circuit obtained after retiming is also presented in Figure 2.4(d).

Note that the retiming technique does not change the function of the original algorithm, instead it simply changes the position of the delay elements as illustrated on the example in Figure 2.4.

Several properties can be observed when performing the retiming of a DFG. Below 4 of the most important are presented (proofs for the given properties are given on Appendix A).

Property 2.2.1. *The weight of a retimed path* $\mathsf{P} = \mathsf{V}_0 \rightarrow \mathsf{V}_1 \rightarrow \ldots \rightarrow \mathsf{V}_k$, w^r_{P}, *in a circuit does not depend on the nodes it goes through,* $\mathsf{V}_1 \ldots \mathsf{V}_{k-1}$, *but only on the first and last nodes,* V_0 *and* V_k, *and on the original path weight,* w_{P}. *The retimed path is therefore defined as:*

$$w^r_{\mathsf{P}} = w_{\mathsf{P}} + r_{\mathsf{V}_k} - r_{\mathsf{V}_0} \tag{2.3}$$

Property 2.2.2. *The weight of a closed path* $\mathsf{P}_{\circlearrowright} = V \rightarrow \ldots \rightarrow V$, $w^r_{\mathsf{P}_{\circlearrowright}}$, *does not change after retiming.*

Property 2.2.3. *The loop and iteration bounds of a DFG do not change by retiming the circuit.*

Property 2.2.4. *Retiming all nodes in a path* $\mathsf{P} = \mathsf{U}_1 \rightarrow \cdots \rightarrow \mathsf{U}_\mathsf{N}$ *by the same value* k *(*$\mathtt{r}_{\mathsf{U}_1} = \cdots = \mathtt{r}_{\mathsf{U}_\mathsf{N}} = k$*) does not change the weight of the edges within* P.

2.3 Unfolding

The *unfolding* technique, or *loop unrolling* as designated in compiler theory, consists on transforming a DFG to allow the execution of multiple iterations of the algorithm in just one loop cycle. For this matter, *unfolding factor* is defined as the number of times a loop is unrolled, i.e. the number of iterations to perform per cycle. However this transformation generates a number of nodes/edges in the final DFG proportional to the unfolding factor. Therefore, it increases the complexity of the DFG and requires more computational resources to execute a single iteration of the unfolded algorithm. The advantage of this technique is that it decreases the time required to perform one iteration down to a minimum value equal to the iteration bound. This happens in two likely circumstances. Firstly when the computational time of the circuit's critical path, i.e., the computational time of all nodes in a path with zero delay edges is longer than T_∞. A particular example is when the computational time of a single unit exceeds the iteration bound. The second circumstance is originated from the fact that T_∞ is not always an integer value. In such cases, it is always possible to unfold the circuit a sufficient number of times to make the iteration time equal to the iteration bound. The application of the unfolding technique is described in Algorithm 2.1.

To illustrate the unfolding technique, let us consider again the IIR filter depicted in Figure 2.4. By applying 2-unfolding to the retimed DFG of the filter (see Figure 2.4(c)) one obtains the DFG shown in Figure 2.5(a) whose mathematical equations can be extracted from the block diagram of Figure 2.5(b):

$$\begin{array}{rcllll} y[2n] & = & a_0 \cdot x[2n] & + & a_1 \cdot x[2n+1-D] & + & b_1 \cdot y[2n+1-D] \\ y[2n+1] & = & a_0 \cdot x[2n+1] & + & a_1 \cdot x[2n] & + & b_1 \cdot y[2n] \end{array}$$

Since each delay D in the 2-unfolded block diagram will now represent two delays ($D = 2$) in the initial filter, it results in:

$$\begin{array}{rcllll} y[2n] & = & a_0 \cdot x[2n] & + & a_1 \cdot x[2n-1] & + & b_1 \cdot y[2n-1] \\ y[2n+1] & = & a_0 \cdot x[2n+1] & + & a_1 \cdot x[2n] & + & b_1 \cdot y[2n] \end{array} \tag{2.4}$$

Algorithm 2.1 Unfolding

To apply the J-unfolding to a DFG two steps must be performed:

1. For each node U in the original DFG, draw J nodes in the unfolded DFG, $\mathsf{U_0} \cdots \mathsf{U_{J-1}}$.
2. For each edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$ with weight $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$, draw J edges $\langle \mathsf{U_i} \rightarrow \mathsf{V_j} \rangle$ with weight $w^U_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$ delays such that

$$j = (i + w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}) \% J,$$

$$w^U_{\langle \mathsf{U_i} \rightarrow \mathsf{V_j} \rangle} = \left\lfloor \frac{i + w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}}{J} \right\rfloor,$$

% represents the remaining result from the division operation and $\lfloor x \rfloor$ the floor of x, i.e., the largest integer less than or equal to x and $0 \leq i < J$.

which corresponds to the original function.

Some important properties can shown from this example (proofs are presented in Appendix A). The first concerns the number of delays in the circuit. Comparing the initial block diagram represented in Figure 2.4(d) and the unfolded block diagram of Figure 2.5(b), we can notice that the total number of delays has not changed.

Property 2.3.1. *The sum of the delays of the J edges $\langle \mathsf{U_i} \rightarrow \mathsf{V_j} \rangle$ generated from J-unfolding a DFG is the same as the delay from the original edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$.*

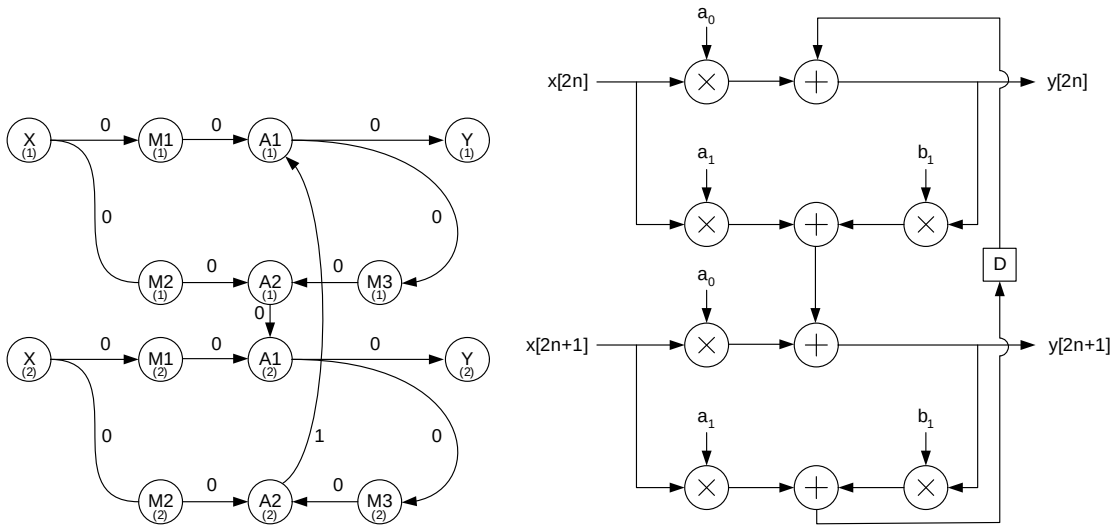
Lemma 1. *Unfoldind preserves the number of delays in a DFG.*

Property 2.3.2. *By J-unfolding a DFG, a cycle $\mathsf{P} = \langle \mathsf{A} \rightarrow \mathsf{A} \rangle$ with w_P delays is transformed into $\mathsf{gcd}\,(J, w_\mathsf{P})$ cycles $\mathsf{P_U} = \langle \mathsf{A_i} \rightarrow \mathsf{A_i} \rangle$, each including $\frac{J}{\mathsf{gcd}(J, w_\mathsf{P})}$ unfolded copies of node A and having a total of $\frac{w_\mathsf{P}}{\mathsf{gcd}(J, w_\mathsf{P})}$ delays where $gcd(a, b)$ represents the greatest common divisor between a and b.*

Property 2.3.3. *By J-unfolding a DFG, the Loop Bound of a cycle is increased by a factor of J:*

$$LB^U{}_\mathsf{P} = J \times LB_\mathsf{P} \tag{2.5}$$

Lemma 1. *By J-unfolding a DFG the iteration bound increases from T_∞ to $J \cdot T_\infty$.*



(a) Data-Flow Diagram. **(b)** Block diagram.

Figure 2.5: 2-unfolding of a first order IIR filter.

2.4 Folding

In typical DSP architectures, each iteration of the algorithm requires the computation of several operations of the same type. Therefore, the units required to perform such operations can usually be multiplexed in order for each one to execute multiple operations. This technique, usually known as *folding*, allows for the reduction of the circuit area by decreasing the number of computational units required to implement the DSP algorithms. Folding corresponds to the inverse of the unfolding operation.

Before performing the folding transformation, one must first group the set of nodes in the DFG by its computational type, so that the folding algorithm is supplied with operations that can be folded into a single functional unit. The algorithm to fold a DFG by a factor of J is described in Algorithm 2.2.

Theorem 1. *The scheduling of nodes* $\{\mathsf{U}_1, \cdots, \mathsf{U}_\mathsf{N}\}$ *at c-steps* $\{\mathsf{s}_{\mathsf{U}_1}, \cdots, \mathsf{s}_{\mathsf{U}_\mathsf{N}}\}$ *is valid if the following condition is verified:*

$$\mathsf{s}_{\mathsf{U}_\mathsf{i}} - \mathsf{s}_{\mathsf{U}_\mathsf{j}} \leq \mathsf{P}_{\mathsf{U}_\mathsf{i}} - N \cdot w_{\langle \mathsf{U}_\mathsf{i} \rightarrow \mathsf{U}_\mathsf{j} \rangle} \quad \forall i, j : 0 < i, j \leq N \tag{2.6}$$

To better understand the folding algorithm, let us apply it to the DFGs of Figures 2.4(b) and 2.4(c). Only one computational unit of each type is to be used: 1 multiplier and 1 adder. Also the nodes a_0, a_1 and b_1 have to be created in order to generate the constant variables for nodes $\mathsf{M1}$, $\mathsf{M2}$ and $\mathsf{M3}$, respectively.

Algorithm 2.2 Folding

The folding of a given DFG comprises the following steps:

1. define the number of computational units to be used for each node type;
2. schedule all nodes in the original DFG to a computational time between 0 and $N-1$ (see subsection 2.4.1), where N represents the folding order; by doing so, it is possible to define the folding factor, FF, as the quotient between the number of nodes in the DFG and the folding order N;
3. compute the number of cycles that each value must be stored, since it is generated by node U until it is consumed by a directly connected node V:

$$w^*_{\langle \mathsf{U}\rightarrow\mathsf{V}\rangle} = N \cdot w_{\langle \mathsf{U}\rightarrow\mathsf{V}\rangle} - \mathsf{P}_\mathsf{U} + \mathsf{s}_\mathsf{V} - \mathsf{s}_\mathsf{U}$$

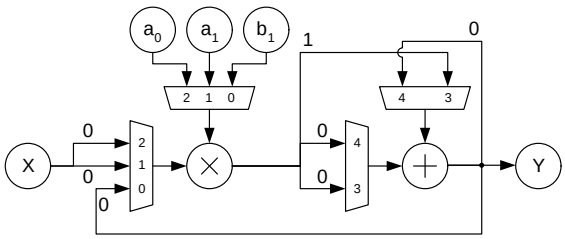
where P_x represents the number of pipeline stages of the unit executing x and s_x the schedule time of node x.

Table 2.1: Schedule times for folding of first order IIR filter

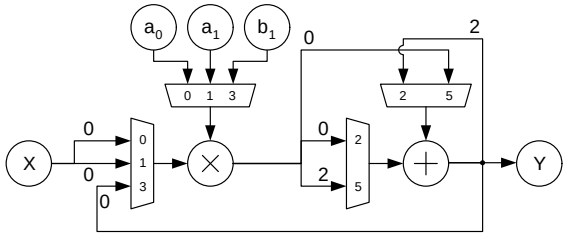
DFG	s_{A1}	s_{A2}	s_{M1}	s_{M2}	s_{M3}
Original	4	3	2	1	0
Retimed	2	5	0	1	3

Assuming that the execution times of the multiplier and adder are 2 and 1 c-steps respectively, and that both computational units use pipeline, the schedule times of Table 2.1 are assigned to the nodes using an optimal scheduling. The minimum time obtained to execute one iteration of the original and retimed algorithm (the folding order) is thereffore $N = 4$ and $n = 5$ c-steps, respectively. Following Algorithm 2.2, the storing cycles for each edge are:

$$\text{Original DFG}: \begin{cases} w^*_{\langle \mathsf{M1}\rightarrow\mathsf{A1}\rangle} = 5\times 0 - 2 + 4 - 2 = 0 \\ w^*_{\langle \mathsf{M2}\rightarrow\mathsf{A2}\rangle} = 5\times 0 - 2 + 3 - 1 = 0 \\ w^*_{\langle \mathsf{M3}\rightarrow\mathsf{A2}\rangle} = 5\times 0 - 2 + 3 - 0 = 1 \\ w^*_{\langle \mathsf{A2}\rightarrow\mathsf{A1}\rangle} = 5\times 0 - 1 + 4 - 3 = 0 \\ w^*_{\langle \mathsf{A1}\rightarrow\mathsf{M3}\rangle} = 5\times 1 - 1 + 0 - 4 = 0 \end{cases}$$



(a) Original data-flow graph



(b) Retimed data-flow graph

Figure 2.6: Folding of first order IIR filter.

$$\text{Retimed DFG}: \begin{cases} w^*_{\langle \mathsf{M1}\to\mathsf{A1}\rangle} = 6\times 0 - 2 + 2 - 0 = 0 \\ w^*_{\langle \mathsf{M2}\to\mathsf{A2}\rangle} = 6\times 0 - 2 + 5 - 1 = 2 \\ w^*_{\langle \mathsf{M3}\to\mathsf{A2}\rangle} = 6\times 0 - 2 + 5 - 3 = 0 \\ w^*_{\langle \mathsf{A2}\to\mathsf{A1}\rangle} = 6\times 1 - 1 + 2 - 5 = 2 \\ w^*_{\langle \mathsf{A1}\to\mathsf{M3}\rangle} = 6\times 0 - 1 + 3 - 2 = 0 \end{cases}$$

The resulting folded circuits for both the original and the retimed DFGs are shown in Figure 2.6. One important aspect of the resulting circuits concerns the number of delays. While the non-retimed DFG had two delays and leads to only one delay when folded, the retimed DFG which had only 1 delay leads to a total of 4 delays in the folded circuit. This outcomes from the fact that the critical path of the non-retimed circuit is $\mathtt{t}_{\mathsf{M3}} + \mathtt{t}_{\mathsf{A2}} + \mathtt{t}_{\mathsf{A1}} = \mathtt{t}_{\mathsf{M2}} + \mathtt{t}_{\mathsf{A2}} + \mathtt{t}_{\mathsf{A1}} = 4$ c-steps in opposition to the critical path of the retimed circuit, which is $\mathtt{t}_{\mathsf{M1}} + \mathtt{t}_{\mathsf{A1}} + \mathtt{t}_{\mathsf{M3}} + \mathtt{t}_{\mathsf{A2}} = 6$ c-steps. In the first case, it was impossible to schedule the architecture in 4 c-steps because of the structural restrictions (only 1 multiplier is provided), thus leading to an architecture scheduling of 5 c-steps. This leads to some important considerations to fold circuits using a minimum number of registers. To achieve a minimum number of registers in the folded circuit, it may be more important to retime it for reducing the critical path than for retiming for reducing the number of registers in the original circuit.

2.4.1 Scheduling of nodes to execution units

After deciding on the total number of execution units to use in the folded architecture, it is necessary to schedule each node to a c-step of the execution unit. Since this problem has been shown to be NP-complete [6], different algorithms with distinct complexity factors have been proposed to reach both the optimal and sub-optimal solutions. These algorithms can be divided into two classes: i) *time-constrained* algorithms, whose main goal is to minimise the number of units required to schedule the operations in a DFG using a fixed number of c-steps; and ii) *resource-constrained* algorithms, whose main objective is to compute the iterations of the given DFG with minimal c-steps, by using a pre-defined number of computational units.

The simplest scheduling techniques belong to the class of resource-constrained algorithms and are the As Soon As Possible (ASAP) and the As Late As Possible (ALAP). These algorithms work by iteratively scheduling the operations in the first or last possible c-step possible, respectively. While these two resource-constrained techniques are very simple, they also serve as initialisation steps for several other algorithms, such as List-Based Schedulers (LBSs) [14,15] or Force Directed-Schedulers (FDSs) [16].

List-Based Schedulers (LBSs) [14, 15] are resource-constrained algorithms that operate similarly to ASAP schedulers. The difference from the latest is that while LBSs use a priority function to set the order by which nodes are to be scheduled (with ASAP and ALAP nodes are scheduled in a random order). These functions can assume different forms, such as the mobility of nodes or the weight of the maximum computational path containing the node. A particular case of these algorithms are the Static List-Based Scheduler (SLBS) [14], where a static list of priorities is created before the beginning of the scheduling process.

Force Directed-Schedulers (FDSs) [16] are time-constrained algorithms whose strategy is to assign similar operations into different c-steps, balancing their distribution and reducing the total number of functional units required. It uses the distribution graphs representing the concurrency of all operations to each c-step to reduce the number of functional units allocated.

Genetic algorithms have also been proposed to solve time-restricted scheduling problems [9]. These algorithms are based on the simulation of natural evolution of genetics with the purpose of minimising the computational units used. In each iteration, these algorithms generate a population of solutions and, by using a given fitness criteria, evaluate the best solutions. From those, a new set of solutions is computed for the next iteration, but using this non-linear approach. Despite such algorithms tend to achieve good solutions, they commonly stop at local minima,

especially for large problems.

Optimal scheduling techniques have also been proposed, such as those based on *Integer Linear Programming* (ILP) [10, 12] techniques. To apply ILP, the DFG is transposed into a set of inequations capable of describing the scheduling problem. The procedure for performing time-constrained scheduling using an ILP solver is presented in Algorithm 2.3.

Algorithm 2.3 Translating the scheduling problem into a ILP problem

To describe the scheduling problem for a ILP solver, create a variable s_k^{U} for each node U in the DFG and each c-step $k \in [0; T-1]$, where T represents the time-constraint, i.e., the number of c-steps available to execute one iteration of the algorithm. Apply the following constraints to the scheduling variables:

1. **Scheduling constraints**: condition each variable s_k^{U} to the values 0 or 1, where the latest indicates that node U is scheduled to c-step k:

$$0 \leq s_k^{\mathsf{U}} \leq 1 \quad \forall \mathsf{U}, k: \ \mathsf{U}: \ k \in [0; T-1]$$

   Also, since each node can only be scheduled to one c-step, add the constraints to the ILP problem:

$$\sum_{k=0}^{T-1} s_k^{\mathsf{U}} = 1 \quad , \ \forall_{\mathsf{U}}$$

2. **Dependence constraints**: for all edges $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$ with weight $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} = 0$, schedule node V in a c-step equal or superior to the scheduling c-step of node U plus the execution time of node U, $\mathtt{t}_{\mathsf{U}}$:

$$\sum_{k=0}^{T-1} k \times s_k^{\mathsf{V}} - \sum_{k=0}^{T-1} k \times s_k^{\mathsf{U}} \geq \mathtt{t}_{\mathsf{U}} \quad , \forall_{\mathsf{U},\mathsf{V}} : w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} = 0$$

3. **Resource constraints**: if there is a resource-constraint limiting the number of functional units of a given type $\mathcal{U}$ to be less than or equal to $N_{\mathcal{U}}$, then define the following extra inequations:

$$\sum_{\mathsf{U} \in \mathcal{U}} s_k^{\mathsf{U}} \leq N_{\mathcal{U}} \quad , \forall_k : k \in [0; T-1]$$

While by using an ILP solver it is guaranteed that an optimal solution will be found, the dimension of the ILP problem rapidly increases: ignoring the resource constraints, a problem with N nodes and E edges with a time constraint of T c-steps, generates an ILP problem with $N \times T$ variables, $2 \cdot N \cdot T + N$ scheduling constraints and E dependence constraints. Even considering that some branch and bound tech-

niques can be applied [10,17], since generally an ILP is a NP-complete problem, the CPU execution times grow very fast. Therefore, optimal ILP solvers are infeasible for large problems.

In summary, one can conclude that as the optimality of the scheduling solutions improves the complexity of the algorithm is increased. While this may not be a crucial factor when the DFG is small, it becomes more and more important as it grows. In particularly, if the number of nodes rises to 10,000 and beyond, the use of very complex scheduling algorithms can be impractical, because they require too long execution times. It is thus necessary to investigate simpler algorithms that are able to produce sub-optimal solutions for scheduling large DFGs in a reasonable amount of time.

3

Algorithms for High-Level Synthesis

Contents

3.1 Critical Path Analysis 23
3.2 Retiming 27
3.3 Scheduling 32
3.4 Register Allocation 43
3.5 Conclusions 47

The ATAS main target is the synthesis of time-constrained, large-scale DSP problems, where a considerably large number of computations (over 10,000 per iteration) is required. One of the tasks required for HLS, the scheduling of processing nodes to functional units, has been shown to be NP-complete [6]. Thus, implementing optimal solutions is not an option since the time required for performing such task becomes non-practical to solve large problems. Therefore, the effort towards developing and implementing HLS techniques focused mainly on designing algorithms able to perform these tasks efficiently, having low complexity (able to scale to large DFGs) and having modest memory requirements.

This chapter describes the algorithms developed and programmed for performing the tasks required for HLS. Furthermore, the complexity of these algorithms is also analysed. The developed application works by performing the following tasks. Firstly, during *critical path analysis*, described in Section 3.1, the architecture is analysed towards determining for each node U the minimum number of c-steps required for: *i)* generating its own variables and *ii)* computing all variables dependent on the variable generated by node U. Then, the *retiming* task (see Section 3.2) attempts to balance all the critical paths, by changing the weights of the edges in the graph, but still maintaining the functionality of the algorithm. Following, as described in Section 3.3, the most complex task during HLS follows: *scheduling* assigns each node to be executed at one particular c-step. Scheduling is made such that nodes are distributed as evenly as possible amongst all c-steps, thus reducing the total number of functional units required to compute each iteration of the DSP algorithm. The *register allocation* is introduced in section 3.4. While the last two tasks are performed together in some HLS tools (like in [18]), it was chosen not to do it in the scope of the ATAS project, due to the corresponding increase in complexity of the problem that would lead to an increase of the computational time. In this sense three algorithms are presented for node scheduling: *i* the Adapted Static List Based Scheduler (ASLBS), *ii* the Time-Constrained List-Based Scheduler (TCLBS) and *iii* the FDS. The first two were developed in the scope of this thesis and the last is the literature typical scheduler for time-constrained problems. For register allocation it was used *lifetime-analysis* and *forward-backward* register allocation. These are known techniques which guarantee the allocation of the minimum number of registers for a given scheduling solution. To generate the final architecture, a *binding* task, which connects functional units, registers and multiplexers to form a data path, is still needed. However this step does not involve any special computational effort and represents only the translation of the output of the scheduling and register allocation tasks into a RTL description, such as VHDL. This last tasks is presented

in Chapter 4.

3.1 Critical Path Analysis

The first task on HLS is to analise the DFG and its critical path. This task allows: *i)* to detect any non-computable loops, i.e. closed paths with a loop bound equal to zero (see Section 2.1), *ii)* to determine all the critical paths of the circuit, required for the retiming process, and *iii)* to compute the scheduling bounds of each node, i.e., minimum and maximum c-steps where each node can be assigned for execution. Step *iii)* is equivalent to the scheduling of all nodes using ASAP and ALAP algorithms with infinite computational resources, respectively. The difference between the maximum and the minimum c-steps where nodes can be scheduled is also called *node mobility*.

To perform this task two values are computed for each node in the DFG:

- the number of *Execution Steps* (ES) represents the minimum number of c-steps required for computing all the variables of node U since the start of the iteration, i.e. the maximum computational time $t_{\mathsf{P_i}}$ of all paths $\mathsf{P_i}$ ending in node U with weight $w_{\mathsf{P_i}} = 0$:

$$\mathtt{ES}_{\mathsf{U}} = max\{t_{\mathsf{P_i}}\} \qquad \forall_{\mathsf{P_i}=\mathsf{V_i}\rightarrow\cdots\rightarrow\mathsf{U}} : w_{\mathsf{P_i}} = 0 \tag{3.1}$$

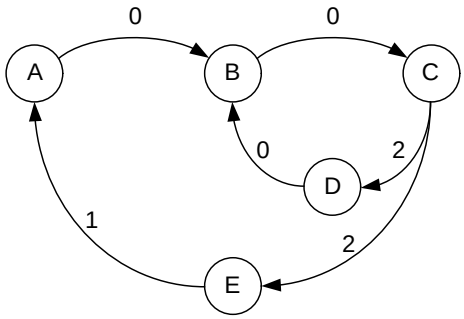
- the number of *Remaining Steps* (RS) represents the minimum number of c-steps required for computing all the variables dependent of the computation of node U until the end of the iteration, i.e. the maximum computational time of all paths $\mathsf{P_{ij}}$ starting in node U (excluding U) with weight $w_{\mathsf{P_{ij}}} = 0$:

$$\mathtt{RS}_{\mathsf{U}} = max\{t_{\mathsf{P_{ij}}}\} \qquad \forall_{\mathsf{P_{ij}}=\mathsf{V_i}\rightarrow\cdots\rightarrow\mathsf{V_j}} : w_{\langle\mathsf{U}\rightarrow\mathsf{V_i}\rangle} = 0 \wedge w_{\mathsf{P_{ij}}} = 0 \tag{3.2}$$

As an example, consider the computation of both ES and RS for the nodes B and E in the DFG presented in Figure 3.1. There are two paths $\mathsf{P_{B1}}$ and $\mathsf{P_{B2}}$ ending in node B with weight equal to 0: $\mathsf{P_{B1}} = \mathsf{A} \rightarrow \mathsf{B}$ and $\mathsf{P_{B2}} = \mathsf{D} \rightarrow \mathsf{B}$. Therefore, the number of execution steps of node B is $\mathtt{ES}_{\mathsf{B}} = max\{\mathtt{t}_{\mathsf{A}} + \mathtt{t}_{\mathsf{B}}; \mathtt{t}_{\mathsf{D}} + \mathtt{t}_{\mathsf{B}}\} = 5$ while the number of remaining steps is $\mathtt{RS}_{\mathsf{B}} = max\{\mathtt{t}_{\mathsf{C}}\} = 1$. On the other hand, no path P leads to or starts at node E having $w_{\mathsf{P}} = 0$. Thus $\mathtt{ES}_{\mathsf{E}} = \mathtt{t}_{\mathsf{E}} = 2$ and $\mathtt{RS}_{\mathsf{E}} = 0$.

Critical path analysis is performed in four phases as shown in Algorithm 3.1. It starts with an initialisation phase, where two lists of nodes are created: *i)* the `es_ready` list containing all nodes $\mathsf{U_A}$ such that the weight of all edges $\langle\mathsf{V_i} \rightarrow \mathsf{U_A}\rangle$ is equal or greater than 1:

$$\mathsf{U_A} \in \mathtt{es_ready} \text{ if } \forall_{\mathsf{V_i}}\ w_{\langle\mathsf{V_i}\rightarrow\mathsf{U_A}\rangle} \geq 1 \tag{3.3}$$



Node	Comp. time (c-steps)	ES (c-steps)	RS (c-steps)
A	2	2	2
B	1	5	1
C	1	6	0
D	4	4	2
E	2	2	0

Figure 3.1: Result of the Critical path analysis on a sample DFG.

and *ii)* the `es_wait` list containing all nodes $\mathsf{U_B}$ not in `es_ready` list:

$$\mathsf{U_B} \in \texttt{es_wait} \text{ if } \exists_{\mathsf{V_i}}\ w_{\langle \mathsf{V_i} \rightarrow \mathsf{U_B} \rangle} = 0. \tag{3.4}$$

Also, in this first stage, for each node U in the DFG a counter, $\texttt{pred}_\mathsf{U}$, is initialised with the number of preceding nodes V: $w_{\langle \mathsf{V} \rightarrow \mathsf{U} \rangle} = 0$. In the second phase, the number of steps for each node is computed. The procedure is made in a loop, where in each iteration a node U in the `es_ready` list is chosen and for every node V successor of node U, i.e. for every node with an edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$ with weight $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} = 0$, the value $\texttt{ES}_\mathsf{V}$ is updated and the $\texttt{pred}_\mathsf{V}$ counter is decremented. Once the counter reaches zero, the node is transfered from the `es_wait` list to the `es_ready` list. During this stage the preparation for computing $\texttt{RS}_\mathsf{U}$ is made. In this sense, while the algorithm is updating the values of $\texttt{ES}_\mathsf{V}$, it initialises the counter $\texttt{succ}_\mathsf{U}$ with the number of nodes V such that $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} = 0$. At the end of the cycle, if the counter $\texttt{succ}_\mathsf{U}$ is zero, then the node U is inserted into `rs_ready` list. If however the counter is not zero, then the node will be inserted into `rs_wait` list of nodes. In the third phase the algorithm checks for any infeasible loop l ($w_\mathsf{l} = 0$). If any node remains in the `es_wait` list, then it belongs to a closed path $\mathsf{P}_{\circlearrowright}$ composed of one or more nodes, with weight $w_{\mathsf{P}_{\circlearrowright}} = 0$. In this case, the algorithm has found an infeasible loop and therefore returns with an error. The fourth stage computes the RS. It works similarly with the computation of the ES: on each iteration of the cycle, the algorithm picks a node U from the `rs_ready` list and, for each predecessor node V of node U, it updates $\texttt{RS}_\mathsf{V}$ and decreases $\texttt{succ}_\mathsf{V}$. Once $\texttt{succ}_\mathsf{V}$ reaches zero, the node

Algorithm 3.1 Critical path analysis

```
1:  /* PHASE 1. Initializations: */
2:  Create four empty lists of nodes: es_ready, es_wait, rs_ready and rs_wait
3:  for all nodes U do
4:     if ∄_⟨V_i→U⟩ : w_⟨V_i→U⟩ = 0 then
5:        Insert U into node list es_ready
6:     else
7:        Insert U into node list es_wait
8:     end if
9:     Set Pred_U as the number of nodes V_i such that w_⟨V_i→U⟩ = 0
10:    Set ES_U = t_U and RS_U = 0
11: end for
12:
13: /* PHASE 2. Computing Execution Steps (ES) */
14: for all nodes U in es_ready list do
15:    for all edges ⟨U → V⟩: w_⟨U→V⟩ = 0 do
16:       ES_V = max{ES_V; ES_U + P_V}
17:       Pred_V = Pred_V − 1
18:       if Pred_V = 0 then
19:          Move V from es_wait list to es_ready list
20:       end if
21:    end for
22:    /* Preparation for computing Remaining Steps (RS) */
23:    if ∄V : w_⟨U→V⟩ = 0 then
24:       Insert U into rs_ready list
25:    else
26:       Insert U into rs_ready list
27:    end if
28:    Set Succ_U as the number of nodes V_i such that w_⟨U→V_i⟩ = 0
29: end for
30:
31: /* PHASE 3. Architecture feasibility test */
32: if wait list is not empty then
33:    ERROR: infeasible loop found
34: end if
35:
36: /* PHASE 4. Computing Remaining Steps (RS) */
37: for all nodes U in rs_ready list do
38:    for all edges ⟨V → U⟩: w_⟨V→U⟩ = 0 do
39:       RS_V = max{RS_V; RS_U + P_U}
40:       Succ_V = Succ_V − 1
41:       if Succ_V = 0 then
42:          Move V from rs_wait list to rs_ready list
43:       end if
44:    end for
45: end for
```

is transfered from the `rs_wait` list to the `rs_ready` list.

During the computation of node ES and RS values, nodes are inserted into two lists sorted accordingly to ES and RS. This allows to speedup the computation of the retiming, scheduling and register allocation tasks. Node insertion is achieved by using an insertion sort algorithm where, in the case of tie, nodes are placed after the last node with the same value. This also guarantees that in case of tie of the ES or RS value between two nodes U and V connected by a zero weight edge ($w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} = 0$), this dependency will prevail in the sorted lists.

Based on the two computed values, it is possible to determine the minimum and maximum c-steps S^{U}_{min} and S^{U}_{max}, respectively, for each node to be scheduled while still allowing to compute the algorithm being synthesised within the specified maximum latency T_{iter}. These bounds can be computed as:

$$S^{\mathsf{U}}_{min} = \mathtt{ES}_{\mathsf{U}} - \mathtt{P}_{\mathsf{U}} \tag{3.5}$$

$$S^{\mathsf{U}}_{max} = \mathtt{RS}_{\mathsf{U}} \tag{3.6}$$

Complexity of Algorithm 3.1

The complexity of Algorithm 3.1 is analysed bellow for all the phases. In phase 1, to compute the initial values of each counter $\mathtt{succ}_{\mathsf{U}}$ and $\mathtt{pred}_{\mathsf{U}}$, it is necessary to sweep all input edges of all nodes. Since the ATAS tool uses sorted lists, the time to compute phase 1 of the algorithm is:

$$\sum_{i=1}^{N} k \times E_i$$

where N is the number of nodes in the DFG, E_i is the number of inputs of node N_i and k is a constant value. This expression can also be written as

$$N \times \frac{\sum_{i=1}^{N} k \times E_i}{N} = k \times N \times \langle E \rangle$$

where $\langle E \rangle = avg\{E_i\}$ is the average value of inputs for all nodes. Thus, the worst-case complexity of phase 1 is $O(N)$. The complexity of phase 2 is defined by the insertion sort algorithm, which has a worst-case complexity $O(N)$. Therefore, the complexity of this phase is $O(N^2)$. It is important to note that the cost of removing a node from a list is constant. Phase 3 has constant complexity and phase 4, similarly with phase 2, can be computed in $O(N^2)$. Thus the worst-case complexity of Algorithm 3.1 is:

$$O(N^2) \tag{3.7}$$

However, due to the structure of the critical path analysis algorithm, nodes are usually inserted at the tail of the ES or RS lists. The insertion sort technique used for implementing this algorithm takes advantage of this fact. It searches the position where the node must be inserted starting from the list tail. Therefore, the complexity of the insertion sort algorithm is typically constant. The best-case complexity of Algorithm 3.1 can therefore be described as:

$$O(N) \tag{3.8}$$

3.2 Retiming

As introduced in Section 2.2, through retiming it is possible to re-arrange the edge weights in a graph. The retiming operation can be applied to achieve one of the major objectives: decreasing the critical path of a circuit, reducing the number of registers required or decreasing the number of buses in the generated circuit. In this section, a retiming algorithm for reducing the critical path of a circuit is presented.

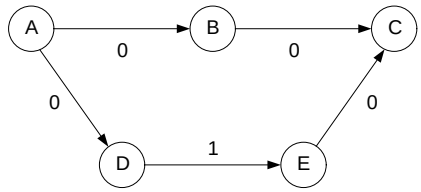
Retiming a set of nodes with a unitary retiming value causes the weights of the edges entering this set of nodes to be increased by one and the weights of the edges originated in the set to be decreased by one. Some restrictions can be found for the retiming values in (2.2): for example, all edge weights must be non-negative (a negative value of an edge would result in a non-causal circuit). This restriction can be written in the form of an inequation: for every edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$ connecting nodes U and V with weight $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$, the following feasibility constraint is defined:

$$\mathtt{r}_{\mathsf{U}} - \mathtt{r}_{\mathsf{V}} \leq w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} \tag{3.9}$$

Since the goal of the presented algorithm is to reduce the critical path of the circuit, more constraints must be considered. Thus, if the desired critical path is T_{iter}, for all paths $\mathsf{P} = \mathsf{U} \rightarrow \cdots \rightarrow \mathsf{V}$ with weight w_{P} (sum of the weight of all edges in P) and computational time t_{P} (sum of computational weights of all nodes in P), the following critical path constraint is defined:

$$\mathtt{r}_{\mathsf{U}} - \mathtt{r}_{\mathsf{V}} \leq w_{\mathsf{P}} - 1 \quad \text{if } t_{\mathsf{P}} > T_{iter} \tag{3.10}$$

In physical terms, this constraint specifies that all paths with a computational time longer than T_{iter} must have at least one register dividing it to fulfil the iteration time T_{iter}. Note that if $t_{\mathsf{P}} > n \times T_{iter}$ with $n \geq 2$ and it is possible to retime the circuit in order to fulfil the time constraint, then a set of n or more constraints will be defined between the nodes in the path P.



Computational time (c-steps)
$\mathtt{t}_\mathsf{A} = 3$ $\mathtt{t}_\mathsf{B} = 3$ $\mathtt{t}_\mathsf{C} = 3$ $\mathtt{t}_\mathsf{D} = 3$ $\mathtt{t}_\mathsf{E} = 3$

Figure 3.2: Example DFG.

Generating all the critical path constraints requires one to know the weight and computational time of all paths in the DFG, w_P and t_P respectively. However, some simplifications can be made. For instance, consider the example DFG of Figure 3.2. There are two paths starting on node A and ending on node C: $\mathsf{P}_1 = \mathsf{A} \rightarrow \mathsf{B} \rightarrow \mathsf{C}$ and $\mathsf{P}_2 = \mathsf{A} \rightarrow \mathsf{D} \rightarrow \mathsf{E} \rightarrow \mathsf{C}$, where $w_{\mathsf{P}_1} = 0$, $w_{\mathsf{P}_2} = 1$, $t_{\mathsf{P}_1} = 9$ and $t_{\mathsf{P}_2} = 12$. For a DFG synthesis with iteration time lower than 9 c-steps, two critical path constraints should be generated involving the two nodes A and C:

$$\begin{array}{ll} \mathsf{P}_1\text{:} & \mathtt{r}_\mathsf{A} - \mathtt{r}_\mathsf{C} \leq -1 \\ \mathsf{P}_2\text{:} & \mathtt{r}_\mathsf{A} - \mathtt{r}_\mathsf{C} \leq 0 \end{array}$$

However, since the constraint originated from path P_1 is more restrictive than the one originated from path P_2, the second one can be dropped. This kind of analysis leads to an initial constraint reduction as suggested by [19], and prunes the number of constraints for the retiming algorithm to

$$\mathtt{r}_\mathsf{U} - \mathtt{r}_\mathsf{V} \leq W(\mathsf{U},\mathsf{V}) - 1 \quad \text{if } D(\mathsf{U},\mathsf{V}) > T_{iter} \tag{3.11}$$

where $W(\mathsf{U},\mathsf{V})$ is the minimum weight of all paths P_i starting in node U and ending in node V, and $D(\mathsf{U},\mathsf{V})$ is the maximum computational time of all paths $\mathsf{P}_\mathsf{j} \in \{\mathsf{P}_\mathsf{i}\}$ such that $w_{\mathsf{P}_\mathsf{j}} = W(\mathsf{U},\mathsf{V})$.

To compute the minimum weight of all paths P_i, a shortest-path algorithm can be used. The fastest known algorithm is the *Floyd-Warshall* algorithm [20]. It computes the all-pairs shortest path problem in $O(N^3)$ time. However this algorithm also requires $O(N^2)$ memory to solve this problem, thus making it infeasible for large problems. To reduce the memory requirements, a single point shortest path algorithm was developed based on a mix between the *Dijkstra* [20] and the *Bellman-Ford* [20] algorithms. While it requires N iterations for the complete process, it computes each iteration in $O(N\langle E\rangle)$ requiring only $O(N)$ memory. Note that the algorithm complexity is lower than both *Dijkstra* and the *Bellman-Ford* algorithms. This happens because the developed algorithm assumes that: *i)* no negative-weight

edges exist in the graph; and *ii)* the weight of all cycles in the graph is greater than or equal to one. This algorithm presents three main differences regarding the original *Bellman-Ford* algorithm, allowing it to take advantage of the above two constraints. The first difference between the two algorithms concerns the $W(\mathsf{U},\mathsf{V_i})$ actualisation method: in every iteration k, the *Bellman-Ford algorithm* updates the shortest path for each node $\mathsf{V_i}$ by analysing if any input edge $\langle \mathsf{V_j} \rightarrow \mathsf{V_i} \rangle$ corresponds to the shortest path to node $\mathsf{V_j}$:

$$W^{k-1}(\mathsf{U},\mathsf{V_j}) + w_{\langle \mathsf{V_j} \rightarrow \mathsf{V_i} \rangle} < W^{k}(\mathsf{U},\mathsf{V_i})$$

The developed algorithm does the inverse operation, i.e. for every iteration k it takes all nodes $\mathsf{V_i}$ and if a shortest path to node $\mathsf{V_j}$ is found through an output edge $\langle \mathsf{V_i} \rightarrow \mathsf{V_j} \rangle$

$$W^{k-1}(\mathsf{U},\mathsf{V_i}) + w_{\langle \mathsf{V_i} \rightarrow \mathsf{V_j} \rangle} < W^{k}(\mathsf{U},\mathsf{V_j})$$

then it updates $W(\mathsf{U},\mathsf{V_j})$. The second difference is that the implemented algorithm computes $W(\mathsf{U},\mathsf{V_i})$ and $D(\mathsf{U},\mathsf{V_i})$ for all nodes $\mathsf{V_i}$ at the same time. The last difference in the algorithms corresponds to the ability to perform additional constraint pruning. Let us consider again the example DFG of Figure 3.2. By using the shortest path algorithm as presented by [19], the complete set of critical path constraints for a maximum iteration time $T_{iter} = 3$ c-steps is:

$$\begin{array}{ll}
a) & \mathtt{r_A} - \mathtt{r_B} \leq -1 \\
b) & \mathtt{r_A} - \mathtt{r_C} \leq -1 \\
c) & \mathtt{r_A} - \mathtt{r_D} \leq -1 \\
d) & \mathtt{r_A} - \mathtt{r_E} \leq \ \ 0 \\
e) & \mathtt{r_A} - \mathtt{r_C} \leq \ \ 0 \\
f) & \mathtt{r_B} - \mathtt{r_C} \leq -1 \\
g) & \mathtt{r_D} - \mathtt{r_E} \leq \ \ 0 \\
h) & \mathtt{r_D} - \mathtt{r_C} \leq \ \ 0 \\
i) & \mathtt{r_E} - \mathtt{r_C} \leq -1
\end{array} \tag{3.12}$$

However, for constraints *a)* and *f)* to be verified, constraint *b)* is automatically verified. In fact, constraints *a)* and *f)* are more restrictive than constraint *b)*. Thus it is possible to exclude constraint *b)* without changing the solution set to the retiming problem. Following the same analysis, it is also possible to prune constraints *d)*, *e)* and *h)* due to constraint sets $\{c,g\}$, $\{c,g,i\}$ and $\{g,i\}$, respectively. Generalising, for all paths $\mathsf{P_{jk}} = \mathsf{V_j} \rightarrow \cdots \rightarrow \mathsf{V_i} \rightarrow \cdots \rightarrow \mathsf{V_k}$, if $D(\mathsf{V_j},\mathsf{V_i}) > T_{iter}$ then only the constraint

$$\mathtt{r}_{\mathsf{V_j}} - \mathtt{r}_{\mathsf{V_i}} \leq W(\mathsf{V_j},\mathsf{V_i}) - 1 \tag{3.13}$$

needs to be considered and all the other constraints

$$\mathtt{r}_{\mathsf{V_j}} - \mathtt{r}_{\mathsf{V_n}} \leq W(\mathsf{V_j},\mathsf{V_n}) - 1 \quad , \ \forall_{n \in]i;k]}$$

can be dropped. Consequently, once the first constraint between nodes $\mathsf{V_j}$ and $\mathsf{V_i}$ is found for a path P_{jk}, the algorithm does not need to compute the shortest path for any node $\mathsf{V_l}$ in that path which is located beyond the restricting node $\mathsf{V_i}$. This constraint pruning, originally presented by [21], greatly improves the performance of the retiming algorithm. For instance, for a 256-point complex Fast Fourier Transform (FFT) [22][1] implemented using the decimation in time radix-2 method, it is possible to reduce the critical path constraint set from over 983 millions to around 62 thousand. It represents a reduction in the number of constraints in almost 16,000 times. It also significatively decreases the computation time of the retiming algorithm in 206 times (from almost 6 hours to about 1.5 minutes[2]) and reduces the memory requirements in around 16,000 times, in a way directly proportional with the number of constraints.

The implemented shortest path algorithm is described in Algorithm 3.2. To store all the generated constraints, the retiming algorithm has to allocate memory dynamically. In order to avoid system calls, this allocation is made in chunks, where each chunk can store up to N constraints. This is an important step to reduce the computation time, since the number of constraints depends on the DFG and may be very large (see Section 4.3).

After computing the set of constraints for the retiming algorithm, the problem can be described as a ILP. However, since it does not have a minimisation function one can solve it by computing the single-point shortest-path on a graph G', which represents the set of constraints [20]. This new graph G' can be obtained by following the next three steps:

1. draw all nodes of the DFG in graph G';

2. for all constraints

$$\mathtt{r}_{\mathsf{U}} - \mathtt{r}_{\mathsf{V}} \leq x$$

   draw an edge from node V to node U with weight x;

3. draw an artificial node A and an edge from this node to all other nodes in the graph with zero weight.

A solution for the ILP exists if there is no negative cycle in the graph, and it can be solved by computing the shortest path $W(\mathsf{A}, \mathsf{U})$ from the artificial node A to all

[1]The DFG of the 256-point complex FFT has 4096 multiplications and 6144 additions/subtractions, totalling 10,240 nodes.

[2]Benchmark made with a AMD AthlonXP 2000+ with 512MB of main memory running the SuSE 9.3 operation system.

Algorithm 3.2 Compute shortest path and maximum delay from node U to all nodes V

```
1:  /* Initializations: */
2:  for all nodes V do
3:     Set W(U,V) = ∞
4:     Set D(U,V) = −1
5:  end for
6:  Set W(U,U) = 0
7:  Set D(U,U) = t_U
8:  Set list= {U}
9:  Set nxt_list= ∅
10:
11: /* Perform computation */
12: while list≠ ∅ do
13:    /* Process all nodes in list */
14:    for all nodes V ∈ list do
15:       for all edges ⟨V → W⟩ do
16:          /* Check if an update on the path weight W is required */
17:          if W(U,W) > W(U,V) + w_⟨V→W⟩ then
18:             W(U,W) = W(U,V) + w_⟨V→W⟩
19:             D(U,W) = D(U,V) + t_W
20:             /* Prune constraints: only insert node W into nxt_list if a constraint
                has not yet been found. */
21:             if D(U,W) < T_iter then
22:                Insert W into nxt_list
23:             end if
24:             /* Check if an update on the path computational weight D is required */
25:          else if (W(U,W) = W(U,V) + w_⟨V→W⟩) and (D(U,W) < D(U,V) + t_W)
             then
26:             D(U,W) = D(U,V) + t_W
27:             /* Prune constraints: only insert node W into nxt_list if a constraint
                has not yet been found. */
28:             if D(U,W) < T_iter then
29:                Insert W into nxt_list
30:             end if
31:          end if
32:       end for
33:    end for
34:    /* Update list */
35:    list ← nxt_list
36:    nxt_list= ∅
37: end while
```

nodes U [20]:

$$\mathtt{r}_{\mathsf{U}} = W(\mathsf{A},\mathsf{U}) \quad , \forall_U \tag{3.14}$$

To compute the shortest path $W(\mathsf{A}, \mathsf{U})$, $\forall_U$, the *Bellman-Ford* algorithm [20] can be used, thus reaching a solution in $O(N^3)$.

3.3 Scheduling

The scheduling techniques designed for the ATAS application were developed for the synthesis of complex time-constrained DSP problems, where a considerably large number of computations (typically over 10,000 per iteration) are required. When scheduling such a large DFGs, the computational times are high for algorithms returning close to optimal solutions. In those cases, it is necessary to use simple algorithms and to trade optimality for speed. The goal was therefore to develop a method having low complexity and being able to schedule large DFGs in a reasonable amount of time. With this purpose, two types of algorithms were developed and evaluated in terms of the scheduling results and the computational time: *i)* simple algorithms based in scheduling schemas such as ASAP or ALAP and *ii)* more complex algorithms based on computing the approximated probability of scheduling each node U to each c-step x, $P(S^{\mathsf{U}} = x)$. In this sense, several scheduling methods were developed and implemented, namely a TCLBS was developed and the FDS [13] was implemented for comparison purpose. Also, a resource-constrained List-Based Scheduler, designated in this thesis by ASLBS, was adapted to solve time constrained problems.

In an early phase, all the three scheduling algorithms compute the scheduling bounds, i.e. the minimum and maximum c-step, S_{min} and S_{max}, respectively, in which each node can be scheduled while allowing for the time constraint to be met, as described in Section 3.1.

In an attempt to assign the nodes to different c-steps, the first two algorithms, TCLBS (depending on the implementation) and FDS, are based on computing the probabilities of these assignments. With this approach, the decision on which node should be assigned to which c-step is made by evaluating the probability for that assignment to interfere with the scheduling of another node. However, this probability is hard to compute: dependencies between nodes cause the probability to schedule a node in the DFG to depend on several other nodes, or even on all nodes of the DFG. In order to simplify the computation, instead of using the real probability, an approximation $P^{(n)}(S^{\mathsf{U}} = x)$ is computed by considering only n levels of node dependencies. For $n = 0$, this approximation is equal to the ratio between the number of functional units available to compute U at c-step x and the total number

of functional units available to compute U in c-step interval [S^{U}_{min} ; S^{U}_{max}]:

$$P^{(0)}(S^{\mathsf{U}} = x) = \begin{cases} \frac{free(x)}{\sum_{s=S^{\mathsf{U}}_{min}}^{S^{\mathsf{U}}_{max}} free(s)} & , S^{\mathsf{U}}_{min} \leq x \leq S^{\mathsf{U}}_{max} \\ 0 & , otherwise \end{cases} \tag{3.15}$$

where $free(x)$ represents the number of functional units available for processing a node at c-step x. Note that, if an infinite number of units are considered, then the expression for the probability can be simplified to

$$\hat{P}^{(0)}(S^{\mathsf{U}} = x) = \begin{cases} \frac{1}{S^{\mathsf{U}}_{max} - S^{\mathsf{U}}_{min} + 1} & , S^{\mathsf{U}}_{min} \leq x \leq S^{\mathsf{U}}_{max} \\ 0 & , otherwise \end{cases} \tag{3.16}$$

For 1 level of dependencies ($n = 1$), the approximated probability will therefore be a function of the number of units available to compute all nodes U and $\mathsf{V} \in \mathcal{D}_{\mathsf{U}}$, where $\mathcal{D}_{\mathsf{U}}$ represents all nodes with an edge to/from U with weight equal to zero:

$$\mathcal{D}_{\mathsf{U}} = \left\{ \mathcal{P}red_{\mathsf{U}} \bigcup \mathcal{S}ucc_{\mathsf{U}} : \mathsf{V} \in \mathcal{S}ucc_{\mathsf{U}} \Leftrightarrow w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} = 0 \ , \ \mathsf{V} \in \mathcal{P}red_{\mathsf{U}} \Leftrightarrow w_{\langle \mathsf{V} \rightarrow \mathsf{U} \rangle} = 0 \right\} \tag{3.17}$$

An approximated value for the probability is then computed as the ratio betwen the number of valid assignments of all nodes $\mathsf{V} \in \mathcal{D}_{\mathsf{U}}$, such that node U is scheduled to c-step x, and the total number of valid assignments of all nodes U and V:

$$P^{(1)}(S^{\mathsf{U}} = x) = \frac{N^{(1)}(S^{\mathsf{U}} = x)}{\sum\limits_{s=S^{\mathsf{U}}_{min}}^{S^{\mathsf{U}}_{max}} N^{(1)}(S^{\mathsf{U}} = s)} \tag{3.18}$$

where:

$$N^{(1)}(S^{\mathsf{U}} = x) = free(x) \prod_{\mathsf{V} \in \mathcal{D}_{\mathsf{U}}} N^{(0)}(S^{\mathsf{V}} | S^{\mathsf{U}} = x) \tag{3.19}$$

and $N^{(0)}(S^{\mathsf{V}} = s | S^{\mathsf{U}} = x)$ represents the number of possible assignments of node V conditioned by the scheduling of node U to c-step x

$$N^{(0)}(S^{\mathsf{V}} | S^{\mathsf{U}} = x) = \sum_{s=S'^{\mathsf{V}}_{min}}^{S'^{\mathsf{V}}_{max}} N^{(0)}(S^{\mathsf{V}} = s) \tag{3.20}$$

with

$$\begin{cases} S'^{\mathsf{V}}_{min} = S^{\mathsf{V}}_{min} \wedge S'^{\mathsf{V}}_{max} = min \left\{ \begin{array}{c} S^{\mathsf{U}} - \mathsf{P}_{\mathsf{V}} \\ S^{\mathsf{V}}_{max} \end{array} \right\} & , if \ \mathsf{V} \in \mathcal{P}red_{\mathsf{U}} \\ S'^{\mathsf{V}}_{min} = max \left\{ \begin{array}{c} S^{\mathsf{U}} + \mathsf{P}_{\mathsf{U}} \\ S^{\mathsf{V}}_{min} \end{array} \right\} \wedge S'^{\mathsf{V}}_{max} = S^{\mathsf{V}}_{max} & , if \ \mathsf{V} \in \mathcal{S}ucc_{\mathsf{U}} \end{cases} \tag{3.21}$$

Generalising the above equations, the probability of scheduling a node U based on n levels of dependencies is:

$$P^{(n)}(S^{\mathsf{U}} = x) = \frac{N^{(n)}(S^{\mathsf{U}} = x)}{\sum_{s=S^{\mathsf{U}}_{min}}^{S^{\mathsf{U}}_{max}} N^{(n)}(S^{\mathsf{U}} = s)} \tag{3.22}$$

where,

$$N^{(n)}(S^{\mathsf{U}} = x) = free(x) \prod_{\mathsf{V} \in \mathcal{D}_{\mathsf{U}}} N^{(n-1)}(S^{\mathsf{V}} | S^{\mathsf{U}} = x) \tag{3.23}$$

and

$$N^{(n-1)}(S^{\mathsf{V}} | S^{\mathsf{U}} = x) = \sum_{s=S'^{\mathsf{V}}_{min}}^{S'^{\mathsf{V}}_{max}} N^{(n-1)}(S^{\mathsf{V}} = s) \tag{3.24}$$

It is important to note that as n tends to infinity, so does the approximated probability trends to the real probability. However, this formula still does not take into account that several nodes may be scheduled to a single type of functional units therefore increasing the level of dependencies for the computation of the probability. Nevertheless, experimental results presented in Chapter 5 show that this approximation is good enough for scheduling nodes efficiently.

3.3.1 Force-Directed Schedulers

Force-directed scheduling [13,16] is a time-constrained algorithm whose strategy consists in assigning similar operations into different c-steps, balancing its distribution and reducing the total number of functional units required. This algorithm operates in five phases, as it is presented in Algorithm 3.3.

In the first phase, the scheduling bounds are computed as described in Section 3.1. In the second phase, the Distribution Graph, that estimates the required number of functional units to perform the algorithm being synthesised, is computed for each functional type. The Distribution Graph $\mathtt{DG}_{\mathcal{U}}(x)$ for the operations of type $\mathcal{U}$ is computed by adding the approximated probabilities $\hat{P}^{(0)}(S^{\mathsf{U}} = x)$ amongst all nodes $\mathsf{U} \in \mathcal{U}$ for each c-step $x \in [0\ ;\ T_{iter} - 1]$:

$$\mathtt{DG}_{\mathcal{U}}(x) = \sum_{\mathsf{U} \in \mathcal{U}} P^{(0)}(S^{\mathsf{U}} = x) \tag{3.25}$$

In the third phase, the FDS computes the force for all nodes to be scheduled to all c-steps. This force is the sum of two components, a Self-Force (SF) and a Dependence-Force (DF),

$$\mathtt{F}_{\mathsf{U}} = \mathtt{SF}_{\mathsf{U}} + \mathtt{DF}_{\mathsf{U}} \tag{3.26}$$

where the SF of node U to c-step i is computed as

$$\mathtt{SF}_{\mathsf{U}}(i) = \mathtt{DG}_{\mathcal{U}}(i) - \sum_{j=S^{\mathsf{U}}_{min}}^{S^{\mathsf{U}}_{max}} \frac{\mathtt{DG}_{\mathcal{U}}(j)}{S^{\mathsf{U}}_{max} - S^{\mathsf{U}}_{min} + 1} \tag{3.27}$$

The dependence force represents the sum of the forces applied when the scheduling of a node U limits the scheduling of its dependent nodes, $\mathcal{D}_{\mathsf{U}} = \{\mathcal{P}red_{\mathsf{U}} \bigcup \mathcal{S}ucc_{\mathsf{U}}\}$, thus reducing the mobility of a preceding and/or succeeding node. Therefore, if the scheduling of a node U limits any nodes $\mathsf{V} \in \mathtt{D}_{\mathsf{U}}$ to be scheduled into the c-step range $[S'^{\mathsf{V}}_{min}\ ;\ S'^{\mathsf{V}}_{max}]$, as expressed in (3.21), the dependence force is computed as the sum of self-forces in the limited interval:

$$\mathtt{DF}_{\mathsf{U}}(i) = \sum_{\mathsf{V} \in \mathcal{D}_{\mathsf{U}}} \sum_{j=S'^{\mathsf{V}}_{min}}^{S'^{\mathsf{V}}_{max}} \mathtt{SF}_{\mathsf{V}}(j) \tag{3.28}$$

Summarising, the force of node U to c-step i is computed as

$$\mathtt{F}_{\mathsf{U}}(i) = \mathtt{DG}_{\mathcal{U}}(i) - \sum_{j=S^{\mathsf{U}}_{min}}^{S^{\mathsf{U}}_{max}} \frac{\mathtt{DG}_{\mathcal{U}}(j)}{S^{\mathsf{U}}_{max} - S^{\mathsf{U}}_{min} + 1} + \sum_{\mathsf{V} \in \mathcal{D}_{\mathsf{U}}} \sum_{j=S'^{\mathsf{V}}_{min}}^{S'^{\mathsf{V}}_{max}} \mathtt{SF}_{\mathsf{V}}(j) \tag{3.29}$$

In the fourth phase of the algorithm, the mobility of node X is reduced by centring it on c-step x such that the force $\mathtt{F}_{\mathsf{X}}(x)$ is the lowest amongst all (node,c-step) pairs:

$$\mathtt{F}_{\mathsf{X}}(x) \leq \mathtt{F}_{\mathsf{Y}}(j) \quad , \forall_{\mathsf{Y},j} :\ \mathsf{Y} \in DFG \ \wedge\ j \in [0\ ;\ T_{iter} - 1] \tag{3.30}$$

This mobility reduction can be achieved by decreasing the interval $[S^{\mathsf{X}}_{min}\ ;\ S^{\mathsf{X}}_{max}]$ by a constant value k:

$$[S^{\mathsf{X}}_{min}\ ;\ S^{\mathsf{X}}_{max}] = [x - \Delta x + k\ ;\ x + \Delta x - k]$$

where $\Delta x = min\left\{|x - S^{\mathsf{U}}_{min}|; |x - S^{\mathsf{U}}_{max}|\right\}$, or a percentage α of the interval

$$[S^{\mathsf{X}}_{min}\ ;\ S^{\mathsf{X}}_{max}] = [x - \lfloor 2 \cdot \alpha \cdot \Delta x \rfloor\ ;\ x + \lfloor 2 \cdot \alpha \cdot \Delta x \rfloor]$$

where $\alpha \in [0; 1[$ and $\lfloor z \rfloor$ represents the floor of z. Once the two limits become equal, $S^{\mathsf{X}}_{min} = S^{\mathsf{X}}_{max}$, the node is considered to be scheduled. Note that this algorithm does not count the number of units already allocated. Instead it successively updates the distribution graph with a more precise approximated value of the number of functional units required. Consequently, once the node is scheduled, the new probability $\hat{P}^{(0)}(S^{\mathsf{U}} = x)$ becomes

$$\hat{P}^{(0)}(S^{\mathsf{U}} = s) = \begin{cases} 1 & \text{, if } s = x \\ 0 & \text{, otherwise} \end{cases} \tag{3.31}$$

Algorithm 3.3 Force-Directed Scheduling

1: */* Phase 1 - Compute scheduling bounds */*
2: Compute node scheduling bounds S_{min} and S_{max}
3: */* Phase 2 - Compute Distribution Graphs */*
4: **for all** Node types $\mathcal{U}$ **do**
5: **for all** c-steps x in interval $[0\ ;\ T_{iter}-1]$ **do**
6: Compute Distribution Graph, $\mathtt{DG}_{\mathcal{U}}(x) = \sum_{\mathsf{U}\in\mathcal{U}} P^{(0)}(S^{\mathsf{U}} = x)$
7: **end for**
8: **end for**
9: **repeat**
10: */* Phase 3 - Compute node forces */*
11: **for all** Nodes U of type $\mathcal{U}$ **do**
12: **for all** c-steps x in interval $[S^{\mathsf{U}}_{min}\ ;\ S^{\mathsf{U}}_{max}]$ **do**
13: Compute node Force,
$\mathtt{F}_{\mathsf{U}}(x) = \mathtt{DG}_{\mathcal{U}}(x) - \sum_{j=S^{\mathsf{U}}_{min}}^{S^{\mathsf{U}}_{max}} \frac{\mathtt{DG}_{\mathcal{U}}(j)}{S^{\mathsf{U}}_{max}-S^{\mathsf{U}}_{min}+1} + \sum_{\mathsf{V}\in\mathcal{D}_{\mathsf{U}}} \sum_{j=S'^{\mathsf{V}}_{min}}^{S'^{\mathsf{V}}_{max}} \mathtt{SF}_{\mathsf{V}}(j)$
14: **end for**
15: **end for**
16: */* Phase 4 - Reduce node mobility */*
17: Choose the (node,c-step) pair (U,x) with the lowest force $\mathtt{F}_{\mathsf{U}}(x)$
18: Reduce the mobility on U and centre it around c-step x
19: Update the mobility on all nodes V contained in a path P such that $w_{\mathsf{P}} = 0$ and $\mathsf{U}, \mathsf{V} \in \mathsf{P}$
20: */* Phase 5 - Update Distribution Graphs */*
21: Update DG of all node types
22: **until** mobility of all nodes is reduced to 1

As soon as all nodes are scheduled, $\{\mathtt{DG}_{\mathcal{U}}\}_{max}$ represents the number of functional units of type $\mathcal{U}$ required for synthesising the algorithm.

It should be noted that the original algorithm [13] suggests reducing the mobility to one ($S^{\mathsf{X}}_{min} = S^{\mathsf{X}}_{max}$) instantaneously at each mobility reduction phase. However, since the attempt to schedule a node immediately makes this algorithm *greedy* [16], scheduling should be made by reducing the mobility of each node in steps, as presented herein and shown in Chapter 5.

After reducing the mobility of a node, the scheduling bounds, S_{min} and S_{max}, for all nodes are updated and consequently the values of the distribution graphs must be re-computed. Afterwards the forces are recomputed for all node forces and the mobility of nodes is reduced.

Complexity of Algorithm 3.3

The complexity of the first phase of the FDS algorithms is $\Theta(N)$, since the actual value of the scheduling bounds has already been computed by the critical

path analysis algorithm. To compute the distribution graphs a constant number of computations for all c-steps in the scheduling bounds of all nodes must be performed. Since the complexity to compute the scheduling bounds depends on T_{iter} and on the DFG, the worst-case will be $O(N \times T_{iter})$. Computing the node force to a c-step x depends on all node inputs and outputs, and for each of the latest on the value of the distribution graph on all c-steps in its scheduling bounds. Since the number of edges in the DFG is limited by the average number of inputs per node and the algorithm requires to compute the force on all c-steps, the complexity is upper bounded by $O(N \times \langle E \rangle \times T_{iter}^2)$. By assuming that the node mobility is always set to 1 at phase 4 (the original algorithm), the complexity of the FDS is

$$O(N^2 \times \langle E \rangle \times T_{iter}^2) \tag{3.32}$$

Considering that the mobility of nodes is decreased by a factor of $\alpha \in [0; 1[$, then up to $\lceil log_{\frac{1}{\alpha}} M \rceil$ iterations will be needed to reduce the mobility of a node from M to 1. Since the node mobility depends on T_{iter}, the worst case complexity is

$$O\left(N^2 \times \langle E \rangle \times T_{iter}^2 \times \left\lceil \log_{\frac{1}{\alpha}}(T_{iter}) \right\rceil\right) \tag{3.33}$$

3.3.2 Adapted Static-Based List Scheduler

The ASLBS herein presented was developed within the scope of this thesis. It is an adaptation of a resource-constrained scheduler to solve time-restricted problems. The original algorithm belongs to the class of list-based schedulers such as those proposed in [13–15]. In these algorithms, the goal is to minimise the number of c-steps used to compute all the operations in a DFG. To accomplish this goal, nodes are sorted in a list accordingly to a dynamic or static priority function, depending if the list is updated during scheduling or not. The presented approach uses the principle applied in list scheduling algorithms, but solves time-constrained problems by iteratively running the Static List-Based Scheduler (SLBS) until the time constraint is satisfied, and adding more functional units at each new iteration.

The ASLBS operates by executing the three phases depicted in Algorithm 3.4. In the first phase, the scheduling bounds are computed and an initial number of functional units is allocated – the number required to perform an initial iteration of the SLBS. This initial value is computed based on the minimum number of units required to schedule all the operations in the DFG, assuming that units will be used with maximum efficiency and nodes will be distributed evenly amongst both functional units and c-steps. In this sense, for each node type, the number of functional units initially allocated will be the ceil of the ratio between the number of

Algorithm 3.4 Adapted Static List-Based Scheduling

1: /* *Phase 1 - Initializations* */
2: Compute node scheduling bounds S_{min} and S_{max}
3: **for all** Node types $\mathcal{U}$ **do**
4: Allocate an initial number of units, $N^{\mathcal{U}}_{alloc} = \frac{count\{\mathsf{U}\in\mathcal{U}\}}{count\left\{x\in[0;T_{iter}-1]\ :\ \exists_{\mathsf{U}\in\mathcal{U}}\ S^{\mathsf{U}}_{min}\leq x\leq S^{\mathsf{U}}_{max}\right\}}$
5: **end for**
6: **loop**
7: /* *Phase 2 - Perform Static List-Based Scheduling* */
8: **for all** nodes U in RS list, ordered decreasingly **do**
9: Schedule node U in the first c-step S^{U} possible such that $S^{\mathsf{U}} \geq S^{\mathsf{U}}_{min}$
10: **for all** nodes $\mathsf{V} \in \mathcal{S}ucc_{\mathsf{U}}$ **do**
11: $S^{\mathsf{V}}_{min} = min\{S^{\mathsf{V}}_{min}; S^{\mathsf{U}} + \mathsf{P}_{\mathsf{U}}\}$
12: **end for**
13: **end for**
14: /* *Phase 3 - Evaluate time-constraint* */
15: **if** $\forall_{\mathsf{U}}:\ max\left\{S^{\mathsf{U}}\right\} \leq T_{iter}$ **then**
16: Return
17: **end if**
18: **end loop**

nodes for the given type and the number of c-steps in the time-constrained interval $[0\ ;\ T_{iter}-1]$, which can be used to schedule nodes of that type. In phase two, the SLBS operates as follows: in each iteration, the algorithm takes the node with the maximum RS value and assigns it in the first possible c-step available. Since the SLBS is a resource-constrained algorithm, it returns the minimum number of c-steps it used to schedule all nodes using the allocated number of units, T_{SLBS}. The next phase of the ASLBS is to compare T_{SLBS} with the time constraint T_{iter}. If $T_{SLBS} \leq T_{iter}$, the SLBS is already able to fulfil the time restriction and the algorithm ends. However, if the condition is not respected, more units must be allocated and a new iteration of the SLBS must be performed. To compute the number of units per node type to be used in the next SLBS iteration, the algorithm compares: *i)* the maximum number of units allocated in the previous iteration (M_{max}); *ii)* the number of units that it really used (M); and *iii)* the time constraint failure ratio T_{SLBS}/T_{iter}. The number of units per node type, M'_{max}, to be used in the next SLBS iteration will then be:

$$M'_{max} = \begin{cases} M & , M < M_{max} \\ M \times \frac{T_{SLBS}}{T_{iter}} & , M \geq M_{max} \end{cases} \tag{3.34}$$

Complexity of Algorithm 3.4

The ASLBS is based on iterating a static based list scheduler multiple times until the time constraint is satisfied. The complexity of the ASLBS is therefore the product between the complexity of the SLBS and the number of iterations performed to satisfy the time constraint.

The complexity of the SLBS is defined by the instructions required to *i)* retrived the first node in the *RS* order list, *ii)* find the first c-step available and *iii)* assign the node to that c-step. Therefore, in the worst case the complexity of the SLBS algorithm is $O(N \cdot \langle E \rangle)$.

For computing the complexity of the algorithm, one must consider its asymptotic growth, i.e., the variation of the scheduling time for a $T_{iter} > T_0$ and $N > N_0$. However, since for any DFG it is always possible to schedule nodes with only one unit for each node computational type, asymptotically only one iteration of the SLBS needs to be performed.

Considering however that T_{iter} is small, the maximum number of times the SLBS is performed can be computed by adding one unit per iteration until a maximum of N units are allocated. Since initially $\left\lceil \frac{N}{T_{iter}} \right\rceil$ units are allocated, the maximum number of iterations k is such that the following condition is true:

$$\left\lceil \frac{N}{T_{iter}} \right\rceil + (k-1) = N \tag{3.35}$$

Re-writting (3.35) in terms of k, results

$$k = 1 + N - \left\lceil \frac{N}{T_{iter}} \right\rceil = 1 + N \cdot \frac{T_{iter}}{T_{iter}} - \left\lceil \frac{N}{T_{iter}} \right\rceil \quad \Leftrightarrow$$

$$k = 1 + \left\lceil N \cdot \frac{T_{iter} - 1}{T_{iter}} \right\rceil \tag{3.36}$$

Again, note that (3.36) is an upper bound valid for a small iteration time and is not valid when T_{iter} is large, where the condition stating that N functional units are required for scheduling is no longer valid.

However, the algorithm initialisation stage, which computes the initial number of functional units allocated does depend on the iteration time. Thus, the complexity of the algorithm is:

$$O(T_{iter} \cdot N) \tag{3.37}$$

3.3.3 Time-Constrained List-Based Scheduler

Following an approach similar to the Force-Directed Scheduling algorithm, the TCLBS attempts to distribute nodes amongst c-steps to minimise the number of

scheduled functional units. However, instead of considering that an infinite number of units are available, the TCLBS algorithm tries to schedule nodes to a fixed number of units. If the initial set of allocated units is not enough to perform scheduling, the algorithm then allocates extra units of the required functional types.

The TCLBS executes the 9 phases as depicted in Figure 3.3 and in Algorithm 3.5. In phase a) the mobility, $S_{min}^{N_j}$ to $S_{max}^{N_j}$, of all nodes N_j in the DFG is computed. Following, phase b) allocates an initial number of units per computational type, by assuming that the units will be used with maximum efficiency. Thus, the algorithm divides the number of nodes of each computational type by the number of c-step where the units can be used, which is determined through the scheduling bounds. The last initialisation phase c) creates the list of nodes required for scheduling. It contains all nodes except those generating constant values or representing DFG inputs or outputs. The final six steps of the algorithm operate in a cycle basis until all nodes have been scheduled. Phase d) selects the next node N_j to be scheduled and phase e) checks if there are available units in $[S_{min}^{N_j}\ ;\ S_{min}^{N_j}]$ c-step interval for the scheduling of node N_j. If it is true, then phase f) follows by assigning the node to a processing unit/c-step pair and in phase h) the scheduling bounds are updated for all nodes not yet scheduled. However, if in phase e) it is detected that the node cannot be allocated due to a resource constraint, a new unit of the required type is allocated and the algorithm returns to the previous phase.

Algorithm 3.5 Time-Constrained List-Based Scheduler

1: Compute node scheduling bounds S_{min} and S_{max}
2: Allocate initial set of units
3: Construct `ToSchedule` list of nodes
4: **while** `ToSchedule` list is not empty **do**
5: Choose a node for scheduling
6: Schedule node to a c-step
7: Update scheduling bounds
8: Remove the scheduled node from `ToSchedule` list
9: **end while**

Several methods were implemented in order to evaluate which one would be the best to decide about the next node to schedule and in which c-step it should be executed, phases d) and f) of Figure 3.3 respectively.

To choose the next node to be scheduled, the following four methods were implemented:

- based on the computed ES and RS values, the four methods `MIN_ES`, `MAX_ES`, `MIN_RS` and `MAX_RS` were implemented, corresponding to the the minimum and

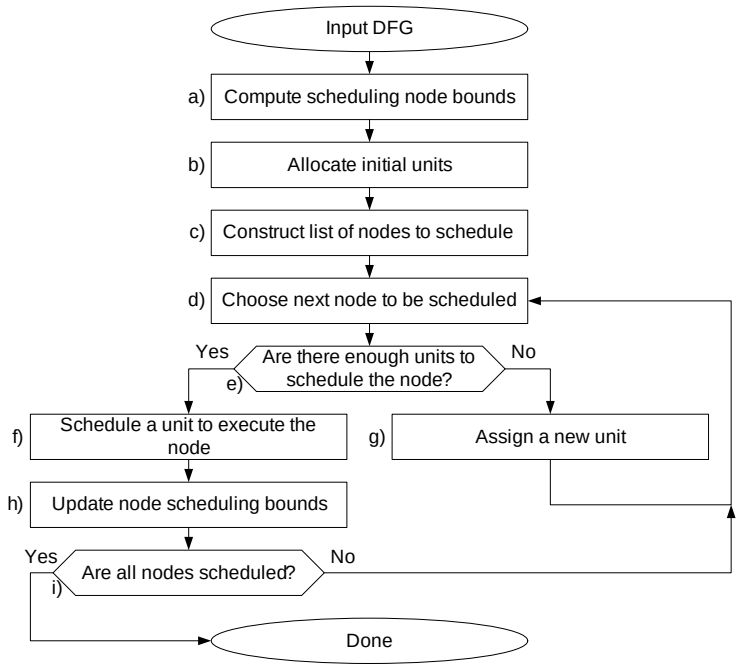


Figure 3.3: Main computing phases of the TCLBS algorithm.

maximum values of ES and RS, respectively.

- based on the minimum number of c-steps available for scheduling a node, `AVAIL_ASAP` and `AVAIL_ALAP` methods were implemented, which differ in the tie solving procedure. The first retrieves the node with smaller computational path amongst the tied candidates, thus privileging an ASAP type scheduling. The second retrieves the node with the highest computational path privileging an ALAP type scheduling. Also, these algorithms give priority to any node that has only 1 or less units available in its scheduling bounds.

For deciding the c-step of execution and the functional unit performing the computation of the node operation, different algorithms were considered by using different heuristics:

- `AVAIL`: schedule the node `U` in the first c-step with a unit available for execution;

- `FREE`: schedule the node `U` in the c-step with maximum number of units available to perform the required computation;

- `PROB`: schedule the node `U` in the c-step which probabilistically will be used to compute the lowest number of nodes of the same type; it operates by *i)* constructing the Distribution Graph, `DG`(s), of all unscheduled nodes of type

$\mathcal{U}$ ($\mathsf{U} \in \mathcal{U}$) for all c-steps s

$$\mathtt{DG}_{\mathcal{U}}(s) = \sum_{\mathsf{v} \in \mathcal{U}} P^{(0)}(S^{\mathsf{v}} = s) \tag{3.38}$$

where $P^{(0)}(S^{\mathsf{V}} = s)$ is computed as shown in (3.15); and *ii)* by scheduling the node in c-step S^U where

$$\mathtt{DG}_{\mathcal{T}}(S^{\mathsf{U}}) \geq \mathtt{DG}_{\mathcal{T}}(r) \quad , \forall_{r \in [S^{\mathsf{U}}_{min}; S^{\mathsf{U}}_{max}]} \tag{3.39}$$

- `FORCE`: schedule the node U based on the c-step which maximises the probability of scheduling all preceding and succeeding nodes, as shown in (3.22).

Two tie break decisions were designed for each type of scheduling approach: scheduling of the node closest to the minimum, or to the maximum c-step of the scheduling bounds, leading to the following 8 c-step choosing methods `AVAIL_FIRST`, `AVAIL_LAST`, `PROB_FIRST`, `PROB_LAST`, `FREE_FIRST`, `FREE_LAST`, `FORCE_FIRST` and `FORCE_LAST`.

Complexity of Algorithm 3.5

The complexity of the TCLBS depends on the adopted algorithms for: *i)* choosing the next node to schedule, *ii)* choosing the c-step where the node is assigned, and *iii)* updating the scheduling bounds at the end of each iteration. The latest is $O(N)$.

In terms of complexity, the node choosing methods `MIN_ES`, `MAX_ES`, `MIN_RS` and `MAX_RS` are all $\Theta(1)$ since they use lists ordered by ES and RS to choose the node. On the other hand, the `AVAIL_ASAP` and `AVAIL_ALAP` methods need to look for the node with the lowest mobility and/or the node with 1 or less units available in its scheduling bounds. Thus, their complexity is increased to $\Theta(N \times T_{iter})$.

The complexity of the c-step choosing methods can be computed observing that: *i)* `AVAIL` depends only on finding one unit available for processing the respective node, thus the worst-case complexity can be written as $O(T_{iter})$; *ii)* `FREE` requires to find the c-step with maximum number of units available for scheduling, thus its execution time complexity is $O(T_{iter})$; *iii)* the time to assign a c-step using the `PROB` method is $O(N \times T_{iter})$, since it requires to scan all c-steps on all nodes of the type of computational unit of the node in scheduling; at last, *iv)* the `FORCE` technique has a worst-case complexity $O([T_{iter} \times N]^n)$, where n represents the level of dependency used for computing the approximated probability, see (3.22), (3.23) and (3.24). The complexity of this algorithm is therefore the maximum one amongst the node choosing methods, the c-step choosing method and the scheduling bounds update method.

Table 3.1: Complexity of the TCLBS algorithm in the different scheduling options.

C-step choosing method	Node choosing method	
	MIN_ES / MIN_RS / MAX_ES / MAX_RS	AVAIL_ASAP / AVAIL_ALAP
AVAIL	$O(N \cdot T_{iter})$	$O(N^2 \cdot T_{iter})$
FREE	$O(N \cdot T_{iter})$	$O(N^2 \cdot T_{iter})$
PROB	$O(N^2 \cdot T_{iter})$	$O(N^2 \cdot T_{iter})$
FORCE	$O(N^{n+1} \cdot T_{iter}^n)$	$O(N^{n+1} \cdot T_{iter}^n)$

The complexity values for each case are shown in Table 3.1. Note that, if using the AVAIL_ASAP or the AVAIL_ALAP techniques as node choosing method, the complexity of the TCLBS algorithm does not depend on the c-step choosing method (assuming the FORCE technique is applied using $n = 1$).

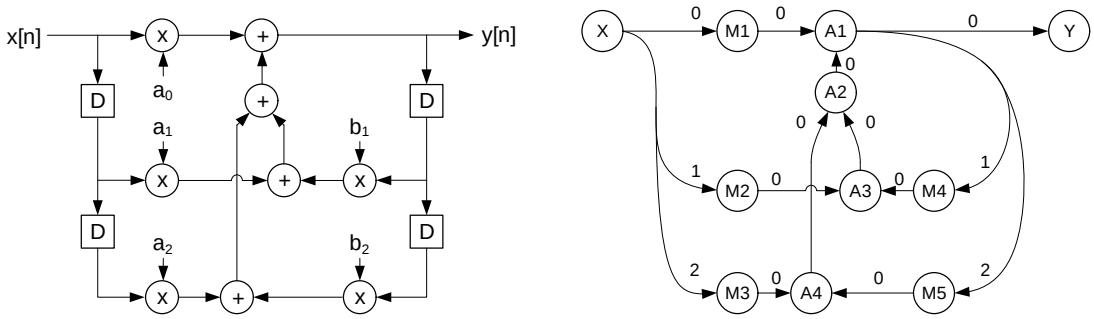
3.4 Register Allocation

To minimise synthesis time, the techniques adopted to implement register allocation are independent from the scheduling process. For computing the number of required registers, the *lifetime analysis* [23] technique was implemented. It uses the concept of variables, i.e. the output values of computational nodes, to determine the minimum number of registers required for implementing the algorithm represented in the DFG. A variable is alive from the c-step it is produced until the c-step it is consumed: if a node U is executed in c-step S^{U} and has a computational weight $\mathtt{w}_{\mathsf{U}}$, then its output variables are generated at c-step $S^{\mathsf{U}} + \mathtt{w}_{\mathsf{U}}$; if a node output variable v_{U} is used by nodes V, W and X, then the variable v_{U} is consumed at c-step x, where

$$x = max \left\{ \begin{array}{l} T_{iter} \cdot w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle} + S^{\mathsf{V}} \\ T_{iter} \cdot w_{\langle \mathsf{U} \rightarrow \mathsf{W} \rangle} + S^{\mathsf{W}} \\ T_{iter} \cdot w_{\langle \mathsf{U} \rightarrow \mathsf{X} \rangle} + S^{\mathsf{X}} \end{array} \right\}$$

Using this technique, the maximum number of variables alive during each c-step represents the minimum number of registers required for the implementation of the algorithm described by the DFG.

To illustrate this lifetime analysis, let us consider the application of Algorithm 3.6 to the second order IIR filter depicted in Figure 3.4. By performing scheduling using the SLBS with 2 functional units (1 multiplier and 1 adder), the scheduling results and the c-steps of variable generation and consumption are presented in Table 3.2. Let us now consider the variable produced by node $\mathsf{A1}$, v_{A1}. Since node $\mathsf{A1}$ is executed at c-step 7, its variable is produced at c-step 8. This variable will be



(a) Block Diagram

(b) Data-Flow Graph (without constant nodes)

Figure 3.4: DFG a first order IIR filter and its scheduling values.

Table 3.2: Scheduling results, in c-steps, of the second order IIR filter.

Node	A1	A2	A3	A4	M1	M2	M3	M4	M5
Computational weight	1	1	1	1	2	2	2	2	2
Scheduling	7	6	3	5	4	0	2	1	3
Variable generation	8	7	4	6	6	2	4	3	5
Variable consumption	19	7	6	6	7	3	5	3	5
Variable storing	8-18	-	4-5	-	6	2	4	-	-

consumed by nodes M4 and M5 at c-steps $8 \times 1 + 1 = 9$ and $8 \times 2 + 3 = 19$, respectively. Therefore the variable v_{A1} is required to be stored from c-step 8 to c-step 18, as shown in the circular lifetime chart presented in Figure 3.5. Note that c-steps 8 to 15 are coincident with c-steps 0-7 on the chart but represent the storing of variables generated at iteration $n - 1$. Similarly c-steps 16-18 are coincident with c-steps 0-3 and represent the storing of a variable generated at iteration $n - 2$. By performing the same analysis to all the other variables, it is possible to compute the minimum number of registers as the maximum number of variables that are required to be stored in each c-step. In the example of the second order IIR filter, the minimum number of registers is 3: required to provide storing of variables $\{v_{\mathsf{A1}}[n-1], v_{\mathsf{A1}}[n-2], v_{\mathsf{M2}}[n]\}$ and $\{v_{\mathsf{A1}}[n-1], v_{\mathsf{A3}}[n], v_{\mathsf{M3}}[n]\}$ at c-steps 2 and 4, respectively.

After life-time analysis, *forward-backward register allocation* is used to assign variables to registers [24]. Using this technique, variables are allocated in the first possible register available at the c-step they are produced, and are propagated in a forward manner until they are dead. If a variable reaches the last register in the list, it is propagated back to the first possible register available. Also if multiple variables are produced at the same c-step, they are assigned to registers in

Algorithm 3.6 Life-time analysis algorithm

1: **for all** types of variables $\mathcal{V}$ **do**
2: Set $R[i] = 0,\ \forall_{i \in [0\ ;\ T_{iter}-1]}$
3: **for all** variables $\mathsf{v} \in \mathcal{V}$, generated in node U **do**
4: Get generation c-step $S^{\mathsf{v}}_{gen} = S^{\mathsf{U}} + \mathsf{P}_{\mathsf{U}}$
5: Get consumption c-step $S^{\mathsf{v}}_{cons} = max\{S^{\mathsf{V}}\},\ \forall_{V \in \mathsf{S}_{\mathsf{U}}}$
6: **for** $i \in [0\ ;\ T_{iter} - 1]$ **do**
7: $R[i] = R[i] + \left\lceil \frac{S^{\mathsf{v}}_{cons} - i}{T_{iter}} \right\rceil - \left\lceil \frac{S^{\mathsf{v}}_{gen} - i}{T_{iter}} \right\rceil$
8: **end for**
9: **end for**
10: Allocate $R^{\mathcal{V}}_{max} = max\{R[i]\}_{i \in [0\ ;\ T_{iter}-1]}$ registers for variables of type $\mathcal{V}$
11: **end for**

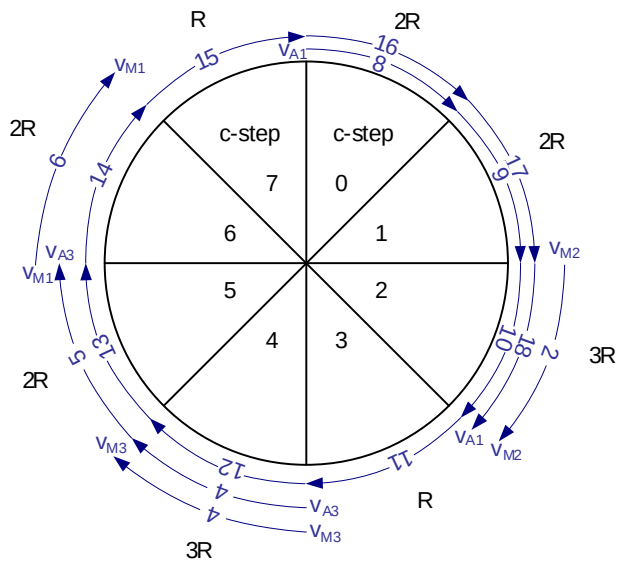


Figure 3.5: Circular lifetime chart of the second order IIR filter.

decreasingly longevity order. This technique, represented in Algorithm 3.7, allows to reduce the number of multiplexers by trying to allocate registers in a circular order: $R1 \rightarrow R2 \rightarrow \cdots \rightarrow RN \rightarrow R1$.

Algorithm 3.7 Forward-backward register allocation algorithm

1: **for all** types of variables $\mathcal{V}$ **do**
2: **for all** variables $\mathsf{v} \in \mathcal{V}$, generated by node U, ordered by increasingly $(S^{\mathsf{v}}_{cons} - S^{\mathsf{v}}_{gen})$ **do**
3: Assign the first free register $R_{S^{\mathsf{v}}_{gen}}$ to store variable v in c-step S^{v}_{gen}
4: **for** $s \in [S^{\mathsf{v}}_{gen} + 1\ ;\ S^{\mathsf{v}}_{cons} - 1]$ **do**
5: **if** register $((R_{s-1} + 1)\ \%\ R^{\mathcal{V}}_{max})$ is available at c-step $(s\ \%\ T_{iter})$ **then**
6: Assign register $((R_{s-1}+1)\%R^{\mathcal{V}}_{max})$ to store variable v at c-step $(s\%T_{iter})$
7: **else**
8: Assign the first free register R_s to store variable v in c-step $(s\%T_{iter})$
9: **end if**
10: **end for**
11: **end for**
12: **end for**

By applying this technique to the second order IIR filter of Figure 3.4, the register allocation presented in Figure 3.6 is obtained. In the first step, register allocation is performed to node $\mathsf{A1}$. Since all registers are free, the variable v_{A1} will be firstly propagated to register $R1$. This variable will then be propagated to the following registers from c-step 8 to 18: $R1 \rightarrow R2 \rightarrow R3 \rightarrow R1 \rightarrow R2 \rightarrow R3 \rightarrow R1 \rightarrow R2 \rightarrow R3 \rightarrow R1 \rightarrow R2$. As can be seen, circular order on register assignment is not interrupted by reaching the last c-step. This allows to decrease both the number and dimension of multiplexers. Following the same procedure, variables v_{M2}, v_{A3}, v_{M3} and v_{M1} are assigned to registers: $v_{\mathsf{M2}} \rightarrow R1$, $v_{\mathsf{A3}} \rightarrow R1 \rightarrow R2$, $v_{\mathsf{M3}} \rightarrow R3$ and $v_{\mathsf{M1}} \rightarrow R2$.

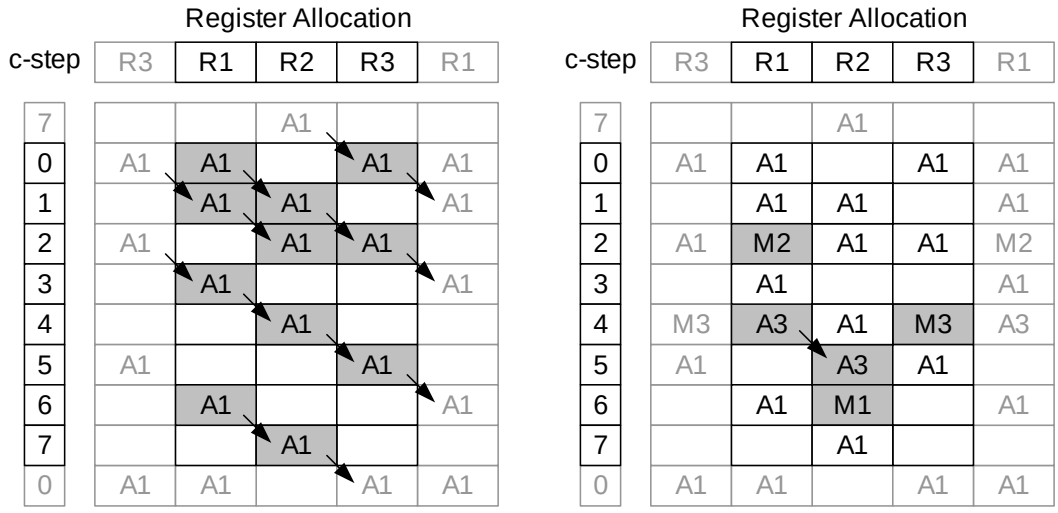
Complexity of Algorithm 3.6 and Algorithm 3.7

The time required to perform life-time analysis and to compute the minimum number of registers depends on the number of variables created during the DFG, the number of output edges per node and the time constraint T_{iter}, as shown in Algorithm 3.6:

$$O(N \times (T_{iter} + \langle E_V \rangle)) \tag{3.40}$$

where $\langle E_V \rangle$ represents the average number of edges originated per variable. However, in common DSP applications and architectures, $T_{iter} > \langle E_V \rangle$. Therefore, (3.40) can be written as:

$$O(N \times T_{iter}) \tag{3.41}$$



(a) Register allocation for the output variable of node A1

(b) Register allocation for the output variables of nodes A3, M1, M2 and M3

Figure 3.6: Register allocation of the second order IIR filter.

Since node ordering is achieved by using an insertion sort algorithm, whose complexity is $O(N^2)$, the implementation of the forward-backward register allocation is

$$O(N^2 + N \times T_{iter}) \tag{3.42}$$

Thus, in the worst case, the time to perform register allocation grows quadratically with the number of nodes and linearly with the iteration time T_{iter}.

3.5 Conclusions

In this chapter the four main phases for HLS were presented and its worst-case complexity was analysed. Firstly, for *critical path analysis*, it was presented an algorithm able to compute the ES and RS of a node in $O(N^2)$. For *retiming*, a new algorithm was developed for creating the ILP constraints, which defines the conditions of the retiming problem. This algorithm is able to compute the constraints in $O(N^2 \cdot \langle E \rangle)$ and reduces the computational time of the ILP solver by pruning most of the constraints. However, the complete retiming algorithm is upper bounded by the ILP solver, which is $O(N^3)$. For scheduling the graph nodes, three algorithms were presented. The FDS is a typical time-constrained scheduler which performs scheduling with a quadratic dependency on both the number of nodes and the iteration time T_{iter}, in the worst-case. The ASLBS is an adaptation of a resource-constrained algorithm. Nevertheless, it is the simplest of all scheduling algorithms

with a $O(N \cdot T_{iter})$ worst case complexity. The third scheduling technique was also developed within the scope of this thesis. The TCLBS is a list based scheduler which can be configured for different node selection and c-step assignment techniques. Depending on the configuration, the complexity of this scheduling algorithm varies between $O(N \cdot T_{iter})$ and $O(N^{n+1} \cdot T_{iter}^{n})$, where n is the dependence level considered to approximately compute the probability in (3.22). Finally, for register allocation two algorithms are presented: *lifetime analysis* and *forwards-backward register allocation*. The first algorithm computes the minimum number of registers required for a given scheduling solution in $O(N \times (T_{iter} + \langle E_V \rangle))$. The second algorithm tries to minimise the number of multiplexers by allocating registers using a circular order. The complexity of this algorithm is quadratic with the number of nodes and linear with the iteration time.

4

Tool Architecture and Implementation

Contents

4.1 Describing the DFG 52
4.2 Data structures 56
4.3 Programming the developed tools 64
4.4 Conclusions 68

Based on the techniques presented in Chapter 3, a tool capable of transforming a graph description of a DSP algorithm into a RTL description was implemented. One of the goals was for the programmed tools to have short execution time, while allowing to easily add new features and/or algorithms. Thus, all the code was written in C language, and specific data structures were created considering code extensibility.

The main components of this tool are represented in Figure 4.1. The tool starts by reading the DFG from a text file, whose structure is shown in Section 4.1. While reading the code, the tool performs a series of feasibility checks, returning error if any of the checks fails. Amongst the feasibility errors that can be detected are: *i)* nodes with undefined types, *ii)* duplicated names of nodes or node types, *iii)* negative edge weights, *iv)* edges originating in or leading to an undefined node or port. All functional units are considered to operate in pipeline, where the outputs of functional units are registered. In this sense, the computational weight of nodes can directly be translated into the number of pipeline stages of the unit performing the node's execution. If it is not the case, there is no guarantee that the resulting VHDL design is feasible since it may be possible to generate combinational loops. Note that pipelining the functional units is an implementation aspect that does not alter the function of the algorithm being implemented. The functional units description is made using VHDL. There is no error checking performed within the ATAS tool. Instead, if any error exists in the VHDL code of a functional unit, it will be translated onto the final architecture file and will lead to a VHDL synthesis error. Also, specifying a different number of pipeline stages in the functional unit description and on the VHDL code will result on an invalid implementation of the algorithm. After reading the description of the DFG, the next component of the tool computes the weight of the critical paths in the circuit (see Section 3.1), and checks for the existence of zero weight closed paths. If any computational path is longer than the predefined maximum latency, the DFG must be retimed in order to fulfil the time-constraint, by using the algorithm described in Section 3.2. If the retiming algorithm fails to re-distribute delays, the constraint is impossible to be fulfilled and the ATAS tool will terminate with an error. After performing the retiming task, the weight of all paths in the DFG is updated. The next phase consists in scheduling nodes to functional units by using one the algorithms specified in Section 3.3. This task may however fail because of resource-constraints that were specified and restrict the maximum number of functional units that could be allocated for any given operation. If it succeeds, the tool will continue by performing register allocation as presented in Section 3.4. The final task is to build the data-path by creating: *i)* the

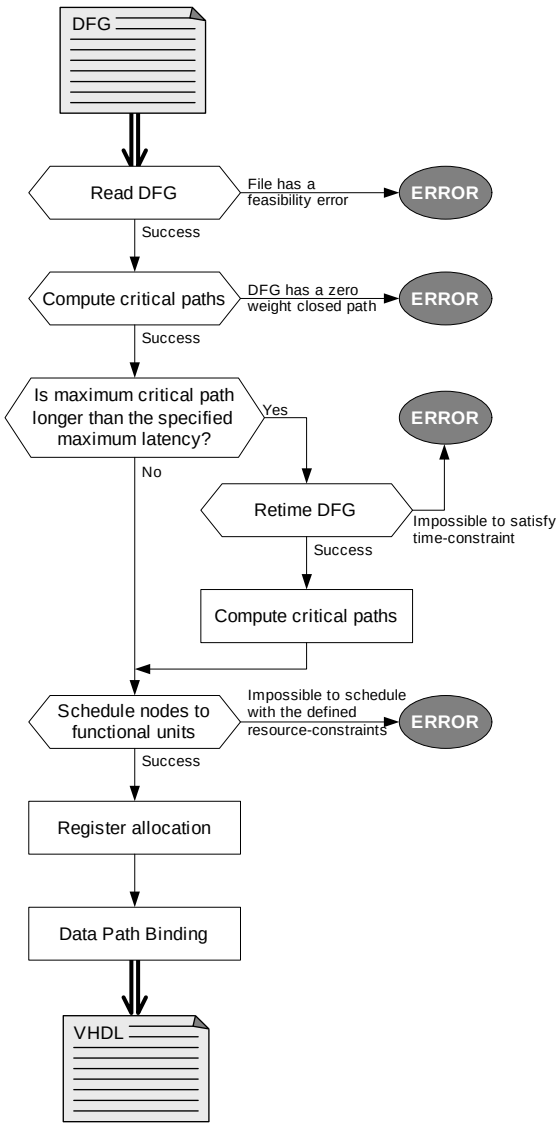


Figure 4.1: Architecture of the ATAS tool.

multiplexers; *ii)* the connections between functional units and registers; and *iii)* the control logic.

This chapter is organised as follows. Section 4.1 explains the structure of the text file describing the DFG to translate into RTL. Section 4.2 describes the data structures adopted for the ATAS tool. Following, the most important implementation aspects are discussed in Section 4.3. Also in this section, the structure of the VHDL file created, after data-path binding, is presented. Finally, Section 4.4 concludes the chapter.

4.1 Describing the DFG

The ATAS application receives the DFG of the system to be synthesised in the form of a text file, with information about input and output ports, computations inside nodes and connections between nodes. It can thus be divided into 4 blocks: *i)* declaration of global ports, *ii)* declaration of functional units, corresponding to node types, *iii)* declaration of the nodes in the DFG and *iv)* declaration of edges in the DFG. All blocks are limited by `Define` and `End` identifiers. Also, excepting VHDL description of the functional units, uppercase is ignored and treated exactly the same way as lowercase. Command termination is identified as an end of line.

To illustrate the description of a DFG to input on the ATAS tool, consider the example DFG of Figure 4.2, which represents a 5th-order elliptic wave filter. In each iteration n, it receives as input a sample value $x[n]$ and outputs a filtered value $y[n]$. The global definition, including input and output ports, is made within the `DefineGlobals` and `EndGlobals` marks as shown in Figure 4.3. The following fields can be declared in the global definition section:

- name of a global clock signal (optional, default value is `CLOCK`)

```
clock alias <name>
```

- name of a global reset signal (optional, default value is `RESET`)

```
reset alias <name>
```

- maximum latency for the computation of one iteration of the algorithm

```
max_latency <value>
```

- external feedback information: boolean value representing if there is any feedback from an output to an input (created in an external VHDL file)

```
feedback <0|1>
```

Besides the above fields, the inputs and outputs must be defined within subsections `Inputs` and `Outputs`, as shown in Figure 4.3. Within each subsection multiple ports may be defined (one for each line) until the declaration of the end of section (`EndInputs` or `EndOutputs`). The declaration of each port is as follows

```
<signal name> <signal description>
```

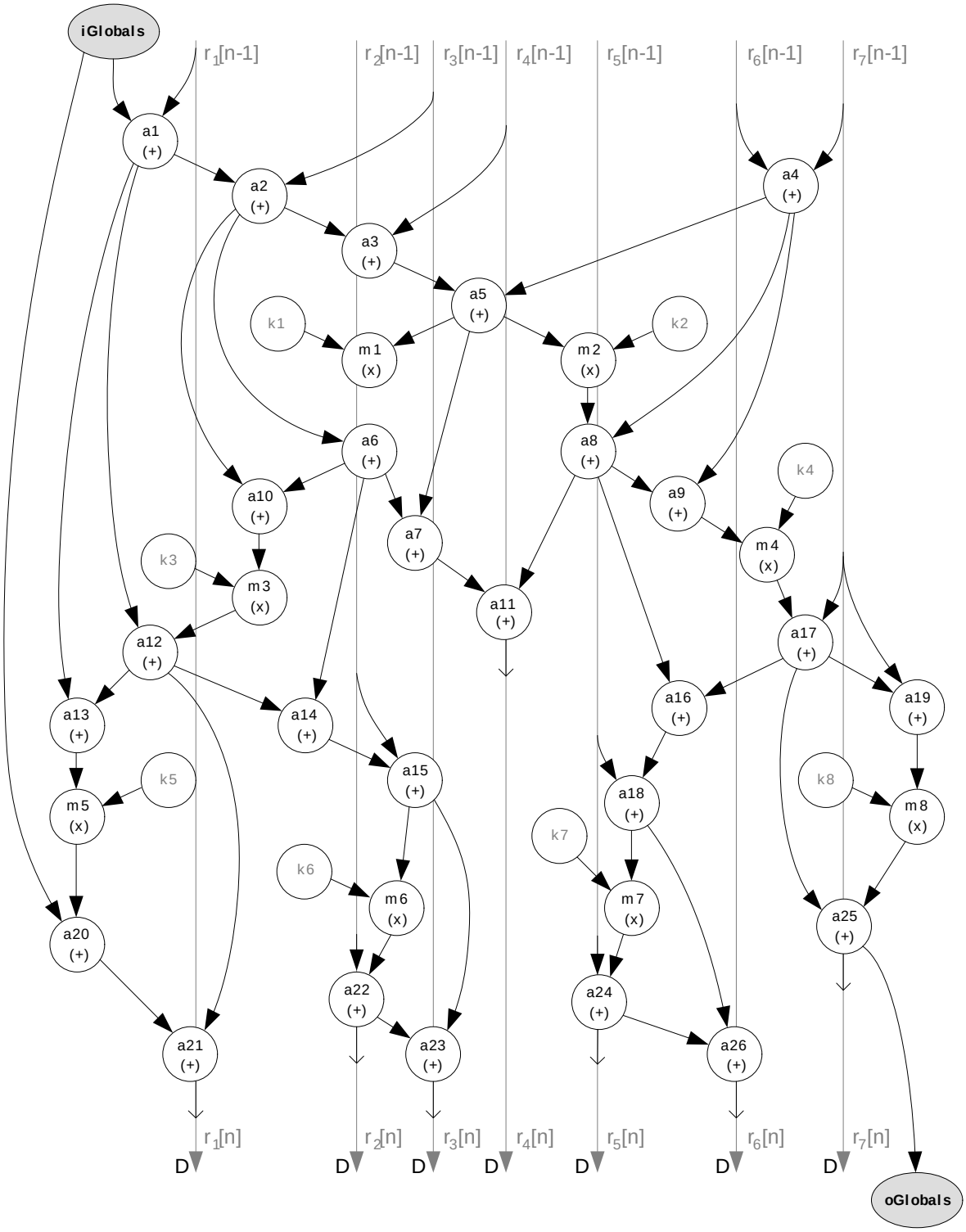


Figure 4.2: DFG of the 5th order elliptical wave filter.

```
DefineGlobals
   clock   alias   clk
   reset   alias   rst
   max_latency     17
   feedback        0
   DefineInputs
      X   std_logic_vector(11 downto 0)
   EndInputs
   DefineOutputs
      Y   std_logic_vector(11 downto 0)
   EndOutputs
EndGlobals
```

Figure 4.3: Global declarations for the 5th order elliptic wave filter.

```
DefineType tp_mult
   pipeline   1
   DefineInputs
      A   std_logic_vector(11 downto 0)
      B   std_logic_vector(11 downto 0)
   EndInputs
   DefineOutputs
      Z   std_logic_vector(11 downto 0)
   EndOutputs
   DefineBehavioral
      process(clk)
         variable aux: std_logic_vector(23 downto 0);
      begin
         if rising_edge(clk) then
            aux := {A} * {B};
            {Z} <= aux(23 downto 12);
         end if;
      end process;
   EndBehavioral
EndType
```

Figure 4.4: Declaration of a functional unit type: multiplier.

```
DefineNodes
   tp_mult m1 m2 m3 m4 m5 m6 m7 m8
   tp_k    k1 k2 k3 k4 k5 k6 k7 k8
   tp_add  a1  a2  a3  a4  a5  a6  a7  a8  a9  a10
   tp_add  a11 a12 a13 a14 a15 a16 a17 a18 a19 a20
   tp_add  a21 a22 a23 a24 a25 a26
EndNodes
```

Figure 4.5: Declaration of the nodes to describe the 5th order elliptical wave filter.

Since the signal declarations have to be translated into VHDL, both the signal name and signal description must be synthesisable by the target VHDL compiler.

The declaration of node types, which are translated into functional units after synthesis, is made by a `Type` block, as shown in Figure 4.4. Inside the type block the field `pipeline`, that indicates the number of pipeline stages of the functional unit type, must be specified:

```
pipeline <#stages>
```

The ATAS application assumes that logic is equally distributed between pipeline stages where a pipeline stage is a logic block followed by a register.

Within the `Type` block, it is also required to describe the node type input and output ports and the function unit computation. The ports are described similarly with the `Global` ports, i.e. within `Inputs` and `Outputs` sub-blocks. The functional unit computation is made in the VHDL language within a `Behavioral` or `Structural` sub-block. All signals that refer to a node type input and/or output must be enclosed within `{}`. If a `Structural` description is made, then no instance name should be defined as one will be automatically assigned. The `Structural` description should start with the component name followed by the definition of the generic map and the port map.

Definition of nodes, made within a `Nodes` block, must follow the declaration of node types, i.e. to declare a node X of type $\mathcal{X}$ one must first declare $\mathcal{X}$ and then X. The declaration of nodes for the 5th order elliptical wave filter is shown in Figure 4.5. The syntax for declaring nodes is as follows:

```
<name of node type> <node name> <node name> ...  <node name>
```

The last stage to describe an algorithm for the ATAS application is to declare the connection between nodes. While connections are not required to be made at the end of the ATAS input file, they must follow the declaration of the source and destination node. Connections are declared within the `Connections` block, one for

```
DefineConnections
   Globals,X   a1,A        0
   Globals,X   a20,A       0
   a21,Z       a1,B        1
   ...
   m7,Z        a24,B       0
   a24,Z       a26,A       0
   a25,Z       Globals,Y   0
EndConnections
```

Figure 4.6: Excerpt of the 5th order elliptical wave filter connection declaration.

each line, with the following syntax:

`<src node>,<src port> <dest node>,<dest port> <edge weight>`

where `<src node>` and `<src port>` represent the names of node and port generating the variable being communicated, `<dest node>` and `<dest port>` represent the node and port receiving the variable and `<edge weight>` is the weight of the edge in the DFG (must be a non-negative value). An excerpt of the declaration of node connections for the 5th order elliptical wave filter is shown in Figure 4.6.

4.2 Data structures

In order to describe the DFG corresponding to the algorithm to be implemented, specific data structures were created to speed up the execution of the ATAS application. Two important aspects were taken into consideration when designing the data structures: i) quick searching capabilities by using sorted lists to speedup execution; and ii) ease of future extensions of the tools. This lead to the use of hierarchical data structures with embedded ordering vectors. The main structures that were defined are shown in Figure 4.7 and have the following meaning:

- `arch_def` is the architecture base data type; this is the root structure of the created data types and includes pointers to the the following lists: node types, nodes, edges (connections between nodes) and type of variables;
- `type_def` defines the type of operation represented in each node (addition, subtraction, multiplication, division, ...); each node type includes a pointer to the list of these nodes in the DFG (e.g. type *multiplier* points to a list of all nodes performing multiplications in the DFG);
- `node_def` stores the node relative information, i.e. the operations executed in the DFG ; in order to facilitate the search of connections to/from any node,

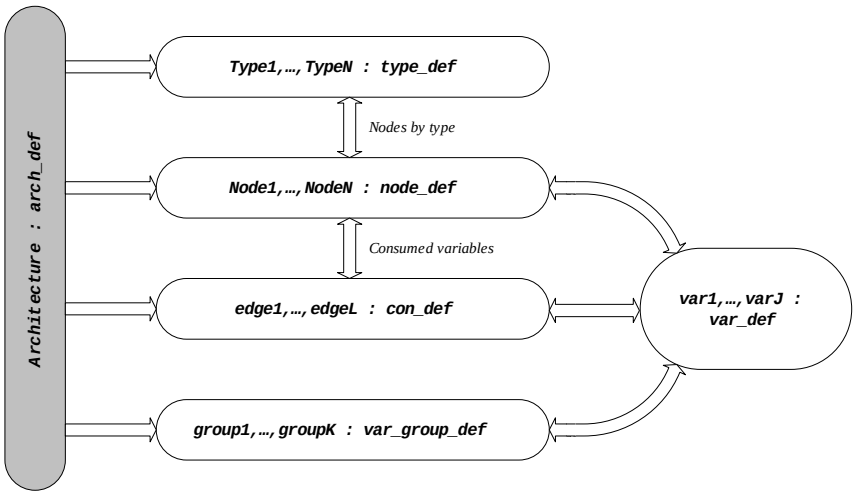


Figure 4.7: Hierarchy of the implemented data structures.

it includes a pointer to the list of edges connected to the node inputs and another pointer to the list of variables it generates;

- `var_def` stores the information concerning node variables, i.e. information about each output of each node, such as the consuming nodes;

- `var_group_def` groups the variables by definition (such as `std_logic_vector` or `integer`); this information is essential to fold multiple variables into a single register.

- `con_def` represents the DFG edges; it communicates architecture variables (node outputs) to the consuming nodes; each edge includes information regarding the variable being communicated and the node and port consuming the variable.

To ease the development of the code, all the created lists are circularly doubly linked: each element in the list points to both to the next and previous elements in the list, including head and tail. This was implemented by making all the `type_def`, `node_def` and `con_def` structures an inheritance to an ordering structure `index_def` comprised of three fields: the ordering number, `value`, and the pointers to the adjacent elements in the list, `prev` and `next`. Furthermore, it allowed the development of generic manipulating functions over the lists such as insertion, removal, ordering or displaying (usefull mostly during the debug process).

The structure of the `arch_def` data type is shown in Figure 4.8 and includes the following fields:

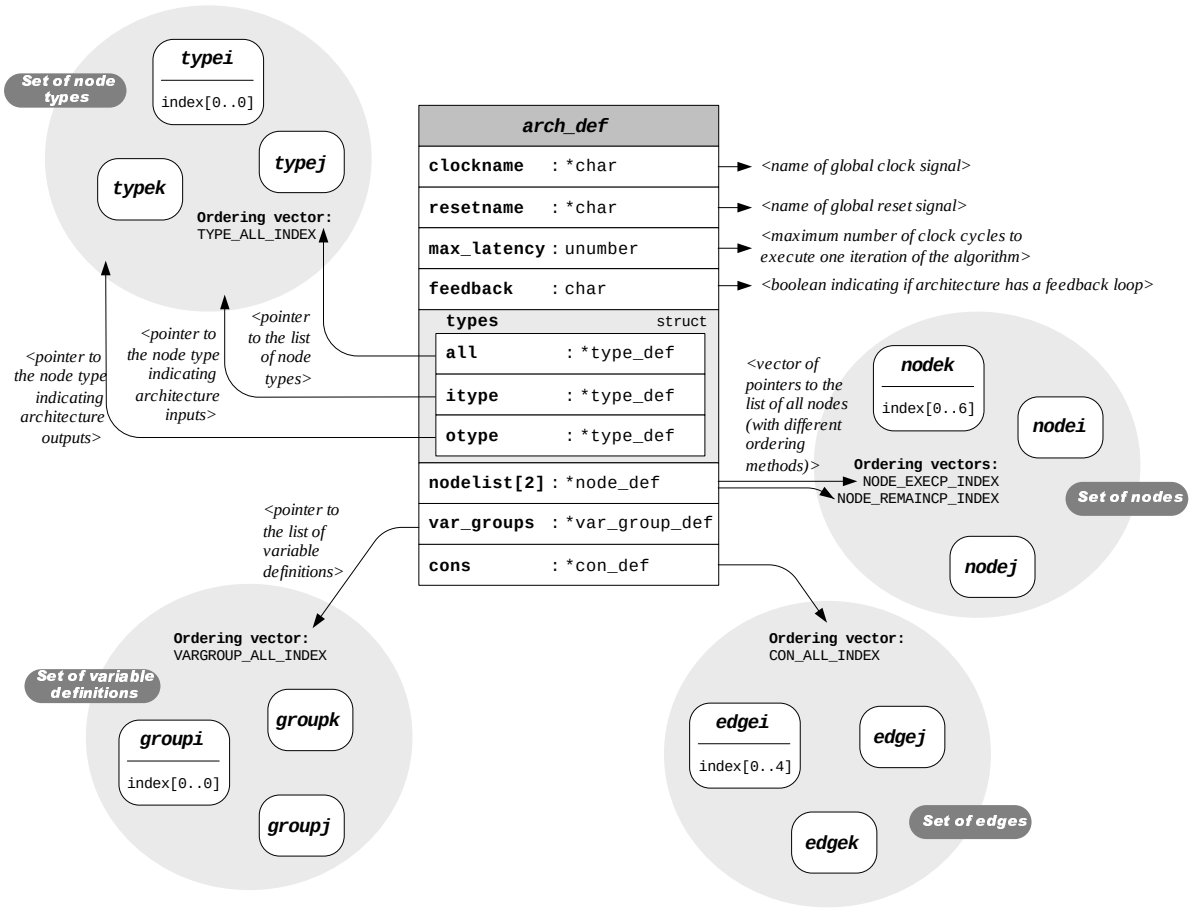


Figure 4.8: Structure of the data type which stores the architecture.

- `clockname` and `resetname` hold the name of the clock and reset signals used in the description of the circuit corresponding to the DFG in VHDL;
- `max_latency` indicates the maximum number of clock cycles available for the implementation of the algorithm (zero means infinite time);
- `feedback` is a boolea variable which states weather any of the outputs of the DFG is feedback into an input; this is only necessary if the connection is not written on the DFG and means that the retiming function cannot add delays to the circuit, which would increase the latency of the circuit;
- `types` is a structure with three fields: two pointers to the node types describing DFG inputs and outputs, `itype` and `otype` respectively, and another pointer to the list of all node types, `all`;
- `nodelist` is an array of pointers to the list of all nodes; the list is the same in all cases, but it is ordered differently;

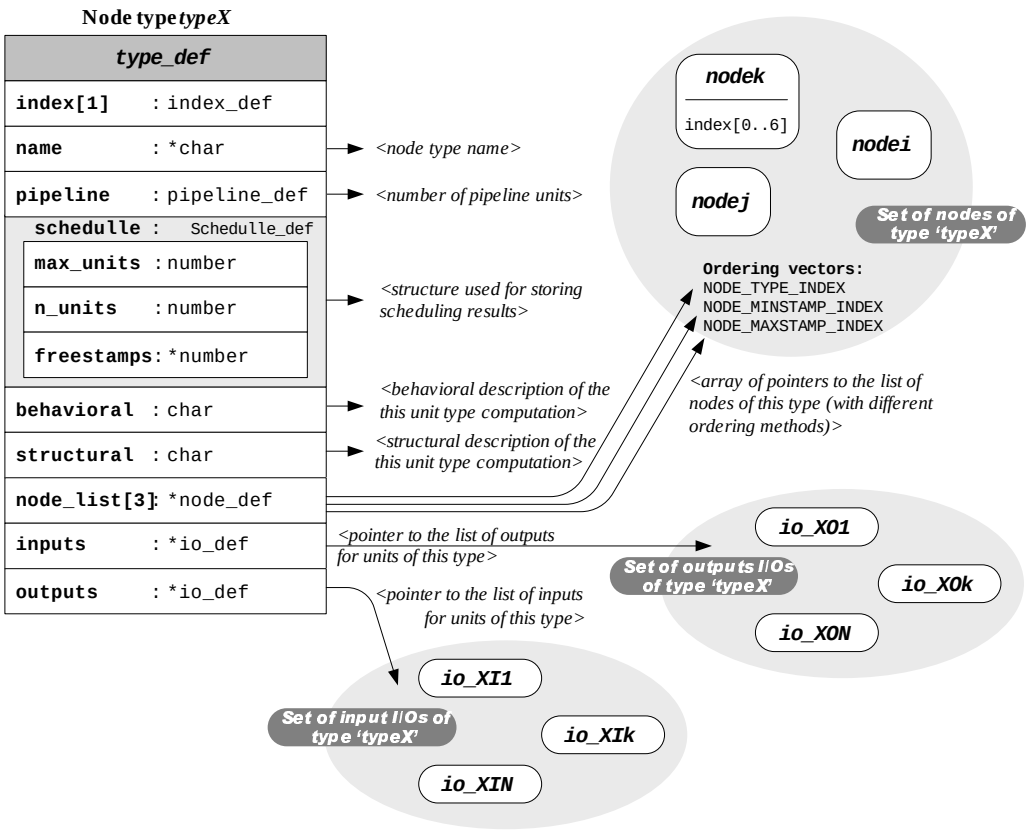


Figure 4.9: Structure of the data type to store node type relative information.

- `var_groups` and `cons` are pointers to the lists containing all types of variable definitions and all edges, respectively;

The node types, stored in the `type_def` structure, includes the nine fields shown in Figure 4.9:

- `index` is an array of `index_def` structures, which are used for listing the node types; in the current implementation, the array has only one element since the implemented algorithms do not require the ordering of the list of node types.

- `name` is a pointer to an array of characters storing the name of the type, thus requiring to allocate memory when filling the field;

- `pipeline` stores the number of pipeline stages for this computational unit, therefore specifying the number of clock cycles required for computing the result; the value 0 means that the unit does not perform any computation and the variables produced by nodes of this type can be consumed on the same c-step as they are generated;

- `schedule` is an embedded structure which is used for node scheduling;

- `behavioral` and `structural` describe the behavioural and strutural VHDL implementation of the units of this type, respectively;
- `node_list` is an array of pointers to the circularly doubly-linked list of nodes of this type; the lists are used dynamically along the execution of the ATAS application, specifically `NODE_MINSTAMP_INDEX` and `NODE_MAXSTAMP_INDEX` are used for ordering nodes by their scheduling bounds, during the scheduling process;
- `inputs` and `outputs` are pointers to the circularly doubly-linked lists of input and output ports, as shown in Figure 4.9.

The `schedule` structure has three fields. The value of the `max_units` is specified in the input file (see Section 4.1). This field is used to state the maximum number of functional units of this type that can be used when implementing the algorithm. If the scheduling algorithm is unable to assign all nodes using up to `max_units`, the scheduling is considered to have failed. If the implementation should have no restrictions on the number of functional units, the variable should be assigned with the value zero. The fields `n_units` and `freestamps` are used during the scheduling process. The first is an integer that indicates the total number of units already scheduled and the second is a pointer used by the scheduling algorithm. The relation between `n_units` and `max_units` is the following:

$$\texttt{n_units} \leq \texttt{max_units} \quad \text{, if } \texttt{max_units} > 0.$$

The circularly doubly-linked lists that store the definition of the input and output ports of the computational units described are composed by `io_def` structures. These have 4 fields: a `name` indicating the external name of the input/output port and its VHDL definition, `def`, and both `next` and `prev` pointers for the next and previous elements of the lists.

The nodes, representing the computations in the algorithm being implemented, are stored in the `node_def` structure shown in Figure 4.10. These are internally ordered by means of the `index_def` structure in up to 7 different ways. While some of the ordering methods may vary with the execution of the ATAS application, 3 have fixed ordering methods: `NODE_TYPE_INDEX` is the index to the list of nodes of the same type and `NODE_EXECCP_INDEX` and `NODE_REMAINCP_INDEX` represent the indexes to the lists of all nodes ordered by ES and RS, respectively. The other fields of the `node_def` structure are:

- `name` stores the assigned node name;

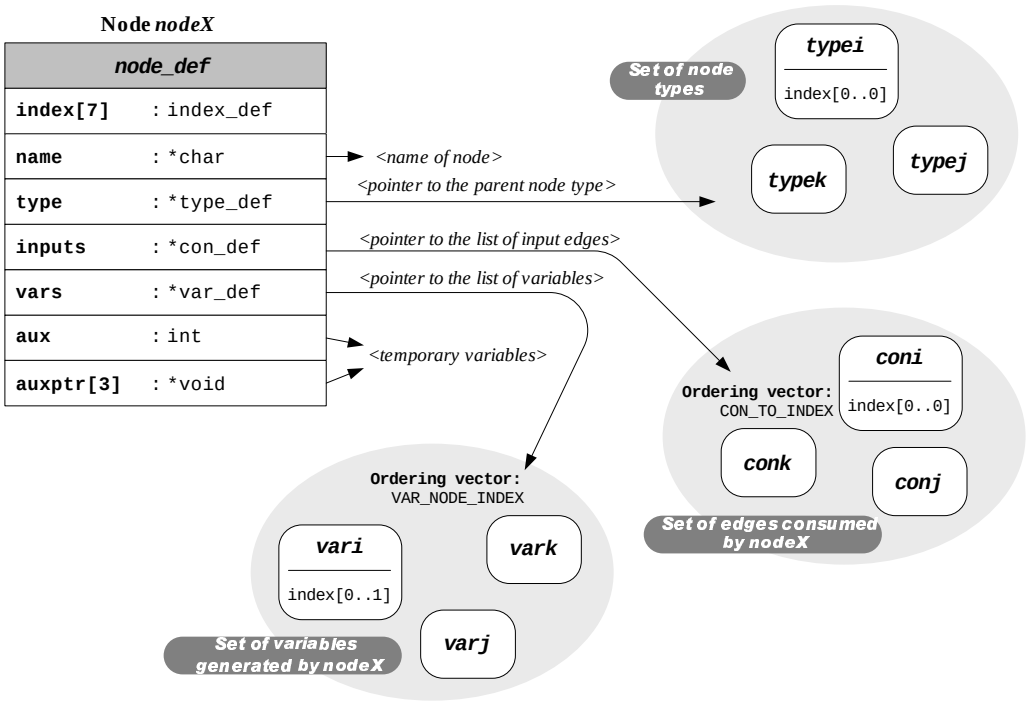


Figure 4.10: Structure of the data type to store node relative information.

- `type` points to the node type;
- `inputs` points to the circularly doubly-linked list storing the edges that lead to this node;
- `vars` points to the circularly doubly-linked list of variables generated by this node (different variables are produced in different output ports);
- `aux` and `auxptr` are auxiliary variables.

The edges, or connections between nodes, are defined by a structure named `con_def`. For an edge $\langle \mathsf{U} \rightarrow \mathsf{V} \rangle$ with weight $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$ communicating a variable v generated by node U, the fields of the `con_def` structure will hold:

- `var` is a pointer to the variable being communicated, v;
- `to_node` is a pointer to the node consuming the variable, V;
- `to_io` is a pointer to the port receiving the variable in node V;
- `delay` represents the edge weight, $w_{\langle \mathsf{U} \rightarrow \mathsf{V} \rangle}$;
- `reg` points to the register which will communicate v to the unit processing node V.

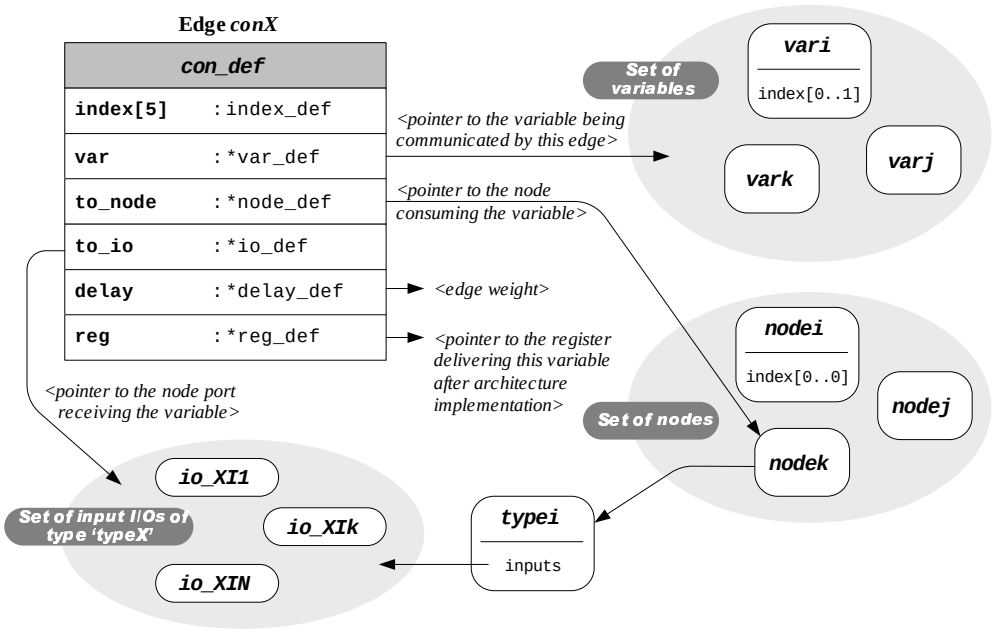


Figure 4.11: Structure of the data type which defines the edges.

To store register information, the `reg_def` structure was created with 2 fields: an `id` which is a unique identifier for the register and a `stamp` array of pointers to the functional unit or register transmitting the value to be stored at each c-step in interval $[0; T_{iter}[$.

There are two more structures involved in the problem description: one for describing variables, `var_def`, and another for describing variable definitions, `var_group`. The first structure is shown in Figure 4.12 and includes the following fields:

- `group` is a pointer to the variable group structure;
- `node` points to the node generating the variable;
- `port` points to the node output port where the variable is generated;
- `gen_stamp` and `cons_stamp` are auxiliary variables used for the register allocation holding the c-step of variable generation and the c-step of variable consumption (by the node executing the latest);
- `cons` is a structure with two pointers to the edges communicating the variable; `list` is the main pointer and `bck` is an auxiliary pointer to perform temporary edge manipulation;

The other structure, `var_group`, is used to store the definition of an input or an output port and is used for register assignment only. However, it is of great

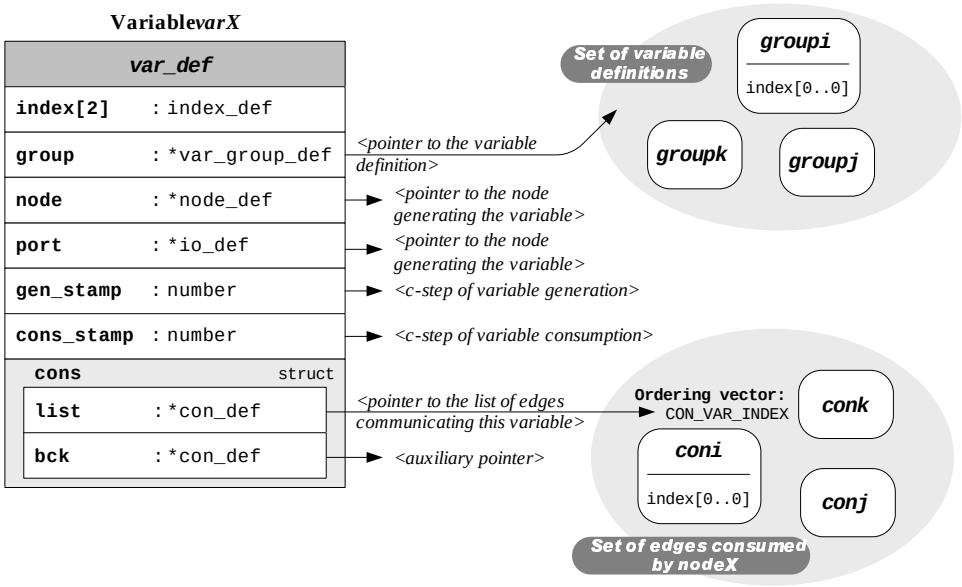


Figure 4.12: Structure of the data type which store variable relative information.

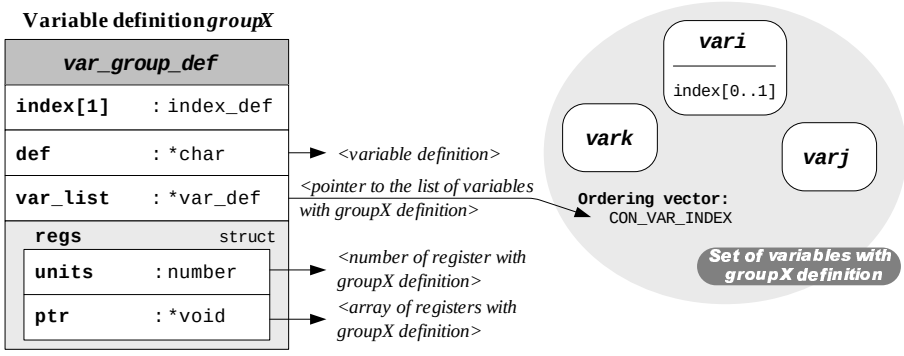


Figure 4.13: Structure of the data type which stores variable definitions.

importance since it allows to detect which variables can be allocated to the same type of registers. It has the following fields

- `def` defines the type of variable;
- `var_list` points to the list of variables of this type;
- `reg` is a structure which includes the number of registers allocated, `units`, and a dynamic array of pointers to the registers sharing the same definition.

4.3 Programming the developed tools

To achieve a set of expandable and re-usable tools, the implementation of the algorithms was made in a modular way. Each of the modules was implemented in a different file and operates accordingly to a predefined interface.

Execution of the ATAS application starts by reading the DFG from an input file. At this point there are no restrictions yet (this is the module which creates the data structures). Program execution continues with the *critical path analysis* algorithm, which takes as input the structure of the DFG. It computes the ES and RS values for each node in the DFG (as described in Algorithm 3.1). At the end of its execution, the module creates two distinct lists that include all the nodes in the DFG. The lists are ordered by ES and RS. The pointers to the next and previous elements in the lists and the ordering values, ES or RS, are stored in the indexing structures `index[NODE_EXECCP_INDEX]` and `index[NODE_REMAINCP_INDEX]`, respectively. After the critical path analysis, the retiming is performed. However, retiming is only executed if the critical path of the DFG is longer than the specified maximum latency (T_{iter}). If this module is required to be executed, the algorithm changes the weights of the node edges. Therefore, upon successful retiming, the ES and RS of all nodes must be recomputed. Following retiming, the scheduling task takes place. To perform this task several modules, each separated into a distinct files were created: one module for implementing the FDS algorithm, another module for implementing the ASLBS algorithm, and 3 more modules for implementing the TCLBS algorithm. The latest three modules store the core of the scheduler, the techniques for choosing the node to be scheduled and the techniques for allocating a node to a c-step. The selection of the scheduling method is made using the command line. Consequently, to avoid tests during the executing of the program, the scheduler is statically identified and a pointer to the function executing scheduling is saved. Moreover, all scheduling algorithms assume that the lists `arch.nodelist[NODE_LIST_BYEXECCP]` and `arch.nodelist[NODE_LIST_BYREMAINCP]` are ascendingly ordered by ES and

RS, respectively. At the end of the execution, the schedulers are required to fill in the `schedulle.n_units` field in the node type structure (see Figure 4.9) with the number of functional units allocated. The scheduler must also fill in the `auxptr[0]` and `auxptr[1]` fields of the node structures (see Figure 4.10), which indicate the c-step and functional unit performing the node computation, respectively. This information is required for the register allocation module, which is executed after performing the scheduling task. The register allocation is therefore responsible for creating the registers and for assigning its inputs, i.e. for filling the `regs` structure on the variable group structures (see Figure 4.13). At the end of execution the ATAS application executes the *datapath binding*. This module creates the control unit and writes the datapath in VHDL, as described in subsection 4.3.2.

The developed algorithms and tools were written in the C programming language, corresponding to more than 9,000 lines of code distributed amongst 44 files (22 source code files and 22 header files).

4.3.1 Speeding up execution

In order to increase the performance of the ATAS tool, several techniques were applied, namely for memory allocation purposes and for reading the input file. After implementing the tool, it was observed that one of the most time consuming tasks was the reading of the input file and the performing of the initial feasibility tests. The problem is that the tool is required to perform a series of operations in order to validate and translate the DFG into the internal description. Namely, for every new node type it needs to check if there is any name duplicates; for every node it needs to check for both duplicated names and needs to find the node type it belongs to; for every edge it requires to find the nodes and ports at both ends of the edge. Moreover, for every new structure created additional memory space must be allocated, thus requiring a great amount of system calls: proportional to $T + N + E$, where T, N and E represent the number of node types, the number of nodes and the number of edges, respectively. To decrease the number of system calls, therefore speeding up the execution of the program, a memory management system was designed. It operates by allocating large memory banks (4kB) and then by attributing the space as it is required, as shown in Figure 4.14. Three pointers are used to keep track of the allocated space: `base`, `ptr` and `next` which point to the first memory bank, the last memory bank and the next memory location, respectively. In order to keep track of all the allocations, the first element in each bank always points to the next memory bank, thus allowing to free all the allocated space if necessary. Using this system, when memory is required, the first step is to check if the current memory

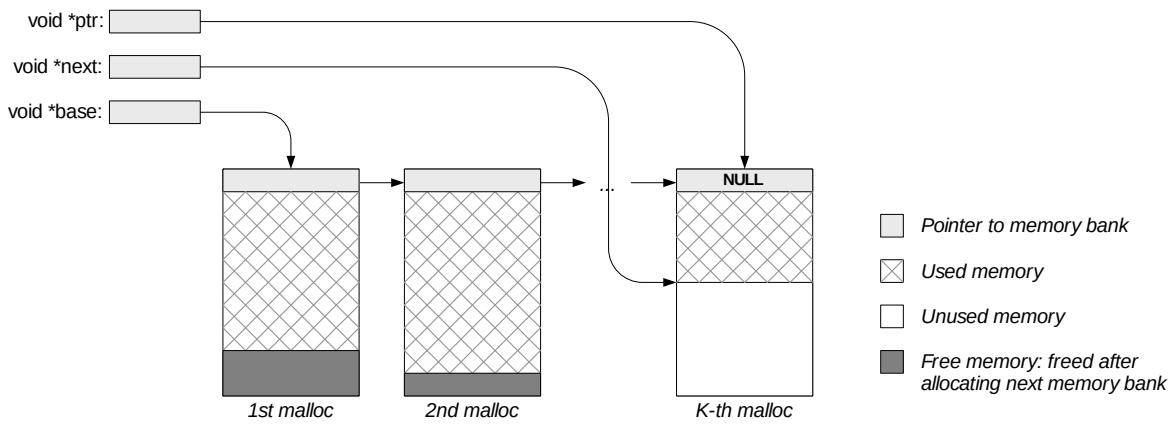


Figure 4.14: Memory allocation technique.

bank k as enough free space for the request. If it does not, the following 3 steps are performed: *i)* freeing the remaining space in bank k; *ii)* allocating of a new bank $k+1$; *iii)* update all required pointers, namely the first position of banks k and $k+1$ will point to bank $k+1$ and `NULL`, respectively, `ptr` will point to bank $k+1$, and `next` will point to `ptr + sizeof(void*)`. After the initial check, the `allocate_memory` function assigns the caller the memory space pointed by `next` and increases the pointer by `size`.

It is important to note that, while this method allows faster execution times, it has a drawback: memory cannot be released in parts, only as a whole. Thus this memory allocation scheme should be used only when a large chunk of memory of unknown size will be required. Examples of this case are the reading of the DFG from file and during retiming to allocate space for critical path constraints (see Section 3.2).

Experimental results show that for reading a random DFG of 10,000 nodes from file, this memory management system allowed a reduction in the execution time of about 66%. However, this scheme proved not to be enough as the CPU execution time for reading DFGs of this dimension was still around 50 s. The problem was related to the high number of string comparisons required to be performed, namely for:

1. validation of node types (node type name must be unique), requires $\frac{T^2}{4}$ comparisons;

2. validation of nodes (node name must be unique), requires $\frac{N^2}{4}$ comparisons;

3. creating of a new node requires to search for node type, thus corresponding to $\frac{N \times T}{2}$ comparisons on average;

4. creating edges requires to search for the origin and destination node of the edge, translating in $2 \times N \times E$ comparisons.

In order to decrease the number of string comparisons, it was decided to implement a temporary hashtable during the reading of the file. One difficulty is however detected when the hash table is created: one cannot know how large it needs to be. To overcome this problem a dynamic hash table was implemented.

The implementation of the hash function followed the work made by Jenkins [25] and Hsieh [26]. Jenkins first developed work to create hash functions which could be able to encode strings for tables of any size. The problem is that creating the usual prime-sized hashtable requires addressing the table with the remainder of the division by a prime number. This is not only a very slow operation, as it does not fully avoids collisions. Experimental evaluation made by both Jenkins and Hsieh showed that the `SuperFastHash` function developed by the latest [26] outperforms all other hashing functions, both in terms of speed as in terms of distribution. This function maps a byte array of variable dimension into a 32-bit integer.

The implementation of the hashtable was made by means of an array with the table size. Each element in the array points to a list of nodes/node types which are hashed by the same number. The array is initially created with size $S = 1024$ and grows as the number of stored elements grows. This is achieved by creating a load factor: if the number of elements in the hash table grows beyond 75% of the array size, S, the table is completely re-hashed into a new one twice as big ($S' = 2 \times S$). Nevertheless this does not involve re-computing the hashing values as those are stored with the elements. While this procedure uses more memory, it allows to decrease execution time when increasing the size of the hashtable. Since the Hsieh `SuperFastHash` function has a uniform distribution, translation from the 32-bit integer into a K-bit integer ($K = log_2(S)$) can be made by simply extracting the K less significant bits from the 32-bit key.

Experimental evaluation of the hash table showed a reduction on CPU execution time of 93% in the DFG reading module.

4.3.2 Data path binding: VHDL code generation

After performing node scheduling and register allocation, the last step for the ATAS tool is to create the multiplexers and to generate the circuits datapath, i.e. to connect registers to functional units to multiplexers. Also, a unit must be described for controlling the communication of the data registers and functional units.

A simplification of the generated data path is shown in Figure 4.15. For each in-

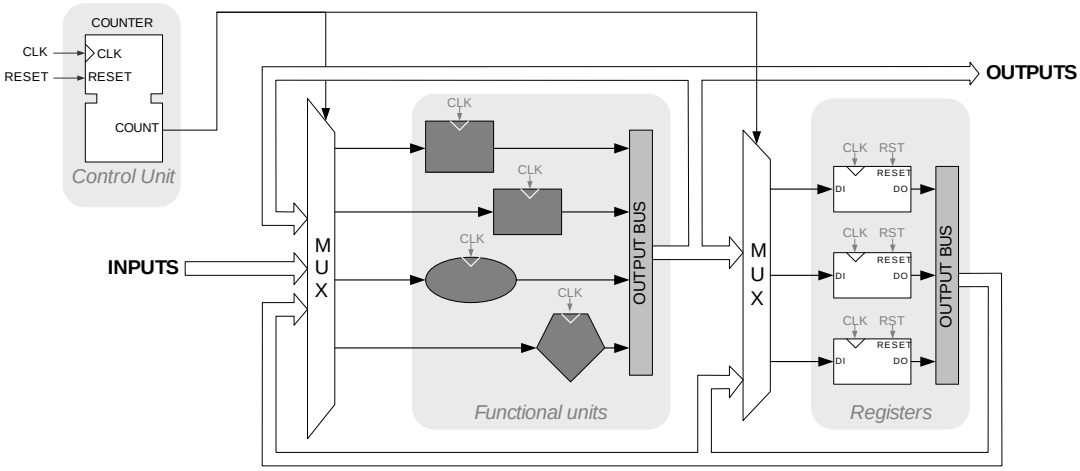


Figure 4.15: Structure of the generated VHDL code.

put port of each functional unit allocated during the scheduling phase, a multiplexer is created. For each register created during register allocation, a multiplexer is also generated. Each multiplexer has up to T_{iter} inputs and 1 output and is controlled by a counter. If a functional unit/register is not allocated for execution/store at a given c-step x, then the multiplexer input at that c-step is not defined, therefore simplifying logic. The control unit holds the information about the current c-step. It is implemented using a single counter which cyclicly counts from 0 to $T_{iter} - 1$.

After the development stage of the ATAS tool, the implemented algorithms were tested by synthesising several DSP algorithms. The VHDL code generated by the tool was then mapped into a Xilinx XC3S400 FPGA [27] device using Xilinx ISE 7 [28]. Tests showed that the generated VHDL code was not only error-free but also warning-free (considering that the descriptions for the functional units does not have errors or warnings). The code was then simulated in ModelSim Xilinx Edition III [29], where the behaviour of the synthesised architecture was verified and proved to work accordingly to the description of the DSP algorithm.

4.4 Conclusions

This chapter described the flow of execution of the ATAS tools and detailed the most important aspects of its implementation, namely the format of the file user to read the DFG, the data structures and the characteristics of the implementation which allow to significantly decrease the execution time. The speed of execution was really one of the most important aspects taken into consideration. With this purpose, it was also designed a specific memory management system and hashtables were implemented allowing to decrease the CPU execution time on about 66% and

93%, respectively. The flexibility was also a parameter taken into consideration. Thus, all lists are circularly doubly-linked. This allowed for the use of the same functions for sorting, accessing or inserting independently on whether it is a list of nodes, of node types, of variables or of edges. This was achieved by creating an `index` structure which is the basis for all the sorting procedures.

5

Experimental Results

Contents

5.1 Evaluating Benchmarks 72
5.2 Performance of the Algorithms 77
5.3 Comparative evaluation with optimal scheduling 92
5.4 Improving scheduling results 93
5.5 Conclusions 94

5.1 Evaluating Benchmarks

To evaluate the performance of the implemented algorithms for solving large problems, well-known digital signal processing algorithms were described in the DFG form (see Section 4.1), namely: *Artificial Neural Networks* (ANNs) [30] and *Fast Fourier Transforms* (FFTs) [22]. However, since the intention was not to bias the results for a particular type of DSP problem, a random network generator was developed in the C programming language, as described in subsection 5.1.3. While this set of benchmarks (the Artificial Neural Networks (ANNs), the Fast Fourier Transforms (FFTs) and the random networks) allows for the validation of algorithms for large scale problems, they are not feasible for comparison with optimal solutions, such as those obtained with ILP, because they are too large. To compare the proposed algorithms with optimal computational intensive approaches, two typical scheduling benchmarks, defined in the 1992 High-Level Synthesis Conference, were used: a 5th order Elliptical Wave Filter (EWF) [31] and an 8 point 1-D Discrete Cosine Transform (DCT) [32], as presented in subsection 5.1.4.

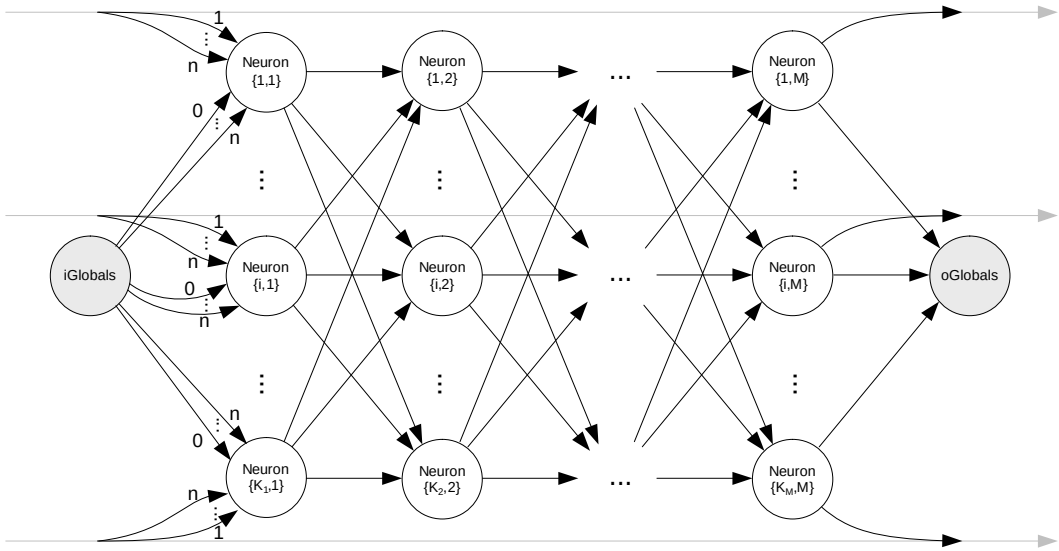
5.1.1 Artificial Neural Network

The ANNs were designed with a multiple layer topology $K_1 \times K_2 \times \cdots \times K_M$, where K_i represents the number of cells in layer i, as shown in Figure 5.1(a).

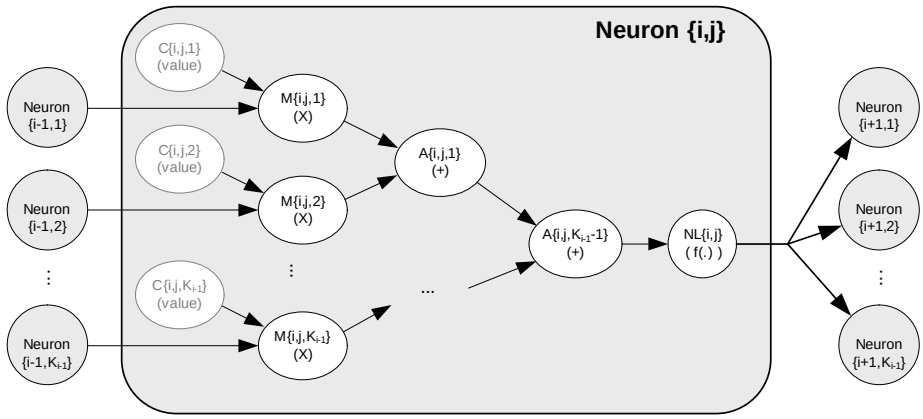
Each ANN Neuron in layer i, $NEURON_{\{i,j\}}$ ($i \in [2; M]$ and $j \in [1; K_M]$), receives K_{i-1} inputs, corresponding to all the outputs of layer $i-1$, and performs the computation shown in Figure 5.1(b). Firstly, it weights all the inputs by multiplying them with a constant coefficient $C_{i,j}$. Afterwards, it computes the sum of all products and applies a sigmoid function, $f(\cdot)$, to the sum. The description of all generated ANNs considers an optimised DFG using basic components. In this sense, there are only three types of operations: 2-value multiplications, 2-value additions and sigmoids. For performing the sum of products, a tree of adders is built such as to minimise the DFG critical path. The sigmoids are considered to be computational expensive operations and thus are implemented by means of look-up tables. The ANN has one input, $x[n]$, and K_M outputs, $y_1[n] \cdots y_{K_M}[n]$. However, it is considered to have time information: its output value depends on the latest L inputs. Thus, the inputs to the first layer are in fact:

$$x[n], \cdots, x[n-L] \tag{5.1}$$

The ANN benchmarks do not include any of the typical learning structures, i.e. the computational blocks which serve to adapt the ANN weights $C_{\{i,j\}}$. It is assumed



(a) Global Data-flow graph.



(b) Data-flow graph of each neural cell.

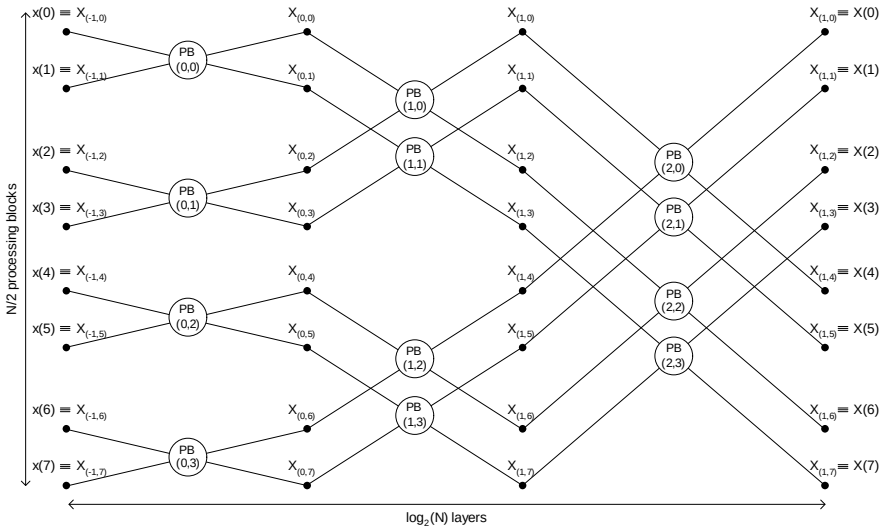
Figure 5.1: The structure of the Artificial Neural Networks.

that the intention is to implement the ANN after the learning stage.

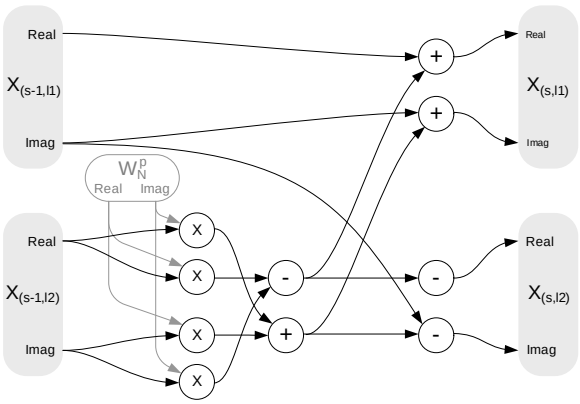
5.1.2 Fast Fourier Transform

The Fast Fourier Transform herein presented is computed by using the decimation-in-time Coley and Tukey algorithm [22], which is able to reduce the amount of required computations by $N^2 - \frac{N}{2} \times log_2 N$. By using this method, the generated DFG will have a butterfly appearance as shown in Figure 5.2(a).

The FFT of a N point vector is represented by a DFG with $M = log_2 N$ stages, each with $N/2$ identical Processing Blocks (PBs). Each processing block $k \in [0; N/2 - 1]$ at stage $s \in [0; M - 1]$, $PB_{(s,k)}$, takes two complex values, $\mathbf{X}_{(s-1,l_1)}$



(a) Global Data-flow graph of an 8-point Fast Fourier Transform.



(b) Data-flow graph of each FFT processing block.

Figure 5.2: Structure of a Fast Fourier Transform.

and $\mathbf{X}_{(s-1,l_2)}$, computed in stage $s-1$, to compute the two two values $\mathbf{X}_{(s,l_1)}$ and $\mathbf{X}_{(s,l_2)}$

$$\begin{cases} \mathbf{X}_{(s,l_1)} = \mathbf{X}_{(s-1,l_1)} + \mathbf{X}_{(s-1,l_2)} \times \mathbf{W}_{(s,l2)} \\ \mathbf{X}_{(s,l_2)} = \mathbf{X}_{(s-1,l_1)} - \mathbf{X}_{(s-1,l_2)} \times \mathbf{W}_{(s,l2)} \end{cases} \tag{5.2}$$

where $\mathbf{W}_{(s,l2)}$ is a constant value,

$$p = (k \,\%\, 2^s) \cdot 2^{M-s-1} \tag{5.3}$$

and

$$\begin{cases} l_1 = (k \,\%\, 2^s) + \lfloor k \,/\, 2^s \rfloor \times 2^{s+1} \\ l_2 = (k \,\%\, 2^s) + \lfloor k \,/\, 2^s \rfloor \times 2^{s+1} + 2^s \end{cases} \tag{5.4}$$

However, this computation is made in the complex domain. Thus $\mathbf{X}$ and $\mathbf{W}$ assume a vector form $\mathbf{Z}$, where

$$\mathbf{Z} = \begin{bmatrix} \mathbb{R}(Z) \\ \mathbb{I}(Z) \end{bmatrix} \tag{5.5}$$

and $\mathbb{R}(Z)$ and $\mathbb{I}(Z)$ represent the real and imaginary parts of Z, respectively. Consequently it is possible to re-write (5.2) as

$$\begin{cases} \mathbf{X}_{(s,l_1)} = \mathbf{X}_{(s-1,l_1)} + \widehat{\mathbf{X}}_{(s-1,l_2)} \times \mathbf{W}_{(s,l2)} \\ \mathbf{X}_{(s,l_2)} = \mathbf{X}_{(s-1,l_1)} - \widehat{\mathbf{X}}_{(s-1,l_2)} \times \mathbf{W}_{(s,l2)} \end{cases} \tag{5.6}$$

where

$$\widehat{\mathbf{X}} = \begin{bmatrix} \mathbb{R}(X) & -\mathbb{I}(X) \\ \mathbb{I}(X) & \mathbb{R}(X) \end{bmatrix} \tag{5.7}$$

The detailed DFG of each processing block is illustrated in Figure 5.2(b). The design of the global DFG for a complex N-point FFT is made by using only two types of operations: additions/subtractions and multiplications. For a N point FFT there will be $N/2 \times log_2 N$ PBs, each with 4 multiplications and 6 additions/subtractions.

5.1.3 Random Networks

In order to analyse the performance of the developed algorithms for a general problem, a program was developed to generate random networks based on a uniform distribution. This program, written in the C programming language, takes as arguments the number of different node types to create and the total number of computational nodes.

The program starts by generating the required node types, where, by using a uniform distribution, it assigns each type a random execution time between 1 and 10 c-steps. Each node type will be assigned with a random number of input ports varying from 1 to 3. Following, the program iteratively creates the N nodes, and assigns each with a unique number $\mathtt{n}_{\mathsf{U}}$ from 1 to N. Nodes are created sequentially from 1 to N and are associated with a given type by using a uniform distribution. Immediately after selecting the type $\mathcal{U}$ for the node U, the edges leading to U are created. The assignment of edges is made in two steps. In the first step, the program randomly chooses the weight for the edge $\langle \mathsf{V} \rightarrow \mathsf{U} \rangle$, by using the probability distribution shown in Figure 5.3. This distribution was constructed such that the majority of nodes is connected with a zero weight edge. In the second step, it randomly selects a node V to generate the variable transmitted by the edge. This node is randomly chosen from a set of nodes depending on the pre-assigned edge

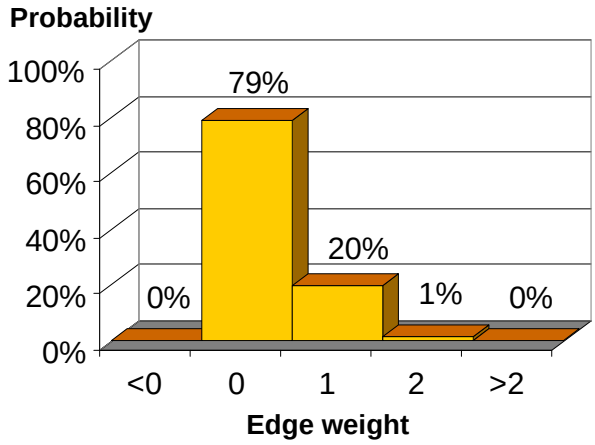


Figure 5.3: Probability distribution of edge weights.

weight. If the edge weight is greater than zero, then all nodes have equal probability to be the edge source. However, if the edge weight is zero one must guarantee that no zero weight closed paths are generated. This is guaranteed by selecting a node from the following set: *i)* the global input node, internally named `iGlobals` and *ii)* any node previously allocated (its unique number must be in range $[1; \mathtt{n_U} - 1]$). During this stage, edges are created for transmitting a variable to each of the node input ports. The generated DFG has only 1 input value. Nevertheless it may have more than one output value.

After the generation of the DFG, the program checks which nodes do not have output edges. Following, it creates zero weight edges from these nodes to the architecture output, internally named `oGlobals`. The last node, the one with the unique number N, will always be connected to `oGlobals`. Note that the number of architecture inputs/outpus does not influence in any way the behaviour of the algorithms (with the exception of the binding algorithm which generates the VHDL code).

For the generation of the random networks it is assumed that all variables are of the same type, thus only one type of registers are required for the generation of the VHDL description.

5.1.4 Typical Scheduling Benchmarks

As research intensified in the areas of HLS, specifically on scheduling algorithms, it became important to define a set of benchmarks to evaluate the proposed techniques and algorithms. As a result, in the 1992 International Conference of High-Level Synthesis, a set of benchmarks were proposed for this task, namely an Elliptical Wave Filter (EWF) and an 8-point Discrete Cosine Transform (DCT).

The EWF is defined with a 5-th order IIR structure optimised to reduce the number of operations. The DFG of the filter, shown in Figure 4.2, is described using 26 additions and 8 multiplications, totalling 34 operations.

The DCT is a traditional technique used in image processing for lossy data compression. The implementation of the 8-point DCT in this benchmark is made by performing a total of 44 operations: 32 additions and 12 multiplications.

5.2 Performance of the Algorithms

In order to evaluate the performance of the developed algorithms, and to analyse how computational time scales with the increase in the number of nodes and in the maximum latency, experimental results were obtained using the ANN, the FFT and the random networks benchmarks. The first two benchmarks were used to evaluate the algorithms for solving DSP problems. By using the random benchmarks it is possible to compare the results of the algorithms for the general case, not biasing the results for a regular problem, which is the case of a DSP. It is expected that the translation of a complete algorithm, programmed in a high level language, to a graph to be more similar to a random network, with multiple and irregular dependencies, than with the ANN or FFT benchmarks, which are highly regular.

For the ANN benchmark, it was used a topology with two cells per layer. The ANN input signals are $x[n]$ and its delayed version $x[n-1]$, and the output signals are $y_1[n]$ and $y_2[n]$. Each layer has eight computational nodes (4 multipliers, 2 adders and 2 sigmoids) and four zero-weight nodes generating constant values. The characteristics of the seven benchmarks used to evaluate the computational time of each algorithm are presented in Table 5.1.

The functional unit execution time considered was: 1 c-step for adders, 2 c-steps for multipliers and 1 c-step for sigmoids (it is considered that the implementation of the sigmoids is based on a look-up table). Since the performance of the HLS algorithms depends on the structure of the DFG, the evaluation using the random benchmarks was based on the average results amongst 500 networks. Also, since for these benchmarks the critical path varies from network to network, the maximum latency was defined as a percentage, α, of the computational weight of the longest path. Thus for benchmark R1 α was set to 1 while for benchmark R2 α was varied from 1 to 20.

To analyse the performance of the algorithms, graphs will be shown presenting both the average execution time and the number of completed instructions[1] to ex-

[1]Instructions committed at the Control Data Bus (CDB) [33].

Table 5.1: Benchmarks used to evaluate the computational time.

Benchmark: ANN	**#Layers** (1)	**#Nets** (2)	#Nodes* (3) = 8 · (1) · (2)	Critical path (4) = 4 · (1)	**Latency** (T_{iter}) (5)
A1	**3–1501**	1	**24–12008**	**12–6004**	4
A2	501	1	4008	2004	**2–2000**
A3	10	**2–1024**	**160–81920**	40	60†
A4	10	50	4000	40	**80–1200**

Benchmark: FFT	**FFT size**‡ (1)	*#Inputs* (2) = $2^{(1)}$	*#Nodes** (3) = 10 · (1) · (2)	*Critical path* (4) = 4 · (1)	**Latency** (T_{iter}) (5)
F1	**1–10**	2–1024	**20–102400**	4–40	6–60†
F2	**5**	32	1600	20	**40 – 1500**

Benchmark: Random networks	**#Types** (1)	**#Nodes** (2)	*#Networks* (3)	*Critical path* (4)	**Latency** (T_{iter}) (5)
R1	3	10–2000	500	*variable*	(4)
R2	3	50	500	*variable*	(4) × [1; · · · ; 20]

* *Excluding constant nodes.*
† *Value kept proportional to the critical path:* (5) = (4) ∗ 1.5.
‡ *A size of x corresponds to a 2^x point FFT.*
Values in bold indicate the parameter being varied.

ecute the algorithm, over five iterations (except for the random networks, which results are already random). The presented values were obtained by using the Performance Application Programming Interface (PAPI) [34] tool to access the processor performance counters. While temporal results are highly dependent on the machine (clock frequency, branch prediction, cache organisation, ...) the number of completed instructions allows to measure the performance of the algorithms in any machine using the Intel Architecture IA-32 [35].

The experimental evaluation was made in a personal computer based on a AMD K7 processor running at 1.6 GHz and the SuSE 9.3 operating system. The details on the machine are presented in Table 5.2.

5.2.1 Reading the input file

The evaluation of the CPU execution time for reading the DFG is performed only by varying the number of nodes, using benchmark A1. Results for latency variations are not presented since this parameter does not influence in any way the

Table 5.2: Description of the Personal Computer used to assess the performance of the ATAS application algorithms.

CPU	AMD K7 2000+
Frequency	1.611 GHz
Instruction TLB	256 entries, 4-way associative
Data TLB	256 entries, 4-way associative
L1 instruction Cache	Write through policy, 64 KB, 2-way associative, 64 B lines
L1 data cache	Write back Pseudo LRU policy, 64 KB, 2-way associative, 64 B lines
L2 cache	Write through Pseudo LRU policy, 256 KB, 16-way associative, 64 B lines
Main Memory	512 MB

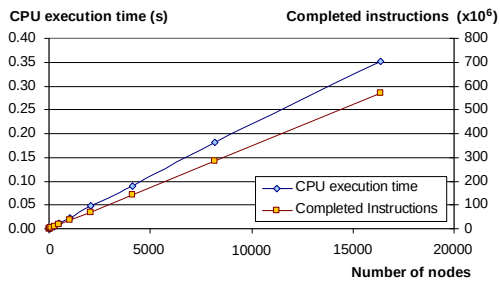


Figure 5.4: Performance of the algorithm to read the DFG (benchmark A1).

computational time.

By analysing the plots in Figure 5.4, it can be concluded that the algorithm is able to read DFGs with a considerably large number of nodes within a reasonable amount of time: it takes about 0.35s to read a DFG with 16,000 nodes. Also, it is possible to observe that the time grows linearly with the number of nodes (the algorithm takes approximately 21.4μs to read each node). Nevertheless, it must be noticed that the computational time is also highly dependent on the number of edges. Assuming that the number of edges per node is limited to a small value, as it is the case with traditional computational units like adders or multipliers, one can directly relate the computational time with the number of nodes and the structure of the DFG.

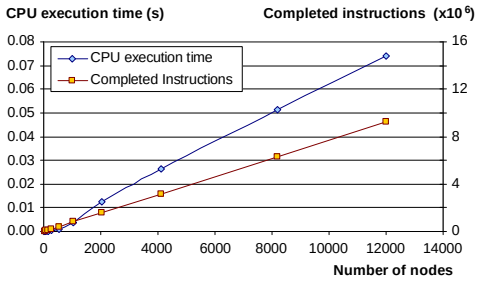


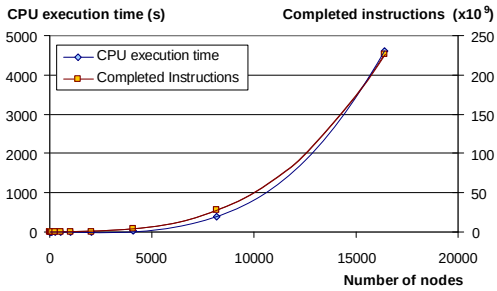
Figure 5.5: Performance of the algorithm to analyse the critical path (benchmark A1).

5.2.2 Critical Path Analysis

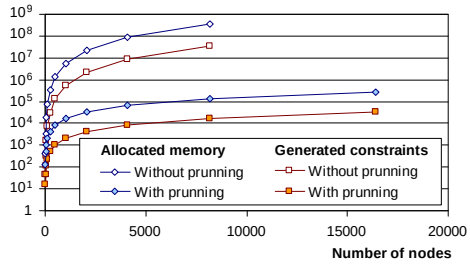
As it was seen in Section 3.1, the complexity of the algorithm that computes the critical path depends only on the number of nodes and on the structure of the DFG. Thus, by using benchmark A1, it is possible to evaluate how the algorithm computational time scales with the increase in the number of nodes. The obtained results are presented in Figure 5.5. As can be seen in the figure, the time to analyse the paths in a graph grows linearly with the number of nodes. It takes approximately 74 ms to analyse the DFG of an ANN with 16,384 nodes, scaling to about 594 ms for an ANN with 81,920 nodes (result taken from benchmark A3). Note that the computational time has a worst-case quadratic dependency with the number of nodes: the algorithm uses an insertion sort algorithm to create an ordered list of nodes. However, as was explained in Section 3.1, in the average case all the nodes are inserted on the tail of the ES and RS lists. However the tool uses circular doubly-linked lists which allows for the insertion sort algorithm to try to insert the nodes at the last position. Therefore, in the average case, the complexity of the critical path analysis algorithm is $O(N)$. This is the case presented in Figure 5.5.

5.2.3 Retiming

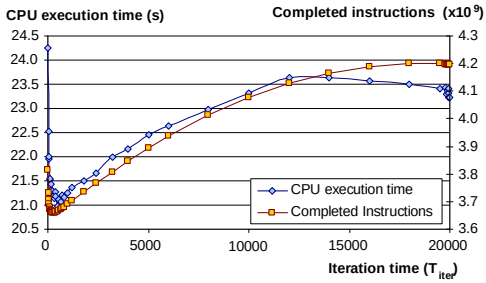
To analyse the performance of the retiming algorithm, benchmarks A1 and A2 were used, to assess the time to execute the algorithm for a different number of nodes and different iteration times, respectively. Results are shown in Figure 5.6. Specifically, in Figure 5.6(b) and Figure 5.6(d), results are presented with the number of generated constraints and the amount of memory allocated in the original retiming algorithm, and by applying constraint pruning. As can be observed, the performance of the retiming algorithm was a greatly improvement by applying constraint pruning: the new algorithm generates only 16,368 constraints for an ANN with 8,192 nodes in contrast with the 33 milion constraints generated by the old algorithm. In fact, the experimental results show that by applying constraint prun-



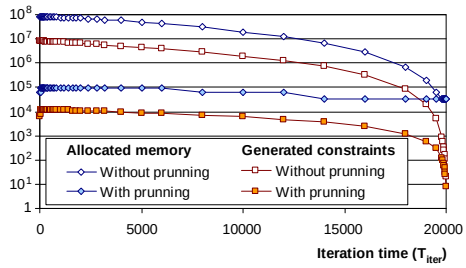
(a) Performance *vs* number of nodes (benchmark A1).



(b) Constraints *vs* number of nodes (benchmark A1).



(c) Performance *vs* iteration time (benchmark A2).



(d) Constraints and memory allocation *vs* iteration time (benchmark A2).

Figure 5.6: Performance of the retiming algorithm.

ing the number of constraints rises linearly with the number of nodes, instead of exhibiting a quadratic behaviour. This also has an effect on the amount of memory allocated by the retiming algorithm and on the execution time: while the old algorithm required approximately 330 MB of memory and took over 1800 s to process an ANN with 8,192 nodes, the new version only required 128 KB of memory and the CPU execution time was less than 390 s.

When varying the iteration time, the advantages of the constraint pruning technique are also observed. For the best case, $T_{iter} = 2$ c-steps, the original algorithm generated about eight million constraints. However, by applying constraint pruning, this value is reduced to only six thousand. This leads to a great reduction in both the allocated memory (from 79 MB to 63 kB) and the execution time (from 190 s to 24 s). However, as the iteration time increases from 2 c-steps to the computation time of the critical path, the number of generated constraints is reduced. Therefore the effectiveness of the constraint pruning is reduced: for $T_{iter} = 1998$ c-steps, the reduction in the number of constraints is still about 60%.

The minimum execution time depicted in Figure 5.6(c) has to do with a balance between the increase and decrease in execution time of 2 cycles executing serially:

the generation of the ILP constraints and the solving of these inequations. On the one hand, as the iteration time is reduced from $T_{iter} = 1998$ c-steps to $T_{iter} = 6$ c-steps the number of constraints increases continually. As this happens the constraint pruning process lowers the number of shortest-path searches, thus reducing the CPU execution time. On the other hand, since more constraints are being generated, the ILP solving algorithm takes increasingly more time. With $T_{iter} < 40$ c-steps, the increase in the computational time for the ILP solving algorithm overcomes the decrease in computational time for the constraint generation algorithm. As a result, the overall execution time increases.

5.2.4 Time Constrained List Based Scheduling algorithms

Before evaluating how the scheduling algorithms scale with time, it is important to analyse the number of units allocated for each algorithm: an algorithm which simply allocates too many functional units (resulting in a folding factor of one[2]) will always be bad, it does not matter how fast it runs. To achieve this goal, performance analysis starts by analysing the scheduling results of the TCLBS algorithm using the several techniques for choosing the next node to be scheduled and for choosing the c-step the node is allocated (see Section 3.3). To perform this analysis, benchmark A3 was used. The obtained results are partially presented in Figure 5.7. For the `FORCE` techniques only node dependency level of $n = 1$ is presented (results varying n will be shown later in this subsection).

Analysing the scheduling results, the first conclusion that can be drawn is that there are node choosing techniques that simply cannot be applied in conjugation with some c-step choosing methods: by attempting to minimise the number of resulting functional units in opposite directions they nullify their effects resulting in a bad scheduling algorithm. To illustrate this effect, let us consider that `MAX_ES` is used for choosing the next node to schedule and `AVAIL_FIRST` is used for selecting the c-step to perform the operation. The first technique, `MAX_ES`, will always select the nodes at the end of a critical path to be scheduled first: for a zero weight path $\mathsf{A} \rightarrow \mathsf{B} \rightarrow \mathsf{C}$, the order by which nodes are scheduled is $\mathsf{C}, \mathsf{B}, \mathsf{A}$. Consider that no more dependencies exist in the DFG for the scheduling of nodes A, B and C besides the edges $\langle \mathsf{A} \rightarrow \mathsf{B} \rangle$ and $\langle \mathsf{B} \rightarrow \mathsf{C} \rangle$. For a maximum iteration time of 5 c-steps and an execution time for the functional units performing operations A, B and C of 1

[2]A folding factor of one implies that one functional unit will be used for each operation (node) in the DFG.

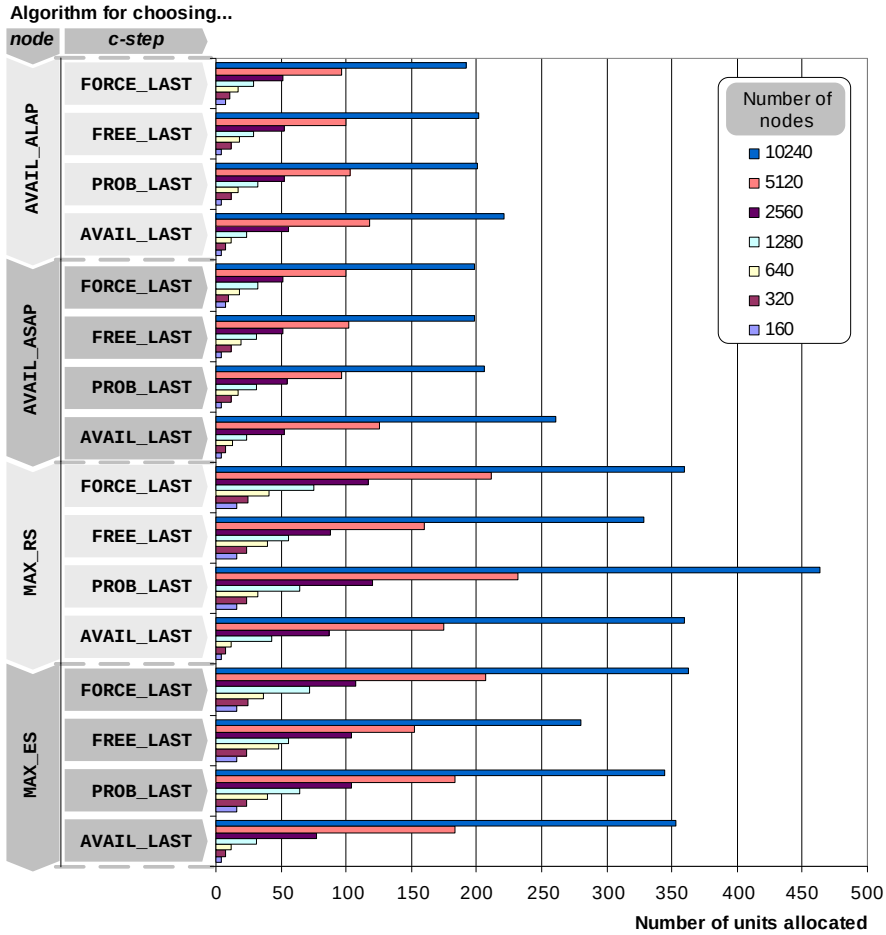


Figure 5.7: Performance of the TCLBS algorithms (benchmark A3).

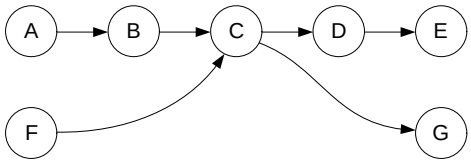
c-step, the mobility for each node is:

$$\left[\ S^{\mathsf{A}}_{min}\ ;\ S^{\mathsf{A}}_{max}\ \right] = [\ 0\ ;\ 2\ ] \tag{5.8a}$$

$$\left[\ S^{\mathsf{B}}_{min}\ ;\ S^{\mathsf{B}}_{max}\ \right] = [\ 1\ ;\ 3\ ] \tag{5.8b}$$

$$\left[\ S^{\mathsf{C}}_{min}\ ;\ S^{\mathsf{C}}_{max}\ \right] = [\ 2\ ;\ 4\ ] \tag{5.8c}$$

When choosing a c-step to execute C (the first assigned by `AVAIL_FIRST`), intuitively one should allocate it to c-step 4, to reduce the mobility of all other nodes as least as possible. However, the `AVAIL_FIRST` will always try to schedule it to the first possible c-step, i.e. c-step 2. Using similar considerations, one should not use any `*_FIRST` (`AVAIL_FIRST`, `PROB_FIRST`, `FREE_FIRST` and `FORCE_FIRST`) technique with a choosing method that gives priority to the nodes at the end of the processing path: `MAX_ES`, `MIN_RS` and `AVAIL_ALAP`. An identical restriction can also be set for c-step allocation techniques of type `*_LAST` (`AVAIL_LAST`, `PROB_LAST`, `FREE_LAST` and `FORCE_LAST`). These should not be used together with the node choosing methods that privilege nodes at the beginning of the processing path: `MIN_ES`, `MAX_RS` and



(a) Structure of the DFG.

Node	A	B	C	D	E	F	G
Execution Steps (ES)	1	2	3	4	5	1	4
Remaining Steps (RS)	4	3	2	1	0	3	0

(b) Execution and Remaining steps: ES and RS.

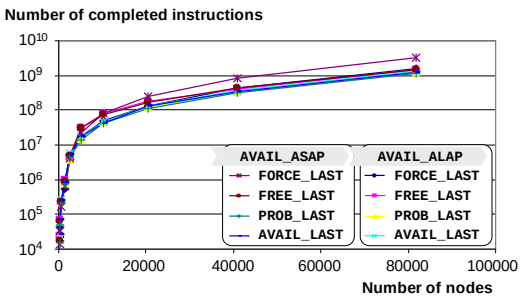
Figure 5.8: Example DFG.

`AVAIL_ASAP`.

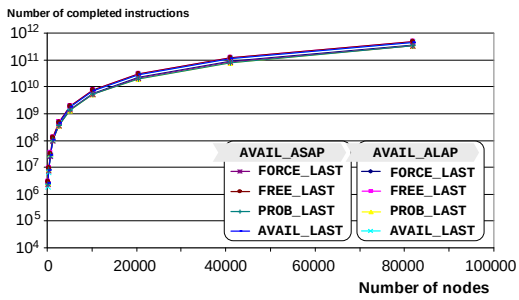
On the other hand, other techniques should be eliminated: `MIN_ES` and `MIN_RS`. To understand the reason for the elimination of these techniques consider the DFG of Figure 5.8. Assuming that the computational weight of all nodes is the same and equal to one c-step, the value of the ES and RS paths are those shown in Figure 5.8(b). The scheduling of this DFG should privilege the nodes in the critical path: $\mathsf{A} \rightsquigarrow \mathsf{E}$. However, by using the `MIN_ES` method, nodes A and F have the same priority. Therefore, it may lead to an erroneous choice on the node to schedule. When using the `MAX_RS` method, a similar situation will occur with this DFG: since node G has the same RS value as node E, it can be scheduled before node G, which may lead to a possible increase in the number of functional units allocated.

The results presented in Figure 5.7, evidence that the `AVAIL_ASAP` and `AVAIL_ALAP` algorithms are always better than the `MAX_ES` and `MAX_RS` algorithms. This is due to the fact that the `MAX_ES` and `MAX_RS` techniques do not consider the number of available functional units for selecting the node to schedule. The `AVAIL_ASAP` and `AVAIL_ALAP` algorithms on the other hand will always privilege the nodes with the least number of functional units available in its scheduling bounds, either because the nodes are in the critical path or because the scheduling of any other node somehow restricts the scheduling of the nodes.

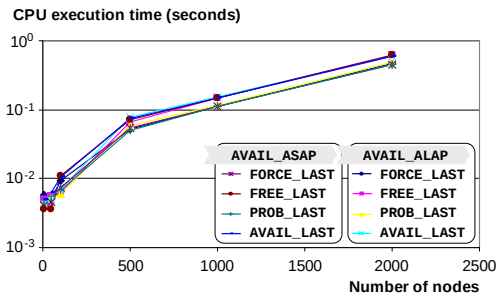
As seen in Section 3.3, the complexity of the TCLBS algorithm does not depend on the node choosing technique if using the `AVAIL_ASAP` or `AVAIL_ALAP` as node choosing techniques. Experiments have also shown this result. As can be observed in Figure 5.9, the time to execute the TCLBS algorithm grows similarly for all the c-step choosing techniques (there is only a small difference in the scaling factor). This behaviour can also be observed when comparing the results of the TCLBS for



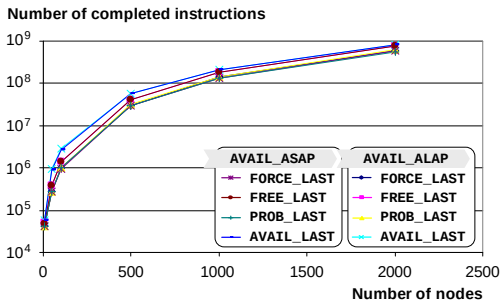
(a) Time *vs* number of nodes (benchmark A3).



(b) Instructions *vs* number of nodes (Benchmark A3).



(c) Time *vs* number of nodes (benchmark R1).



(d) Instructions *vs* number of nodes (benchmark R1).

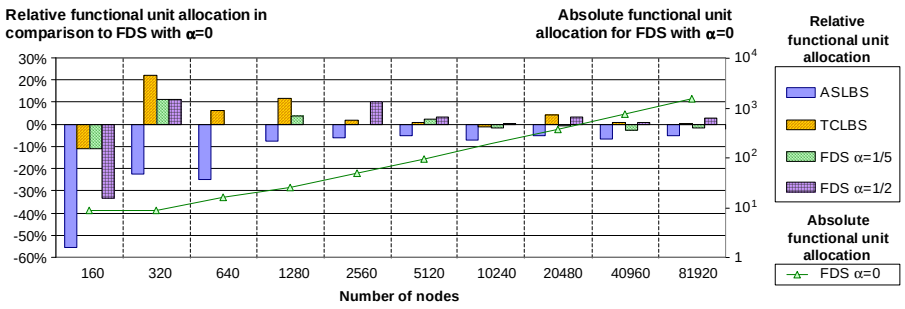
Figure 5.9: Computational time of the TCLBS algorithms.

benchmark R1, as shown in Figure 5.9.

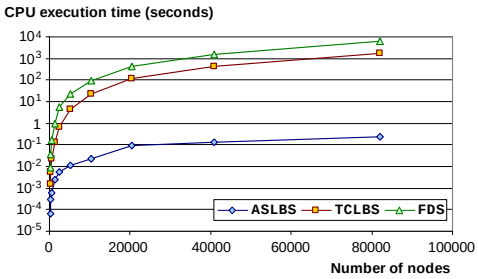
So far all the analysis for the `FORCE` c-step choosing technique has been made using a probability dependence level $n = 1$. Therefore, the question is now: should the results improve if n increases and thus more nodes are considered for computing the probability? The problem is that the computation of the probability value is made using approximations which discard many dependencies. Consequently the results obtained by increasing n do not significantly improve: results for $n = 2$ show a small increase in the performance. Nevertheless, the computational time also significantly increases due to the complexity of the algorithm growing exponentially with n.

5.2.5 Comparative analysis of the scheduling algorithms

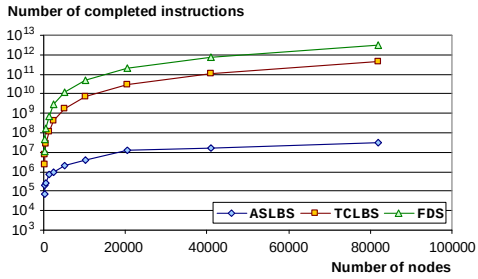
After analysing the results for the TCLBS algorithm, using the various proposed node and c-step choosing techniques, a comparative evaluation of the different scheduling algorithms, namely the TCLBS, the ASLBS and the FDS, was performed. For the TCLBS, only the results for the `AVAIL_ALAP/FORCE_LAST` tech-



(a) Allocation functional units *vs* number of nodes.



(b) Time *vs* number of nodes.



(c) Instructions *vs* number of nodes.

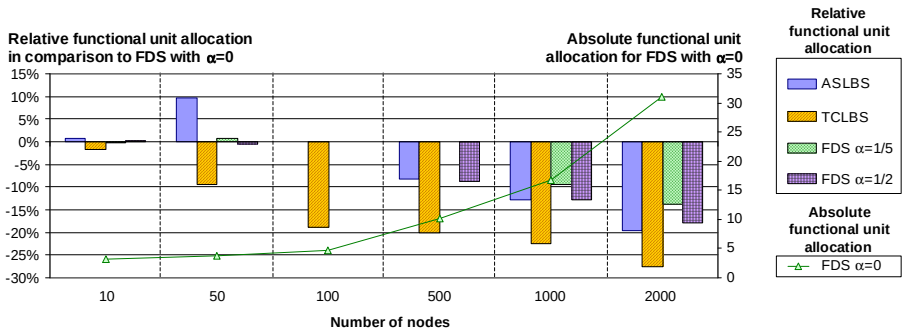
Figure 5.10: Performance of the scheduling algorithms (Benchmark A3).

niques are presented, since this pair is the one leading to the least number of allocated units and the CPU execution time does not significatively change with the c-step choosing technique. The presented results for the FDS algorithm use a mobility reduction factor of $\alpha = 0$ (the original algorithm), $\alpha = \frac{1}{5}$ and $\alpha = \frac{1}{2}$. As it will be seen later, although the results may improve using higher mobility reduction factors, $0 < \alpha < 1$, the CPU execution time also increases. Figure 5.10 and Figure 5.11 present the results for benchmarks A3 and R1. Results for the FFT benchmarks are presented in Appendix B. The allocation of functional units is compared with the one obtained by using the original FDS algorithm ($\alpha = 0$). Thus, the relative functional unit allocation for an algorithm `ALG` is computed as

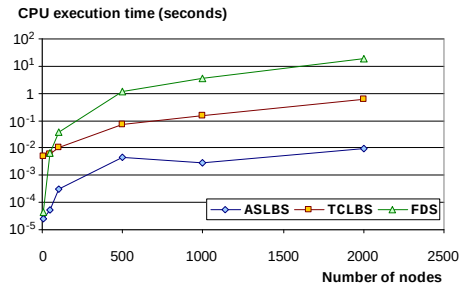
$$M_{\texttt{ALG}}/M_{\texttt{FDS}} - 1 \tag{5.9}$$

where $M_{\texttt{ALG}}$ and $M_{\texttt{FDS}}$ represent the number of functional units allocated by algorithms `ALG` and FDS with $\alpha = 0$, respectively.

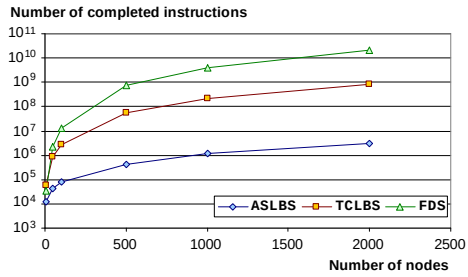
As can be seen, the ASLBS algorithm is able to perform scheduling much faster than the TCLBS and the FDS algorithms, taking less than 25 ms to schedule 80.000 nodes. However, for general networks with more than 50 nodes (benchmark R1), the TCLBS allocates over 10% less functional units than the other algorithms. The drawback of the TCLBS is that it is greedy. Therefore, while it can achieve very



(a) Allocation functional units *vs* number of nodes.



(b) Time *vs* number of nodes.



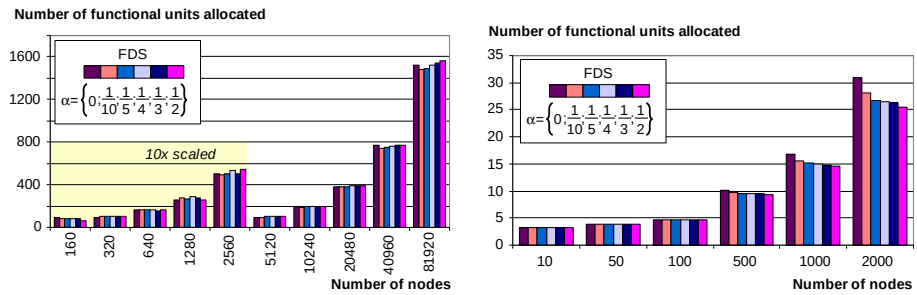
(c) Instructions *vs* number of nodes.

Figure 5.11: Performance of the scheduling algorithms (Benchmark R1).

good schedules in the general case it may also use more units in other cases, as illustrated in the results presented in Figure 5.10 for benchmark A3. On the other hand, the FDS algorithm not only presents the worse scheduling solutions, but it also takes more time than the other algorithms.

In Section 3.1 it was reported that by using a mobility reduction factor α larger than zero it was possible to reduce the greediness of the FDS algorithm. By varying α between 0 and $\frac{1}{2}$ it is possible to analyse the real effectiveness of this mobility reduction factor. The results for benchmarks A3 and R1 are presented in Figure 5.12. From this results it is possible to conclude that the scheduling solutions significatively change by increasing the value of α. However, one can also notice that the results are not always improved with a higher values of α, as shown for benchmark A3. Also, despite the number of functional units allocated decreases in the general case, both the ASLBS and the TCLBS algorithms still present better results. Plus, as α increases so does the CPU execution time: increasing α from 0 to $\frac{1}{10}$, $\frac{1}{5}$, $\frac{1}{4}$, $\frac{1}{3}$ and $\frac{1}{2}$ for a random network of 2000 nodes corresponds to an increase in the CPU execution time by a factor of 1.64, 1.72, 1, 99 and 2.34 times, respectively.

When comparing the scheduling algorithms by varying the maximum latency

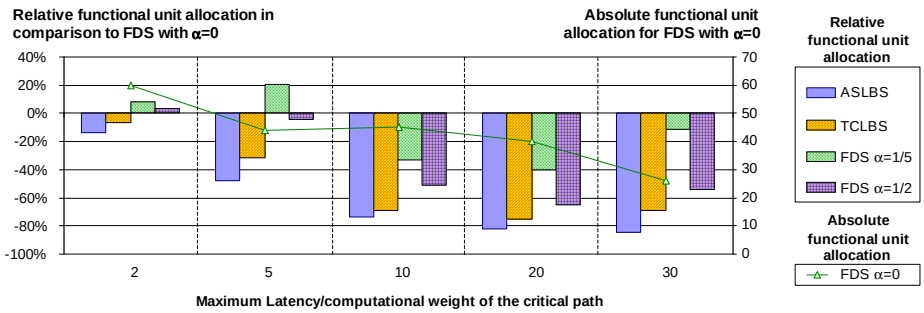


(a) Benchmark A3. **(b)** Benchmark R1.

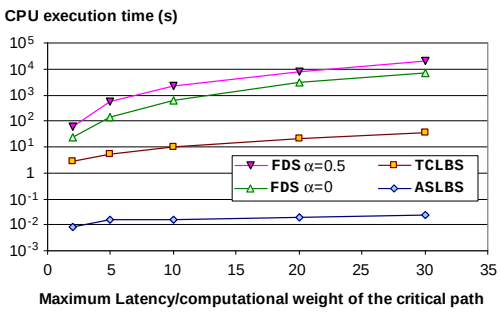
Figure 5.12: Performance of the FDS algorithm by varying the mobility reduction factor.

(T_{iter}), the FDS results become even worse: as predicted, it exhibits a higher dependency with T_{iter} than the TCLBS or the ASLBS. Furthermore, the extra computational time required does not lead to better scheduling solutions, as it can be seen by comparing the results of benchmarks A4 and R2 in Figure 5.13 and Figure 5.14. Results for benchmark F1 are presented in Appendix B. Optimally, one would expect that the folding factor[3] would increase proportionally with the increase of the maximum latency: if x units are allocated with a maximum latency T_{iter}, then only $\frac{x}{k}$ units would be needed for a maximum latency of $k \cdot T_{iter}$. Looking at the results for benchmark A4, the increase of the maximum latency from 80 c-steps to 1200 c-steps increase the folding factor of the ASLBS by 13 times. However, the increase in the folding factor for the TCLBS, the FDS with $\alpha = 0$ and the FDS with $\alpha = \frac{1}{2}$ is only of 7, 2.3 and 5.2 times, respectively. Furthermore, the increase in the computational time with T_{iter} grows much slower for the ASLBS than for the other algorithms: for benchmark A4, it increases 2.75 times from $T_{iter} = 80$ c-steps to $T_{iter} = 1200$ c-steps while the TCLBS, the FDS with $\alpha = 0$ and the FDS with $\alpha = \frac{1}{2}$ increase 12.5, 318.8 and 345.6 times, respectively. Also, by comparing the results for the random networks, presented in Figure 5.14, it is possible to conclude that the FDS algorithm is unable to reduce the number of functional units to the minimum of 3 (one per each type of operation). In fact, even by increasing the maximum latency to 20 times the computational weight of the critical path, the FDS algorithm still schedules almost 10 functional units in average. On the other hand, both the TCLBS and ASLBS algorithms were able to reach a scheduling solution of only 3 allocated functional units in almost every network with a maximum latency of 5 times the computational time of the critical path. This results shows that the FDS algorithms not only provides worse schedules and requires more computational time, but it is

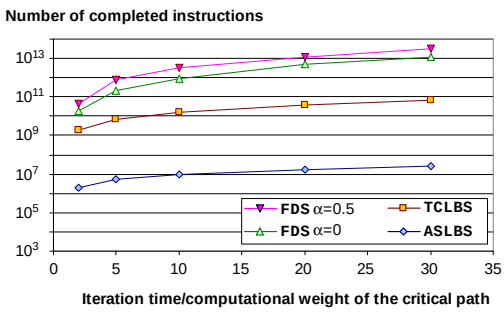
[3]Equal to the total number of operations divided by the total number of allocated functional units (see Section 2.4).



(a) Functional units allocation *vs* number of nodes.



(b) Time *vs* number of nodes.



(c) Instructions *vs* number of nodes.

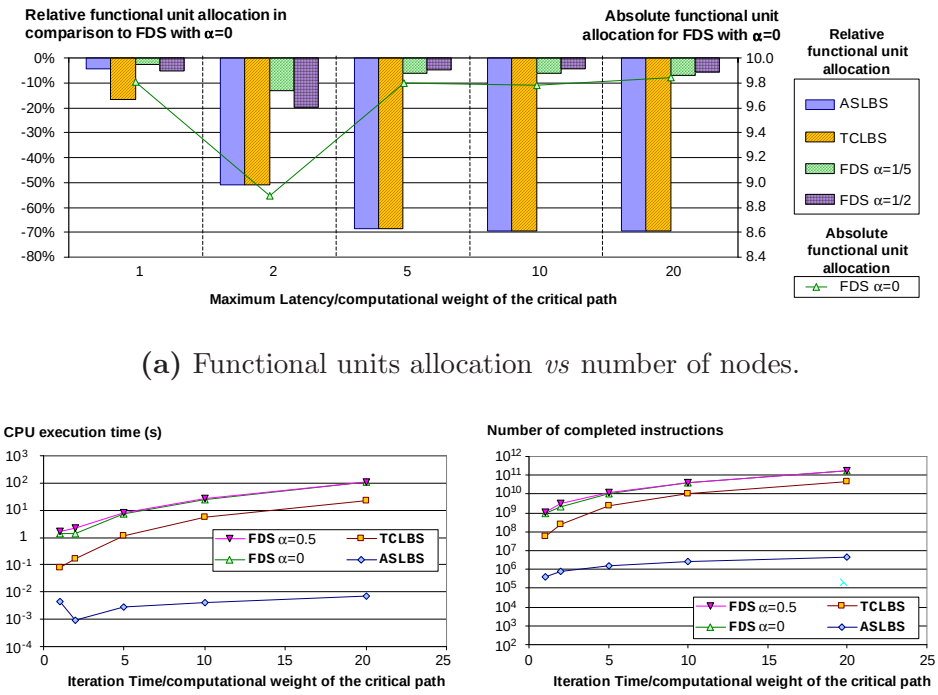
Figure 5.13: Performance of the scheduling algorithms (Benchmark A4).

alsounable to produce reasonable scheduling solutions when the maximum latency increases.

5.2.6 Register allocation

So far the analysis of the different scheduling algorithms has been made by comparing the CPU execution time and the number of functional units allocated. However, the number of registers required by each algorithms also an important feature. To perform this analysis, results for benchmarks A3 and R1 are presented for comparing the dependency between the algorithms and the number of nodes. Also benchmarks A4 and R2 are used to evaluate the algorithms on their dependency with the iteration time T_{iter}. These results can be found in Figure 5.15. Results for benchmarks F1 and F2 are presented in Appendix B.

The line in the plots represents the number of registers allocated after scheduling using the original FDS algorithm ($\alpha = 0$). The bars represent the percentage of registers allocated in comparison with the reference, after performing node scheduling by using the algorithms ASLBS, TCLBS, FDS with $\alpha = \frac{1}{5}$ and $\alpha = \frac{1}{2}$. The computation of the relative register allocation values for a scheduling algorithm `ALG`



(a) Functional units allocation *vs* number of nodes.

(b) Time *vs* number of nodes.

(c) Instructions *vs* number of nodes.

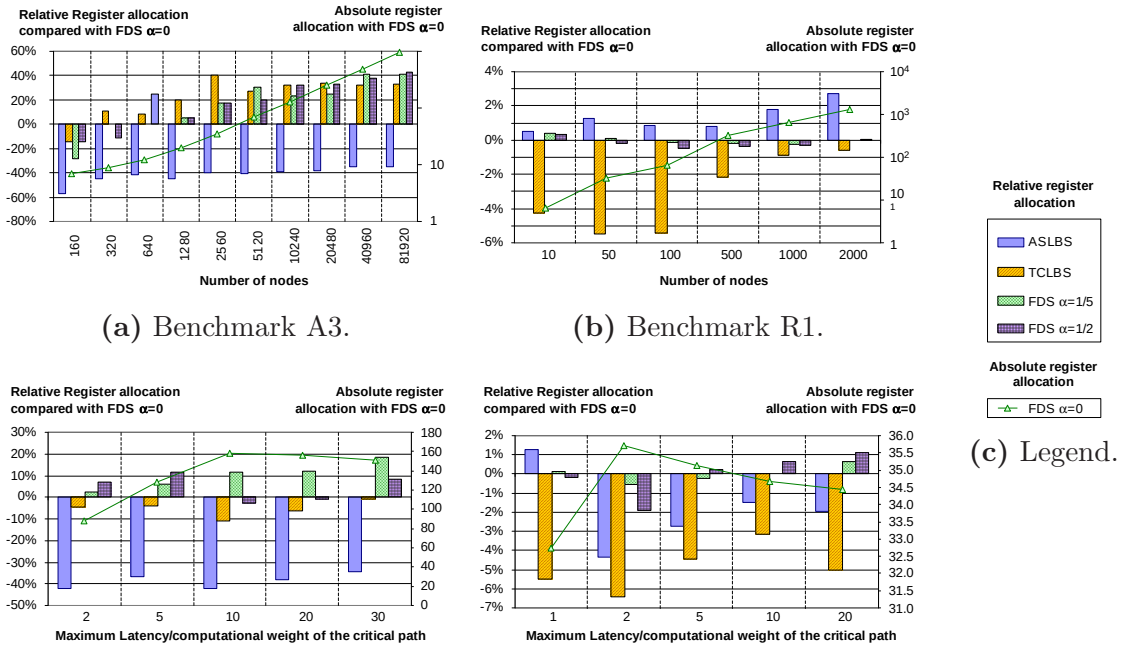
Figure 5.14: Performance of the scheduling algorithms (Benchmark R2).

is

$$M_{\texttt{ALG}}/M_{\texttt{FDS}} - 1 \tag{5.10}$$

where $M_{\texttt{ALG}}$ and $M_{\texttt{FDS}}$ represent the number of registers allocated by algorithms `ALG` and FDS with $\alpha = 0$, respectively.

By comparing the results of benchmarks A3 and R1, the first conclusion to be drawn is that the number of register allocated strongly depends on the solution provided by the different scheduling algorithms. Examining the results presented in Figure 5.15, it is possible to see that in the general case (benchmarks R1 and R2) the ASLBS leads to the allocation of more registers when the number of functional units is similar (compare the functional unit allocation results in Figure 5.11 and Figure 5.14 and the register allocation results in Figure 5.15). This increase is even stronger when comparing the register allocation after scheduling using the TCLBS and the ASLBS algorithms. Even in this case, the increase is less than 7%. Another important characteristic is that the number of allocated registers grows linearly with the number of nodes, as can be seen in Figure 5.15(a) and Figure 5.15(b). However, the number of registers also has a linear dependency with the iteration time T_{iter}. Although scheduling techniques reduce the number of functional units as T_{iter} grows, variables have to be stored for longer. Thus the number of registers may increase.



(a) Benchmark A3. (b) Benchmark R1. (c) Legend.

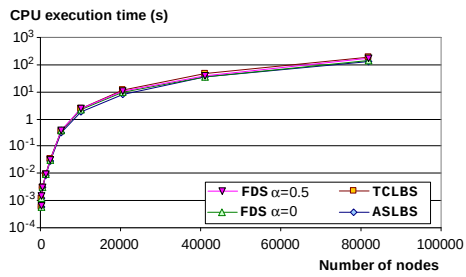
(d) Benchmark A4. (e) Benchmark R2.

Figure 5.15: Results on register allocation after applying different scheduling algorithms.

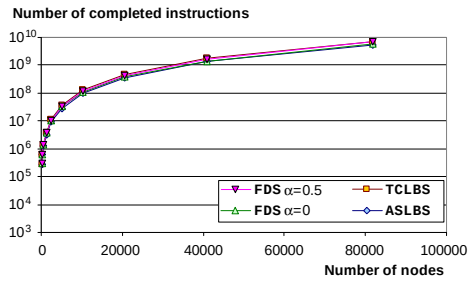
Regarding of CPU execution time, experiments show that the time is mostly dependent on the number of registers required. Thus, for the same network, scheduled using different algorithms, the CPU execution time tends to slightly increase if the scheduling solution requires more registers. Even so, the growth in the execution time is similar for all scheduling algorithms, as can be seen in Figure 5.16.

5.2.7 Describing the architecture in VHDL code

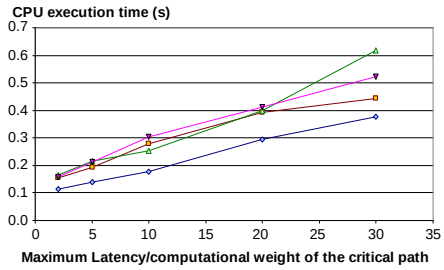
The algorithm to convert the generated data path into VHDL is almost straightforward. Thus the performance of this algorithm is mainly dependent on the number of functional units and registers allocated as can be seen from the results of benchmarks A3 and A4 presented in Figure 5.17. Therefore, since the increase in the number of nodes increases the number of functional units and the number of registers, the time to generate the VHDL increases. On the other hand, as the latency increases two factors contribute in opposite directions: schedulers tend to allocate less functional units, but the number of registers tends to increase. Overall, the decrease in the first factor is not enough to overcome the second factor. Thus, the algorithm tends to take slightly more time to execute as the maximum latency T_{iter} increases.



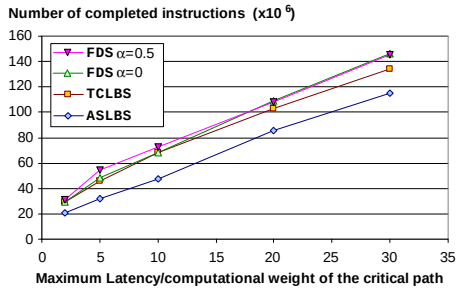
(a) Time *vs* number of nodes (benchmark A3).



(b) Instructions *vs* number of nodes (benchmark A3).



(c) Time *vs* number of nodes (benchmark A4).



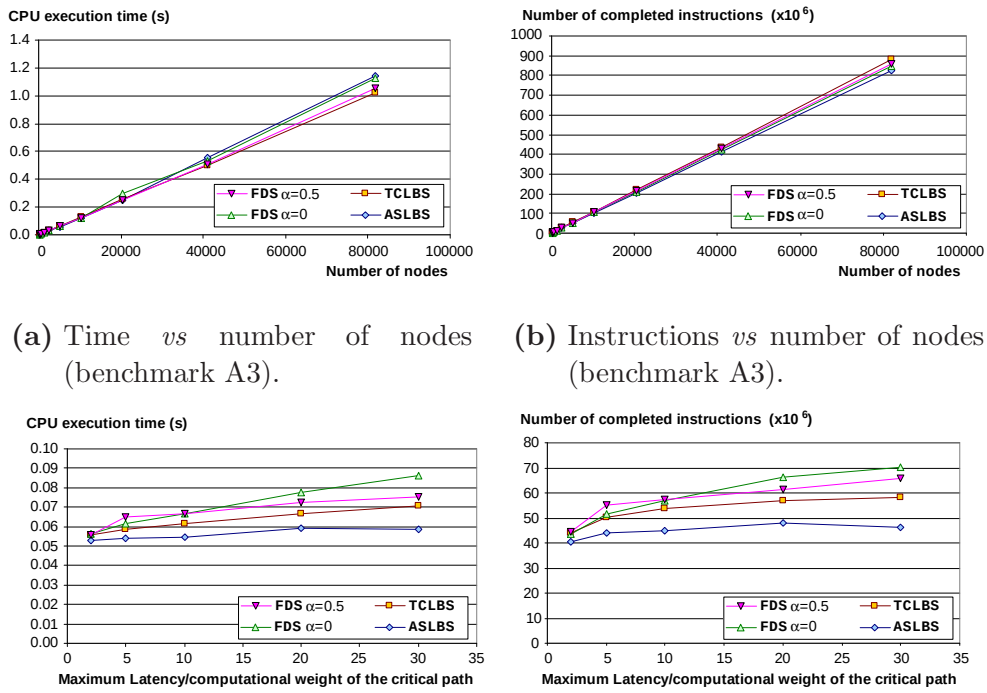
(d) Instructions *vs* number of nodes (benchmark A4).

Figure 5.16: Time to perform register allocation

5.3 Comparative evaluation with optimal scheduling

Eventhough the proposed time-constrained schedulers (the TCLBS and the ASLBS) present better results in terms of allocated functional units and CPU execution time than the commonly literature proposed FDS, it is still important to compare these algorithms with other schedulers. To perform this task, two other benchmarks were written into the ATAS application input language: a 5th-order Elliptical Wave Filter (EWF) [31] and a 1-D 8 point Discrete Cosine Transform (DCT) [32]. These are typical benchmarks used for evaluating scheduling algorithms. The results of the proposed schedulers are compared against: *i)* a time-constrained ILP-based scheduler [12]; *ii)* a genetic scheduler [18]; and *iii)* the Force-Directed Scheduler [13, 16] as described in Section 3.3.

The EWF and DCT benchmark results presented in Table 5.3 show that both the TCLBS and the ASLBS are able to achieve optimal solutions in most cases. Again the TCLBS and the ASLBS seem to behave better than the FDS. They allocate less functional units and lead to the use of less registers. In many cases they are even able to reach optimal solutions.



(a) Time *vs* number of nodes (benchmark A3).

(b) Instructions *vs* number of nodes (benchmark A3).

(c) Time *vs* number of nodes (benchmark A4).

(d) Instructions *vs* number of nodes (benchmark A4).

Figure 5.17: Time to generate the VHDL code.

5.4 Improving scheduling results

The results presented so far have proven that the fastest scheduler is the ASLBS. In terms of functional units this scheduler also behaves very well, either producing the best scheduling solutions or requiring no more than 8% more functional units. However, for the same cases, it is possible to further improve the results. Considering that the designer is willing to spend time trying to achieve a better scheduling, the best solution would be to run more then one scheduler and then to choose the result using less resources. In this case, the suggestion would be to firstly use the ASLBS, which is very fast, and then to improve the results by using the TCLBS. The relative improvement on the allocation of functional units for benchmark R1 is shown in Figure 5.18, where the improvement metric, is defined as

$$Improvement = \frac{M - M_{comb}}{M} \times 100\% \tag{5.11}$$

and M and M_{comb} represent the number of functional units for the single scheduling algorithm (ASLBS or TCLBS) and for the combined algorithm (best of ASLBS and TCLBS), respectively. As it can be observed, the improvement on large random networks is over 20% and 10% when compared with the results of the ASLBS and

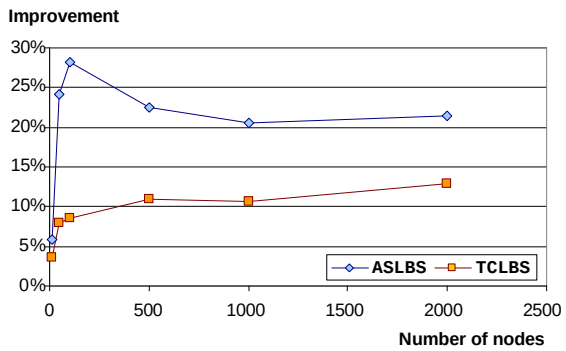


Figure 5.18: Performance of a ASLBS+TCLBS combined scheduling (benchmark R1).

the TCLBS, respectively. This advantage arises from the fact that the relative results of both algorithms varies from network to network. Thus, by using the best of both algorithms, it is possible to obtain a scheduling which is always better than each of them alone. Also, note that the relative increase in CPU execution time in comparison to the TCLBS algorithm is almost 0: the ASLBS algorithm by about 1000 times faster than the TCLBS algorithm. In terms of the number of registers there is no penalty: the results are similar to those obtained when performing TCLBS scheduling alone.

5.5 Conclusions

In this chapter, the performance of the HLS algorithms was evaluated based on the execution time and, in the case of the scheduling algorithms, in the amount of resources allocated. To perform this evaluation, two types of benchmarks were presented: those describing regular problems, ANNs and FFTs, and those describing irregular problems. While the first type represent the usual type of algorithms found in DSP, the second one describes better the type of DFGs obtained when translating a general algorithm, wrote in a high-level language, into a graph description. For each benchmark, evaluation was made by changing the number of nodes in the DFG and by varying the maximum iteration time allowed for the scheduling of the operations in the DFG.

Experimental results shown that by using the memory management system and hashtables, described in Section 4.3, the execution time of the algorithm to read the DFG from a file grows linearly with the number of nodes, taking approximately 21.4μs per node. On the other hand, the execution time of the algorithm to analyse the critical path grows quadratically with the number of nodes in the DFG. Through experimental evaluation of the retiming algorithm it was shown that the imple-

mented constraint pruning technique is not only able to significantly reduce the execution time, but also to decrease the memory requirements. For an ANN with 8,192 nodes, the execution time was reduced in over 78% while the memory requirements were decreased in over 99.96%. Evaluation of the several scheduling techniques allowed to conclude that the TCLBS works best by using the `AVAIL_ALAP/FORCE_LAST` pair. Also, the comparison with the two other schedulers, the ASLBS and the typical FDS, showed that the two developed algorithms are better than the FDS algorithm. These results are more significant when varying the maximum iteration time. In those cases, the FDS has always presented the worse results. Moreover, the ASLBS is not only the faster but it is also the one that allocates less functional units and registers for the regular problems. Nevertheless, results also show that for the general case, the random networks, the solution obtained by the ASLBS algorithm could be further improved in over 20% by attempting to schedule with the TCLBS algorithm and then choosing the best solution.

Table 5.3: Heuristic comparison for the EWF and 1-D DCT benchmarks.

Algorithm	EWF [31]		1-D DCT [32]	
	c-steps	**Resources**	**c-steps**	**Resources**
ILP-Based Scheduler*† [12]	17	3 ±,1 ×, - R	10	3 ±,3 ×, - R
	18	2 ±,1 ×, - R	12	3 ±,2 ×, - R
	19	2 ±,1 ×, - R	13	2 ±,2 ×, - R
	21	2 ±,1 ×, - R	20	2 ±,1 ×, - R
	28	1 ±,1 ×, - R	32	1 ±,1 ×, - R
Genetic Scheduler† [18]	17	3 ±,2 ×,10 R	10	- ±,- ×, - R
	18	2 ±,2 ×,10 R	12	- ±,- ×, - R
	19	- ±,- ×, - R	13	- ±,- ×, - R
	21	2 ±,1 ×, 9 R	20	- ±,- ×, - R
	28	- ±,- ×, - R	32	- ±,- ×, - R
FDS $\alpha = 0$†	17	3 ±,2 ×, 8 R	10	4 ±,2 ×,14 R
	18	2 ±,1 ×, 8 R	12	3 ±,2 ×,17 R
	19	2 ±,1 ×, 9 R	13	3 ±,2 ×,18 R
	21	2 ±,1 ×, 8 R	20	2 ±,1 ×,21 R
	28	2 ±,2 ×,10 R	32	2 ±,1 ×,20 R
FDS $\alpha = \frac{1}{4}$†	17	3 ±,2 ×, 8 R	10	4 ±,2 ×,14 R
	18	2 ±,1 ×, 8 R	12	3 ±,2 ×,19 R
	19	2 ±,1 ×,10 R	13	3 ±,2 ×,16 R
	21	2 ±,1 ×, 8 R	20	2 ±,2 ×,19 R
	28	2 ±,1 ×,11 R	32	2 ±,1 ×,22 R
TCLBS†	17	4 ±,3 ×, 8 R	10	4 ±,2 ×,16 R
	18	2 ±,1 ×, 8 R	12	3 ±,2 ×,19 R
	19	3 ±,1 ×, 9 R	13	3 ±,2 ×,19 R
	21	2 ±,1 ×,10 R	20	2 ±,1 ×,21 R
	28	2 ±,1 ×,10 R	32	1 ±,1 ×,19 R
ASLBS†	17	3 ±,2 ×, 8 R	10	4 ±,2 ×,15 R
	18	3 ±,2 ×, 9 R	12	3 ±,2 ×,15 R
	19	2 ±,1 ×, 8 R	13	3 ±,2 ×,15 R
	21	2 ±,1 ×, 9 R	20	2 ±,2 ×,19 R
	28	1 ±,1 ×,10 R	32	1 ±,1 ×,23 R

* *Resource contrained algorithm* — *ALU Units:* ±
† *Time contrained algorithm* — *Mult. Units:* ×
- Not Available — *Registers: R*

6

Conclusions

Contents

6.1 Future Work 100

Within the scope of this thesis, the core components of the ATAS tool for automatic generation of efficient hardware structures were proposed and programmed. The developed components comprise six tasks: *i)* reading the DSP algorithm from a file; *ii)* analysing the critical path; *iii)* retiming the DFG; *iv)* scheduling nodes to functional units; *v)* allocating the registers; and *vi)* binding of the data path into a VHDL description. All the developed algorithms were designed for synthesising complex architectures with 10,000 or more operations.

Within the developed algorithms, special attention was given to scheduling, due to typical scheduling algorithms being developed to solve small problems and thus seek for optimal solutions. However, since scheduling is a NP-complete problem, it is not likely to find an optimal algorithm that can be scaled to solve large problems. Moreover, there is no knowledge of the development of high-level synthesis scheduling algorithms for problems as complex as 10,000 nodes. The presented research is therefore focused on the development and evaluation of sub-optimal algorithms able to perform time-constrained scheduling for complex problems. With this purpose, two schedulers were developed: the Time-Constrained List-Based Scheduler (TCLBS) and the Adapted Static List Based Scheduler (ASLBS). These were compared with the most commonly scheduler used in the literature, the Force Directed-Scheduler (FDS), which was also implemented for comparison purposes. Experimental results show that, despite the ASLBS being a simple scheduling algorithm, it is able to solve the problem in a very small amount of time while still obtaining better results than the FDS. For example, for the ANN benchmarks, the ASLBS is able to achieve the best results amongst the tested algorithms, tooking over 1000 times less computational time. Results also show that, in the general case, the FDS algorithm is unable to increase the folding factor as the maximum iteration time increases. Even for regular problems, results evidence that the increase in the folding factor is much smaller than in the ASLBS or the TCLBS algorithms. For an ANN with 4,000 nodes and with an increase of 15 times in the iteration time, the ASLBS and the TCLBS algorithms increase the folding factor in 13 and 7 times, respectively. Nevertheless, the FDS algorithm only increases the folding factor in 5.2 times, even taking over 10^6 and 10^2 times more CPU execution time than the ASLBS and the TCLBS algorithms, respectively. Experimental results concerning the synthesis of general problems also show that it is possible to reduce the number of allocated functional units in the general case by performing dual scheduling (ASLBS and TCLBS) and retrieving the best result of the two. This technique, which has an execution time similar to the TCLBS, is able to improve the results obtained by performing single scheduling with the ASLBS algorithm more than 20% for large problems.

Comparing the performance of the proposed algorithms against optimal algorithms, it was possible to conclude that both the ASLBS and the TCLBS algorithms are able to achieve optimal solutions in most cases. Also, when the optimal solution is not obtained, by using the best solution between the ASLBS and the TCLBS algorithms, at most one functional unit is allocated above the optimum.

For the retiming task, a new algorithm was also developed. It uses constraint pruning techniques to reduce the computational time required to build the ILP problem. For an ANN with 8,192 nodes, by reducing the number of constraints in over 99.96%, it decreases the memory requirements in the same amount while decreasing the CPU execution time in more than 78%.

The development of the ATAS tool had two main goals: fast execution of the developed algorithms and simplicity in adding new features and algorithms. With this purpose several techniques were employed. Namely, the C-code data structures were designed to allow fast access to all node dependencies. To read the a DFG from file, hashtables were implemented so has to allow the reduction of the CPU execution time in 93%. Also, in the scope of several algorithms, such as file reading and retiming, a memory allocation scheme was designed. Such scheme works by allocating memory in large chunks and, by reducing the number of accesses to system functions, it allows to reduce the CPU execution time of the implemented algorithms of up to 66%.

Globally, the techniques developed for the ATAS tool are able to design an efficient structure for a complex algorithm in a short amount of time: it takes about 200 ms to synthesise an ANN with 2,000 nodes. When scaling for 80,000 nodes, the execution time increases quadratically to about 214 s. Also, it is able to schedule the operations of a 1024 point FFT (composed by 51,200 nodes) in 147 s. However, this time does not include retiming, since this algorithm is not indispensable: it is only required if the there is a path with no delays and a computational weight larger than the iteration time. Nevertheless, if retiming is required, the execution time grows in about 10 s for the 2,000 nodes ANN. In fact this is the most time-consuming task of the ATAS tool. Thus, it is one of the areas where new fast algorithms should be investigated.

Validation of the VHDL code generated by the ATAS tool was also performed. By using Xilinx ISE 7, it was verified that the tool generates no VHDL errors nor warnings. Moreover, the generated code was also simulated in order to verify that the generated architectures operate accordingly to the algorithm described by the input DFG.

6.1 Future Work

The developed techniques represent the initial developing stage of a more broad project: the development of a tool able to generate efficient hardware structures from a high level description of a DSP algorithm. In this sense, much work has to be done. Firstly, as defined in the project, the tool inputs and outputs should be redraw in order to support XML definitions. This will allow for a better integration with future tools that translate a high level description of the DSP algorithm in a DFG. Secondly, techniques for unfolding, such as the ones presented in Section 2.3 should be implemented. While this will increase the complexity of the HLS problem, it will also allow for the scheduling algorithm to explore more parallelism leading to a more efficient allocation of functional units and registers. Moreover, the retiming algorithm should be reviewed and expanded. On the one hand, improving the pruning of constraints should further reduce the CPU execution time. On the other hand, it makes sense to implement an ILP solver, therefore expanding the retiming algorithm to achieve either a minimum area or minimum period architectures. Concerning the scheduling problem, a new algorithm should also be developed to allow module selection depending on the constraints, i.e., selecting between fast and area consuming and slow but area efficient functional units. Finally, better support should be implemented for pipelined structures where data enters the architecture at a different rate then the algorithm iterations are computed.

Bibliography

[1] E. F. Deprettere, E. Rijpkema, P. Lieverse, and B. Kienhuis, "High Level Modeling for Parallel Executions of Nested Loop Algorithms," in *12th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'00)*, 2000, pp. 79–92.

[2] S. Rele, V. Jain, S. Pande, and J. Ramanujam, "Compact and Efficient Code Generation Program Restructuring on Limited Embedded DSPs," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 20, no. 4, pp. 477–494, 2001.

[3] G. Almasi, "MaJIC: A MATLAB just-in-time compiler," Ph.D. dissertation, University of Illinois, June 2001.

[4] J. M. P. Cardoso and H. C. Neto, "Compilation for FPGA-Based Reconfigurable Hardware," *IEEE Des. Test*, vol. 20, no. 2, pp. 65–75, 2003.

[5] R. Allen, "Compiling High-Level Languages to DSPs," *IEEE Signal Processing Magazine*, vol. 22, no. 3, pp. 47–56, May 2005.

[6] M. R. Garey and D. S. Johnson, *Computers and intractability : a guide to the theory of NP-completeness*. W.H. Freeman, 1979.

[7] "Algorithms and Tools for Automatic hardware Synthesis (ATAS)," http://sips.inesc-id.pt/projects/atas.

[8] "Extensible Markup Language (XML)," http://www.w3.org/XML/.

[9] E. Torbey and J. Knight, "Performing Scheduling and Storage Optimization Simultaneously Using Genetic Algorithms," in *MWSCAS '98: Proceedings of the 1998 Midwest Symposium on Systems and Circuits*. Washington, DC, USA: IEEE Computer Society, 1998, p. 284.

[10] M. Narasimhan and J. Ramanujam, "On Lower Bounds for Scheduling Problems in High-Level Synthesis," in *DAC '00: Proceedings of the 37th Conference*

on Design Automation (DAC'00). Washington, DC, USA: IEEE Computer Society, 2000, pp. 546–551.

[11] S. Chaudhuri and R. Walker, "ILP-based scheduling with time and resource constraints in high level synthesis," in *Proceedings of the Seventh International Conference on VLSI Design*, 1994, pp. 17–20.

[12] M. Narasimhan and J. Ramanujam, "Improving the Computational Efficiency of ILP-Based Scheduling Problems in High-Level Synthesis," in *Proc. International Conference on Computer Aided Design (ICCAD-98)*, 1998.

[13] P. G. Paulin and J. P. Knight, "Algorithms for High-Level Synthesis," *IEEE Des. Test*, vol. 6, no. 6, pp. 18–31, 1989.

[14] A. W. D. Gajski, N. Dutt and S. Lin, *High-Level Synthesis Introduction to Chip and System Design.* Kluwer Academic Publishers, 1994.

[15] A. Sllame and V. Drabek, "An efficient list-based scheduling algorithm for high-level synthesis," in *Euromicro Symposium on Digital System Design*, September 2002, pp. 316–323.

[16] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, J. van Meerbergen, and A. van der Werf, "Improved force-directed scheduling in high-throughput digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, pp. 945–960, 1995.

[17] P. Brucker, B. Jurisch, and B. Sievers, "A branch and bound algorithm for the job-shop scheduling problem," *Discrete Appl. Math.*, vol. 49, no. 1-3, pp. 107–127, 1994.

[18] E. Torbey and J. Knight, "Performing Scheduling and Storage Optimization Simultaneously Using Genetic Algorithms," in *Proceedings of the 1998 Midwest Symposium on Systems and Circuits.* Washington, DC, USA: IEEE Computer Society, 1998.

[19] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[20] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algoritms.* MIT Press, 1999.

[21] N. Maheshwari and S. Sapatnekar, "Optimizing large multiphase level-clocked circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 9, pp. 1249–1264, September 1999.

[22] E. C. Ifeachor and B. W. Jervis, *Digital Signal Processing*, 2nd ed. Prentice-Hall, 2002.

[23] R. S. A. Aho and J. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[24] K. Parhi, "Systematic Synthesis of DSP data format converters using life time analysis and forward-backward register allocation," *IEEE Transactions on Circuits and Systems - II*, vol. 39, no. 7, pp. 423–440, 1992.

[25] R. J. J. Jr., "Hash Functions for Hash Table Lookup," *Dr. Dobb's Journal*, September 1997.

[26] P. Hsieh, "Hash functions," http://www.azillionmonkeys.com/qed/hash.html, 2004.

[27] Xilinx, Inc., "Spartan-3 FPGA Family: Complete Data Sheet," http://direct.xilinx.com/bvdocs/publications/ds099.pdf.

[28] ——, "Xilinx ISE 7," http://www.xilinx.com/products/design_resources/design_tool/grouping/logic_design_prod.htm.

[29] Model Technology, "ModelSim Xilinx Edition III," http://www.xilinx.com/ise/verification/mxe_details.html.

[30] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.

[31] H. W. S.Y. Kung and T. Kailath, *VLSI and Modern Signal Processing*. Prentice-Hall, Inc., 1985.

[32] S. Dutta and W. Wolf, "A circuit-driven design methodology for video signal-processing datapath elements," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 2, pp. 229–240, 1999.

[33] D. A. Patterson and J. L. Hennessy, *Computer architecture: a quantitative approach*, 3rd ed. Morgan Kaufmann Publishers Inc., May 2002.

[34] "Performance Application Programming Interface (PAPI)," http://icl.cs.utk.edu/papi.

[35] Intel Corporation, "IA-32 Intel Architecture Software and Developer's Manuals," http://www.intel.com/design/pentium4/manuals/index_new.htm.

A

Proofs of High-Level Synthesis properties

Contents

A.1 Retiming 106
A.2 Unfolding 107
A.3 Folding 109

A.1 Retiming

Proof of Property 2.2.1. The weight of the retimed path can be written as,

$$w^r_{\mathsf{P}} = w^r_{\langle \mathsf{V}_0 \to \mathsf{V}_1 \rangle} + \ldots + w^r_{\langle \mathsf{V}_{k-1} \to \mathsf{V}_k \rangle} = \sum_{i=0}^{k-1} w^r_{\langle \mathsf{V}_i \to \mathsf{V}_{i+1} \rangle}$$

Using (2.2), this weight can be re-written as a function of the weight of the edges in the original path and of the retiming values for each node in the path:

$$w^r_{\mathsf{P}} = \sum_{i=0}^{k-1} w_{\langle \mathsf{V}_i \to \mathsf{V}_{i+1} \rangle} + r_{\mathsf{V}_{i+1}} - r_{\mathsf{V}_i} =$$

$$= \sum_{i=0}^{k-1} w_{\langle \mathsf{V}_i \to \mathsf{V}_{i+1} \rangle} + \sum_{i=0}^{k-1} r_{\mathsf{V}_{i+1}} - \sum_{i=0}^{k-1} r_{\mathsf{V}_i} =$$

$$= w_{\mathsf{P}} + \sum_{i=1}^{k} r_{\mathsf{V}_i} - \sum_{i=0}^{k-1} r_{\mathsf{V}_i}$$

$$w^r_{\mathsf{P}} = w_{\mathsf{P}} + r_{\mathsf{V}_k} - r_{\mathsf{V}_0}$$

□

Proof of Property 2.2.2. From Property 2.2.1, if the starting and ending nodes of a path are the same, then

$$w^r_{\mathsf{P}_{\circlearrowleft}} = w_{\mathsf{P}_{\circlearrowleft}} + r_{\mathsf{V}} - r_{\mathsf{V}} = w_{\mathsf{P}_{\circlearrowleft}}$$

□

Proof of Property 2.2.3. As seen in section 2.1, the loop bound depends on the computational time and on the cycle path weight. However, retiming does not change the computational time of each unit and from Property 2.2.2 the weight of a closed path remains constant after retiming. Therefore, the loop bound of a cycle is not changed by retiming a circuit. Moreover, since the iteration bound is a particular case of the loop bound, it also remains constant. □

Proof of Property 2.2.4. From (2.2), if the retiming value of the starting and ending nodes of the edge $\langle \mathsf{U} \to \mathsf{V} \rangle$ in the retimed path is the same, then

$$w^r_{\langle \mathsf{U} \to \mathsf{V} \rangle} = w_{\langle \mathsf{U} \to \mathsf{V} \rangle}$$

□

A.2 Unfolding

Proof of Algorithm 2.1. Consider a node in the original DFG U with a dependence to a node V imposed by an edge with weight $w_{\langle \mathsf{U}\to\mathsf{V}\rangle}$. When applying J-unfolding, a new circuit executing J iterations of the algorithm will be generated. The unfolding transformation therefore re-schedules an iteration $n \in \mathbb{N}$ to be executed at unit $i = n\%J$ and at time $k = \left\lfloor \frac{n}{J} \right\rfloor$, such that $n = k \times J + i$. This operation will also change the dependence $\langle \mathsf{U} \to \mathsf{V}\rangle$ between iterations n and $n + w_{\langle \mathsf{U}\to\mathsf{V}\rangle}$ and recreate it between node $\mathsf{U_i}$ of iteration $\left\lfloor \frac{n}{J} \right\rfloor$ and node $\mathsf{V_j}$ of iteration $\left\lfloor \frac{n+w_{\langle \mathsf{U}\to\mathsf{V}\rangle}}{J} \right\rfloor$ where

$$\begin{cases} i = n\%J \\ j = (n + w_{\langle \mathsf{U}\to\mathsf{V}\rangle})\%J \end{cases}$$

Replacing n by $k \times J + i$ results in:

$$\begin{cases} i = n\%J \\ j = (k \times J + i + w_{\langle \mathsf{U}\to\mathsf{V}\rangle})\%J \end{cases} = \begin{cases} i = n\%J \\ j = (i + w_{\langle \mathsf{U}\to\mathsf{V}\rangle})\%J \end{cases}$$

Moreover, the weight of the new edge $\langle \mathsf{U_i} \to \mathsf{V_j}\rangle$ will be:

$$w_{\langle \mathsf{U_i}\to\mathsf{V_j}\rangle} = \left\lfloor \frac{n + w_{\langle \mathsf{U}\to\mathsf{V}\rangle}}{J} \right\rfloor - \left\lfloor \frac{n}{J} \right\rfloor = \left\lfloor \frac{k \times J + i + w_{\langle \mathsf{U}\to\mathsf{V}\rangle}}{J} \right\rfloor - \left\lfloor \frac{k \times J + i}{J} \right\rfloor =$$

$$k + \left\lfloor \frac{i + w_{\langle \mathsf{U}\to\mathsf{V}\rangle}}{J} \right\rfloor - k - \left\lfloor \frac{i}{J} \right\rfloor = \left\lfloor \frac{i + w_{\langle \mathsf{U}\to\mathsf{V}\rangle}}{J} \right\rfloor - \left\lfloor \frac{i}{J} \right\rfloor$$

Since $0 \le i \le J - 1$, it results that

$$w_{\langle \mathsf{U_i}\to\mathsf{V_j}\rangle} = \left\lfloor \frac{i + w_{\langle \mathsf{U}\to\mathsf{V}\rangle}}{J} \right\rfloor$$

□

Proof of Property 2.3.1. Following Algorithm 2.1, by unfolding a DFG by a factor of J, each edge $\langle \mathsf{U} \to \mathsf{V}\rangle$ with weight $w_{\langle \mathsf{U}\to\mathsf{V}\rangle}$ generates J unfolded edges with a sum of weights

$$W_{\langle \mathsf{U}\to\mathsf{V}\rangle} = \sum_{i=0}^{J-1} \left\lfloor \frac{i + w_{\langle \mathsf{U}\to\mathsf{V}\rangle}}{J} \right\rfloor$$

Considering that $w_{\langle \mathsf{U}\to\mathsf{V}\rangle}$ can be described as

$$w_{\langle \mathsf{U}\to\mathsf{V}\rangle} = a \times J + b$$

where a is the integer division of $w_{\langle \mathsf{U}\to\mathsf{V}\rangle}$ by J and b is the remainder after division, the sum can be re-written as

$$\sum_{i=0}^{J-1} \left(a + \left\lfloor \frac{i + b}{J} \right\rfloor \right) = a \times J + \sum_{i=0}^{J-1} \left\lfloor \frac{i + b}{J} \right\rfloor$$

where

$$\left\lfloor \frac{i+b}{J} \right\rfloor = \begin{cases} 0 & , \frac{i+b}{J} < 1 \\ 1 & , \frac{i+b}{J} \geq 1 \end{cases} = \begin{cases} 0 & , i < J-b \\ 1 & , i \geq J-b \end{cases}$$

Since $0 \leq i \leq J-1$, the sum of weights of the J edges, $W_{\langle \mathsf{U} \to \mathsf{V} \rangle}$, will be

$$a \times J \;+\; \underbrace{0+0+\cdots+0}_{J-b\ zeros} \;+\; \underbrace{1+1+\cdots+1}_{b\ ones} \;= w_{\langle \mathsf{U} \to \mathsf{V} \rangle}$$

□

Proof of Lemma 1. Since for every edge, the sum of delays of the edges generated after unfolding is the same as the delay of the original edge, the total number of delays in the original DFG and the unfolded DFG are the same. □

Proof of Property 2.3.2. A DFG with a closed path P, $\mathsf{A} \to \cdots \to \mathsf{A}$, unfolded by a factor of J originates multiple closed paths $\mathsf{P_U}$, each including $n = p+1$ unfolded representations of A:

$$\mathsf{P_U} = \mathsf{A_i} \to \cdots \mathsf{A_{k_1}} \to \cdots \to \mathsf{A_{k_p}} \to \cdots \to \mathsf{A_i} \quad , p \in [0; J-1]$$

Note that n is not necessarily the unfolding factor, but an unknown variable. The restriction to the loop's existence is found in Algorithm 2.1 and states that the path $\langle \mathsf{A_i} \to \mathsf{A_i} \rangle$ is valid proved that:

$$i = (i + n \cdot w_{\mathsf{P}}) \% J$$

which is true while $n \cdot w_{\mathsf{P}}$ is a multiple of J:

$$n \cdot w_{\mathsf{P}} = m \cdot J \qquad , m \in \mathbb{N}$$

The minimum path $\langle \mathsf{A_i} \to \mathsf{A_i} \rangle$, i.e., the path which transverses less unfolded representations of A, will be the one presenting the lowest number n:

$$N = n_{min} = \left(\frac{m \cdot J}{w_{\mathsf{P}}} \right)_{min}$$

Since n and m must be positive integers, the problem can be summed to finding the minimum value of m such that $\frac{m_{min} \cdot J}{w_{\mathsf{P}}}$ is integer:

$$M = m_{min} = \frac{w_{\mathsf{P}}}{\mathtt{gcd}\,(w_{\mathsf{P}}, J)}$$

where $\mathtt{gcd}\,(a, b)$ is the greatest common divider between a and b.

Therefore,

$$N = \frac{J}{\mathtt{gcd}\,(w_{\mathsf{P}}, J)}$$

Since each loop $\mathsf{P_U}$ contains $J/\mathtt{gcd}\,(w_{\mathsf{P}}, J)$ unfolded representations of A, the total number of closed paths generated from P is

$$\frac{J}{J/\mathtt{gcd}\,(w_{\mathsf{P}}, J)} = \mathtt{gcd}\,(w_{\mathsf{P}}, J)$$

Also, from Algorithm 2.1 the number of delays in the path $\langle \mathsf{A_i} \rightarrow \mathsf{A_i} \rangle$ is

$$w^r_{\mathsf{P}} = \left\lfloor \frac{i + n \cdot w_{\mathsf{P}}}{J} \right\rfloor$$

and replacing n with N results,

$$w^r_{\mathsf{P}} = \left\lfloor \frac{i}{J} + \frac{J \cdot w_{\mathsf{P}}}{\mathtt{gcd}\,(J \cdot w_{\mathsf{P}}, J)} \right\rfloor = \frac{w_{\mathsf{P}}}{\mathtt{gcd}\,(w_{\mathsf{P}}, J)}$$

□

Proof of Property 2.3.3. As defined on Section 2.1, the Loop Bound of a cycle P is defined as

$$LB = \frac{t_{\mathsf{P}}}{w_{\mathsf{P}}}$$

Moreover, as it can be observed from Property 2.3.2, the number of delays of each new unfolded cycle $\mathsf{P_U}$ will be $\frac{w_{\mathsf{P}}}{\mathtt{gcd}(J, w_{\mathsf{P}})}$. Also, the computational time of each unfolded cycle will be increased by the number of times that the original cycle needs to be executed to close the path, i.e. by $\frac{J}{\mathtt{gcd}(J, w_{\mathsf{P}})}$. The unfolded Loop Bound, LB_U, is therefore:

$$LB_U = \frac{\frac{J}{\mathtt{gcd}(J, w_{\mathsf{P}})} t_{\mathsf{P}}}{\frac{w_{\mathsf{P}}}{\mathtt{gcd}(J, w_{\mathsf{P}})}} = J \frac{t_{\mathsf{P}}}{w_{\mathsf{P}}} = J \cdot LB$$

□

Proof of Lemma 1. Since the Iteration Bound is a particular case of the Loop Bound, the proof of this Lemma is directly taken from Property 2.3.3. □

A.3 Folding

Proof of Theorem 1. Let the nodes $\mathsf{U_i}$ and $\mathsf{U_j}$ be scheduled to be executed by units $\mathtt{E}_{\mathsf{U_i}}$ and $\mathtt{E}_{\mathsf{U_j}}$. Set N as the number of c-steps required to perform one iteration of the algorithm. The result of U will be generated at time $t_1 = \mathtt{s}_{\mathsf{U_i}} + \mathsf{P}_{\mathsf{U_i}}$ but is only consumed at time $t_2 = \mathtt{s}_{\mathsf{U_j}} + N \cdot w_{\langle \mathsf{U_i} \rightarrow \mathsf{U_j} \rangle}$, thus requiring the result to be stored for $t_2 - t_1 = N \cdot w_{\langle \mathsf{U_i} \rightarrow \mathsf{U_j} \rangle} - \mathsf{P}_{\mathsf{U_i}} + \mathtt{s}_{\mathsf{U_j}} - \mathtt{s}_{\mathsf{U_i}} = w^*_{\langle \mathsf{U_i} \rightarrow \mathsf{U_j} \rangle}$ clock cycles.

Since the storage time $t_2 - t_1 \geq 0$, the restriction $w^*_{\langle \mathsf{U_i} \rightarrow \mathsf{U_j} \rangle} \geq 0$ must be fulfilled. Re-writting this restriction results:

$$N \cdot w_{\langle \mathsf{U_i} \rightarrow \mathsf{U_j} \rangle} - \mathsf{P}_{\mathsf{U_i}} + \mathsf{s}_{\mathsf{U_j}} - \mathsf{s}_{\mathsf{U_i}} \geq 0 \;\Leftrightarrow\; \mathsf{s}_{\mathsf{U_i}} - \mathsf{s}_{\mathsf{U_j}} \leq \mathsf{P}_{\mathsf{U_i}} - N \cdot w_{\langle \mathsf{U_i} \rightarrow \mathsf{U_j} \rangle}$$

□

B

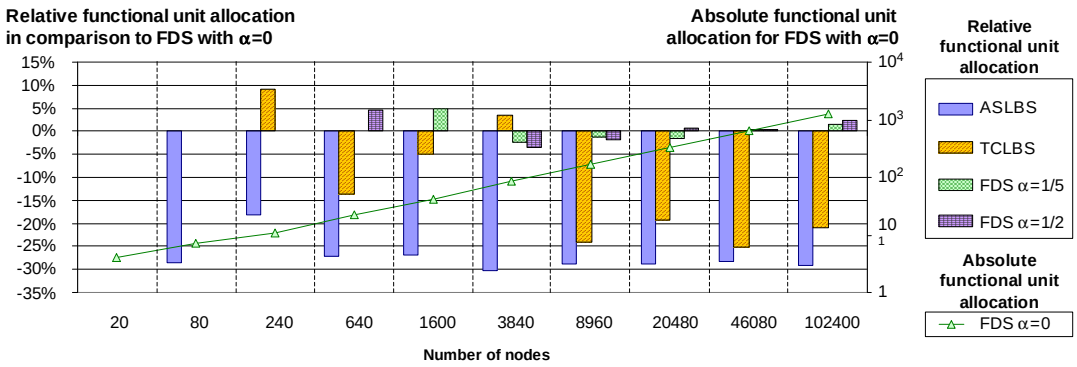
Additional experimental results

Contents

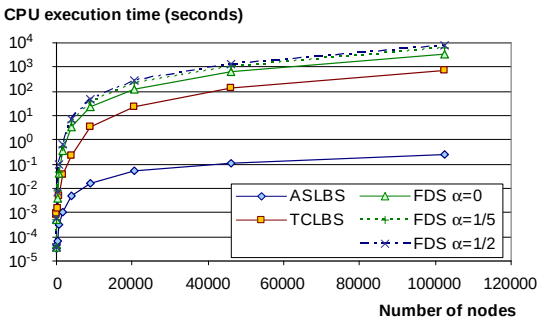
B.1 FFT benchmark 112

B.1 FFT benchmark

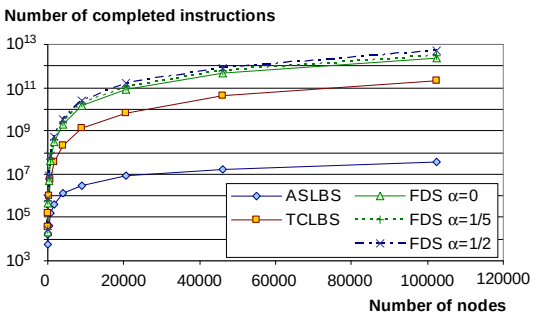
In this appendix the scheduling and register allocation results for benchmarks F1 and F2 are presented (see Table 5.1). In this sense, Figure B.1 shows the results for the scheduling of a DFG computing the FFT of a signal varying from 2 to 1024-points. For this benchmark the iteration time was kept 50% above the computational



(a) Functional unit allocation *vs* number of nodes.



(b) Time *vs* number of nodes.

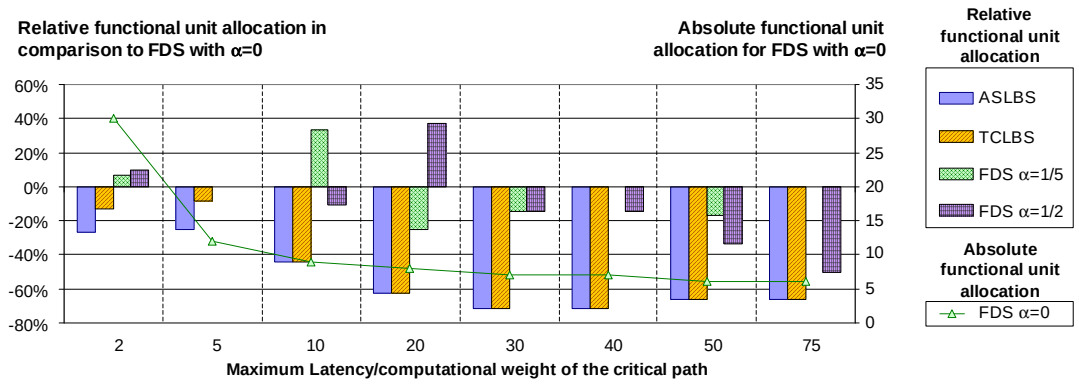


(c) Instructions *vs* number of nodes.

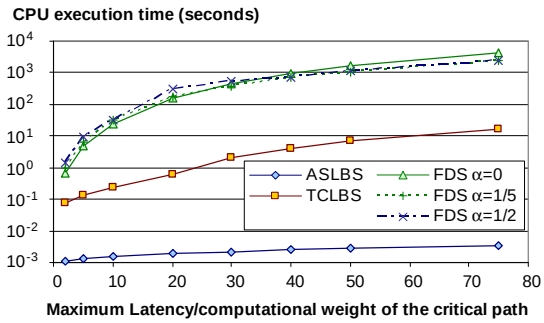
Figure B.1: Performance of the scheduling algorithms (Benchmark F1).

weight of the critical path. Therefore, for the 2-point FFT, with a critical path of 4 c-steps, the specified iteration time was 6 c-steps. Similarly, for the 1024-point FFT, having a critical path weight of 40 c-steps, the specified iteration time was 60 c-steps. Results show that the ASLBS algorithm is not only the quickest one but it also leads to the allocation of the least number of functional units. Comparatively, for a large number of nodes, the TCLBS algorithm produces scheduling solutions which are not much worse than those obtained by the ASLBS algorithm. However it requires considerably more time to reach the scheduling solution. On the other hand, the FDS presents the worse scheduling solutions: corresponds to an increase of up to 30% more functional units than the ASLBS algorithm. Figure B.2 presents the results for the scheduling of a 32-point FFT by varying the iteration time between

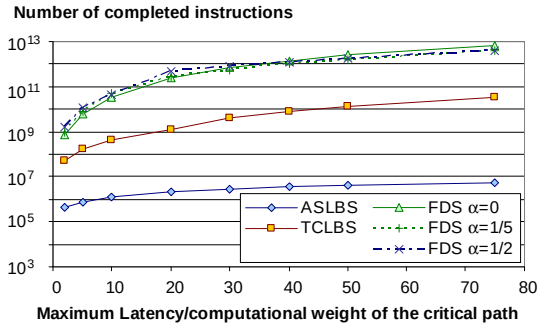
2 and 75 times the weight of the critical path. From Table 5.1, the computational



(a) Functional units allocation *vs* iteration time.



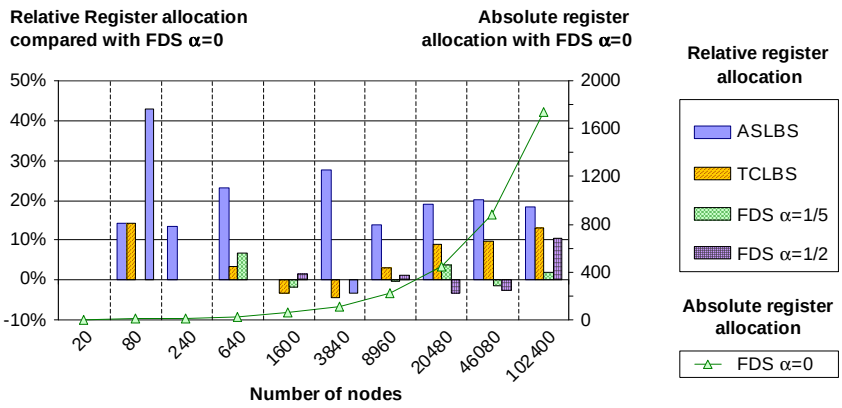
(b) Time *vs* iteration time.



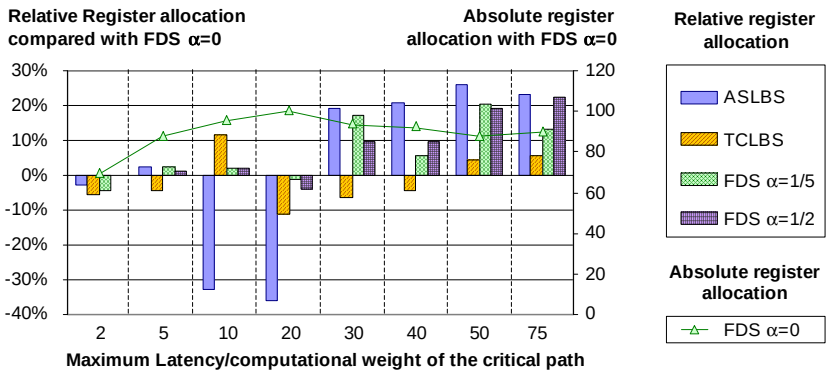
(c) Instructions *vs* iteration time.

Figure B.2: Performance of the scheduling algorithms (Benchmark F2).

weight of the critical path is 20 c-steps. Therefore the specified iteration time was 40, 100, 200, 400, 600, 800, 1000 and 1500 c-steps. The experimental results show that the ASLBS and the TCLBS algorithms are able to reach solutions which use more than 60% less functional units than the FDS with $\alpha = 0$. The advantage of the first two algorithms comes from the fact that, unlike the FDS, they are able to significatively increase the folding factor as the iteration time increases. Thus, the obtained solutions require less functional units. Figure B.3, presents the register allocation results for benchmarks F1 and F2 after performing scheduling with each of the ASLBS, TCLBS and FDS algorithms. This results show that, for the FFT benchmark, the ASLBS algorithm tends to require more registers than any of the other scheduling algorithms: for a FFT with 128 points or more it requires almost 20% more registers than the FDS with $\alpha = 0$. On the other hand, the TCLBS algorithm, which required slightly more functional units than the ASLBS, only uses around 10% more registers than the FDS algorithm. When varying the iteration time, the results are confirmed: the ASLBS requires about 20% more registers than



(a) Benchmark F1: register allocation *vs* number of nodes.



(b) Benchmark F2: register allocation *vs* iteration time.

Figure B.3: Results on register allocation after applying different scheduling algorithms.

the FDS. Curiously, for the FFT benchmark, the FDS tends to use less registers with $\alpha = 0$ than with $\alpha = \frac{1}{5}$ or $\alpha = \frac{1}{2}$.