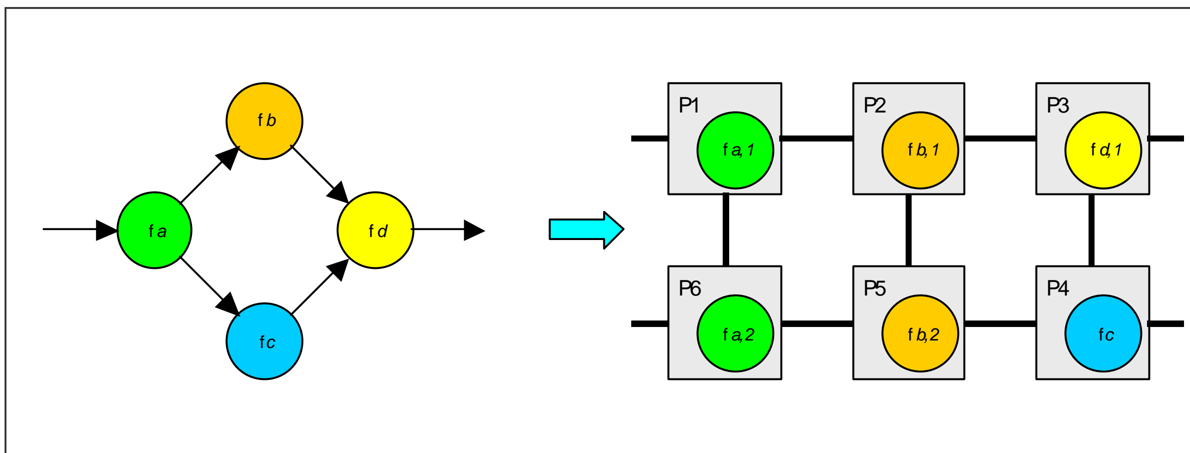
Master Thesis

A Pre-Processing Method for Software Synthesis of Synchronous Dataflow Networks

A.C.M. van den Berg

836856321

June 2009



Open Universiteit Nederland

Department of Computer Science - Master Opleiding Technische Informatica

Thesis committee *Open Universiteit*
Prof. dr. A. Bijlsma
Dr. B.J. Heeren

Thales Nederland
Dr. ir. H. Schurer
Ir. T.S. Schilder

Acknowledgements

It took quite some years to finish this thesis. For this reason, the people involved have changed over the years.

In this acknowledgement I would like to thank Maarten te Boekhorst, my first Open University advisor, for his critical view on the problem definition and valuable feedback during the first phase of this thesis.

I also would like to thank professor Johan Jeuring for his supervision, guidance and help. Johan was involved from the beginning until February 2008 when he became educational director at the University of Utrecht.

At this moment, professor Lex Bijlsma and supervisor Bastiaan Heeren became involved. Although they weren't involved during the execution phase, I would like to thank Bastiaan for his detailed and valuable comments on the draft and final versions of this thesis.

I am also grateful to my tutors from Thales, Hans Schurer and Theo Schilder, for their guidance and support during the execution phase of this thesis.

Last, but not least, I would like to thank my family and friends who encouraged me to finish this last and important part of my study.

Summary

This thesis describes the progress that has been realized on the software synthesis of functional specifications of real-time signal processing applications that run on multi-processor architectures. It has been a challenge for years to bridge the gap between the abstract functional specification and the hardware dependent target code in an automatic way. Although commercial solutions exist for a limited set of hardware platforms, this thesis focuses on a generic approach.

The thesis defines a three-layered approach for the software synthesis process. The most abstract layer, the top layer, exploits the data parallelism in the functional specification and it increases the function granularity in the specification. The middle layer schedules the functions to a hardware architecture and the bottom layer generates the target code for each individual processing node. At each lower layer more hardware details are necessary. This thesis describes the implementation of the hardware independent top layer of the software synthesis process.

In the first part of the thesis an appropriate specification language for signal processing applications is defined. For this purpose, the Synchronous Data Flow (SDF) formalism, that is often used to describe signal processing applications, is extended. In the extension data particles, which are communicated between functions, are not considered atomic. Data particles can carry dimensions that specify the memory organisation of data. The dimensions are used to explore the data parallelism in the functional specification.
Latency and throughput are typical requirements of a real-time signal processing application that need to be optimised. Characteristics that determine latency are modelled in an abstract, hardware independent way. The characteristics are computation complexity and communication delay.

The next part of the thesis describes the design and implementation of a compiler that exploits the data parallelism in the functional specification. The compiler finds function iteration in the functional specification based on the data dimensions. Where appropriate, the compiler creates extra instances of iterated functions and re-distributes the data to the functions accordingly. The result is that the granularity of the functions in the specification will be increased and that latency will be reduced.

The results of the extended formalism are tested with a prototype of a Network Expansion Compiler and a typical signal processing application. The results are successful to a certain extent. An accurate latency reduction cannot be given, but the compiler can determine which functions contribute significantly to the total network latency. These functions are to be dealt with first.
Further improvements should be carried out on the prototype of the compiler to become of practical use.

Samenvatting

Deze scriptie beschrijft de voortgang die is geboekt bij de vertaling van functionele specificaties van real-time signaalverwerkende applicaties naar een software-implementatie die op multi-processorarchitecturen executeren. Dit proces wordt softwaresynthese genoemd. Het is al jaren een uitdaging om het gat tussen de abstracte functionele specificatie en de hardware afhankelijke softwarecode automatisch te dichten. Hoewel er voor een beperkt aantal hardware platformen commerciële oplossingen bestaan, richt deze scriptie zich op een algemene aanpak.

De scriptie definieert een aanpak van drie lagen voor het softwaresynthese proces. De meest abstracte laag, de bovenste laag, maakt gebruik van het in de specificatie aanwezige dataparallellisme en verhoogt de granulariteit van de functies in de specificatie. De middelste laag beeldt de functies af op de hardware architectuur en de onderste laag genereert de softwarecode voor elke afzonderlijke processor. Bij elke laag is er meer kennis nodig van de hardware. De scriptie beschrijft de implementatie van de bovenste laag van het softwaresynthese proces.

In het eerste gedeelte van de scriptie wordt een geschikte specificatietaal voor signaalverwerkende applicaties gedefinieerd. Hiervoor wordt het Synchrone Data Flow formalisme, dat vaak wordt gebruikt om dergelijke applicaties te specificeren, uitgebreid. In de uitbreiding worden datadeeltjes, die tussen de functies worden gecommuniceerd, niet als ondeelbaar beschouwd. De datadeeltjes kunnen dimensies hebben die de organisatie van de data in het geheugen specificeren. De dimensies worden gebruikt om de aanwezige dataparallelliteit in de specificatie te ontdekken.
Doorlooptijd en verwerkingscapaciteit zijn typische eisen die aan een real-time signaalverwerkende applicatie worden gesteld. De karakteristieken die de doorlooptijd bepalen, worden op een abstracte, hardwareonafhankelijke wijze gemodelleerd. Deze karakteristieken zijn de rekencomplexiteit en de vertraging veroorzaakt door communicatie.

Het tweede gedeelte van de scriptie beschrijft het ontwerp en implementatie van een compiler die de dataparallelliteit in de specificatie uitbuit. De compiler zoekt naar iteratie van functies in de specificatie gebaseerd op de data dimensies. De compiler creëert waar mogelijk extra instantiaties van functies die herhaald worden en herverdeelt de data overeenkomstig naar de nieuwe en bestaande functies. Het resultaat is dat de granulariteit van de functies in de specificatie wordt vergroot en dat de doorlooptijd wordt verkleind.

De resultaten van de uitgebreide specificatietaal zijn getest met een prototype van een Network Expansion Compiler en een typische signaalverwerkende applicatie. De resultaten zijn tot op zekere hoogte succesvol. Een nauwkeurige reductie van de doorlooptijd kan niet worden gegeven, maar de compiler is in staat om de functies te vinden die een significante bijdrage leveren aan de totale doorlooptijd van het netwerk. Deze functies moeten als eerste worden aangepakt.
Het prototype van de compiler behoeft nog een aantal verbeteringen om praktisch bruikbaar te worden.

Table of Contents

1. Introduction 8
1.1. Scope and context 8
1.2. Process context 11
1.3. Thesis' structure 12
2. Definitions and background aspects 13
2.1. Introduction 13
2.2. Level of parallelism 13
2.3. Architectures of high performance computers 13
2.4. Memory organization 14
2.5. Network topologies 14
2.6. Software synthesis 16
2.7. Common parallel processing techniques 18
2.8. Expansion and scheduling 22
3. Problem definition 24
4. Hardware independent modelling formalism 25
4.1. Modelling aspects 25
4.2. Specification language of function networks 29
4.3. Execution rate 36
4.4. Distributor and Aggregator functions 38
4.5. Summary 40
5. Network Expansion Compiler 41
5.1. Global design Network Expansion Compiler 41
5.2. Parser 41
5.3. Analyse network 43
5.4. Expand network 46
5.5. Generate output 47
5.6. Implementation 47
6. Results 48
6.1. Software synthesis framework 48
6.2. Testing Network Expansion Compiler 49
6.3. Shortcomings prototype Network Expansion Compiler 55
6.4. Scheduler and code generator 56
6.5. Visualisation 58
7. Conclusions and recommendations 59
7.1. Conclusions 59
7.2. Recommendations Network Expansion Compiler 60
7.3. General recommendations 61

8. Abbreviations ... 62

9. References ... 63

APPENDIX A Specifications ... 64

A.1 Functional Network Specification in Backus-Naur Form ... 64

A.2 DOT-file radar application demo ... 66

A.3 ENDF radar application demo ... 68

A.4 FDF radar application demo ... 71

A.5 ADF radar application demo ... 75

Table of Figures

Figure 1-1 Sensor system 8
Figure 1-2 Mapping of functions to a processor architecture 9
Figure 1-3 Layered method of software synthesis process 11
Figure 2-1 Network topologies 15
Figure 2-2 Data flow diagrams 17
Figure 2-3 Definition latency and throughput 19
Figure 2-4 Increased throughput with a pipeline 20
Figure 2-5 Task parallelism 21
Figure 2-6 Data parallelism 22
Figure 2-7 Mapping of SDF-network on a physical multiprocessor architecture 23
Figure 4-1 DF-diagram of typical radar SP-application 25
Figure 4-2 Homogeneous SDF-diagram of typical radar SP-application 26
Figure 4-3 Multi-rate SDF-diagram of typical radar application 26
Figure 4-4 Multi-rate SDF-diagram with exploited data parallelism 27
Figure 4-5 Extended dataflow diagram of the typical radar SP-application 28
Figure 4-6 Distribution overlap 32
Figure 4-7 Function with scalar inputs and outputs 35
Figure 4-8 Function with multiple dimensions input and output 35
Figure 4-9 Execution rate of 1 36
Figure 4-10 Execution rate larger than 1 37
Figure 4-11 Inconsistent context 37
Figure 4-12 Exception on execution rate dimensions 37
Figure 4-13 Functional dataflow network without parallelisation 38
Figure 4-14 Distribution and aggregation 39
Figure 4-15 Path optimisation 40
Figure 5-1 Processing flow Network Expansion Compiler 41
Figure 5-2 Class diagram Network Expansion Compiler 42
Figure 5-3 Path selection 43
Figure 6-1 Software synthesis framework 48
Figure 6-2 Unprocessed specification of radar demo application 50
Figure 6-3 Processed radar demo application reference (without expansion) 51
Figure 6-4 Expanded radar demo application 54
Figure 6-5 Expanded function network allocated to hexagonal processing architecture 57
Figure 6-6 Alternative specification dimension parameters (1) 58
Figure 6-7 Alternative specification dimension parameters (2) 58
Figure 9-1 Hexagonal processor architecture 75

1. Introduction

1.1. Scope and context

The Thales Business Unit “Surface Radar” (SR) develops and produces radar and optronic sensor systems, mainly for military customers. These sensor systems are used for surveillance (object search) or fire control (target engagement) purposes.

Figure 1-1 shows the architecture of a modern (pulsed) radar system and is explained below.

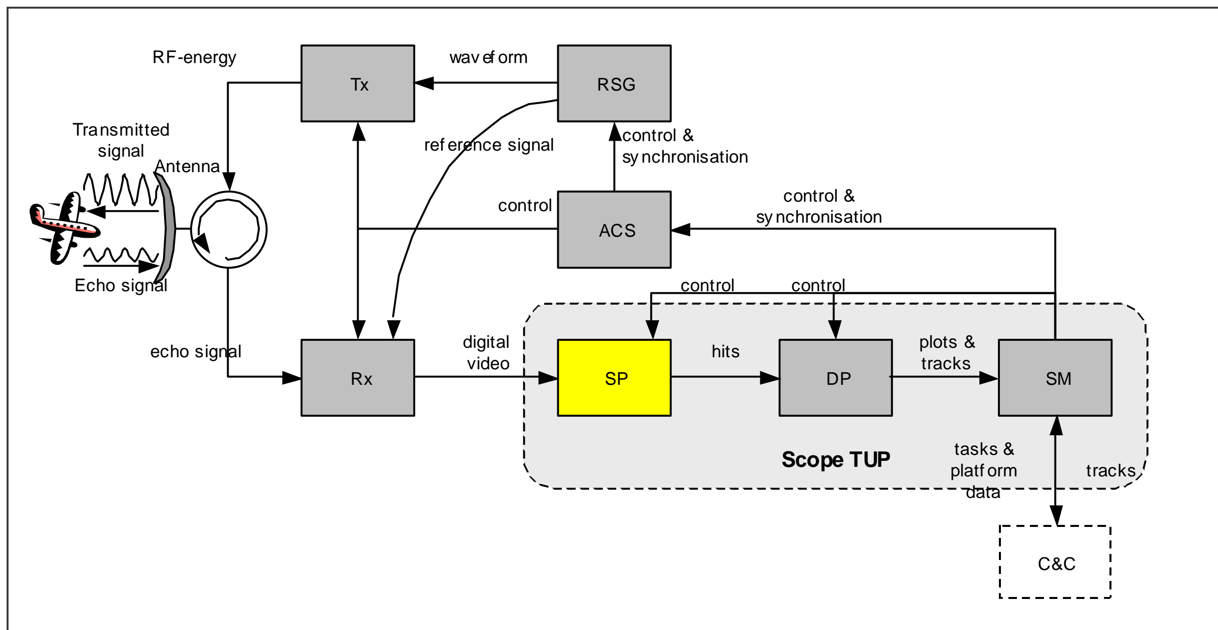


Figure 1-1 Sensor system

The Antenna Control System (ACS) sends *control & synchronisation* data to the Radar Signal Generator (RSG). The *control & synchronisation* data defines the exact waveform and frequency of the radar pulses to be generated by the RSG and determines the exact timing when the *waveform* is to be transmitted. The pulse frequency, the pulse repetition frequency and the number of pulses are import parameters of the control data. The transmitter (Tx) amplifies the *waveform* signal and sends the *Radio Frequency (RF) energy* to the antenna aperture. The antenna aperture has the right geometry to transfer *RF-energy* into free space. As soon as the amplified waveform has been transmitted, the radar system will switch to the listening mode.

When the RF-energy encounters an object, a part of the RF-energy is reflected back to the antenna as an *echo signal*. The reflected energy is routed to the receiver (Rx) where the *echo signal* is transformed to a lower frequency and sampled by an Analogue to Digital Converter (ADC). The ADC in the receiver generates a *digital video* stream that is processed by the Signal Processing (SP). The SP reduces the video stream significantly to detections (*hits*). Hits are detections that have (almost) equal bearing, but are ambiguous in range and speed. The Data Processing (DP) reduces the ambiguous *hits* to *plots* (clusters of hits with identical range, bearing and elevation). *Plots* are correlated with previous transmissions in the same direction and finally, stable plot and track data (*plots and tracks*) are transferred to the Sensor Management (SM). In most cases the SM acts as a gateway to the Command & Control (C&C) system. C&C is not a part of the radar system. It provides all kind of *tasks and platform data* to the sensor. *Tracks* are reported back to the C&C system.

Within the Business Unit Surface Radar, the Technical Unit Processing (TUP) is responsible for the implementation of the Signal Processing, Data Processing and Sensor Management, as indicated in Figure 1-1. “Implementation” stands for the definition of processing architectures and the development of hardware and software functions to implement the functional specifications.

When we zoom in on the Signal Processing, a functional model often specifies the signal processing application. The number of functions in the functional model is relatively small and the data dependencies

between the functions are rather straightforward. The challenge though is to process data streams up to hundreds of MB/s and produce hard real-time results within tens of milliseconds to the Data Processing. Although an optronic sensor system has a different sensor architecture than a radar, Signal Processing is part of this architecture as well and faces similar challenges.

Hard real-time signal applications demand powerful parallel processing architectures and may be implemented by dedicated programmable hardware, multiprocessor architectures, or a combination of both.

Processing architectures built with (programmable) hardware devices like Application Specific Integrated Circuits (ASIC's) or Field Programmable Gate Arrays (FPGA's) offer the ability to develop application dependent processing architectures, i.e. the hardware architecture is mapped to the application's functions. In some cases this is inevitable, e.g. when a generic architecture is not feasible from a technical or economical point of view.

With generic multi processor architectures a mapping must be created between the functions and the processor architecture as illustrated in Figure 1-2.

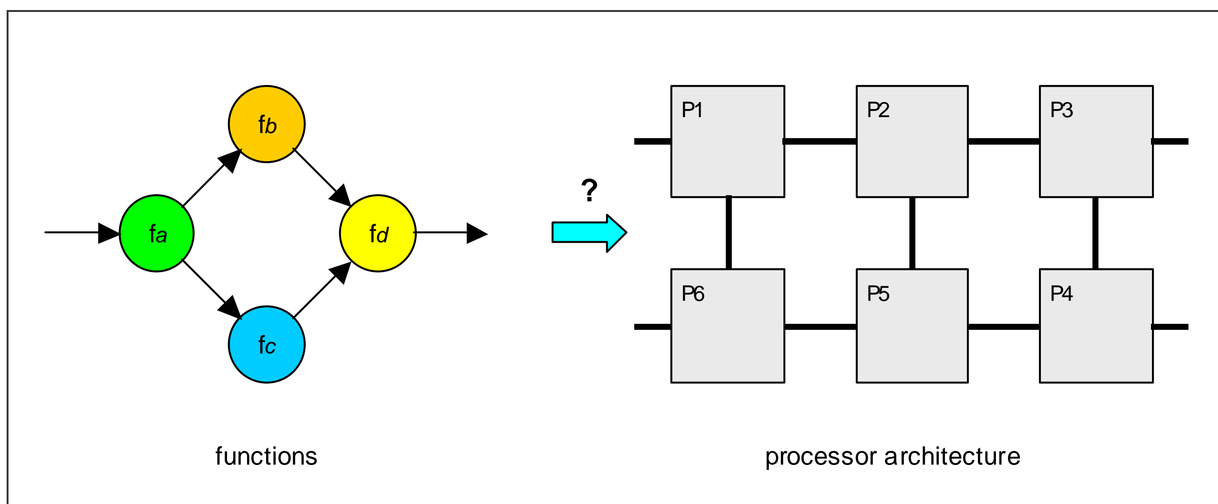


Figure 1-2 Mapping of functions to a processor architecture

As can be seen from the rather simple example in Figure 1-2, the granularity from the function network is smaller than the granularity of the processor architecture. With real-time signal processing applications, this is usually the case. The hardware architecture cannot be used in an optimal way, unless the granularity of the network is increased. This may be possible if the granularity of the data dependencies is further exploited.

Finally, application source code has to be developed and target code has to be allocated to one of the processors. The process of allocating target code to the processing architecture is called *scheduling*. The schedule has to be optimised with respect to the real-time requirements of the application. The whole process of transferring the functional specifications or models into target code is called *software synthesis*.

If one of the elements in the software synthesis process is changed, the whole process has to be repeated. For a long time the software synthesis has been a manual process. It is a fact that the hardware of the multiprocessor hardware evolves more rapidly than the functional specification. Hence, software synthesis has to be performed each time the hardware architecture changes.

As manual software synthesis of the functional specifications is time consuming and a major cost driver, the Technical Unit Processing (TUP) works on a method to automate the software synthesis process. A layered approach seems to be the best approach to decouple the rapid hardware evolvement from the functional specifications.

The method that has been chosen has three layers: a top layer exploits the parallelism in the functional specification and increases the granularity of the function network, the middle layer executes the scheduling and the bottom layer is responsible for the target code generation.

Progress has already been made on target code generation and scheduling where the number of functions is larger than the number of processors. The Thales tool MODERN (Modelling and Design Environment for Relational Networks) was originally developed to support the method, but a commercial tool has replaced it. The commercial tool, however, does not cover the top layer.

Therefore, this thesis focuses on the top layer, where the parallelism of the functional specification will be exploited in order to increase the granularity of the function network, i.e. the functional specification is pre-processed for the scheduler.

Functional models of signal processing applications are often specified by the "Synchronous Data Flow" (SDF) formalism. The formalism is further explained in 2.6.2, but, in short, it defines a network of nodes connected by edges. The nodes represent the functions in the network and the edges the data dependencies between the functions. This thesis extends the data flow formalism to further exploit the data parallelism and to enable the generation of intermediate specifications or models as input for the middle layer.

To summarize: the top layer pre-processes the functional specification before scheduling, hence the title of the thesis: *"A Pre-Processing Method For The Software Synthesis Of Synchronous Data Flow Networks"*.

1.2. Process context

As described in the previous paragraph, this thesis deals with the top layer of the software synthesis process of real-time signal processing applications on multiprocessor architectures.

Figure 1-3 shows the complete process of the software synthesis in relation to the thesis' scope.

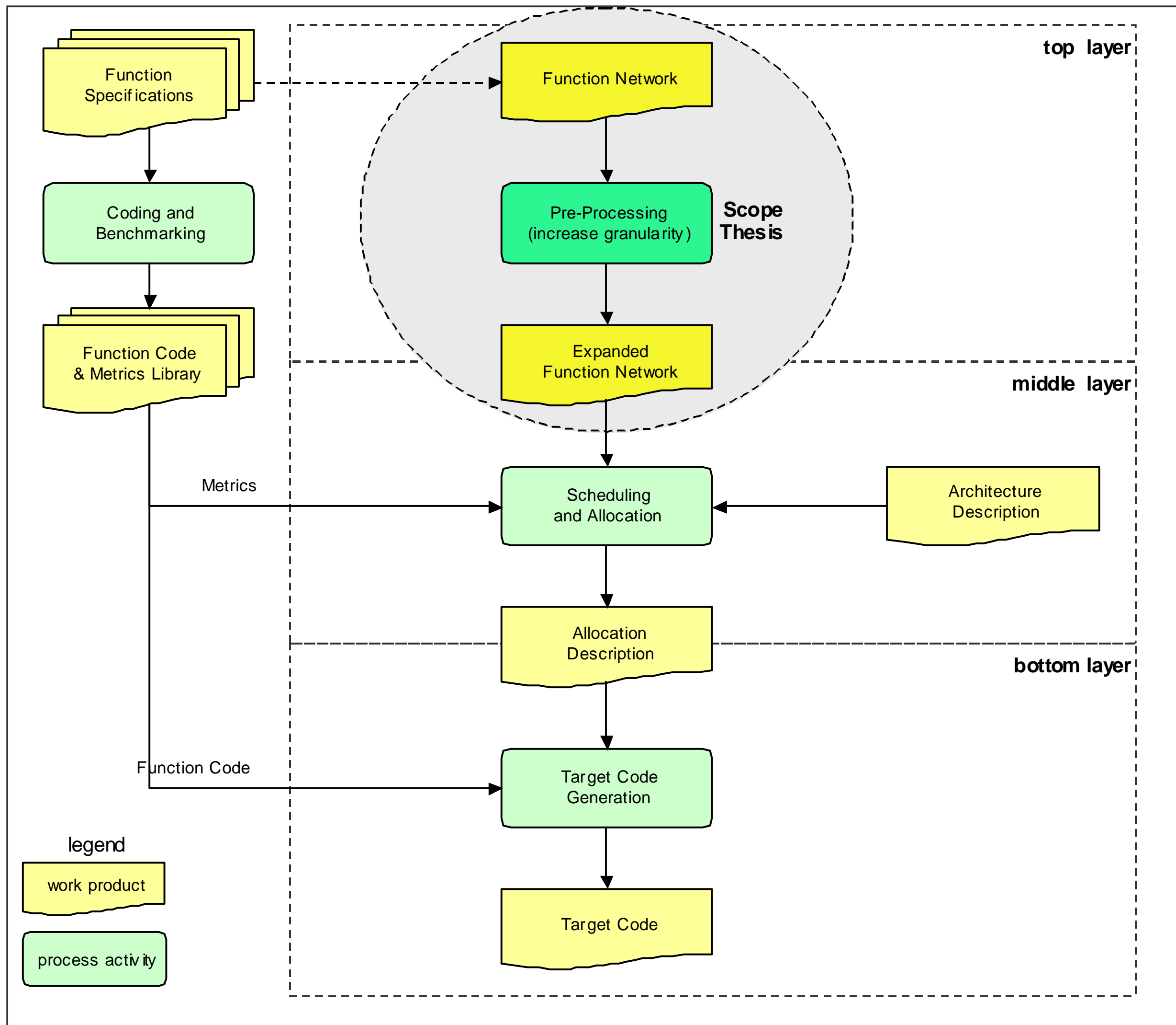


Figure 1-3 Layered method of software synthesis process

For the sake of completeness and as background information the complete process is described hereafter.

The top layer deals with the pre-processing of the functional specification, the *Function Network*. The Function Network specifies a network of functions in an extended data flow notation that will be described in chapter 4. Each function in the network has a *Function Specification* and may be implemented (coded) for different target processors.

Code analysis or target benchmarking determines the scheduling metrics (e.g. number of operations) and are stored in a *Function Code & Metrics Library*. The function specifications are used in every layer of the process. In particular, the input and output specification are relevant for the pre-processing, as will become clear in chapter 4 and 5.

The *Architecture Description* specifies the characteristics of the hardware architecture: the topology of processors and memory, the processing capabilities of individual processors, the interface capabilities, the amount of memory, etc.

In the middle layer, the *Function Network* specification, the *Architecture Description* and the *Function Code & Metrics Library* are used as input for the *Scheduling and Allocation* process. The output of this process is an *Allocation Description* that specifies which function is allocated to which node in the processing architecture.

In the bottom layer, the *Target Code Generation* process takes the *Allocation Description* and the *Function Code Library* as input and generates *Target Code* for each processor.

At the end of the software syntheses the real-time requirements are verified. If they are not met, one or more work products in the process must be modified. When for example the pre-processor has exploited the maximum degree of parallelism in the function network, the implementation of the functions or the hardware architecture must be adjusted.

1.3. Thesis' structure

Now the scope and context are clear, the structure of the rest of this thesis is defined as follows:

As the thesis' scope is rather specialized, chapter 2 will define and explain some relevant background aspects in order to better understand the problem description.

In chapter 3 the problem description of this thesis is defined.

In chapter 4 a modelling formalism is defined to describe functional networks in a formal way. In fact, it is an extension of the Synchronous Data Flow formalism described in paragraph 2.6.2.The formalism is needed for the implementation of a Network Expansion Compiler (NEC) that increases the granularity of the network.

Chapter 5 discusses the design of the NEC that was developed to increase the granularity of the functional network. Where applicable, the design choices will be explained.

In chapter 6 the obtained (test) results are presented and discussed.

In chapter 7 the conclusions and recommendations for further research and NEC improvements are presented.

2. Definitions and background aspects

2.1. Introduction

In this chapter relevant aspects are addressed that were studied during a literature survey and they should help the reader to understand the scope of the thesis' problem that is defined in chapter 3.

First, the level op level of parallelism, or the granularity of parallel tasks, will be addressed in paragraph 2.2.

Although scheduling and the hardware structure of the computer system are outside the scope of this thesis, some background information is given on processor architectures (paragraph 2.3), memory organisations (paragraph 2.4) and processor network topologies (paragraph 2.5). It helps to understand the context of the scheduling aspects as explained in 2.8.

A lot of literature has been published on software synthesis of signal processing applications. Some programming standards have evolved and these are addressed shortly in 2.6.1. The data flow formalism has been used quite often in the literature for the software synthesis of signal processing applications. The data flow formalism will be explained in 2.6.2.

In paragraph 2.7 some common techniques to increase throughput and reduce latency are explained.

2.2. Level of parallelism

According to [1], the performance of a computer system is defined by three factors. The time to execute a program (T) is the product of the number of instructions to execute (n_i), the average number of clock cycles required per instruction (CPI), and the clock cycle time (t_c):

$T = n_i \times CPI \times t_c$

This thesis focuses on lowering the average number of clock cycles by exploiting program level parallelism. A lower average number of clock cycles helps to decrease latency. With *program level parallelism* a single program is broken down into parts where independent sections (tasks) and loop iterations are candidates for exploiting parallelism. In [1] other levels of parallelism are classified as *job level parallelism, instruction level parallelism* and *arithmetic and bit level parallelism*.

A concept related to the level of parallelism is the *granularity* of parallel tasks or operations. A large grain system is one in which the operations that run in parallel are fairly large, in the order of entire programs. Small grain parallel systems divide programs into very small pieces, in some cases only a few instructions. Task level parallelism can be considered as medium grain parallelism and determines the level in this thesis.

2.3. Architectures of high performance computers

The hardware architecture determines to a large extent the performance of a computer system. The taxonomy of Flynn, as defined in [3], is often used for the classification of high-performance computers. The classification is based on the way of manipulating of instruction and data streams and comprises four main architectural classes: SISD, SIMD, MISD and MIMD machines. Van der Steen and Dongarra [3] summarize Flynn's taxonomy in the following way:

- *SISD* machines: Single Instruction Single Data stream machines are the conventional systems that contain one CPU (Central Processing Unit) and hence can accommodate one instruction stream that is executed serially.
- *SIMD* machines: Single Instruction Multiple Data stream machines often have a large number of processing units that all may execute the same instruction on many data items in parallel. Vector or array processors may be considered as a subclass of SIMD machines. They act on arrays of similar data using specially structured CPUs. Today, Graphical Processing Units (GPU's) are a good example of SIMD machines
- *MISD* machines: Multiple Instruction Single Data stream machines act with multiple instructions on a single stream of data. These kinds of machines are sometimes used in fault tolerance systems.

- *MIMD* machines: Multiple Instruction Multiple Data stream machines execute several instruction streams in parallel on different data. The difference with the multi-processor SISD machines lies in the fact that instructions and data are related because they represent different parts of the same task to be executed. There are a large variety of MIMD systems and especially in this class the Flynn taxonomy proves to be not fully adequate for the classification of systems. In the MIMD category, all arrays of processors are lumped together regardless of how they are connected and how they view memory. Since these characteristics can have a dramatic effect on performance, it would be desirable if the taxonomy reflected those differences.

Other types of taxonomies have been defined ([5], [6] and [7]), but no other has become as popular as the Flynn taxonomy.

2.4. Memory organization

The memory organization of multiprocessor architectures is another relevant aspect of parallel systems. According to [3], two memory organisations can be made: shared memory and distributed memory.

2.4.1. Shared memory

Shared memory systems have multiple CPU's all of which share the same address space. This means that the knowledge of where data is stored is of no concern to the user as there is only one memory accessed by all CPU's on an equal basis. The physical connections are quite simple. Most bus structures allow an arbitrary (but not too large) number of devices to communicate over the bus. The problem with shared memory systems is that processors must contend for access to the bus. Local cache memory at each CPU can improve performance, but it also introduces an extra difficulty known as the cache coherence problem.
Shared memory systems can be either SIMD or MIMD.

2.4.2. Distributed memory

In the case of distributed memory systems each CPU has its own associated memory. The CPU's are connected by some network and may exchange data between their respective memories when required. In contrast to shared memory machines the user must be aware of the location of the data in the local memories and will have to move or distribute these data explicitly when needed.
Distributed memory systems have several benefits compared to shared memory systems. First, there is no bus or switch contention. Each processor can utilize the full bandwidth to its own local memory without interference from other processors. Second, the lack of a common bus means there is no inherent limit to the number of processors; the size of the system is now constrained only by the network used to connect processors to each other. Third, there are no cache coherence problems. Each processor is in charge of its own data, and it does not have to worry about putting copies of it in its own local cache and having another processor reference the original.

The major drawback in the distributed memory design is that interprocessor communication is more difficult. If a processor requires data from another processor's memory, it must exchange messages with the other processor. This introduces two sources of overhead: it takes time to construct and send a message from one processor to another, and a receiving processor must be interrupted in order to deal with messages from other processors.

Again, distributed memory systems may be either SIMD or MIMD. Distributed memory systems exhibit a large variety in their topology. A few examples will be discussed in the next paragraph.

2.5. Network topologies

In this paragraph some network topologies are described.

A major consideration in the design of parallel systems is the set of pathways over which the processors, memories, and switches communicate with each other. These connections define the interconnection network, or topology, of the machine. Besides the performance of the processors, memory and switches, the topology has a major influence on the scheduling and mapping of the signal processing application onto the network of the computer system. From [1] the most relevant aspects will be summarized.

The following discussion of the properties of interconnection networks is based on a collection of nodes that communicate via links. In an actual system the nodes can be processors, memories, or switches. Two

nodes are neighbours if there is a link connecting them. Figure 2-1 shows a few examples of network topologies: ring, fully connected, hypercube, star, tree, mesh, crossbar switch and banyan switch.

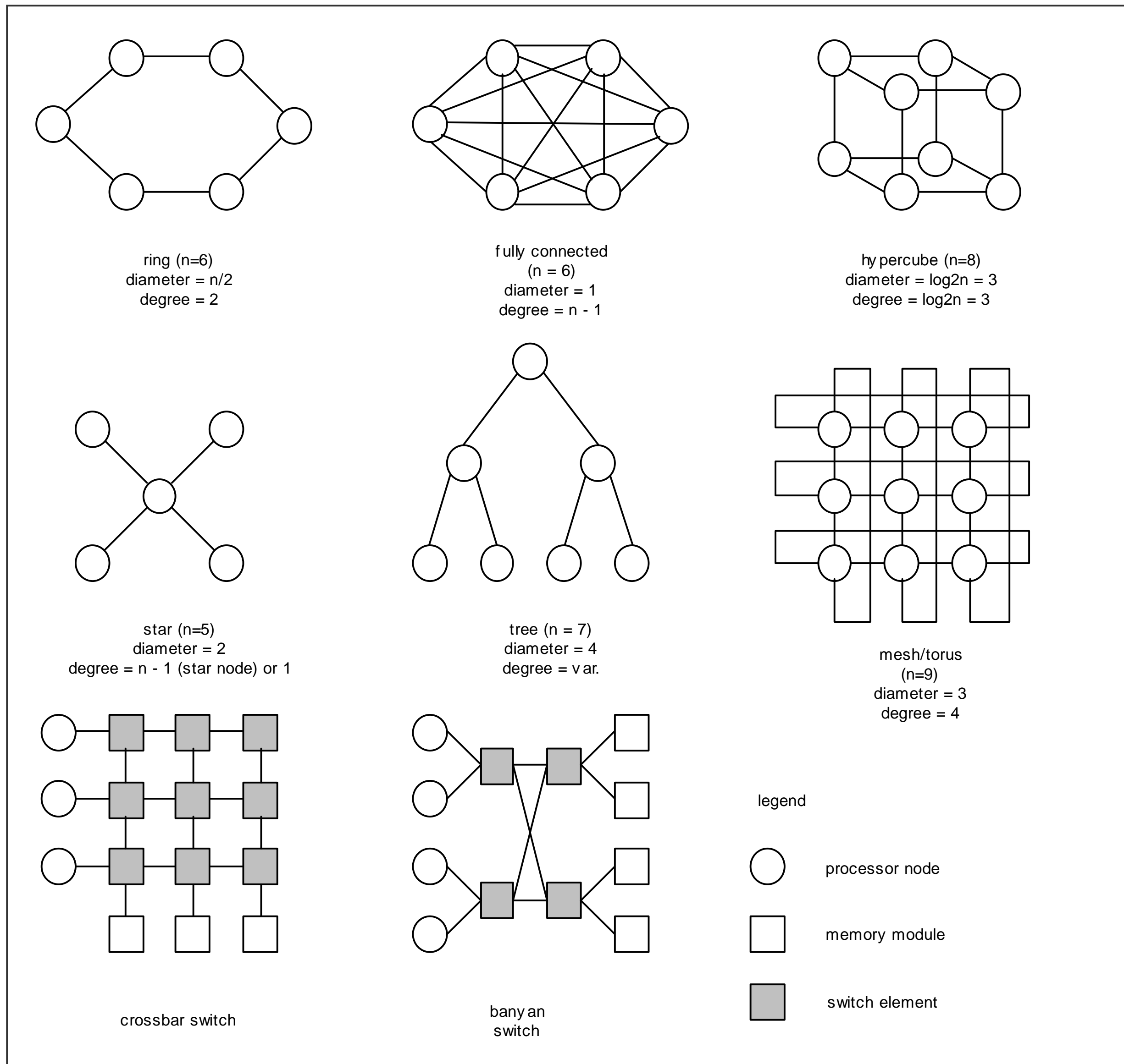


Figure 2-1 Network topologies

The *degree of a node* is defined to be the number of its neighbours. In a *ring topology* each node is connected to only two other nodes, while each node in a *fully connected network* is linked to every other node. In practice the degree of a topology has an effect on cost, since the more links a node has the more logic it takes to implement the connections.

When a node is not connected to every other node, messages may have to go through intervening nodes to reach their final destination. The *diameter of a network* is the longest path between any two nodes. Again the ring and fully connected network show two extremes. A ring of n nodes has diameter n/2, but a fully connected network has a fixed diameter (1) no matter how many nodes there are.

The diameter of a ring grows as more nodes are added, but the diameter of a fully connected network remains the same. On the other hand, a ring can expand indefinitely without changing the degree, but each time a new node is added to a fully connected network a link has to be added to each existing node. Scalability refers to the increase in the complexity of communication as more nodes are added. In a highly scalable topology more nodes can be added without severely increasing the amount of logic required to implement the topology and without increasing the diameter.

A scalable topology that has been used in several parallel processors is the *hypercube*. A communication link between two nodes defines a 1-dimensional "cube". A square with four nodes is a 2-dimensional cube, and a 3D cube has eight nodes. This pattern reveals a rule for constructing an n-dimensional cube: begin with a (n -1)-dimensional cube, make an identical copy, and add links from each node in the original to the corresponding node in the copy. Doubling the number of nodes in a hypercube increases the degree by only 1 link per node, and likewise increases the diameter by only 1.

Another desirable property of interconnection networks is *node symmetry*. A node symmetric network has no distinguished node, that is, the "view" of the rest of the network is the same from any node. Rings, fully connected networks, and hypercubes are all node symmetric. Tree and star topologies are not. A *tree topology* has three different types of nodes, namely a root node, interior nodes, and leaf nodes, each with a different degree. A *star topology* has a distinguished node in the centre that is connected to every other node. When a topology is node asymmetric a distinguished node can become a communications bottleneck.

Another common topology is a *planar (2D) mesh*. This network is basically a matrix of nodes, each with connections to its nearest neighbours. Meshes usually have "wraparound" connections, e.g. the node at the top of the grid has an "up" link that connects to the node at the bottom of the grid. A mesh topology with wraparound connections is often referred to as a *torus*.

The two final interconnection networks discussed in this section are examples of *multistage networks*. Systems built with these topologies have processors on one edge of the network, memories or processors on another edge, and a series of switching elements at the interior nodes. In order to send information from one edge to another, the interior switches are configured to form a path that connects nodes on the edges. The information then goes from the sending node, through one or more switches, and out to the receiving node. The size and number of interior nodes contributes to the path length for each communication, and there is often a "set-up time" involved when a message arrives at an interior node and the switch decides how to configure itself in order to pass the message through.

The first example of a multistage network is the crossbar switch. In a typical application there will be a column of processors on one side and a row of memories on the other side. The switch configures itself dynamically to connect a processor to a memory module. As long as each processor wants to communicate with a different memory there will be no contention. If two or more processors need to access the same memory, however, one will be blocked until the switch reconfigures itself. A crossbar has a short diameter - information needs to pass through only one switching element on a path from one edge to another - but poor scalability. If there are n processors and a like number of memories there are n^2 interior switches. Adding another processor and memory means adding another 2n - 1 interior nodes.

A *banyan network* is a multistage switching network that has the same number of inputs as outputs and interior nodes that are m x m switches. Examples of banyan networks are butterfly networks and omega networks, which are both built from 2 x 2 switches. The diameter of a butterfly is $\log_2 n$, where n is the number of inputs and outputs, and there are $O(n \times \log_2 n)$ switches, so these networks scale more efficiently than a crossbar.

2.6. Software synthesis

The process of transferring a functional specification, model or algorithm of a signal processing application into code that can be executed by the computer system is called software synthesis.

This process may take several intermediate steps from instruction level to functional specification level that may be executed manually or automatically. Assemblers and compilers perform the lowest level steps of the software synthesis. In general, the lower level steps are automated first. With parallel systems an extra difficulty is allocating parts of the code to different processing units.

On instruction level and to a lesser extent on program level compilers already exist to automatically allocate code to a specific processing unit. High Performance FORTRAN and Data Parallel C are examples of programming languages that support parallelism at task level, i.e. independent program sections.

To perform the software synthesis of a real-time application automatically it is crucial that the behaviour of the application can be defined in a formal way. A model of computation is a formalism that *exactly* specifies the behaviour of an application. The University of California at Berkeley has studied a lot of models (Process Networks, Discrete Event, Finite State Machine, Synchronous/Reactive, etc.) and has

developed a system engineering / rapid prototyping tool that supports these models: Ptolemy [8]. Data Flow is a sub formalism of Process Networks and is very appropriate to express parallelism and will be discussed in paragraph 2.6.2.

2.6.1. Standards

There are not many standards for programming parallel architectures. Over the last decade, parallel programming standards like Message Passing Interface (MPI) [9] and OpenMP [10] have evolved to support the portability of parallel programs.

MPI is intended for distributed memory architectures and is a library of functions and macros that can be used in FORTRAN, C, and C++. MPI incorporates a limited set of communication primitives (e.g. MPI_Send, MPI_Receive) and a set of reduction operations (MPI_Reduce, e.g. sum, product, min, max, logical or/and, bit wise or/and, etc.). The real-time extension of MPI, MPI-RT, seems a good candidate for the software synthesis of real-time signal processing applications.

OpenMP (MP stands for Multi Processing) is an API for writing parallel programs with portable shared memory parallelism and makes it easy to write portable multi-threaded programs in FORTRAN, C and C++. OpenMP is a set of compiler directives and library routines that, just like MPI, must be supported by the target platform.

Software synthesis of signal processing applications running on specific hardware architecture is performed automatically if the manufacturer of the architecture implements the MPI or OpenMP libraries. This is a tremendous exercise and will only be done by vendors of processing architectures.

Unfortunately, a commercial solution cannot always be used for different reasons. For example the heterogeneity with other hardware may be too limited or the harsh environment prohibits using a commercial solution. Sometimes a company doesn't want to be dependent on a vendor specific solution or cannot afford the hardware costs.

In this case alternatives have to be developed, like the subject of this thesis.

2.6.2. Data flow

Data flow (DF) is a formalism that describes a network of parallel processes or functions that communicate through unlimited First In First Out (FIFO)-buffers. A FIFO is a buffer where a message that is received first, is handled first. Data flow is a light synchronized parallel model with no notion of global time. The sequence of processes in the network is merely determined by data dependencies. It is an appropriate formalism to describe signal processing applications.

A data flow application can be represented by a program where data is transferred from one function to another by means of function calls. No control mechanisms (e.g. interrupts) are involved.

Data flow models are expressed in data flow diagrams, as shown in Figure 2-2.

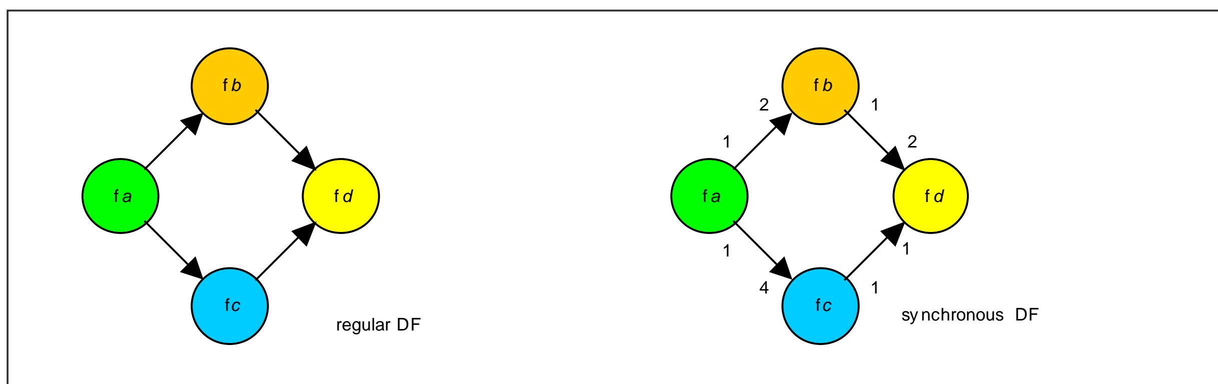


Figure 2-2 Data flow diagrams

A dataflow diagram contains actors that are connected by edges. The actors represent the (parallel) processes. The edges represent the FIFO-buffers where the arrow represents the direction of the data flow.

Data particles or tokens are communicated from one actor to another. When sufficient input tokens are present in the FIFO-buffers an actor " fires". At this moment, the input tokens are consumed and output tokens are produced. In case an actor has no input edges, an actor fires spontaneously.

At the regular dataflow formalism the number of consumed and produced tokens is variable. Without further specification, the token buffers are initially empty. Initial buffer tokens are often used to model delay in the network.

Synchronous Data Flow (SDF) is a further specialization of regular data flow [11]. At each network iteration the number of consumed and produced tokens at each actor is constant. An iteration of the network is defined as the minimum number of firings (greater than zero) after which the initial state of the buffers in the network is reached again.

When all tokens produced and consumed in the network are equal to one (or normalized to one), all actors in the network have an equal firing rate. This particular case is referred to as homogeneous SDF. When actors have different firing rates, the network is said to be multi-rate SDF.

Consider the right example in Figure 2-2. The numbers at the base of the arrow represent the number of tokens produced at each firing, the numbers at the head of the arrow represent the number of tokens consumed at each firing. With this simple example, it can easily be verified that at each network iteration function f_a fires four times, function f_b fires two times and function f_c and f_d fire one time. All the buffers have returned in their initial state after the network iteration.

When no network iteration exists, the application cannot be implemented with finite resources. Suppose f_d consumes 3 instead of 2 tokens at the upper edge. The buffer of the lower edge of f_d will never return in its initial state and will increase forever. Because of memory constraints on the FIFO-buffer, this situation is not feasible.

Checking the existence of a network iteration in a SDF-network is decidable according to [12]. For this reason the SDF-graph is represented by a matrix similar to the incidence matrix associated with directed graphs in the graph theory. It is constructed by numbering each node and arc and assigning a column to each node and a row to each arc. The (i,j)th entry in the matrix is the amount of data produced by node j on arc i each time it is invoked. If node j consumes data from arc i, the number is negative, and if it is not connected to arc i, then the number is zero.

If the functional names of the actors are allocated to a column (f_a =1; f_b = 2; f_c = 3; f_d = 4) and the arcs are allocated to a row (f_a -> f_b = 1; f_a -> f_c = 2; f_b -> f_d = 3; f_c -> f_d = 4), the matrix for the SDF-graph in Figure 2-2 is constructed as follows:

```
1  -2 0  0
1  0  -4 0
0  1  0  -2
0  0  1  -1
```

Given the existence of an iteration of an SDF-application we are able to determine a schedule how to distribute the processes across a multi-processor architecture. The schedule can be determined a priori, i.e. at compile time.

According to [1], many scheduling algorithms for SDF-applications exist. As scheduling is a NP-hard problem heuristics are applied to find a good schedule.

2.7. Common parallel processing techniques

In this paragraph some common parallel processing techniques are discussed that are used to increase throughput and decrease latency: pipelines, task parallelism and data parallelism. These techniques can be applied at all kinds of parallel levels as defined in 2.2.

First, the definition of iteration, iteration time, throughput and latency is given.

2.7.1. Latency and throughput

Consider an endlessly iterated program with four functions, f_a to f_d, in Figure 2-3.The program processes a continuous data stream and is executed on a processing architecture with 6 processors (P_1 to P_6). The lines between the processors represent the communication links.

In this first situation, the program is executed on a single processing unit (P1). Each program execution is called an iteration.

In this example we assume that the data particles to be consumed by f_a are produced equidistant in time. The iteration time of each function is depicted in the simple Gantt chart below. The colours of the execution time correspond with the function colours. The width of a square box is a single time unit. The execution time of function f_a and f_b is 2 units; the execution time of function f_c and f_d is 1 unit. The grey boxes represent the (time equidistant) input data.

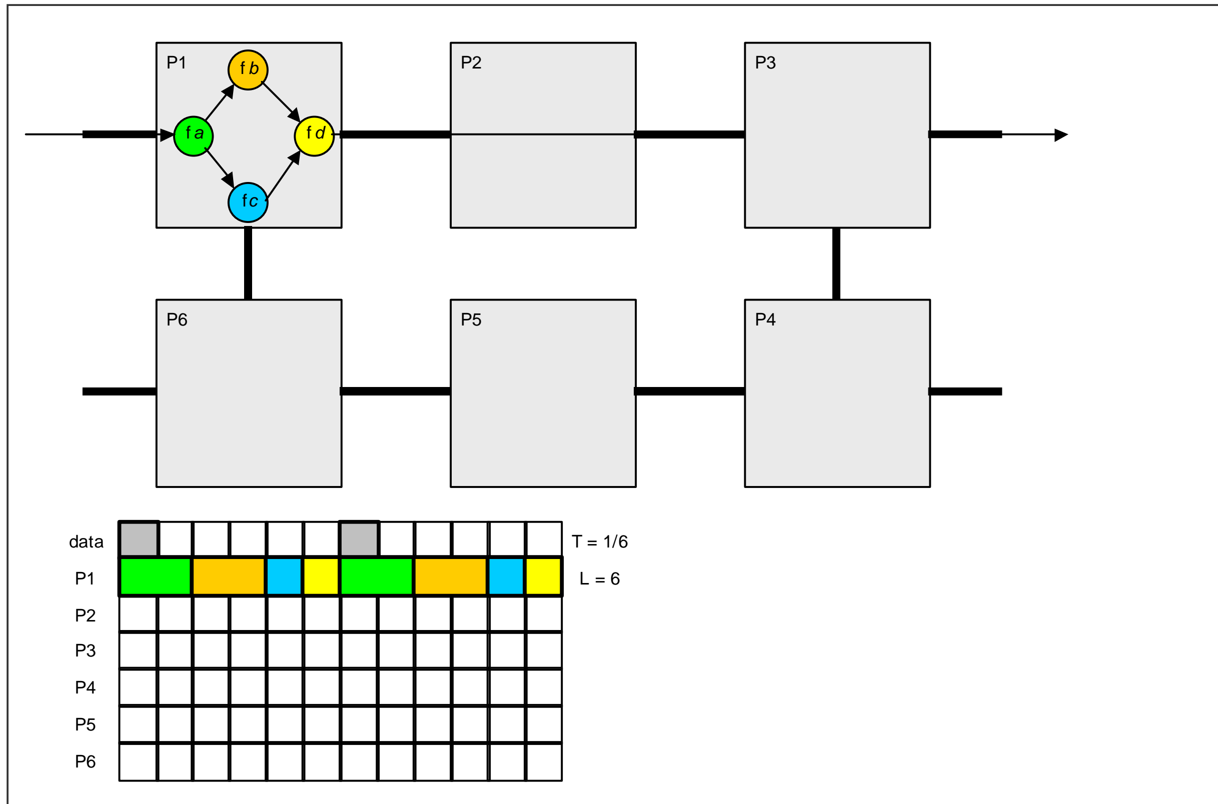


Figure 2-3 Definition latency and throughput

Latency (L) is defined as the time it takes to finish a single execution from the moment the input data is available until output data is produced.

Throughput (T) is defined as the number of iterations per time unit.

In the example in Figure 2-3 the throughput is equal to 1/6 and latency is equal to 6. From the Gantt chart we can clearly see that the usage of the processing resources is far from optimal.

2.7.2. Pipeline

Pipelines are used to increase throughput if resources are available. Latency is not reduced with a pipeline. A common analogy for a pipeline is the assembly line used in manufacturing. The end goal is to increase productivity - the number of instructions executed per second or the number of cars built per day - by dividing a complex operation into pieces that can be performed in parallel. Separate "workers" implement successive steps along the assembly line, and when an item finishes one step it is passed down the line to the next step.

According to [1], the following requirements must be satisfied in a pipelined system:

- A system is a candidate for pipelined implementation if it repeatedly executes a basic function.
- A basic function must be divisible into independent stages that have minimal overlap.
- The complexity of the stages should be roughly similar

The example from Figure 2-3 fulfils the above-mentioned criteria. In the second situation, the program from Figure 2-3 is split up into 3 independent sequential stages and distributed across 3 processors. The

mapping and the results are shown in Figure 2-4. In the corresponding Gantt chart we can see that in this second situation the throughput has doubled (T = 1/4), but that the total latency has not decreased (L = 6).

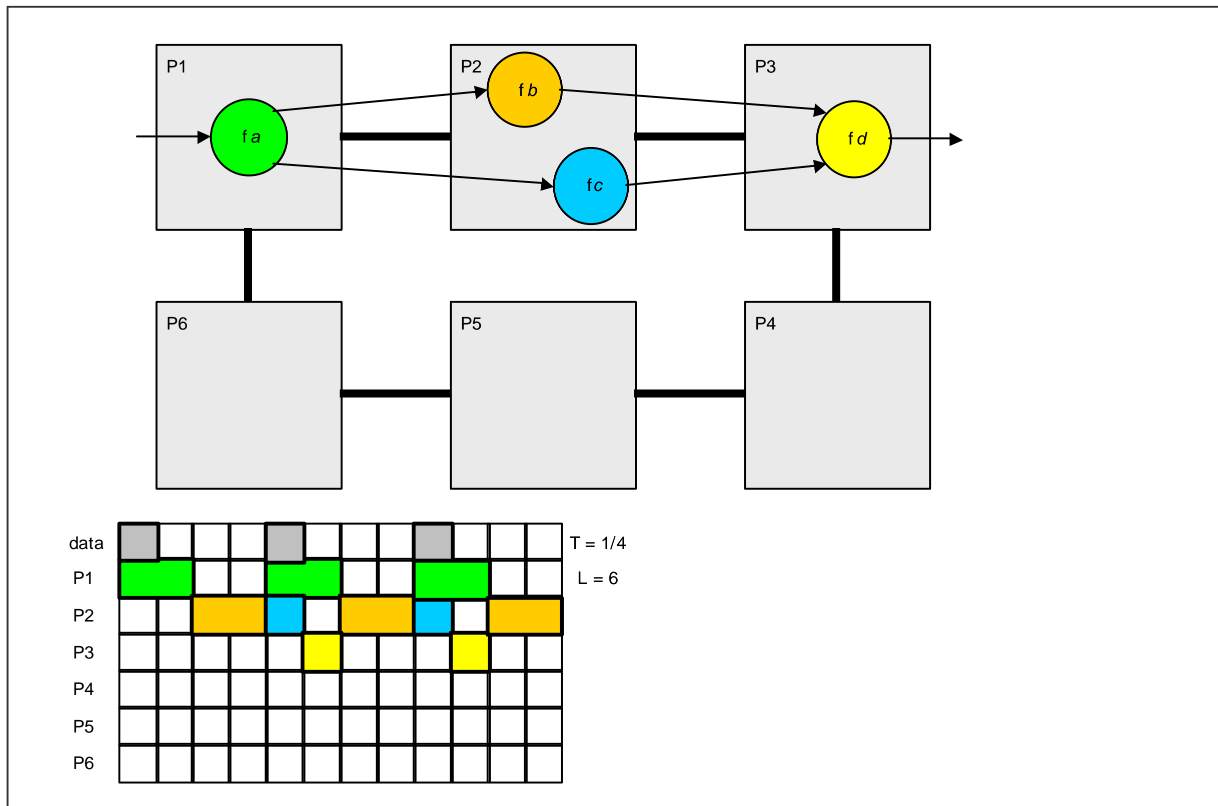


Figure 2-4 Increased throughput with a pipeline

2.7.3. Task parallelism

Task parallelism is defined as tasks or functions that operate on different data sets. The data sets are kept intact and each dataset is processed by a unique task. In the third situation, consider the same program as in Figure 2-4. In Figure 2-5 function f_c is mapped on processor P_5.

The Gantt chart of Figure 2-5 shows that, in addition to the pipeline, function f_b and f_c are performed in parallel. When communication overhead is neglected (which is normally not the case) it can be verified that latency in this third situation has been reduced (L = 5) and that throughput has been quadrupled (T = 1/2) compared to the first situation.

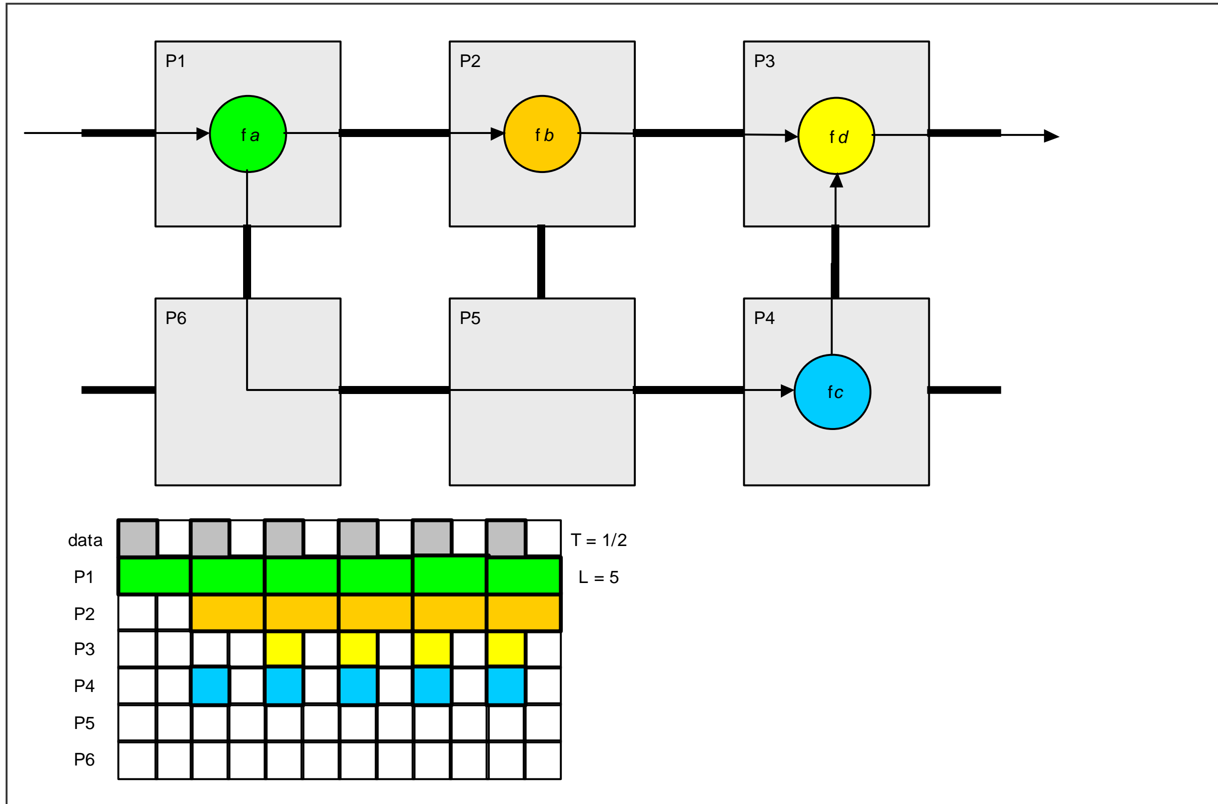


Figure 2-5 Task parallelism

2.7.4. Data Parallelism

Data parallelism is defined as identical functions that operate on different parts of the same data set. In Figure 2-6 we assume that the data is not atomic and we arbitrarily choose function f_a and f_b to be instantiated twice. In this last situation, each instantiation is mapped on a different processor. Half of the original data that was sent to f_a is distributed to $f_{a,1}$ and the other half to $f_{a,2}$. The same distribution is applicable for function f_b. The processing time for $f_{a,1}$, $f_{a,2}$, $f_{b,1}$ and $f_{b,2}$ has also halved.

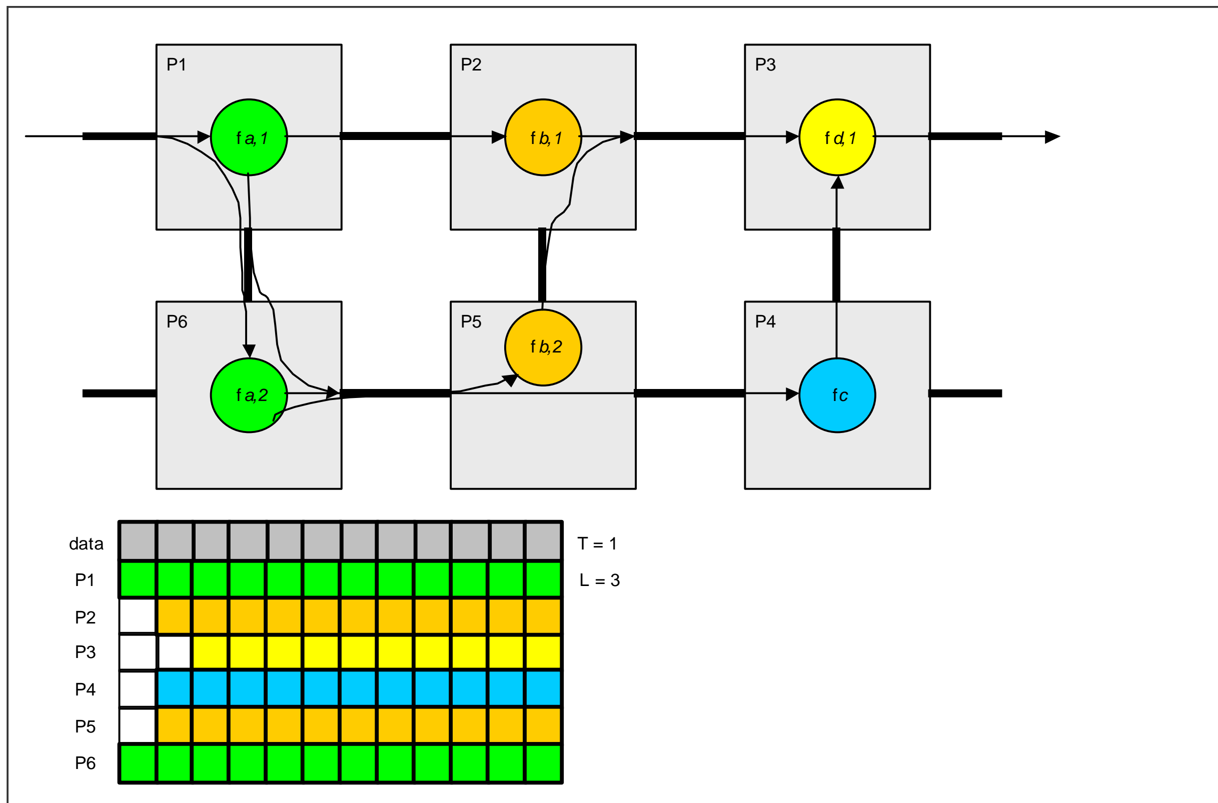


Figure 2-6 Data parallelism

As one can verify in the Gantt chart, the processing resources are used to their maximum. Of course, the distribution and aggregation of data is not costless and will introduce overhead. Data has to be routed through the architecture. When the communication overhead is neglected, it can be verified that the latency has been further reduced (L = 3), and that throughput has been increased (T = 1).

In program code data parallelism can often be discovered in iteration loops. In nested iteration loops the data parallelism can be even greater.

2.8. Expansion and scheduling

The scope of this thesis is in the field of real-time multiprocessor signal processing applications. Typical for these kinds of applications is that the number of functions is relatively small compared to the number of processors in the architecture, as opposed to data processing applications where the number of functions is relatively large compared to the number of processors in the architecture.

We know that every function demands processing and communication bandwidth. To achieve a solution with minimum latency and maximum throughput we generally try to use all processors in the architecture.

In the case that more functions than processors exists, functions must be clustered to achieve a solution with respect to maximum network throughput and minimum network latency. Different clustering algorithms exist to realize this. Earlier research at Thales has been performed on this subject and is described in [1].

In the case that less functions than processors exists, the granularity of the function network can be raised if data parallelism exists. In this situation multiple instances of a function are created, where each instantiation of a function operates on a different part of the data set. Functions that expose data parallelism and contribute the largest part of the latency should be selected first. The granularity of the function network is to be increased to the granularity of the processor architecture. In this thesis the increase of function network granularity is called expanding.

The concept of clustering versus expanding is visualised in Figure 2-7: a simple functional network with four non-identical functions is mapped on a processor-architecture with respectively two and six processors. In the first situation functions are clustered and scheduled on the two processors. In the second situation, the functional network is first expanded by duplicating function f_a and f_d and then scheduled on six processors.

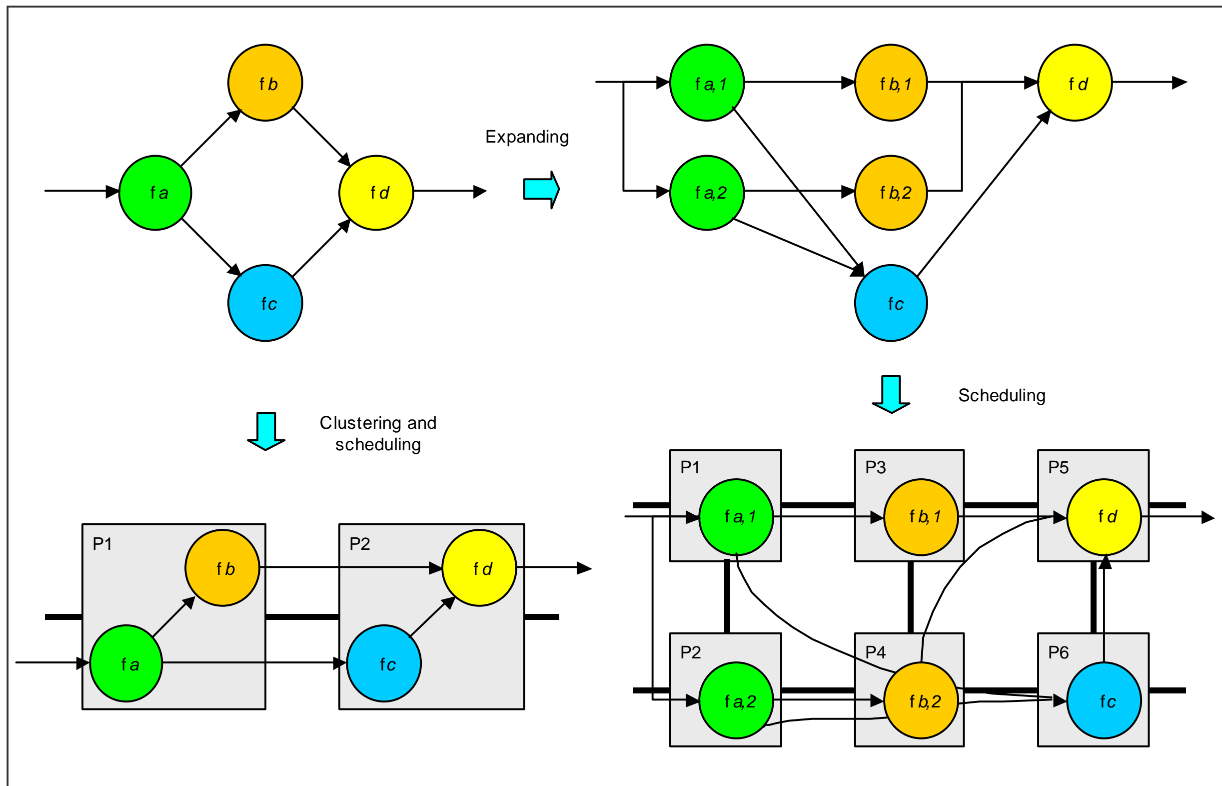


Figure 2-7 Mapping of SDF-network on a physical multiprocessor architecture

In fact, network expansion is all about exploiting the data parallelism in the function network. The scheduler exploits the task parallelism to reduce latency and can create pipelines to increase throughput.

Of course, expansion is not without penalties. Data must be routed to and from each processor. Input data of function f_a must be distributed to processor P_1 and P_2, and output data of function f_a must be aggregated and sent to processor P_3 and P_4, etc. The extra overhead and scalability of the distribution and aggregation depends on the memory architecture of the processor architecture.

3. Problem definition

In chapter 1 of this thesis a three-layered method is defined for the software synthesis of functional specifications of signal processing applications for multiprocessor architectures. The top layer will exploit the data parallelism in the functional specification, the middle layer will schedule the expanded function network and the bottom layer will generate the target code for the hardware architecture.

The main goal of this thesis is to implement the hardware independent top layer, the pre-processor of the scheduler.

In paragraph 2.6.2 it was explained that data flow, and especially synchronous data flow, is an appropriate formalism to specify signal processing applications. Task parallelism can be expressed very well and to some extend data parallelism can be explored in the SDF-diagram, as will be explained in paragraph 4.1. As data particles are considered atomic, nothing is said about the dimensions and memory organisation of the data. When this knowledge is taken into account, it is expected that data parallelism can be further exploited.

Another aspect is the modelling of characteristics that determine the latency in signal processing applications: the number of operations to be performed by each function and communication delay, especially with distributed memory organisations.

Taking the modelling aspects mentioned above into account (data dimensions, number of operations) the SDF-formalism is not detailed enough and therefore, *a formalism in the form of a specification language will be defined in this thesis based on the synchronous data flow formalism*.

Thesis questions to be answered are:

Q1 How should the nodes and data in the extended SDF-formalism be modelled or specified?

Q2 How should latency parameters like number of operations and communication delay be modelled without hardware knowledge?

Q3 How will the top layer be tested?

Q4 How can task parallelism be exploited in the most optimal way for the middle layer, the scheduler? In other words: how can the best granularity of the functional model be found?

Q5 How practical is the approach of the hardware independent top layer?

Questions Q1 and Q2 will be investigated in chapter 4 - Hardware independent modelling formalism.
The design of the compiler will be discussed in chapter 5 - Network Expansion Compiler.
Questions Q3 and Q4 will be considered in chapter 6 - Results.
Question Q5 will be answered in chapter 7 - Conclusions and recommendations.

4. Hardware independent modelling formalism

In this chapter a hardware independent modelling formalism will be defined to specify synchronous dataflow (SDF) applications. The major difference with the standard SDF-formalism is the modelling of data dimensions to exploit the data parallelism in the function network. A specification language will be defined to specify the functional models and a graphical notation is defined for visualisation purposes.

The specification language shows a lot of commonality with regular imperative languages, but the expressive power of this specification language is limited to SDF-applications. In fact, the language supports merely the assignment statements and iteration loops. Conditional statements (if-else, while, etc.) are not supported.
The reason for the definition of a specific language is that the data dimension parameters will be used to exploit the data parallelism in the model to raise the function granularity. No other languages support the dimension parameters.

In paragraph 4.1 a global overview on the modelling aspects is given by means of a typical radar SP-application. The major differences with the SDF-formalism will be explained.
Paragraph 4.2 deals with the details of the specification language. Each section of the specification language will be explained.
Paragraph 4.3 explains the rules for the execution rate of the functions in the network.
The last paragraph of this chapter elaborates on the distribution and aggregation functions when the network is expanded.

4.1. Modelling aspects

The data flow formalism (DF) can be used to describe small grain programs (e.g. instruction level) up to large grain programs (e.g. job level). In this thesis we consider the granularity at task level (medium granularity), i.e. the program is broken down into independent sections, functions or tasks that communicate with each other through data paths.

Without a global notion of time, DF can express precedence of functions very well. Another strong feature of DF is the expression of task parallelism (operations that are executed on different data sets).

In DF nothing is said about the dimensions or memory organisation of the data particles that flow along the data paths. For small grain DF it may be sufficient to consider the data particles as atomic (undividable), but for medium and large grain DF this is often not the case.

In Synchronous Data Flow (SDF) diagrams data parallelism can be exploited to some extend and this will be discussed next.

Consider a typical radar signal processing (SP) application in Figure 4-1 that will be used throughout this thesis. The application is specified in a DF diagram. The triangles represent the input sources and output sinks.

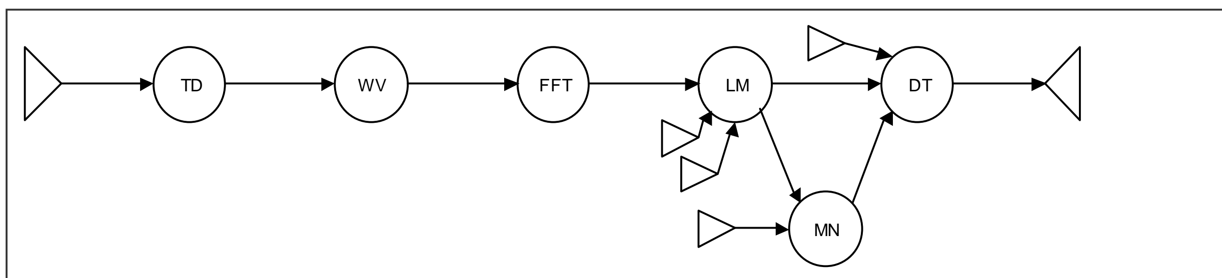


Figure 4-1 DF-diagram of typical radar SP-application

The application itself will not be discussed in detail here. For the reader it is sufficient to have a global notion of the radar video data and what kind of operations the functions perform on the radar video data.

The radar video in this example is a continuous data stream of burst data, a two-dimensional matrix that contains the sampled echoes of the radar transmissions in a particular direction. Multiple transmissions (called sweeps) are sent in this particular direction and the echo of each transmission is sampled during a certain time. The samples in the sweep are range cells that represent a certain range in distance. The first

range cell in a sweep represents the nearest distance to the radar. To say something about the speed of possible objects, the frequency spectrum across the range cells with identical range in the burst data is analysed.

The actors in the diagram have the following functions:

- **TC (Turn Corner)**: transposes the range and sweep data in memory. All range cells with the same range in the burst data will be written consecutively in memory.
- **WV (Weigh Video)**: performs a weighing function on the range data
- **FFT (Fast Fourier Transformation)**: performs a FFT function on the identical range data
- **LM (Logarithmic Modulus)**: Calculates the logarithmic modulus (logmod) of the frequency bins generated by the FFT-function.
- **MN (Mean Normalization)**: estimates the background level of a logmod video from the neighbouring cells.
- **DT (Detect Target)**: detects objects or targets in the logmod video when the value exceeds a certain threshold level compared to the background level of the range cell.

If the burst data is considered as an atomic particle we can define the homogeneous SDF-diagram illustrated in Figure 4-2 (if no number of tokens is specified at the arrow, it is equal to 1).

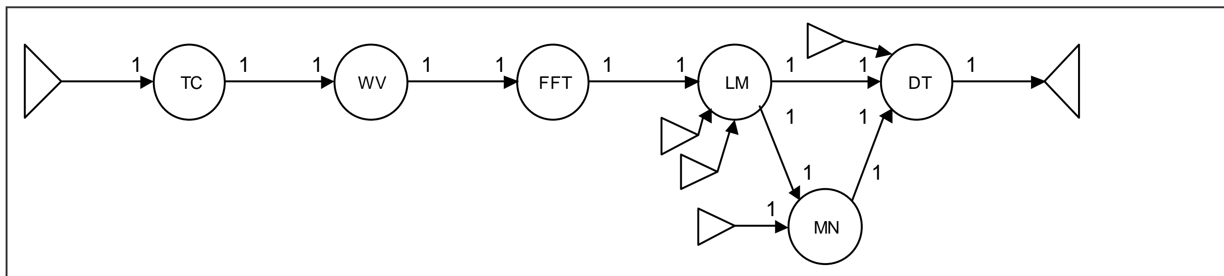


Figure 4-2 Homogeneous SDF-diagram of typical radar SP-application

The execution rate of all functions is equal to 1 and no data parallelism can be revealed.

From the description of the burst data as described before, one can determine that the data is not atomic. With this knowledge we can construct the multi-rate SDF-diagram illustrated in Figure 4-3.

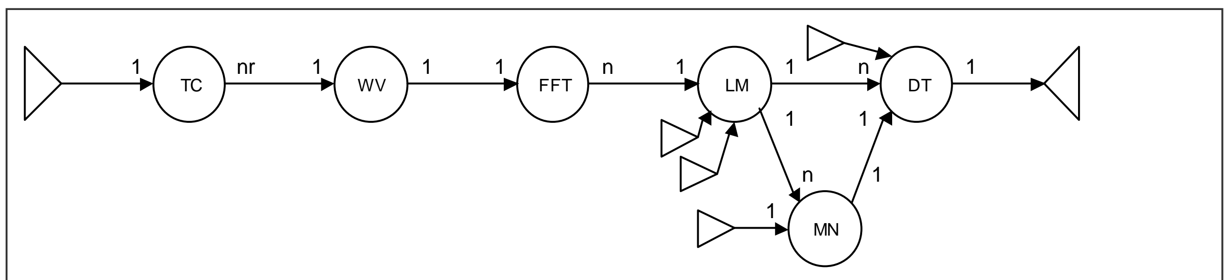


Figure 4-3 Multi-rate SDF-diagram of typical radar application

The TC-function produces nr (**n**umber of **r**ange cells) data particles that represent vectors of range cells with identical range. The WV-function weighs the data elements in the vector and passes this vector to the FFT-function. The FFT-function performs a n-points FFT on the range data and generates n frequency bins. The number of sweeps is normally equal to the number of frequency bins. For each frequency bin the LM-function calculates the logmod value. Finally, for each range the DT-function evaluates the logmod video if a target can be detected above a certain threshold level, calculated by the MN-function.

The execution (or firing) rate of the different functions is decidable and can be determined from the diagram: TC = 1, WV = nr, FFT = nr, LM = nr * n, MN = nr, DT = nr.

The multi-rate character of the SDF-diagram reveals data parallelism. The data parallelism is exploited to a certain extent as is illustrated in the SDF-diagram of Figure 4-4. In this particular case, the output of function TC was arbitrarily split up in two parts, but could have been split up in *nr* parts. The output of function FFT was also split up in two parts, where it could also have been split up in *n* parts.

As can be seen the granularity of the diagram has been increased.

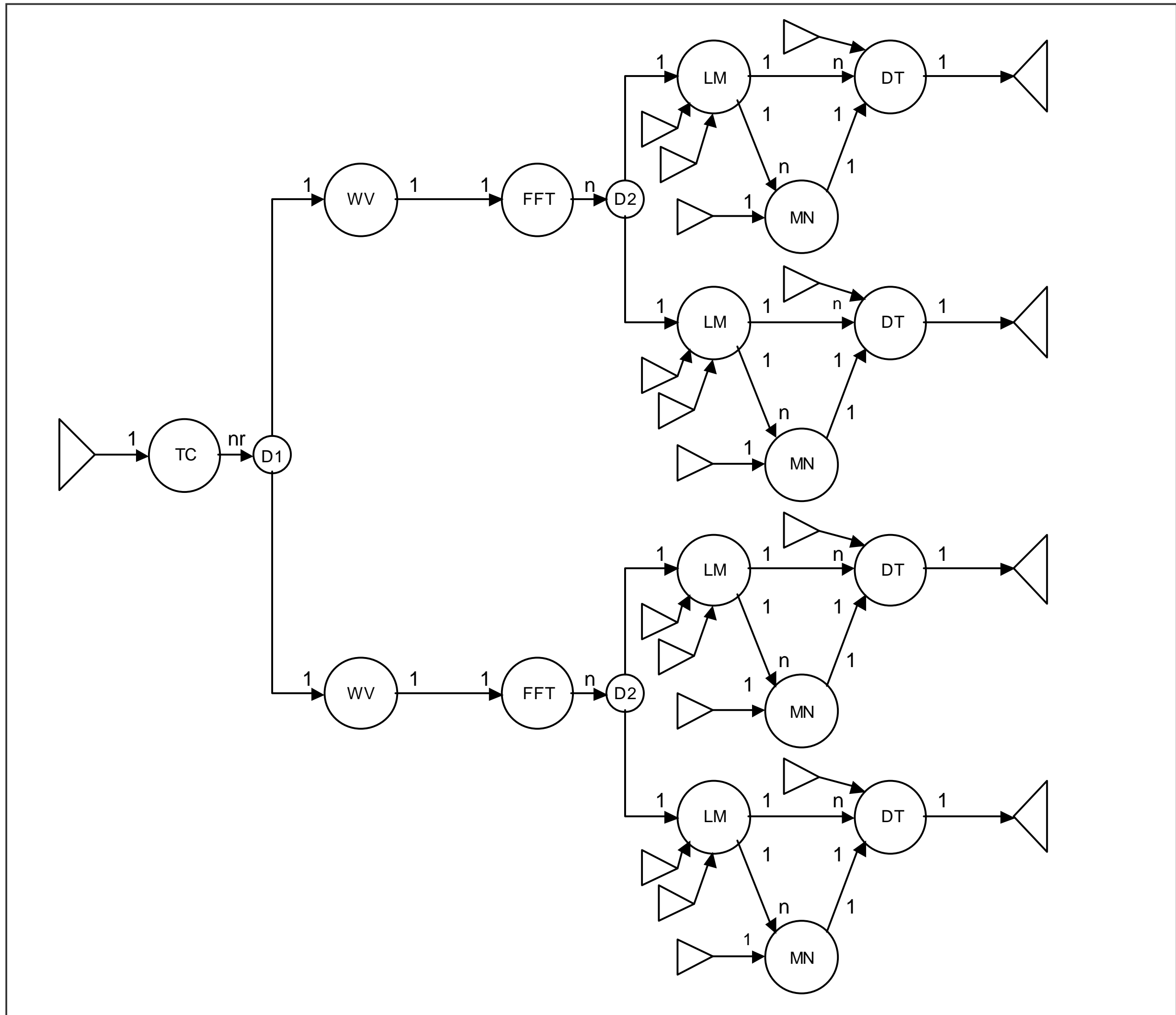


Figure 4-4 Multi-rate SDF-diagram with exploited data parallelism

Distributor nodes D1 and D2 have been added to the diagram to leave the original functions intact and not to worry how the data particles should be distributed.
Suppose the distributor nodes distribute the data particles equally to the successor nodes, than the execution rate of the WV- and FFT-functions has been reduced with a factor 2 and the execution rate of the LM-, DT-, and MN-functions has been reduced with a factor 4.

Although data parallelism can be exploited in multi-rate SDF-diagrams, the type, dimensions and memory organisation of the data cannot be revealed. To implement the functions without knowledge of their context, it is essential to specify the memory organisation (by means of data dimensions).

From the previous examples one may conclude that to exploit data parallelism in data flow specifications it is essential to have knowledge about the data particles (e.g. the burst data). In the thesis this is addressed by the specification of the data interface of the nodes and data particles. When the data parallelism is exploited, the data particles will also act as distributor or aggregator nodes in the network (see paragraph 4.4).

In Figure 4-5, a first impression of the extended dataflow diagram is given for the typical radar SP-application. The blue printed text will be explained in paragraph 4.3 about the execution rate.

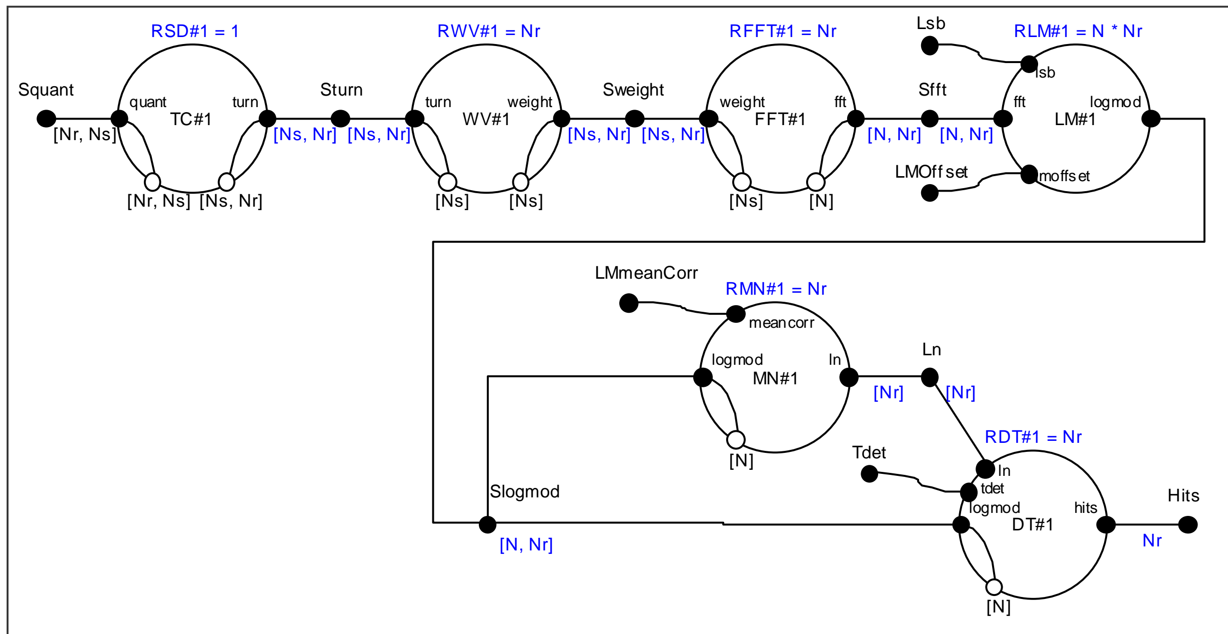


Figure 4-5 Extended dataflow diagram of the typical radar SP-application

To process the extended graph automatically, it will be represented by a specification language. Furthermore, the specification language contains more details as the graphical notation. The graph would become unreadable if all details were represented.

The specifications language and the graphical notation will be explained in detail in the next paragraph.

4.2. Specification language of function networks

When a synchronous data flow graph needs to be processed automatically it is often represented by an incidence matrix as described in 2.6.2. To process the extended graph automatically, an incidence matrix carries too few parameters to support the necessary details for the pre-processor (e.g. the data dimensions).

This is the reason why the extended data flow graphs are represented by a specification language in this thesis. Another advantage of the specification language is that it contains more details than the graphical notation.

In the previous paragraph the graphical notation of the functional model was introduced by means of the radar SP-application example. The text specification of this example is given below. It doesn't have to be studied in detail, it will be explained after the example code.

```
NETWORK demo

/* type definitions */

TYPES:

QUANT_VIDEO = 2 * float;
CORNER_TURN_VIDEO = 2 * float;
WEIGHTED_VIDEO = 2 * float;
FFT_VIDEO = 2 * float;
LSB = float;
LOGMOD_VIDEO = float;
LOGMOD_OFFSET = float;
LM_MEAN_CORRECTION = float;
LN = float;
THRESHOLD_DET = float;
HITS = (float + 2 * float + float);

/* dimension declarations */

DIMENSIONS:

Nr = 1024 : 2 : 1;
Ns = 16 : 2;
N = 16 : 2;

/* data declarations */

INPUTS:

QUANT_VIDEO Squant[Nr, Ns];
LOGMOD_OFFSET LMOffset;
LSB Lsb;
LM_MEAN_CORRECTION LMMeanCorr;
THRESHOLD_DET Tdet;

OUTPUTS:

HITS Hits;

LOCALS:

CORNER_TURN_VIDEO Sturn;
WEIGHTED_VIDEO Sweight;
FFT_VIDEO Sfft;
LOGMOD_VIDEO Slogmod;
LN Ln;

/* used functions */

FUNCTIONS:

TC
(
    INPUTS:
        QUANT_VIDEO quant[Nr, Ns];
    OUTPUTS:
```

```
        CORNER_TURN_VIDEO turn[Ns, Nr];
) : OPS ( 4 * Ns * Nr );

WV
(
    INPUTS:
   CORNER_TURN_VIDEO turn[Ns];
    OUTPUTS:
   WEIGHTED_VIDEO weight[Ns];
) : OPS ( Ns );

FFT
(
    INPUTS:
        WEIGHTED_VIDEO weight[Ns];
    OUTPUTS:
        FFT_VIDEO fft[Ns];
) : OPS ( Ns * log( Ns ) );

LM
(
    INPUTS:
   FFT_VIDEO fft,
        LSB lsb,
   LOGMOD_OFFSET lmoffset;
    OUTPUTS:
        LOGMOD_VIDEO logmod;
) : OPS ( 100.0 );

MN
(
    INPUTS:
   LOGMOD_VIDEO logmod[Ns],
   LM_MEAN_CORRECTION meancorr;
    OUTPUTS:
   LN ln;
) : OPS ( Ns );

DT
(
    INPUTS:
        LN ln,
   LOGMOD_VIDEO logmod,
        THRESHOLD_DET tdet;
    OUTPUTS:
        HITS hits;
) : OPS( 100 );


INTERCONNECTIONS:

/* TurnCorner */

Squant -> TC#1(quant[Nr, Ns]);
TC#1(turn[Ns, Nr]) -> Sturn;

/* WeightFFT */

Sturn -> WV#1(turn[Ns]);
WV#1(weight[Ns]) -> Sweight;

/* FFTVideo */

Sweight -> FFT#1(weight[Ns]);
FFT#1(fft[N]) -> Sfft;

/* LogMod */

Sfft -> LM#1(fft);
LMOffset -> LM#1(lmoffset);
Lsb -> LM#1(lsb);
LM#1(logmod) -> Slogmod;

/* MeasureNoise */
```

```
Slogmod -> MN#1(logmod[N]);
LMMeanCorr -> MN#1(meancorr);
MN#1(ln) -> Ln;

/* DetectTarget */

Slogmod -> DT#1(logmod[N]);
Ln -> DT#1(ln);
Tdet -> DT#1(tdet);
DT#1(hits) -> Hits;
```

A quick look at the radar example above shows that the specification is divided in 5 global parts: type definitions (TYPES), dimension definitions (DIMENSIONS), data declarations (INPUTS, OUTPUTS, LOCALS), function specifications (FUNCTIONS) and the interconnections (INTERCONNECTIONS).

The specification format was derived from the input specification of the scheduler. The new elements are: DIMENSIONS, INPUTS, OUTPUTS and LOCALS. Furthermore, the syntax of the FUNCTIONS and INTERCONNECTIONS part was modified. The main reason was to model the data and their dimensions.

In the next sub paragraphs the details of these sections will be discussed, but first a global overview on the specification language will be given hereafter.

The functional model is a function network description where functions are interconnected by data elements. The interconnections are defined in the INTERCONNECTIONS section of the function network specification.

The data elements are specified in the INPUTS, OUTPUTS and LOCALS sections. Data elements have a type definition declared in the TYPES section. Data elements and types can have dimension parameters that define the size of the dimensions and whether the dimensions can be split up in smaller parts. The dimensions are declared in the DIMENSIONS section.

In the software synthesis method defined in Figure 1-3, the interface of the functions is pre-defined in a library and is re-used in the function network specification. Types and dimensions of the data interfaces of the inputs and outputs are part of the function interface specification. The function interfaces or *formal functions* used in the network are specified in the FUNCTIONS section of the function network specification. Also part of the formal function specification is the number of operations (OPS) parameter. It is one of the thesis questions (Q2) that needs to be answered and will be described in 4.2.5.

Next, the sections of the specification language will be discussed in detail. As can be seen in the example, comments are enclosed between /* and */ markers.

The exact syntax of the language is specified in Backus-Naur Form (BNF), but will not be discussed in detail in this thesis. Where applicable, part of the BNF specification will be used to explain certain examples. For the sake of completeness the BNF-specification is given in APPENDIX A, paragraph A.1.

4.2.1. Network definition

The function network specification starts with the NETWORK keyword followed by the network name (fN in this case). One can imagine that the network may be represented as a function with the same name on a higher hierarchical level, but in this thesis no network hierarchy is assumed. The functions are said to be *primitive*, i.e. they cannot be decomposed into smaller functions in the functional model.

4.2.2. Types section

Data elements and the input and output connections of a function have data types that are declared in the type declaration section that starts with the keyword TYPES. A data type is declared by means of a unique identifier and a type expression. The main purpose of the type specification is to enable type checking and to determine the memory size and organisation of the data.

The type expression is a construction of floats and integers according to the following BNF-specification

```
TypeExpression:
    Factor
    | Factor '+' TypeExpression

Factor:
```

```
     NUMBER '*' Factor
     | Term

Term:
     IDENTIFIER
     | '(' TypeExpression ')'
```

Examples

```
QUANT_VIDEO = 2 * float;
TYPE1 = (integer + 2 * float);
```

Note that the terms float and integer are not reserved key words.

The type information is not shown in the graphical representation.

4.2.3. Dimensions section

The dimension section starts with the keyword DIMENSIONS: each declaration defines the dimension parameters of a global dimension. Global implies that it has a global scope in the network specification and that it can be used throughout the network specification.

A dimension is declared by means of a unique identifier and a dimension expression. The dimension expression contains three parameters separated by colons. The first parameter specifies the size of a vector. The second value specifies the number of segments in which the vector may be subdivided. The third parameter specifies the number of elements that consecutive segments must have in overlap when the segments are distributed to separate functions. The overlap property is used quite often in SP-applications when streaming data must be cut into segments, e.g. for performance reasons. To reduce the boundary effects the neighbour elements are taken into account.

The value of the parameter specifies how many elements must be taken into account. When the third value is omitted, the distribution overlap is assumed to be zero.

In Figure 4-6 the distribution overlap is visualized. Dimension D1 specifies a vector of 8 elements that is subdivided in 4 parts of two elements (as shown by the four different colours). The overlap parameter specifies the number of elements at each neighbouring side of a part that should be taken into account when the part is processed. The total length of the part and neighbouring elements are called segments. At the boundaries of the vector no overlap element is available. Therefore, the first and last segment will have a shorter length. In the example the length of the segments is 4 and 3.

Examples

```
DIMENSIONS:
Nr = 8:2;
D1 = 8:4:1;
```

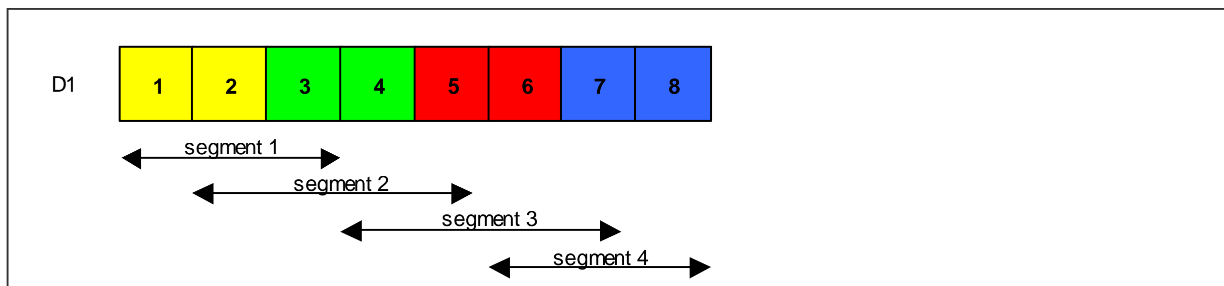


Figure 4-6 Distribution overlap

The dimension identifiers are specified at the function symbol and near the data elements. The values of dimensions are not represented in the graphical representation.

4.2.4. Data element sections

The data elements in the function network are declared in three sub-sections: INPUTS, OUTPUTS and LOCALS. Input data elements define the start of the data flow within the network, output data elements define the end. Local data elements have predecessor and successor functions in the data flow network.

A data element is declared by means of a type and a unique identifier. In the graphical notation a data element is a black dot connected to a function input or output. For example, data elements in Figure 4-5 are *Squant*, *Sturn*, *Sweight*, etc.
The data elements and their dimensions determine the context of the function. From the context the execution rate of the function is determined. This will be explained in paragraph 4.3.

The type of data elements is not represented in the graphical representation.

Examples

```
INPUTS:

QUANT_VIDEO Squant[Nr, Ns];
LOGMOD_OFFSET LMOffset;
LSB Lsb;
LM_MEAN_CORRECTION LMMeanCorr;
THRESHOLD_DET Tdet[N, Nr];

OUTPUTS:

HITS Hits;

LOCALS:

CORNER_TURN_VIDEO Sturn;
WEIGHTED_VIDEO Sweight;
FFT_VIDEO Sfft;
LOGMOD_VIDEO Slogmod;
LN Ln;
```

A remark on the data elements is appropriate here. The target code generator will implement the data elements as distributor or aggregator functions. In a shared memory situation it will be a memory location and introduces no extra communication overhead. On distributed memory architectures it will be implemented as a send- or receive primitive or function and introduces extra latency. This will be further discussed in paragraph 4.4.

4.2.5. Functions section

The FUNCTIONS section declares the *formal functions*, or function interfaces, used in the function network. The formal function specifies the interfaces of a function and the number of operations; its internal behaviour is determined by the implementation of the function. The formal function is instantiated in the INTERCONNECTION section. An instantiation of a formal function is called an *actual function*.

Below the example of the TC-function is given.

```
TC
(
    INPUTS:
        QUANT_VIDEO quant[Nr, Ns];
    OUTPUTS:
        CORNER_TURN_VIDEO turn[Ns, Nr];
) : OPS (4 * Ns * Nr);
```

The given layout is arbitrary; the declaration can also be specified on a single line.

Connections

The data interface of a function is defined by *connections*. A connection is the specification of an input or output of the function. The type and dimensions of the connection are also specified. The order of the dimension identifiers specifies the memory organisation of the data. If a connection has no dimensions, the data interface is considered as a scalar.

Dimensions

The dimensions of the inputs and outputs in the formal function are instantiated by the global dimensions as defined in the network specification (in this example Nr resp. Ns). One has to realize that the implementation of the function has to deal with variable dimension parameters.

Number of operations

The formal function also specifies a "number of operations" (OPS) parameter. It specifies the number of operations on the basis of the dimension parameters.

One of the questions to be answered in this thesis (Q2) is how latency determinant characteristics can be modelled in a hardware independent way.

The idea is to use the order of computational complexity known from the complexity theory. It says that if n is the size of the input, the complexity to solve the problem can be expressed in n. An exact expression can be used (e.g. $n^2 - 3n + 7$) or an order of complexity, where the most dominant term will be used to express the complexity when n becomes very large (e.g. n^2). The "big O" notation is often used to express the order of complexity. Common orders of complexity are $O(1)$, $O(n)$, $O(n^2)$, $O(n^3)$, $O(^{n4})$, $O(2^n)$, $O(\log.n)$, $O(n.\log.n)$.

The BNF-syntax of the OPS parameter is as follows:

```
Operations:
    OPS '(' DimensionExpression ')' ;

DimensionExpression:
    MathExpression
    | DimensionExpression '*' DimensionExpression
    | DimensionExpression '/' DimensionExpression
    | DimensionExpression '+' DimensionExpression
    | DimensionExpression '-' DimensionExpression
    | '(' DimensionExpression ')'
;

MathExpression:
    NUMBER
    | FLOAT
    | DimName
    |  LOG '(' DimName ')'
    |  POW '(' NUMBER ',' DimName ')'
    |  POW '(' DimName ',' NUMBER ')' ;
```

Examples

```
1
2.5 * N
N * N + 2 * N * log(N)
pow(2, N)
pow(N, 3)
(N + M) / (N * M)
```

Graphical notation of functions

The graphical notation was defined in this thesis as an extension to the regular SDF-diagrams. A function in an SDF-diagram is denoted as a circle. In the extended graphical notation, the name of the function is specified in the circle with a suffix to indicate the instantiation number (e.g. #1). Inputs of a function appear per default on the left half of the circumference and are denoted as a black dot. Outputs of a function are also denoted as a black dot and appear per default on the right half of the circumference.

Dimensions of connections are defined by a white dot on the circumference. Multi-dimensional data is specified by a sequence of dimension identifiers separated by commas and enclosed by brackets.

The type specification and OPS parameter are not visualised in the graphical representation.

In Figure 4-7 the example of function LM#1 is given. The LM-function operates on 3 scalar inputs and produces a scalar output. The number of operations is fixed (100.0 in this case) and does not depend on any dimension.

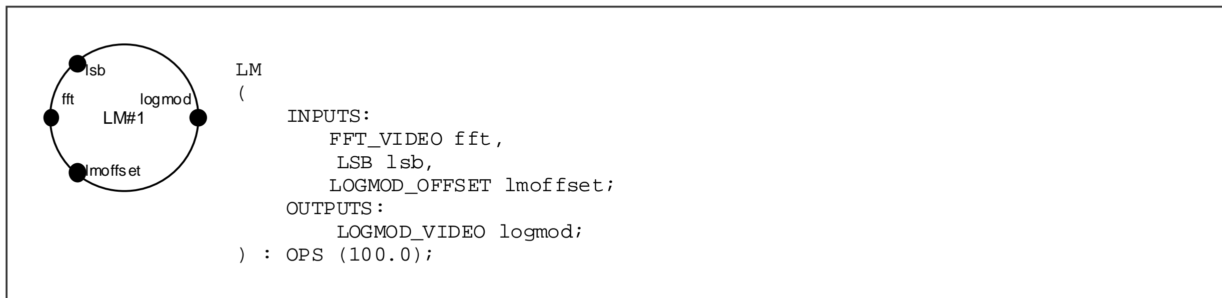


Figure 4-7 Function with scalar inputs and outputs

In Figure 4-8 function TC#1 operates on an input matrix with dimension [Nr, Ns] and produces an output matrix with reversed dimension [Ns, Nr].

The number of operations depends on dimension 4 times Ns times Nr.

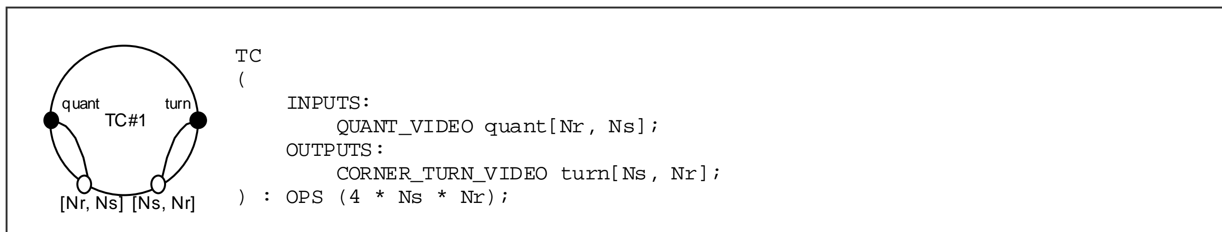


Figure 4-8 Function with multiple dimensions input and output

4.2.6. Interconnections section

The interconnection section starts with the keyword INTERCONNECTIONS and instantiates functions by allocating the input and output interfaces of each function to data elements. Instantiated functions are referred to as actual functions and are identified with their formal function name and an instantiation sequence number separated by a hash sign. In this way a data flow is created between functions and an SDF application is created. Also the data dimensions are allocated to the interfaces of the functions. During compilation of the network, a check must be performed if the data flow is valid and if the global dimensions match within the network.

In the INTERCONNECTIONS section a list of global dimension identifiers is allocated to a list of dimension identifiers of function interconnections. In Figure 4-8 the dimension inputs are illustrated as an open dot at the function circle. Multiple dimension identifiers specify multi-dimensional data. The order of the identifiers specifies how the data is organized in memory.

Examples

```
INTERCONNECTIONS:
…
Sturn -> WV#1(turn[Ns]);
WV#1(weight[Ns]) -> Sweight;
```

When the function specification has been compiled (to be discussed in chapter 5), an interconnection in the INTERCONNECTIONS section may have been attributed with distribution parameters as illustrated below:

```
INTERCONNECTIONS:
…
Sturn -> WV#1(turn[Ns]) [Nr#1];
WV#1(weight[Ns]) -> Sweight [Nr#1];
```

Distribution text *[Nr#1]* has been attributed to the connection. This optional parameter is part of the syntax specification:

```
Interconnection:
    NodeEdgeNode ';'
    | NodeEdgeNode '[' Dimensions ']' ';' ;
```

4.3. Execution rate

Each function in a functional dataflow network has a context determined by the data elements it is connected to. The context determines the execution rate of the function, i.e. the number of times the function is iterated during a single execution of the network. When functions in a particular network have different execution rates, it is called a multi-rate network.

Let us define the list of dimensions of connection c of function f as $LD_{f,c}$ and the list of dimensions of the data element d it is connected to as LD_d. $LD_{f,c}$ has n elements and LD_d has m elements.

The execution rate of function f can be determined from its context if three requirements are met:

1. The length of the dimension list of all function input connections shall be smaller or equal to the length of the dimension list of the data element the function connection is connected to;
2. When $LD_{f,c}$ has n elements and LD_d has m elements, the first n elements of both lists are equal;
3. The connection execution rate list $LR_{f,c}$ of all function input connections shall be equal[1].

The function connection execution rate list LR_f is defined as the $LD_d - LD_{f,c}$, and contains the elements of LD_d with index n + 1 to m.

The execution rate list LR_f of function f is equal to all $LR_{f,c}$. The execution rate r_f of the function f is defined as the product of the dimension size of all elements in LR_f. The function imposes a dimension list to the data elements connected to the output connections of the function and is the union of $LD_{f,c}$ and LR_f, where LR_f determines the last elements. An empty function dimension list results in an execution rate of one (1).

To illustrate this, let us consider the execution rate functions in different contexts.

In Figure 4-9, function WV depends on an input vector with Ns elements and produces also an output vector with Ns elements. In the context Ns is defined as 16.

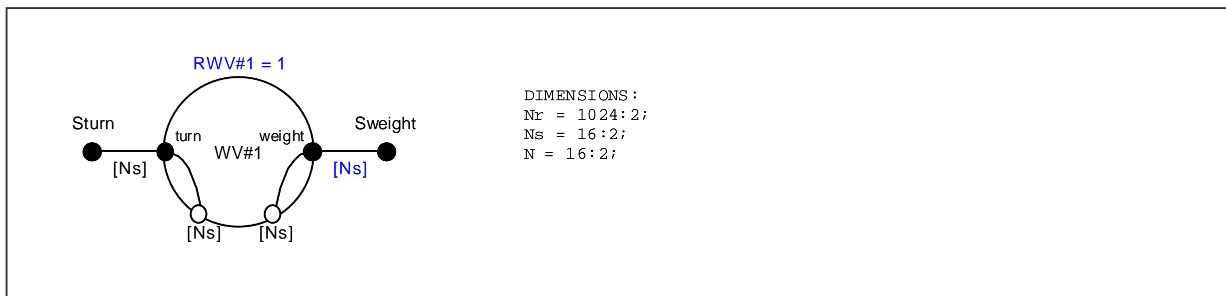


Figure 4-9 Execution rate of 1

In this example it can be verified that $LD_{WV\#1,turn}$ = [Ns] and LD_{Sturn} = [Ns]. The dimension lists comply with the three requirements as stated above. The connection execution rate list $LR_{WV\#1}$ is $LD_{Sturn} - LD_{WV\#1,turn} = \varnothing$ and therefore function WV#1 has an execution rate $r_{WV\#1} = 1$.

The dimension list of output connection *weight* is $LD_{WV\#1,turn}$ = [Ns]. The imposed dimension list $LD_{Sweight}$ is $\varnothing \bigcup$ [Nr] = [Ns] = [16].

The deduced execution rate and output dimension list have been attributed in the example in blue text.

[1] There is an exception to this rule, to be discussed at Figure 4-12

In Figure 4-10 we consider the same function, but now with a different context.

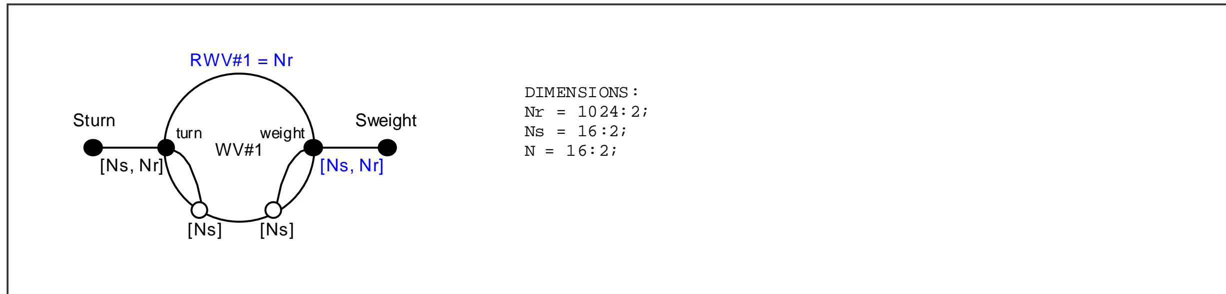


Figure 4-10 Execution rate larger than 1

In this example it can again be verified that $LD_{WV\#1,turn}$ = [Ns], but now LD_{Sturn} = [Ns, Nr]. The dimension lists again comply with the three requirements. The connection execution rate list $LR_{WV\#1}$ is LD_{Sturn} - $LD_{WV\#1,turn}$ = [Nr], and therefore function WV#1 has an execution rate $r_{WV\#1}$ = Nr = 1024.

The dimension list of output connection *weight* is $LD_{WV\#1,turn}$ = [Ns], but now the imposed dimension list $LD_{Sweight}$ is [Ns] $\cup$ [Nr] = [Ns, Nr] = [16, 1024].

In the examples above, the execution rate could be determined in an unambiguous way.

In Figure 4-11, we consider the same example, but now the context is inconsistent.

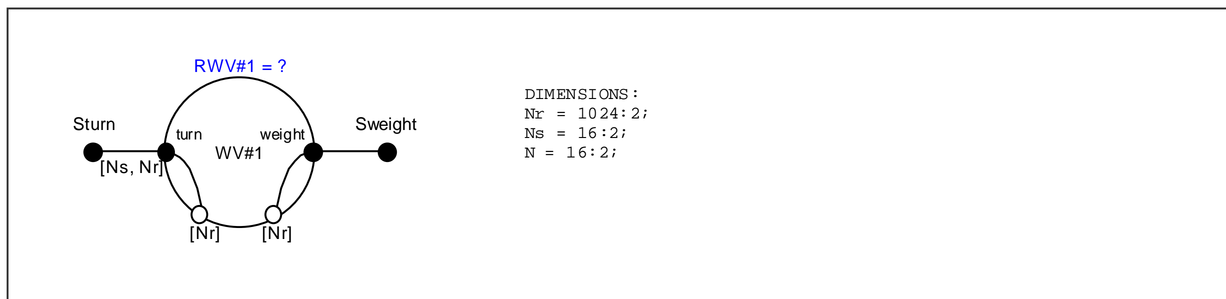


Figure 4-11 Inconsistent context

It can be verified that LD_{Sturn} = [Ns, Nr] and $LD_{WV\#1,turn}$ = [Nr]. The first elements of the lists are not equal (Ns ≠ Nr), and thus the second requirement is not met. Therefore, The execution rate $r_{WV\#1}$ cannot be determined.

There is an exception on inconsistent contexts when some input connections have an empty execution rate list while other input connections don't. Figure 4-12 shows such an example.

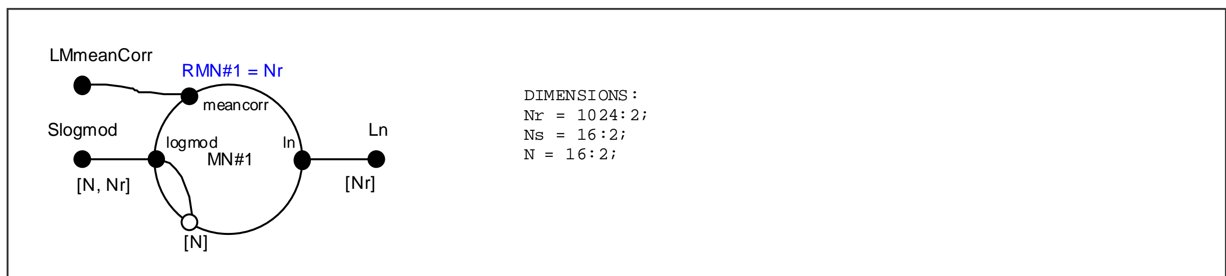


Figure 4-12 Exception on execution rate dimensions

In this example $LR_{MN\#1,logmod}$ = [Nr] and $LR_{MN\#1,meancorr}$ = $\varnothing$, but still the $LR_{MN\#1}$ is [Nr] and $R_{MN\#1}$ = Nr.

The reason for this exception is to model the situation that the value of these particular input connections remain constant during multiple executions of the function. During each execution of the function the old value is retrieved.

This exception gives the opportunity to instantiate a constant in the function.

4.4. Distributor and Aggregator functions

4.4.1. Parallelisation

Each function that has an execution rate greater than one represents an iteration loop in the network. An iteration loop can be unfolded by multiple instantiations of the same function and let them operate in parallel on different parts of the data set. The different parts of the data set need to be distributed to each instantiation and the results must be aggregated again. The results must be aggregated in the correct order, otherwise the functionality of the expanded network might differ from the original network.

Let us consider the functional dataflow network of Figure 4-13. In this example actual function fB#1 has an execution rate of D2 * D1, and is a candidate to be split up.

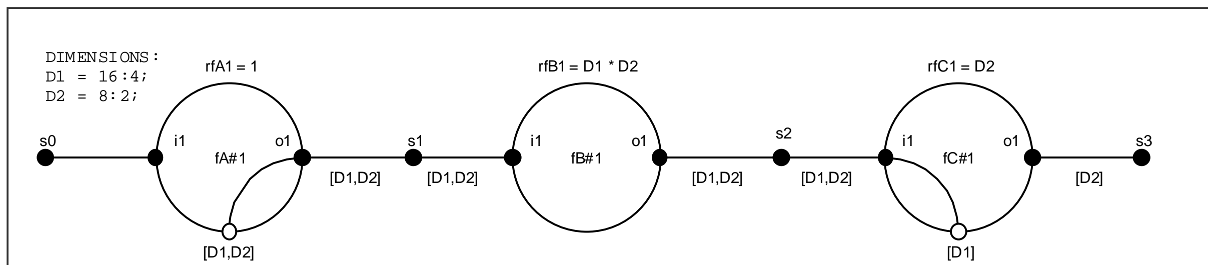


Figure 4-13 Functional dataflow network without parallelisation

In Figure 4-14 the granularity is increased. Extra instances have been created of function fB and fC.

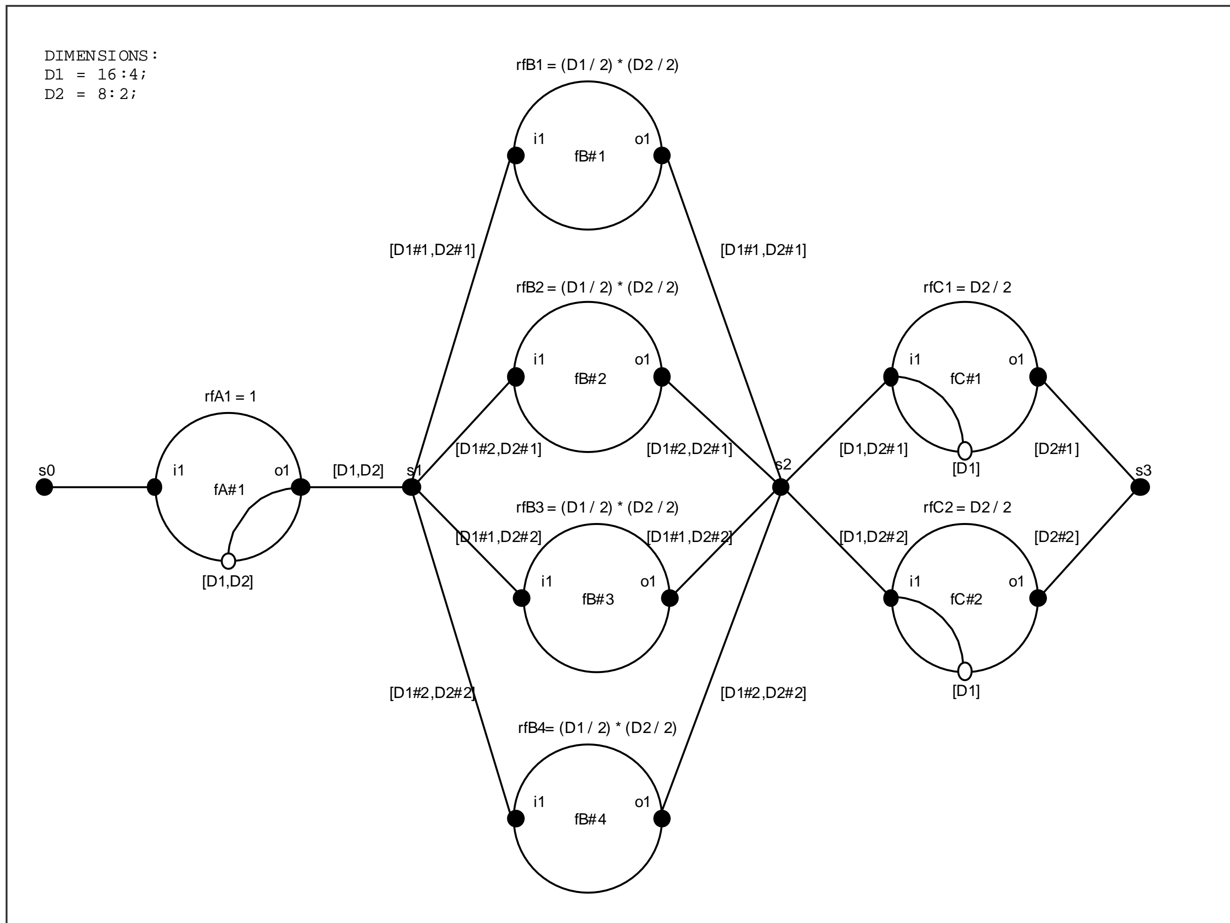


Figure 4-14 Distribution and aggregation

In Figure 4-14, data elements s1 operates as data distributor. Because the distribution parameter of dimension D1 is 4 (D1 = 16:*4*), the data of data element s1 is distributed to 4 instances of function fB.

Data element s2 first aggregates data from the instances of function fB and then distributes data to two instances of function fC (distribution parameter of D2 is 2).

In Figure 4-15 the data paths are optimised, i.e. aggregation and distribution is reduced, with the creation of an extra instance of data element s2.

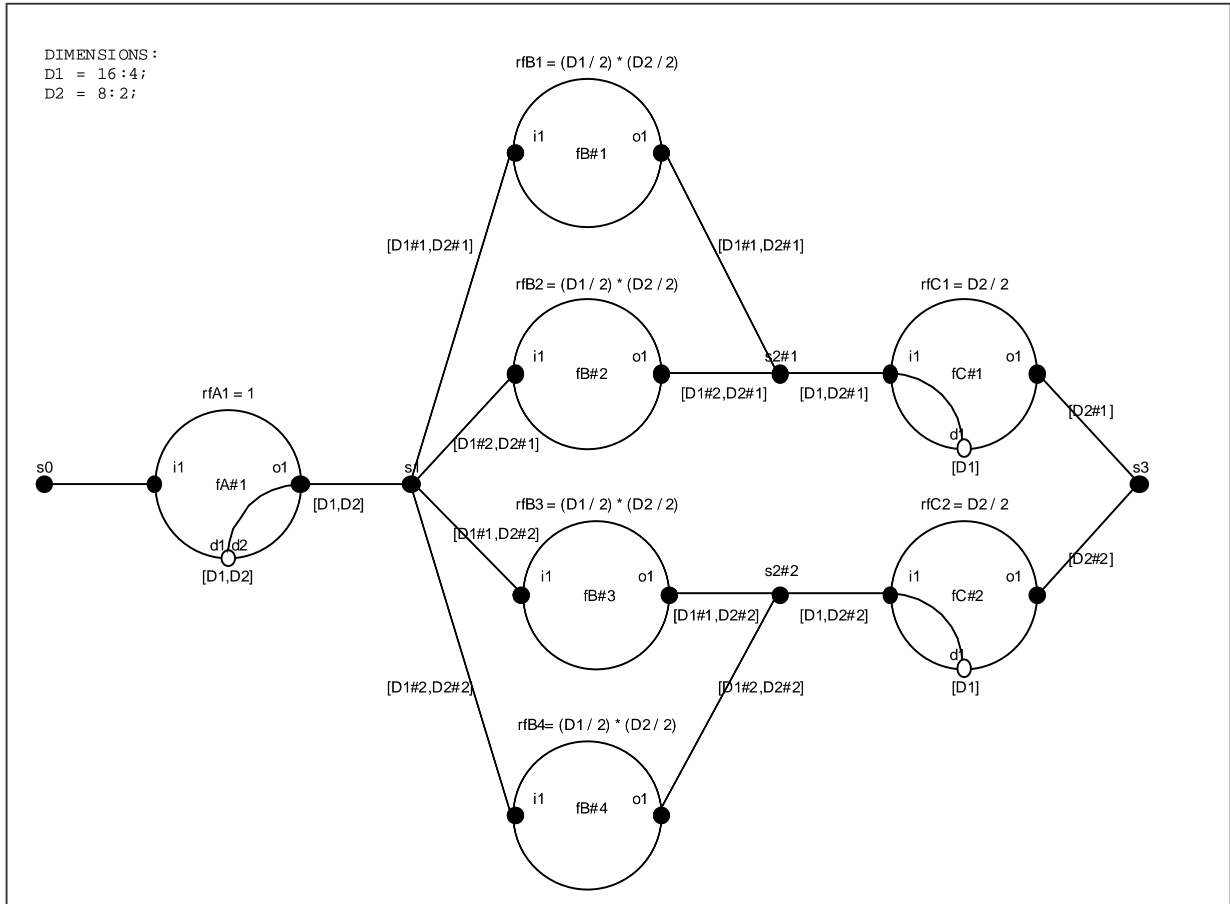


Figure 4-15 Path optimisation

Data element s3 aggregates the output data of function fC again.

With path optimisation the order in which the elements are distributed and aggregated should be considered carefully, as the order of the data might be of relevance in a different part of the network.

For example, the data of s2 is aggregated over the first dimension (D1) first and not over the second (D2) as function fC needs to operate on the first dimension.

The pre-processor must handle this administration. This could be handled by keeping track of the distribution numbers.

4.5. Summary

This chapter explained the rationale of the specification language.

Thesis question Q1 was answered by the way the data elements and data dimensions are modelled in the specification language.

Thesis question Q2 was partly answered by the modelling the OPS parameter. The computation complexity can be expressed with an exact calculation or a order of magnitude. The communication delay will be discussed in paragraph 5.3.3.

The execution rates of each function can be determined from the context. Functions with an execution rate greater than one can be selected to be instantiated multiple times. Data is re-distributed accordingly and execution rates will be updated.

5. Network Expansion Compiler

To implement the pre-processor of the software synthesis process defined in Figure 1-3, a Network Expansion Compiler (NEC) was developed. The design will be discussed in this chapter.

Paragraph 5.1 will describe the global design of the NEC. The major phases during the compilation (parser phase, analysis phase, expansion phase and output generation phase) will be explained in paragraph 5.2 to 5.5. Paragraph 5.6 will address the implementation of the compiler shortly.

5.1. Global design Network Expansion Compiler

In this paragraph the global design of the Network Expansion Compiler (NEC) is presented.

The design of the NEC can be explained from the processing flow as depicted in Figure 5-1.

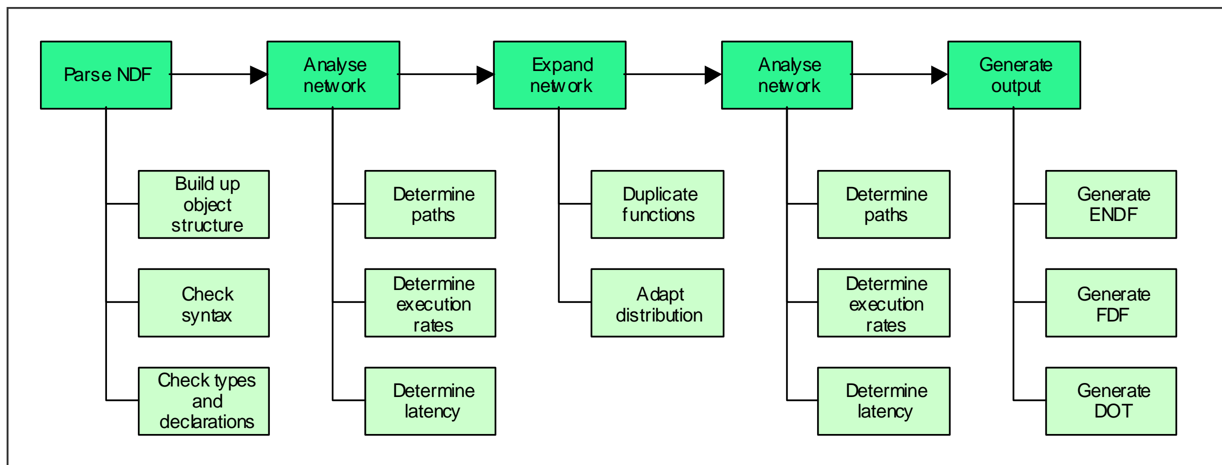


Figure 5-1 Processing flow Network Expansion Compiler

In the first process step the NEC parses the function network specification as discussed in chapter 3 and builds up an object structure of the network. The syntax of the specification is checked and also type checking is performed in the FUNCTIONS and INTERCONNECTION section. On errors the parsing process is stopped.

In the next phase the network is analysed. First, the paths in the network are determined. Then, the context of the functions in the paths is verified. If it is valid, the execution rate parameters of the function will be set. Once the execution rate is determined, the latency can be determined from the OPS parameter and the communication delay.

In the next process phase, the network will be expanded. The granularity will be enlarged by adding extra instances of functions with execution rates greater than one. After this action the distribution parameters of the functions and data elements must be updated.

Next, a second analysis is performed to re-establish the paths, execution rates and latencies in the expanded network.

Finally, in the last phase of the process, the output is generated: an attributed function network specification of the expanded function network. The generated output is input for the middle layer of the software synthesis method, the scheduler.

In the next paragraphs each process step will explained in more detail.

5.2. Parser

In the first process step the function network description is parsed. The input is a Network Description File (NDF).

A parser for the NDF was constructed with the tools Lex and Yacc. Lex and Yacc are common open source tools to generate lexical scanners and parsers (see [13]).
In APPENDIX A paragraph A.1 the complete grammar for the parser is given. Some elements have

already been discussed in paragraph 4.2. The grammar is not very complex and is left for the interested reader to interpret it in its whole.

While parsing the NDF, the syntax is checked and an object structure of the network is build up according to the class diagram of Figure 5-2.

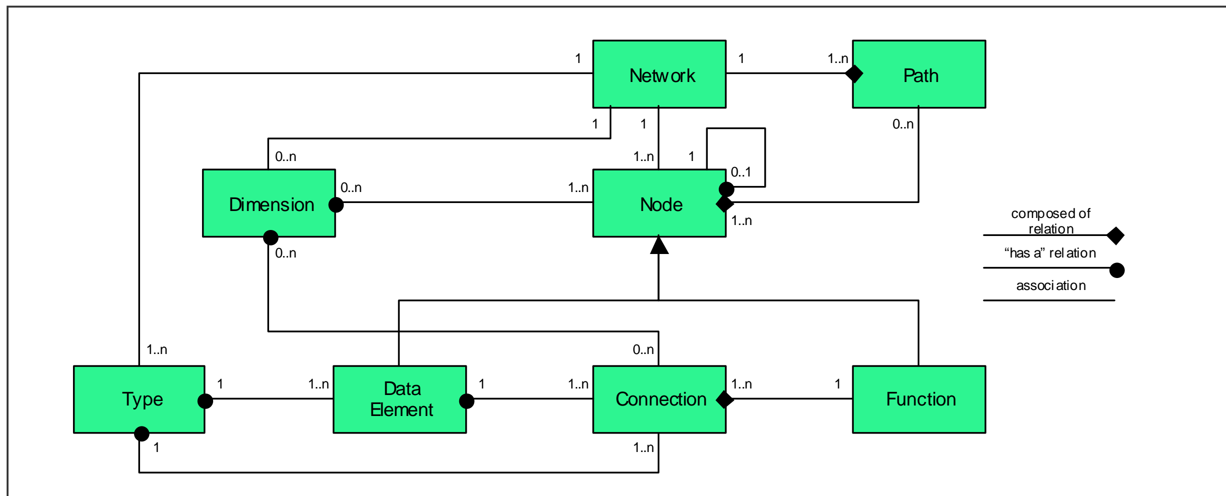


Figure 5-2 Class diagram Network Expansion Compiler

The numbers at the relations specify the multiplicity of the relation.

The class diagram shows the following properties:

- A network is associated with one or more dimensions, types and nodes
- A network has one or more paths.
- A path has one or more nodes
- A node can be either a data element or function.
- A node can have successor and predecessor nodes.
- A node can have dimensions.
- A data element has a type and can have dimensions.
- A data element has a link with one or more connections.
- A function has one or more connections.
- A connection has a type, can have dimensions and always has a link with data element.
- A type is always associated with one ore more data elements and one ore more connections.
- A dimension is always associated with one or more nodes
- A dimension may be associated with one or more connections

When parsing the TYPES section, types are added to the network object. This also applies to the DIMENSIONS section where dimensions are added. When the INPUTS, OUTPUTS and LOCALS sections are parsed the types of the data elements are verified against the earlier type declarations in the TYPES section and added to the Network object.

When the FUNCTIONS section is parsed the functions are added to the network object. Connections are added to the corresponding function object. The types of the connections are checked against the earlier type declarations in the TYPES section.

Finally, when the INTERCONNECTIONS section is parsed, data elements are connected to the corresponding function connection. First, the existence of the formal function is verified, then the identical type of the data element and the function connection.

For each new object type the parser checks if the identifier is unique.

After the NDF has been parsed correctly, the first analysis phase will be executed. This will be discussed in the next paragraph.

On errors, the parser is abandoned and the NEC exits with an appropriate error report.

5.3. Analyse network

In the second process step, the network is analysed. The functions must have a consistent context. As long this is true, the execution parameters of the functions and dimension parameters of the local data elements will be set. This will be explained in sub paragraphs 5.3.1 and 5.3.3.

To do this, all the paths in the function network are determined first and will be discussed in the next sub paragraph.

5.3.1. Path creation

The INTERCONNECTIONS section in the function network specification defines edges between a function connection and a data element or vice versa. From these edges, a directed graph can be constructed. A data flow graph may contain cycles. When it does, the cycle should have delay nodes, but for reasons of simplicity it is not taken into account in this thesis. That makes the graph a Directed Acyclic Graph (DAG). No cycle check will be performed.

Paths in the network are constructed from the edge specifications to be able to determine the latency in the network.

The design of the NEC defines a network path as a linked list of nodes. If a path branches, a new path is created.

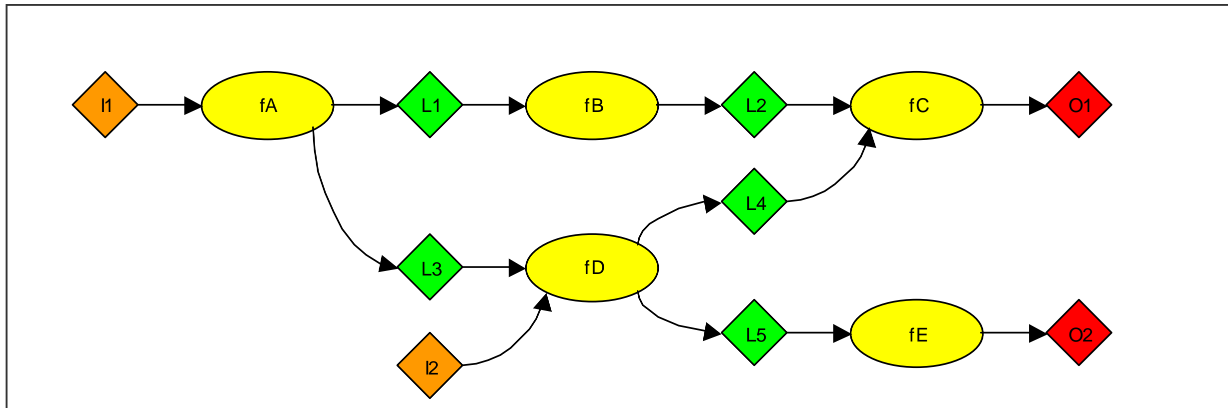


Figure 5-3 Path selection

In the example in Figure 5-3 diamonds and ovals represent nodes. The yellow ovals represent functions, orange diamonds represent input data elements, red diamonds represent output data elements and green diamonds represent local data elements.

The paths that can be discriminated are:

I1 – fA – L1 – fB – L2 – fC – O1
I1 – fA – L3 – fD – L4 – fC – O1
I1 – fA – L3 – fD – L5 – fE – O2
I2 – fD – L4 – fC – O1
I2 – fD – L5 – fE – O2

The recursive algorithm to find and create the paths is specified below:

```
/* find all node paths in network */
find all input nodes
for each input node do
   new current path
   append_path (current path, input node)
end

append_path (current path, node)
   append node to current path
   find successor nodes
   if output node found then
      /* end of path found */
      add current path to network
   else
      /* successor nodes found, append first successor node to current path */
      append_path (current path, first successor node)
      for other successor nodes do
         /* path branches */
         copy current path to new path
         append_path (new path, successor node)
      end
   end
end
```

Because each input data element determines the start of a path (and they are identified after the parsing of the function network specification), the algorithm starts with the input nodes.

First, the input node is added to a new path and then the algorithm searches for its successor nodes. If an output node is found, the path-end has been found and this path is added to the network.
If there are successors, the first successor node found is added to the path. When there are more successor nodes, the path branches, apparently. For each branch the current path is copied and the algorithm continues by appending the corresponding successor node to the path.
The recursion enters each time a node is appended to the path. The recursion ends when a path-end is found.

The number of paths can become fairly large, especially in the expanded network. This can have significant impact in the performance of the NEC. A smarter algorithm could be implemented to filter out the most critical paths

5.3.2. Determine execution rates

When the paths have been determined, the execution rate of each function along each path is determined. A function can be selected when all input nodes have been processed. If this condition is not true, the input node will be processed first. This is done in a recursive way. By default, the input data element nodes have the status processed.

The function execution rate depends on the dimensions of the input data elements. The dimensions of the input data elements are specified by the NDF. The execution rate of the functions and the dimensions of their output connections determine the dimensions of the local and output data elements. In this method, the nodes are processed in a flow that moves from input to output.

As explained in paragraph 4.3, the context of each function must remain valid at all times.

For each input connection of a function a dimension list $LD_{f,i}$ is determined from the NDF. The connection is linked with a data element that has dimension list LD_d. Let us assume that $LD_{f,i}$ has n elements and LD_d has m elements. In a valid context m must be greater or equal to n and the first n elements of $LD_{f,c}$ and LD_d must be equal. The remaining elements of LD_d, elements n + 1 to m, determine the connection rate list $LR_{f,i}$.

If all input connections have an identical $LR_{f,\,i}$ the function execution rate list LR_f is consistent and equal to $LR_{f,i}$. The function execution rate r_f is determined by the product of all dimensions sizes defined in LR_f. If LR_f is empty, r_f is equal to 1.

If the function execution rate list LR_f is determined, the dimension list LD_d of the data element connected to an output connection is defined as the conjunction of the output connection dimension list $LR_{f,o}$ and LR_f, where LR_f determines the highest elements. If LD_d is already defined by a previous function, a check is performed whether both definitions are equal.

The algorithm is defined below.

```
/* determine function execution rates within network */
for all network paths do
   for all nodes in path do
      if predecessors not processed
         determine execution rate list of predecessors
         calculate execution rate of predecessors
      end
      calculate execution rates of input connections
      if execution rates of input connections differ
         report invalid context and exit
      end
      calculate function execution rate and determine function execution rate dimension list
      for all output connections do
         if successor is processed then
            if dimensions of successor node differs then
               report invalid context and exit
            end
         else
            set dimension list of successor nodes
         end
      end
   end
end

calculate execution rates of input connections
{
   for all function input connections do
      if dimension list of connection is larger than dimension list of connected data
      element then
         report invalid context and exit
      end
      for all dimensions of connection do
         if dimension is not equal to corresponding dimension of connected data element
         then
            report invalid context and exit
         end
      end
      connection execution rate = multiply remaining dimensions of connected data
      element
   end
   if input connections have different execution rate then
      report invalid context and exit
   end
}
```

5.3.3. Calculate latency

After the rates have been determined the path latencies are calculated. The network latency is the path with the largest latency.

Each node adds latency to the path due to its calculation time expressed by the number of operations and communication with other functions. To have a generic approach we assume that data is transferred to other functions by message passing.

The number of operations is estimated from the computation complexity as specified in 4.2.5. Although the exact number can be calculated, an order of magnitude is assumed to be sufficient. The big *O* notation is used to specify the number of operations.

The communication delay is calculated from the number of elements to be received and/or sent by the function. The function starts when all input data elements are available. If all data is received and sent through the same channel, the communication latency will be larger than when each input and output have their own communication channel. The NEC assumes that the number of channels is not a bottleneck and that communication delay depends linearly on the largest number of data to be sent or received by a channel.

Furthermore, it is assumed that the data elements add no latency when they act as data distributor or aggregator.

Taking the remarks above into account, the latency is calculated from the operations parameter specified by the formal function and the communication delay by the following algorithm:

```
latency = operations_latency * execution_rate + communication_latency

/*  calculate communication delay */
for all connections do
   for all dimensions
      calculate product of dimension sizes and element type size
   end
   communication_latency = find maximum product
end
```

The number of operations is calculated while the OPS parameter is parsed.

Example

In the Network Description File (NDF) of the radar application demo the operations parameter (OPS) of the function TurnCorner is (4 * Nr * Ns).

Nr stands for “Number of range quants” and Ns stands for “Number of sweeps”.

In the network dimension Nr is equal to 1024. Dimension Ns is equal to 16. The function TurnCorner operates on 4 * Nr * Ns = 4 * 1024 * 16 = 65536 elements. In the current implementation of the NEC, the communication delay is equal to the size of the input data. Therefore, the communication delay of the TurnCorner function is Nr * Ns * typesize = 1024 *16 * 2 = 32768. As the execution rate of the function TC is 1, the total latency of the function TC = 65536 * 1 + 32768 = 98304.

5.4. Expand network

If the network has been analysed the NEC starts to explore the expansion possibilities based on the distribution parameters defined in the dimension declaration in the NDF.

```
DIMENSIONS:

DimA = 16 : 8;
DimB = 32 : 4 : 2;
```

In the example define above, dimension DimA has a size 16 and allows 8 distributions, i.e. the data can be split up in 8 distributions, each distribution has 2 consecutive elements.

DimB has a size of 32 that allows 4 distributions and each distribution has 10 consecutive elements, because the third parameter defines an overlap of 2 elements. In the NEC the overlap feature has not been implemented.

After the analysis of the function network, the function execution rate parameters, declared in the list LR_f, contains dimensions as defined in the DIMENSIONS section.

If the distribution parameter of the first dimension definition allows n distributions ($R_f(1) = n$), the function f will be instantiated n times, including the original instantiation. The input and output data has to be distributed accordingly, where the first part of the data is distributed to the first instantiation, the second part to the second instantiation, etc.

If multiple execution rate parameters are defined in R_f, the number of instantiations will be the product of the distribution parameters.

The NEC traverses the list of functions in the network and instantiates new functions according to the execution rate parameters. New instantiated functions inherit the properties of the original function and are connected to the original input and output data elements. A distribution parameter list is allocated to each connection. The list element defines the dimension identifier, the dimension size, the maximum number of distributions and the overlap parameter.

After this expansion phase the network is analyzed for the second time. During the second analysis, the distribution parameter list is updated. The maximum number of distributions in the DISTRIBUTION section will be set to one (1), because maximum allowed expansion will have been reached. The dimension size of all distributions is the division of the original size by the original maximum number of distributions. The remainder part is allocated to the last distribution.

In the INTERCONNECTION section, each interconnection will be attributed with the distribution list.

5.5. Generate output

The last step of the NEC is the generation of the output, i.e. the Expanded Network Description File (ENDF), the Function Description File (FDF) and the visualisation file (DOT). For each output file the NEC traverses the object structure and generates the text files.

The ENDF is the processed NDF and has the same syntax as the NDF. Execution rates, estimated latency and local dimensions and distributions are attributed to the INTERCONNECTION section.

In the next chapter the results of the NEC will be discussed.

5.6. Implementation

The NEC was implemented in C++ using the Together 4.0 tool environment. There were various reasons for this choice:

- An object oriented language supports the implementation of the object oriented design of the NEC very well;
- The C++ Standard Template Library (STL) can be used, which offers a lot of functions ready to use (e.g. linked list container, iterators, etc.);
- The existing framework was developed in C++;
- For future maintenance at Thales the C++ language was preferred.
- The free Together 4.0 design environment (TogetherSoft Corporation) supports C++ (and Java).

The source code of the NEC is archived at Thales and will not be elaborated on in this thesis.

6. Results

In this chapter the results of this thesis will be discussed.

In paragraph 6.1 the software synthesis framework, which was used as a test environment, is described globally. It answers thesis question Q3 (test the top layer).
Paragraph 6.2 answers thesis question Q4 (exploit the level of task parallelism).
Paragraph 6.3 will address some shortcomings of the NEC prototype.
Finally, in paragraph 6.5 the visualisation tool is discussed shortly.

6.1. Software synthesis framework

In Figure 6-1 the software synthesis framework is depicted and represents the implementation of the method described in Figure 1-3. The framework will be used to test the design and implementation of the compiler. The typical radar application used throughout this thesis will be used to evaluate the results.

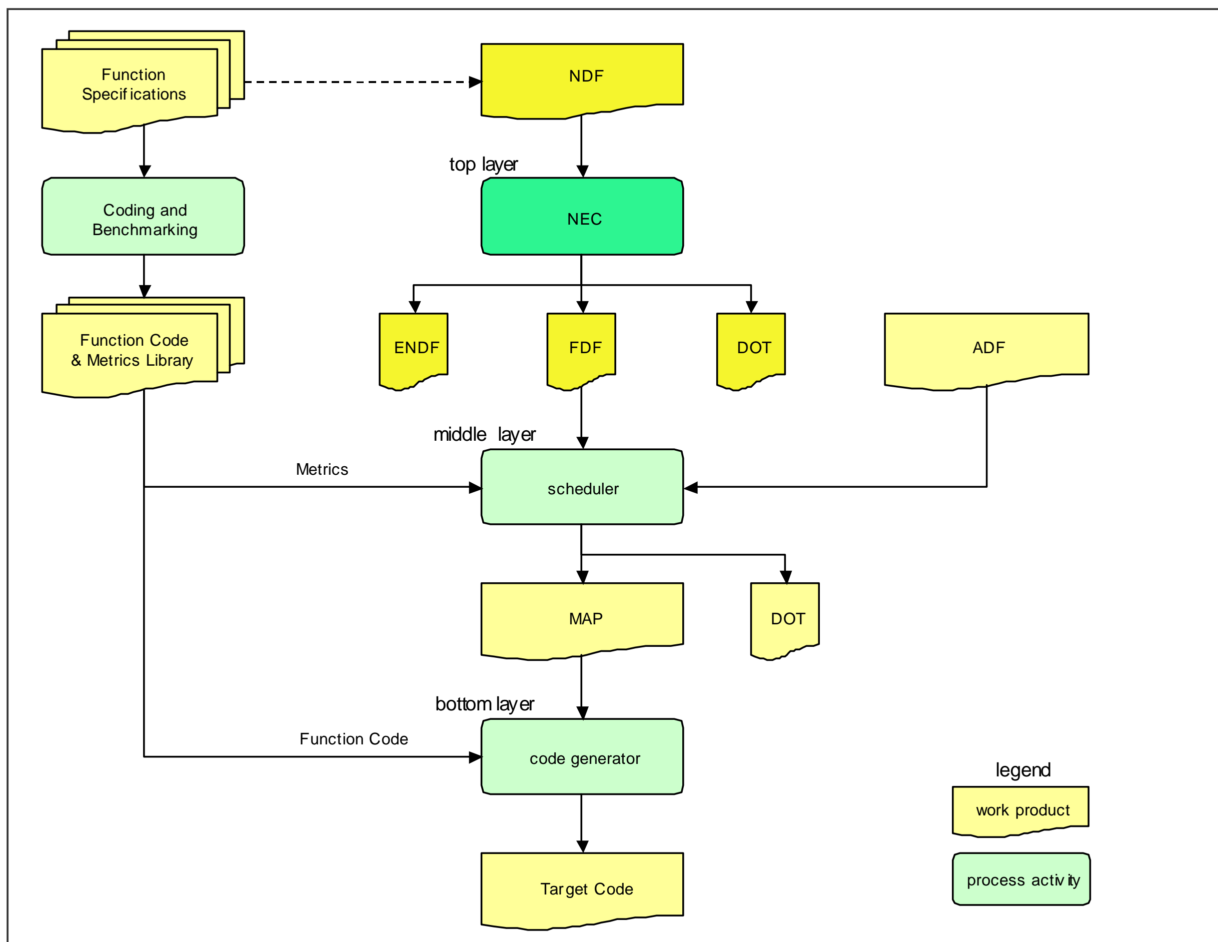


Figure 6-1 Software synthesis framework

The input of the Network Expansion Compiler (NEC) is a text file that specifies the network, the Network Description File (NDF). The main output of the NEC is a text file that specifies the expanded network, the Expanded Network Description File (ENDF).

The ENDF has the same syntax as the NDF, but the degree of parallelisation has been extended and the data distribution is specified in the INTERCONNECTION section. Execution rates, local dimensions and distributions have been attributed to the description.

The NEC generates two side products: the Function Definition File (FDF) and a visualisation file (DOT).

The FDF describes the same function network as the ENDF, but has a different syntax (without data elements, data distribution and dimension properties).

The FDF is generated to be able to use the existing allocation tool (“scheduler”) that has been developed under [1]. Together with the Architecture Description File (ADF) and the metrics library the scheduler allocates, or maps, the expanded network to a physical processor architecture. The output of the scheduler is a map-file (MAP).

The MAP-file is used as input for the code generator. Together with the function code from the library the source code for each processor in the hardware architecture is generated.

For visualization purposes, several DOT files are generated during the process. The reason why the layout engine *dot* ([14]) has been used will be explained in paragraph 6.5.

First, the results of the NEC will be addressed. For the sake of completeness, some results of the scheduler will also be addressed to evaluate the quality of the generated FDF.

6.2. Testing Network Expansion Compiler

The development of the NEC was one of the main goals of this thesis. In this paragraph the results of the NEC are discussed. In paragraph 6.2.1 the exploitation of the data parallelism will be evaluated and answers thesis question Q4.

6.2.1. Exploiting data parallelism

Figure 6-2 shows our radar application demo example (generated by the *dot* layout engine). Although it is less detailed as the graphical specification in Figure 4-5 it gives a good overview of the data flow in the function network.

The functions are represented as yellow ovals. Input data elements are represented as orange diamond shapes, local data elements as green diamond shapes and output data elements as red diamond shapes.

In this figure the network has not yet been processed and this explains why the dimensions parameters are only mentioned at the input data elements. The figure complies exactly with the NDF specification of the typical radar application as defined in paragraph 4.2.

In the next step the NEC processes the NDF. To set a network latency reference to compare the expansion results, the NDF is processed first with the distribution parameters set to 1 for all dimensions, as shown below:

```
DIMENSIONS:

Nr = 1024 : 1 : 1;
Ns = 16 : 1;
N = 16 : 1;
```

With these parameters, no expansion will occur.

After the NEC has processed the NDF successfully, the ENDF is generated. The reference ENDF is visualised in Figure 6-3.

For each function three basic parameters have been added in the yellow oval: execution rate, number of operations per execution and total latency. The latter is calculated by:

function latency = OPS * execution rate + communication latency.

The communication delay is determined by the input or output that has the largest data size, which is determined by (number of elements) * (the size of the data type)

At the bottom of the diagram, the total network latency has been added. It is calculated from the path with the largest latency. In this particular case it is equal to 1904738. This is the latency without expansion and will be compared later with the latency of the expanded network.

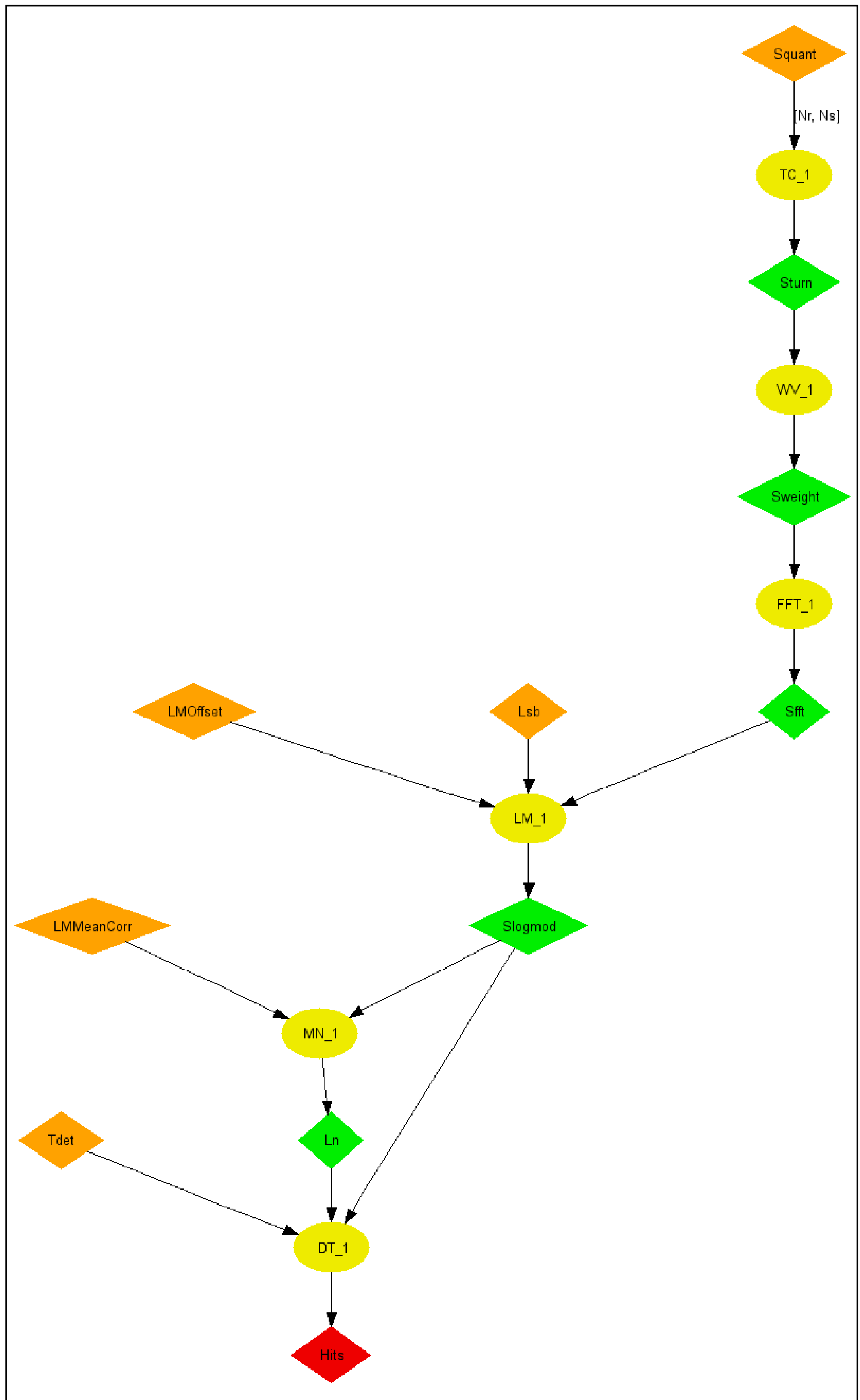


Figure 6-2 Unprocessed specification of radar demo application

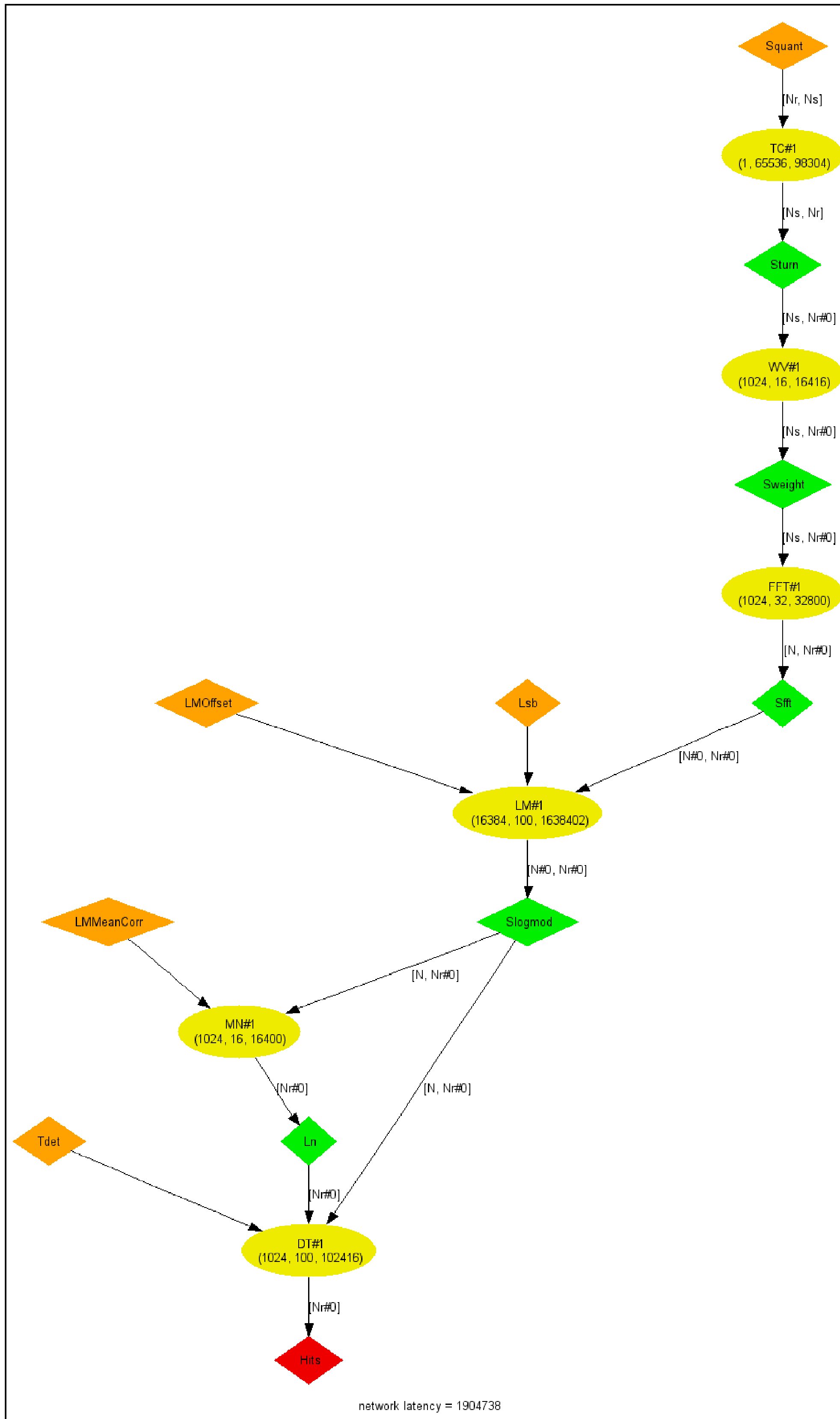


Figure 6-3 Processed radar demo application reference (without expansion)

In Figure 6-3 all arcs have been attributed with dimension and distribution parameters according to the ENDF-syntax. Dimensions that map on the dimension(s) of the function input connections do not have a distribution number (the number after the hash). Dimensions with a distribution number determine the execution rate of the function as explained in paragraph 4.3. If the distribution number is equal to 0 (zero), it implies that it is the single distribution.

The following example will explain this in more detail. Let's consider the TC and WV function in the ENDF:

```
NETWORK demo

TYPES:

QUANT_VIDEO = 2 * float;
CORNER_TURN_VIDEO = 2 * float;
WEIGHTED_VIDEO = 2 * float;
…

DIMENSIONS:

Nr = 1024 : 1 : 1;
Ns = 16 : 1 : 0;
N = 16 : 1 : 0;
…

FUNCTIONS:

TC
(
    INPUTS:
        QUANT_VIDEO quant[Nr, Ns];
    OUTPUTS:
        CORNER_TURN_VIDEO turn[Ns, Nr];
) : OPS ( 65536 );

WV
(
    INPUTS:
        CORNER_TURN_VIDEO turn[Ns];
    OUTPUTS:
        WEIGHTED_VIDEO weight[Ns];
) : OPS ( 16 );
…

INTERCONNECTIONS:

/* TC#1
 * operations  : 65536
 * latency  : 98304
 * rate  : 1
 */

Squant -> TC#1(quant[Nr, Ns]);
TC#1(turn[Ns, Nr]) -> Sturn;

/* WV#1
 * operations  : 16
 * latency  : 16416
 * rate  : 1024 [Nr]
 */

Sturn -> WV#1(turn[Ns]) [Nr];
WV#1(weight[Ns]) -> Sweight [Nr];
```

The data dimensions of the Squant input data element (Nr and Ns) map on the dimensions of the input connection *quant* of the function TC#1. As the function connection execution rate list of the *quant* input ($LR_{TC\#1,squant}$) is empty, TC#1 has an execution rate of 1. At each execution, TC#1 produces a matrix *turn* with Ns x Nr elements, stored in the local data element *Sturn*.

The number of operations of TC#1 is 4 * Nr * Ns. During the parsing of the NDF, the NEC calculates the number of operations of which the value is exported to the ENDF.
OPS = 4 * 1024 * 16 = 65536.

The communication delay is the maximum data size to be sent or received by function TC#1. It is equal to Nr * Ns * type size = 1024 * 16 * 2 = 32768.

The total latency of TC#1 is equal to OPS * execution rate + communication delay = 65536 * 1 + 32768 = 98304.

The execution rate, the number of operations per execution and the total latency are annotated in the ENDF at the INTERCONNECTIONS section.

An equal derivation can be applied to function WV#1. The input connection *turn* of the function WV#1 has a single dimension (Ns) and maps on the first dimension of the local data element *Sturn*. Therefore, no distribution number is added to Ns. The dimension Nr is not mapped on the input connection dimension. As no distribution of segments is allowed, it is attributed with distribution number 0 (zero). Therefore, dimension Nr determines the execution rate of function WV#1 and is equal to 1024. The number of operation is Ns and is equal to 16.

The communication delay is Ns * type size = 16 * 2 = 32.

The total latency of WV#1 is 16 * 1024 + 32 = 16416.

After this explanation the other functions speak for themselves.

Next, the DIMENSIONS section in the NDF is modified as below:

```
DIMENSIONS:

Nr = 1024 : 2 : 1;
Ns = 16 : 2;
N = 16 : 2;
```

Now, the dimensions Nr, Ns and N are allowed to split up in 2 parts.

After the NEC has processed the NDF successfully, the expanded NDF (ENDF) is generated. The ENDF is visualised in the same manner as in the previous examples.

As can be seen in Figure 6-4, extra functions of WV, FFT, LM and MN have been instantiated. The execution rates and latency of these functions have been reduced as a result of the data distribution. Where dimensions have been split up, a distribution number is added to the dimension, e.g. Nr#2 at function WV#2.

As can be seen at the bottom of Figure 6-4, the total network latency has been reduced to 591970. Compared to the reference latency in Figure 6-3 of 1904738 this is a reduction of

(1 – 591970 / 1904738) * 100% = 69%

From this example we can conclude that the network expansion can lead to significant reduction of the network latency. The most significant reduction is caused by extra instances of function LM. If the other functions remain unexpanded, the reduction would still be significant: a total network latency of

1904738 – 1638402 + 409602 = 675936 is still a reduction of 65%.

The feature to find the most significant reduction has not been implemented in the prototype.

A remark should be made about function LM in the network. The dimension parameters of N and Nr allow the data to this function to be distributed over 2 dimensions. With dimension N and Nr distributed in 2 parts, 4 instances of LM should have been created, but this is not implemented in the NEC.

The full ENDF description of this example can be found in APPENDIX A, paragraph A.3.

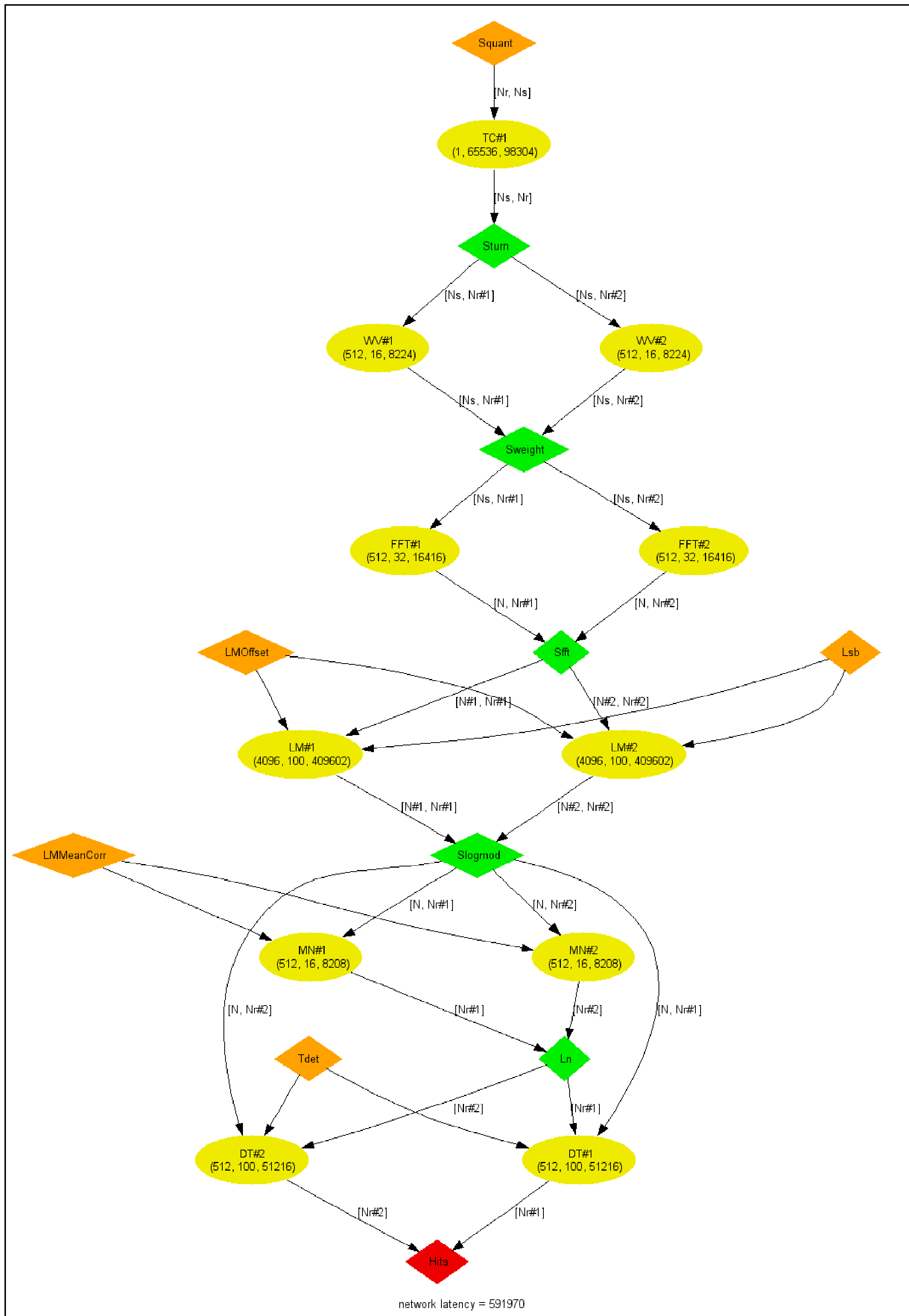


Figure 6-4 Expanded radar demo application

6.3. Shortcomings prototype Network Expansion Compiler

Due to time constraints not all planned features of the compiler have been implemented in the NEC. The shortcomings relate to the communication latency model, distribution and aggregation, path optimisation and data overlap. Also the strict type checking can be considered as a shortcoming.

Communication latency

A simple approach for communication latency is modelled. Each element to be received or sent adds a delay unit (1 operation). It is assumed that send and receive operations can be performed simultaneously. The input or output that receives or sends the largest data size, determines the worst-case communication latency. This simple model works quit well for unexpanded networks, but the overhead created by distribution, aggregation and forwarding of data is not taken into account by the NEC prototype.

Distribution and aggregation

As we have seen in the example of Figure 6-5, distribution/expansion over multiple dimensions is not implemented correctly.

Path optimisation

Path optimisation, as discussed in paragraph 4.4, has not been implemented. In particular, the administration for the path optimisation is to be addressed.

Data overlap

The specification of data overlap as described in 4.2.3 has not been implemented due to time constraints.

6.4. Scheduler and code generator

Not much will be said about the scheduler and the code generator in this thesis, as they were already developed and are outside the scope of the thesis. The reason that the scheduler is mentioned is because the NEC has to interface with the scheduler. The interface is realized by means of the Function Description File (FDF) that already existed as input file for the scheduler. For further information on this subject, the reader is referred to [1].

The Function Description File (FDF) describes the same function network as the ENDF, but has a different syntax. Data elements, dimensions and distribution properties are not part of this syntax. The FDF is generated to use the existing allocation tool.

The function metrics should be imported from the Function Code & Metrics library, but at the time of writing, they are generated from the ENDF for testing purposes. Data elements (inputs, local and outputs) are modelled as distribution and aggregation functions that have no latency, but this should to be adapted in the final implementation.

For information only, the FDF of the expanded radar application demo is given in APPENDIX A, paragraph A.4.

Although the Architecture Description File (ADF) is out of the scope of the thesis, for the sake of completeness an example is given in APPENDIX A, paragraph A.5.

Allocation graph

The scheduler maps the (expanded) function network to the hardware architecture. It also generates a DOT-file to visualize the result. The result of the expanded radar application demo is given in Figure 6-5. The rectangles represent the processors, the ovals the functions and data elements.

A few remarks should be made here:

1. To overcome the differences between the ENDF and FDF, data elements are specified as functions that have no operations;
2. As there are more functions (11 “real” functions) than processors (6), the scheduler has to cluster multiple functions to the same processor. Some functions that were expanded by the compiler (e.g. WV, FFT, MN and DT) are scheduled on the same processor. In this case the expansion did not contribute to a reduction of the latency;
3. A BEGIN and END function have been added. Although not strictly necessary, the developer of the scheduler considered it to be convenient to have a single entry and exit point for the application to minimize the interface with the host environment;
4. The numbers at the directed arrows represent the size of the data. The scheduler uses the size of the data (in bytes) that is transferred between the functions to find a good mapping on the hardware architecture. When functions reside on the same processor the communication delay is ignored;
5. The number of operations of functions that reside on the same processor are added and displayed for each processor.
6. The BEGIN and END function add no extra performance;
7. The scheduler has difficulties to find a schedule when the number of processors is larger than 6. This prevents to show a good example where the number of functions is equal ore close to the number of processors. Due to time constraints, it could not be investigated what the cause of this restriction is.

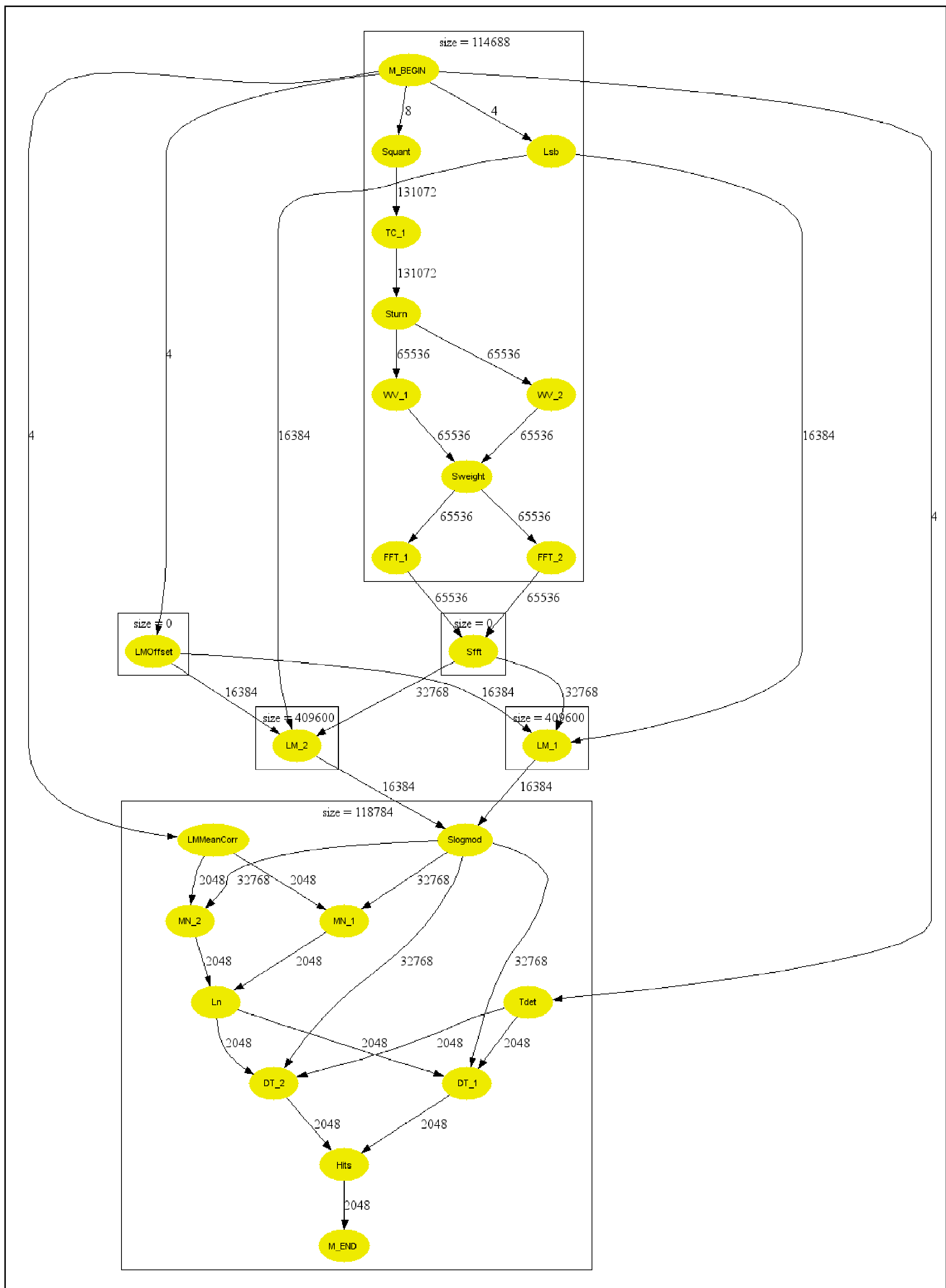


Figure 6-5 Expanded function network allocated to hexagonal processing architecture

6.5. Visualisation

In this paragraph a few short remarks will be made about the visualisation of the function networks. The graphical notation as described in chapter 4 will be evaluated first.

The graphical notation helps to visualise the most important characteristics of the network. The dimension parameters of the function are visualised as a open dot on the circumference of the network and mapped with an arc on the function input and output connections. Although data dimensions can be recognized very quickly, it is quite laborious to draw them manually. In retrospect, this could also have been specified as shown at the right side of Figure 6-6:

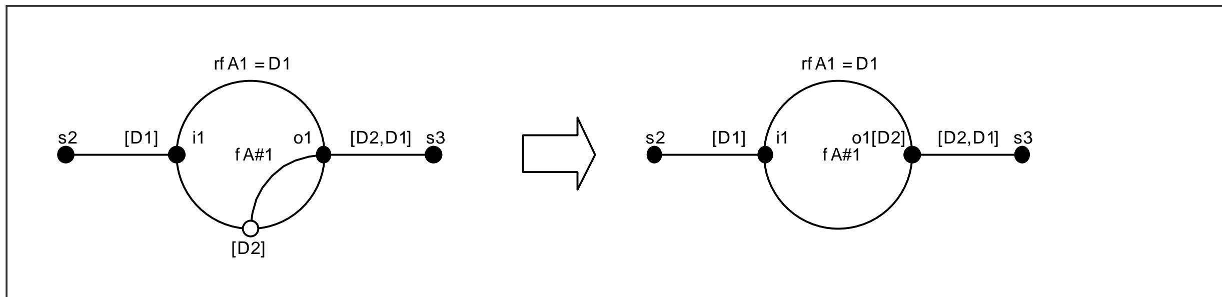


Figure 6-6 Alternative specification dimension parameters (1)

In this situation dimension parameters are specified as text at every input and output. The dimension list of the data element maps at the dimension list at the function connection.

However, the following drawback can be recognized already: the space available in the function circles is rather small to host all the names and dimension lists for the function, inputs and outputs.

To overcome this issue only the global or actual dimensions are specified at the arcs between the data element and the function. The actual dimensions that don't have a mapping on the function's formal dimensions have a postfix that represents the distribution number. When the distribution number is 0 (zero), there is only one distribution.

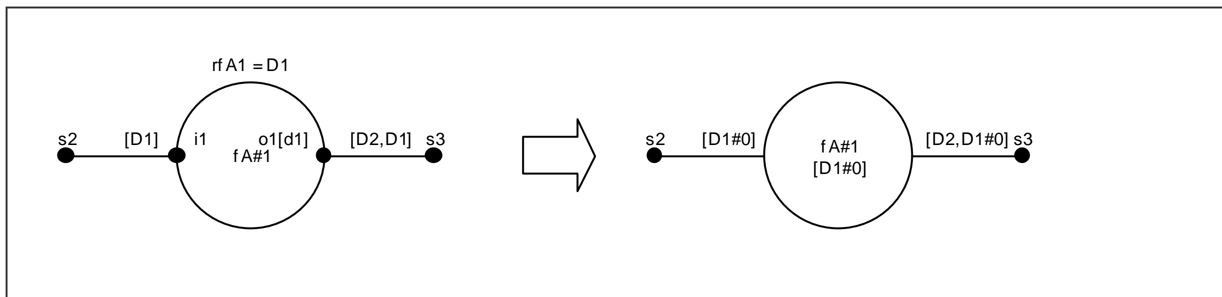


Figure 6-7 Alternative specification dimension parameters (2)

With this convention the layout engine *dot* ([14]) can be used to visualise the function networks. Regular SDF-diagrams are represented very well by *dot* as it is particularly suitable for the representation of Directed Graphs.

All kind of attributes can be specified for the graph, nodes and edges (e.g. labels, colours, ranks, sub clusters, etc.).

When the *dot* engine is run, a graphical output format is generated. Multiple graphical formats can be generated (gif, jpg, png, etc.).

7. Conclusions and recommendations

In this chapter the conclusions of this thesis and recommendations for future work will be discussed.

A Network Expansion Compiler (NEC) was developed. The NEC implements the pre-processor of the software synthesis method for synchronous data flow networks and exploits the data parallelism in the function network.

First the conclusions of this thesis will be drawn in paragraph 7.1. In paragraph 7.2 further recommendations for the NEC will be given and paragraph 7.3 some general recommendations for further study.

7.1. Conclusions

At the time the thesis started, a study was performed on the status of parallel programming standards and (commercial) tools that support the software synthesis of real-time signal processing applications on multiprocessor architectures. Today, *mature tools exist that support parallel programming standards* like MPI and OpenMP, but they are *often dedicated to and optimised for specific hardware architectures*.

To support random hardware architectures a *three-layer software synthesis method* was set-up. The top layer exploits the data parallelism in the function network and increases the granularity of the function network. It has no knowledge about the hardware architecture. The middle layer schedules the expanded function network to the hardware architecture and the bottom layer finally generates the target code for each processor. The method was tested using an existing framework and a limited number of applications and architectures. A typical radar application was chosen to test and evaluate the approach. The conclusion can be drawn that the chosen approach is *a good approach to decouple the functional models from the hardware layer*, which changes more rapidly than the models. This answers thesis questions Q3 and Q5.

To specify the functional models a specification language was defined. The language extends the principles of the Synchronous Data Flow (SDF) formalism with data dimension parameters and data types. *The dimension parameters enable the exploration of data parallelism in the network and have shown to be an enrichment of the SDF-formalism*. In another context the specification language extends the current develop environment at Thales quite well, and this can be considered as a major benefit.
A graphical notation for the specification language was also defined for visualisation purposes. After some small adoptions in the original notation the diagrams can be generated automatically with the *dot* layout engine.
The overall conclusion of thesis question Q1 is that *the chosen method is successful*.

A Network Expansion Compiler (NEC) was developed to implement the pre-processor. The overall conclusion of thesis question Q4 is that *the NEC is successful in exploiting data parallelism.* A few shortcomings in the prototype of the NEC should be addressed to become of practical use.

Thesis question Q2 asks whether latency determinant parameters can be modelled in an abstract manner, independent from the hardware architecture. The number of operations and the communication delay of the functions in the critical path determine the total network latency. For the estimation of the number of operations an exact number can be calculated or an order of magnitude. The latter method is less accurate than an exact calculation, but if the input size of the problem is relatively large, the calculation can give a good impression in which part of the network the granularity should be increased to have a significant reduction of the total latency. Therefore, it is concluded that *latency determinant parameters can be specified in a hardware independent manner and are useful for the defined software synthesis method.*

Thesis question Q4 is about how the NEC can generate the optimal granularity for the scheduler. The simplest approach is to exploit maximum data parallelism and thereby create maximum granularity. The scheduler will have to cluster the functions again to the number of processor nodes. This approach is not very efficient. It is concluded that *optimum granularity can't be determined without having some knowledge about the number of processor nodes*. In the implementation this was realized by setting the distribution parameters in the NDF to the right level. Optimal granularity is achieved when the number of functions is close or equal to the total number of processors, provided that the functions that contribute the largest part of the latency are expanded first.

7.2. Recommendations Network Expansion Compiler

In paragraph 6.3 some shortcomings of the prototype of the NEC were discussed. Besides the fact that some features were not implemented due to time constraints, this paragraph will give some recommendations for further improvement of the NEC.

7.2.1. Network paths

A few NEC-improvements can be made in relation to network paths, e.g. the determination of the critical path can be implemented in a more efficient way, the implementation of path optimisation and cycle detection.

Determine critical path

The current implementation to find the path with the largest latency after expansion is not very efficient. After expansion, every path is traced again to determine the critical path, while all the expanded paths have equal latency. To overcome this, it is sufficient to trace the original paths. The original paths have been attributed with distribution #1 or #0 if it has no distribution. This way the performance to determine the critical path after expansion is equal to the unexpanded network.

Another solution is to connect all input data element to a dummy node BEGIN and all outputs to dummy node END. A standard method or algorithm known from literature could be used to calculate the longest path. Linear algebra can calculate the critical path efficiently.

A useful feature that might save unnecessary or inefficient expansion of functions that don't contribute to the network latency significantly is to address the latency determinant function first.

Path optimisation

To implement path optimisation, the distribution administration can be used. Paths with the same distribution number are eligible for optimisation. Extra instances of data elements must be created in the network to create the optimised paths.

Cycles

From the beginning it was a goal to compile SDF-networks. To a certain extent this has been achieved, but the path finding algorithm of the NEC doesn't take cycles into account. With cycles delay loops can be modelled. When it is implemented, care should be taken that cycles are detected and that the path finding algorithm does not end up in an endless loop.

7.2.2. Hierarchy

As explained in 4.2.1, a node or function in a function network may be atomic, or may represent a function network at a lower hierarchy level. This type of hierarchy is not taken into account in the current implementation. One can imagine that the granularity and parallelism of the network could be enlarged if this lower level granularity is taken into account.

7.2.3. Types

In the current implementation, the NEC is abandoned when conflicting types are found on an interconnection, i.e. between a data element and a function connection. Although this rule can be maintained, it is quite strict.

The conflict between the type of a data element and a function connection can be avoided if the data element has no type. In fact, the data element represents only memory that can hold any kind of data type. If this approach were chosen, the check on the data type would shift to the interconnection of a function input and output and doesn't really solve the problem.

For example, the type QUANT_VIDEO, CORNER_TURN_VIDEO, WEIGHTED_VIDEO and FFT_VIDEO have exactly the same data structure (2 * float), but different functional names. In this case, the FFT function cannot be connected to the TC function, as the CORNER_TURN_VIDEO and FFT_VIDEO have a different name. Of course, the data types in this example can be renamed to a more general name in the network specification, but the idea is to reuse the function specifications in multiple function network specifications.

Therefore, data types should be defined on a global level, e.g. in a central library. This would also support the standardisation of type definitions.

One can imagine that type conflicts can still occur, while two types differ only in name. A type alias might be implemented in the NEC to overcome these kinds of conflicts. The type alias declares that the types are equal.

7.3. General recommendations

The implementation of the compiler was rather laborious. Due to time constraints not all functionality was implemented in the prototype of the NEC: path optimisation, implementation of distribution and aggregation functions and multi dimension expansion were not realized.

Perhaps other solutions could have been chosen to implement the pre-processor. For example: a solution using linear algebra, genetic algorithms or loop unfolding methods. Also the use of a standard parallel programming language might have lead to faster results.

Some general remarks will be made here as recommendations for future work.

MPI, OpenMP and OpenCL

From the literature study in this thesis it was concluded that parallel programming standards like MPI and OpenMP are interesting standards to be followed up.

Recently (2008), the open standard for heterogeneous parallel programming OpenCL (Open Computing Language) has been released. It is an efficient C-based parallel programming model that abstracts the specifics of underlying hardware.

The results of this thesis can be reused in such a development environment, e.g. the NEC could generate output formats for MPI, OpenMP or OpenCL.

Nested Loop Programs

In retrospect, the definition of a specific specification language and graphical language was rather a large effort. Instead, an existing imperative language could have been used to express the function network. After all, expressions of an SDF application are limited to assignment statements and (nested) loops.

Suppose the source code of functions was instantiated in a network program. The compiler could search for iteration loops and unfold them to increase granularity.

In literature these kinds of applications are known as Nested Loop Programs (NLP). All kind of techniques like loop unfolding and index skewing can be used for NLP. For further information on this subject, the reader is referred to [15].

Genetic algorithm for expansion and scheduling

Some steps have been performed to explore an implementation of the pre-processor with a genetic algorithm. The layered approach could be applied again for expansion and scheduling.

Use of the incidence matrix and linear algebra

The use of linear algebra might help to implement the cost functions in an efficient manner. Linear algebra offers a lot of solutions to common problems like critical path finding, path counting, etc. For this purpose, the incidence matrix that is often used to model SDF-diagrams (as described in 2.6.2) can be extended with dimension parameters.

8. Abbreviations

ACS	Antenna Control System
ADF	Architecture Description File
ASIC	Application Specific Integrated Circuit
BNF	Backus-Naur Form
C&C	Command & Control
CPI	Clock cycles Per Instruction
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DF	Data Flow
DP	Data Processing
ENDF	Expanded Network Description File
FDF	Function Description File
FIFO	First In First Out
FPGA	Field Programmable Gate Array
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MODERN	Modelling and Design Environment for Relational Networks
MP	MultiProcessing
MPI	Message Passing Interface
NDF	Network Description File
NEC	Network Expansion Compiler
OPS	Number of Operations
RSG	Radar Signal Generator
Rx	Receiver
SDF	Synchronous Data Flow
SISD	Single Instruction Multiple Data
SIMD	Single Instruction Multiple Data
SM	Sensor Management
SP	Signal Processing
TUP	Technical Unit Processing
Tx	Transmitter

9. References

[1] J. van Bemmel, Static Multi-Processor Scheduling on Generic Architectures, afstudeerverslag Universiteit Twente, November 2000

[2] High Performance Computer Architecture, http://www.ipp.mpg.de/~rfs/comas/Helsinki/helsinki04/CompScience/csep/csep1.phy.ornl.gov/ca/ca.html

[3] A. van der Steen and Jack J. Dongarra, Overview Of Recent Supercomputers, 15th edition, June 2002

[4] M.J. Flynn, Some computer organisations and their effectiveness, IEEE Trans. on computers, Vol. C-21, 9, 1972

[5] J.E. Shore, Second thoughts on parallel processing, Comput. Elect. Eng., 1973

[6] R. Hockney and C. Jesshope, Parallel Computers 2, 1988

[7] K. Hwang, Advanced Computer Architecture, 1993

[8] S. Bhattacharyya et al., The Ptolemy Almagest User Manual, EECS Dept., University of California, Berkeley, 1990 – 1997

[9] P.S. Pacheco, A users guide to MPI, Department of Mathematics, University of San Francisco, San Francisco, 1998

[10] OpenMP C and C++, Application Program Interface, version 2.0, March 2002

[11] E.A. Lee and D.G. Messerschmitt, Synchronous Data Flow, Proceedings of the IEEE, Vol. 75, No. 9, pp.1235-1245, September 1987

[12] E.A. Lee and D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput., Vol. C-36, No. 2, pp. 24-35, January 1987

[13] Thomas Nieman, A compact guide to Lex and Yacc, http://epaperpress.com/lexandyacc/

[14] Emden Gansner and Eleftherios Koutsofios and Stephen North, Drawing graphs with dot, February 4, 2002

[15] Todor Stefanov, Bart Kienhuis, Ed Deprettere, Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instance, Proc. 10th Int. Symposium on Hardware/Software Codesign, Estes Park, Colorado, USA, May 6–8, 2002

APPENDIX A Specifications

A.1 Functional Network Specification in Backus-Naur Form

```
Network:
    NETWORK NetworkName
    TYPES TypeDeclarations
    DIMENSIONS DimensionDeclarations
    INPUTS DataElements
    OUTPUTS DataElements
    LOCALS DataElements
    FUNCTIONS Functions
    INTERCONNECTIONS Interconnections ;

TypeDeclarations:
    /* empty */
    | TypeDeclaration TypeDeclarations ;

TypeDeclaration:
    TypeName '=' TypeExpression ';' ;

TypeExpression:
    Factor
    | Factor '+' TypeExpression ;

Factor:
    NUMBER '*' Factor
    | Term ;

Term:
    IDENTIFIER
    | '(' TypeExpression ')' ;

DimensionDeclarations:
    /* empty */
    | DimensionDeclaration DimensionDeclarations ;

DimensionDeclaration:
    DimensionName '=' NUMBER ':' NUMBER ';' ;
    | DimensionName '=' NUMBER ':' NUMBER ':' NUMBER ';' ;

DataElements:
    /* empty */
    | DataElement DataElements ;

DataElement:
    TypeName DataElementName ';'
    | TypeName DataElementName '[' Dimensions ']' ';' ;

Functions:
    FunctionPrototype ';'
    | FunctionPrototype ';' Functions ;

FunctionPrototype:
    FunctionName '(' INPUTS Connections ';' OUTPUTS Connections ';' ')' ':' Operations ;

Connections:
    Connection
    | Connection ',' Connections ;

Connection:
    ConnectionTypeName
    | ConnectionTypeName '[' Dimensions ']' ;

ConnectionTypeName:
    TypeName ConnectionName ;

Dimensions:
    Dimension
    | Dimension ',' Dimensions ;

Dimension:
    DimensionName
    | DimensionName '#' NUMBER
```

```
    | DimensionName '*' ;

DimensionName:
    IDENTIFIER ;

DimName:
    IDENTIFIER ;

Operations:
    OPS '(' DimensionExpression ')' ;

DimensionExpression:
    MathExpression
    | DimensionExpression '*' DimensionExpression
    | DimensionExpression '/' DimensionExpression
    | DimensionExpression '+' DimensionExpression
    | DimensionExpression '-' DimensionExpression
    | '(' DimensionExpression ')'
;

MathExpression:
    NUMBER
    | FLOAT
    | DimName
    |  LOG '(' DimName ')'
    |  POW '(' NUMBER ',' DimName ')'
    |  POW '(' DimName ',' NUMBER ')' ;

Interconnections:
    Interconnection
    | Interconnection Interconnections ;

Interconnection:
    NodeEdgeNode ';'
    | NodeEdgeNode '[' Dimensions ']' ';' ;

NodeEdgeNode:
    Node
    EDGE Node ;

Node:
    DataElementName
    | FunctionConnection ;

FunctionConnection:
    FunctionName '#' NUMBER '(' ConnectionName ')'
    | FunctionName '#' NUMBER '(' ConnectionName '[' Dimensions ']' ')' ;

NetworkName:
    IDENTIFIER ;

TypeName:
    IDENTIFIER ;

DataElementName:
    IDENTIFIER ;

FunctionName:
    IDENTIFIER ;

ConnectionName:
    IDENTIFIER ;
```

A.2 DOT-file radar application demo

```
digraph G {

/* graph attributes */

rankdir = TB
ratio = 1.3
size = "12,16"
rank = max
fontname = Helvetica
fontsize = 10
label = "network latency = 591970"

/* default function attributes */
node [shape = ellipse, height = 0.5, width = 0.75, fontname = Helvetica, fontsize = 10, style =
filled, color = "yellow2"];

/* function nodes */

TC_1 [label = "TC\#1\n(1, 65536, 98304)"];
WV_1 [label = "WV\#1\n(512, 16, 8224)"];
FFT_1 [label = "FFT\#1\n(512, 32, 16416)"];
LM_1 [label = "LM\#1\n(4096, 100, 409602)"];
MN_1 [label = "MN\#1\n(512, 16, 8208)"];
DT_1 [label = "DT\#1\n(512, 100, 51216)"];
WV_2 [label = "WV\#2\n(512, 16, 8224)"];
FFT_2 [label = "FFT\#2\n(512, 32, 16416)"];
LM_2 [label = "LM\#2\n(4096, 100, 409602)"];
MN_2 [label = "MN\#2\n(512, 16, 8208)"];
DT_2 [label = "DT\#2\n(512, 100, 51216)"];

/* data elements */

/* default data elements attributes */
node [shape = diamond, height = 0.2, width = 0.2, fontname = Helvetica, fontsize = 10, style =
filled, color = "green2"];

Squant[color = "orange1"];
LMOffset[color = "orange1"];
Lsb[color = "orange1"];
LMMeanCorr[color = "orange1"];
Tdet[color = "orange1"];
Hits[color = "red2"];
Sturn;
Sweight;
Sfft;
Slogmod;
Ln;

/* edges */

/* default interconnection attributes */
edge [fontname = Helvetica, fontsize = 10, dir = forward];

/* edges */

Squant -> TC_1 [label = "[Nr, Ns]"];
TC_1 -> Sturn [label = "[Ns, Nr]"];
Sturn -> WV_1 [label = "[Ns, Nr#1]"];
WV_1 -> Sweight [label = "[Ns, Nr#1]"];
Sweight -> FFT_1 [label = "[Ns, Nr#1]"];
FFT_1 -> Sfft [label = "[N, Nr#1]"];
Sfft -> LM_1 [label = "[N#1, Nr#1]"];
Lsb -> LM_1;
LMOffset -> LM_1;
LM_1 -> Slogmod [label = "[N#1, Nr#1]"];
Slogmod -> MN_1 [label = "[N, Nr#1]"];
LMMeanCorr -> MN_1;
MN_1 -> Ln [label = "[Nr#1]"];
Ln -> DT_1 [label = "[Nr#1]"];
Slogmod -> DT_1 [label = "[N, Nr#1]"];
Tdet -> DT_1;
DT_1 -> Hits [label = "[Nr#1]"];
```

```
Sturn -> WV_2 [label = "[Ns, Nr#2]"];
WV_2 -> Sweight [label = "[Ns, Nr#2]"];
Sweight -> FFT_2 [label = "[Ns, Nr#2]"];
FFT_2 -> Sfft [label = "[N, Nr#2]"];
Sfft -> LM_2 [label = "[N#2, Nr#2]"];
Lsb -> LM_2;
LMOffset -> LM_2;
LM_2 -> Slogmod [label = "[N#2, Nr#2]"];
Slogmod -> MN_2 [label = "[N, Nr#2]"];
LMMeanCorr -> MN_2;
MN_2 -> Ln [label = "[Nr#2]"];
Ln -> DT_2 [label = "[Nr#2]"];
Slogmod -> DT_2 [label = "[N, Nr#2]"];
Tdet -> DT_2;
DT_2 -> Hits [label = "[Nr#2]"];

}
```

A.3 ENDF radar application demo

```
NETWORK demo

/* type definitions */

TYPES:

QUANT_VIDEO = 2 * float;
CORNER_TURN_VIDEO = 2 * float;
WEIGHTED_VIDEO = 2 * float;
FFT_VIDEO = 2 * float;
LSB = float;
LOGMOD_VIDEO = float;
LOGMOD_OFFSET = float;
LM_MEAN_CORRECTION = float;
LN = float;
THRESHOLD_DET = float;
HITS = ( float );

/* dimension declarations */

DIMENSIONS:

Nr = 1024 : 2 : 1;
Ns = 16 : 2 : 0;
N = 16 : 2 : 0;

/* data declarations */

INPUTS:

QUANT_VIDEO Squant[Nr, Ns];
LOGMOD_OFFSET LMOffset;
LSB Lsb;
LM_MEAN_CORRECTION LMMeanCorr;
THRESHOLD_DET Tdet;

OUTPUTS:

HITS Hits;

LOCALS:

CORNER_TURN_VIDEO Sturn;
WEIGHTED_VIDEO Sweight;
FFT_VIDEO Sfft;
LOGMOD_VIDEO Slogmod;
LN Ln;

/* used functions */

FUNCTIONS:

TC
(
    INPUTS:
        QUANT_VIDEO quant[Nr, Ns];
    OUTPUTS:
        CORNER_TURN_VIDEO turn[Ns, Nr];
) : OPS ( 65536 );

WV
(
    INPUTS:
        CORNER_TURN_VIDEO turn[Ns];
    OUTPUTS:
        WEIGHTED_VIDEO weight[Ns];
) : OPS ( 16 );

FFT
(
    INPUTS:
        WEIGHTED_VIDEO weight[Ns];
```

```
    OUTPUTS:
        FFT_VIDEO fft[Ns];
) : OPS ( 32 );

LM
(
    INPUTS:
        FFT_VIDEO fft,
        LSB lsb,
        LOGMOD_OFFSET lmoffset;
    OUTPUTS:
        LOGMOD_VIDEO logmod;
) : OPS ( 100 );

MN
(
    INPUTS:
        LOGMOD_VIDEO logmod[Ns],
        LM_MEAN_CORRECTION meancorr;
    OUTPUTS:
        LN ln;
) : OPS ( 16 );

DT
(
    INPUTS:
        LN ln,
        LOGMOD_VIDEO logmod,
        THRESHOLD_DET tdet;
    OUTPUTS:
        HITS hits;
) : OPS ( 100 );


/* network connections */

INTERCONNECTIONS:

/* TC#1
 * operations  : 65536
 * latency  : 98304
 * rate  : 1
 */

Squant -> TC#1(quant[Nr, Ns]);
TC#1(turn[Ns, Nr]) -> Sturn;

/* WV#1
 * operations  : 16
 * latency  : 8224
 * rate  : 512 [Nr#1]
 */

Sturn -> WV#1(turn[Ns]) [Nr#1];
WV#1(weight[Ns]) -> Sweight [Nr#1];

/* FFT#1
 * operations  : 32
 * latency  : 16416
 * rate  : 512 [Nr#1]
 */

Sweight -> FFT#1(weight[Ns]) [Nr#1];
FFT#1(fft[N]) -> Sfft [Nr#1];

/* LM#1
 * operations  : 100
 * latency  : 409602
 * rate  : 4096 [N#1, Nr#1]
 */

Sfft -> LM#1(fft) [N#1, Nr#1];
Lsb -> LM#1(lsb);
LMOffset -> LM#1(lmoffset);
```

```
LM#1(logmod) -> Slogmod [N#1, Nr#1];

/* MN#1
 * operations  : 16
 * latency  : 8208
 * rate  : 512 [Nr#1]
 */

Slogmod -> MN#1(logmod[N]) [Nr#1];
LMMeanCorr -> MN#1(meancorr);
MN#1(ln) -> Ln [Nr#1];

/* DT#1
 * operations  : 100
 * latency  : 51216
 * rate  : 512 [Nr#1]
 */

Ln -> DT#1(ln) [Nr#1];
Slogmod -> DT#1(logmod[N]) [Nr#1];
Tdet -> DT#1(tdet);
DT#1(hits) -> Hits [Nr#1];

/* WV#2
 * operations  : 16
 * latency  : 8224
 * rate  : 512 [Nr#2]
 */

Sturn -> WV#2(turn[Ns]) [Nr#2];
WV#2(weight[Ns]) -> Sweight [Nr#2];

/* FFT#2
 * operations  : 32
 * latency  : 16416
 * rate  : 512 [Nr#2]
 */

Sweight -> FFT#2(weight[Ns]) [Nr#2];
FFT#2(fft[N]) -> Sfft [Nr#2];

/* LM#2
 * operations  : 100
 * latency  : 409602
 * rate  : 4096 [N#2, Nr#2]
 */

Sfft -> LM#2(fft) [N#2, Nr#2];
Lsb -> LM#2(lsb);
LMOffset -> LM#2(lmoffset);
LM#2(logmod) -> Slogmod [N#2, Nr#2];

/* MN#2
 * operations  : 16
 * latency  : 8208
 * rate  : 512 [Nr#2]
 */

Slogmod -> MN#2(logmod[N]) [Nr#2];
LMMeanCorr -> MN#2(meancorr);
MN#2(ln) -> Ln [Nr#2];

/* DT#2
 * operations  : 100
 * latency  : 51216
 * rate  : 512 [Nr#2]
 */

Ln -> DT#2(ln) [Nr#2];
Slogmod -> DT#2(logmod[N]) [Nr#2];
Tdet -> DT#2(tdet);
DT#2(hits) -> Hits [Nr#2];
```

A.4 FDF radar application demo

```
LATENCY = 100 MS
RATE = 20 HZ

TYPES

QUANT_VIDEO_1024_16 = 1024 * 16 * 2 * float
CORNER_TURN_VIDEO_16_1024 = 16 * 1024 * 2 * float
CORNER_TURN_VIDEO_512_16 = 512 * 16 * 2 * float
WEIGHTED_VIDEO_512_16 = 512 * 16 * 2 * float
WEIGHTED_VIDEO_512_16 = 512 * 16 * 2 * float
FFT_VIDEO_512_16 = 512 * 16 * 2 * float
FFT_VIDEO_4096 = 4096 * 2 * float
LSB_4096 = 4096 * float
LOGMOD_OFFSET_4096 = 4096 * float
LOGMOD_VIDEO_4096 = 4096 * float
LOGMOD_VIDEO_512_16 = 512 * 16 * float
LM_MEAN_CORRECTION_512 = 512 * float
LN_512 = 512 * float
LN_512 = 512 * float
LOGMOD_VIDEO_512_16 = 512 * 16 * float
THRESHOLD_DET_512 = 512 * float
HITS_512 = 512 * ( float )
CORNER_TURN_VIDEO_512_16 = 512 * 16 * 2 * float
WEIGHTED_VIDEO_512_16 = 512 * 16 * 2 * float
WEIGHTED_VIDEO_512_16 = 512 * 16 * 2 * float
FFT_VIDEO_512_16 = 512 * 16 * 2 * float
FFT_VIDEO_4096 = 4096 * 2 * float
LSB_4096 = 4096 * float
LOGMOD_OFFSET_4096 = 4096 * float
LOGMOD_VIDEO_4096 = 4096 * float
LOGMOD_VIDEO_512_16 = 512 * 16 * float
LM_MEAN_CORRECTION_512 = 512 * float
LN_512 = 512 * float
LN_512 = 512 * float
LOGMOD_VIDEO_512_16 = 512 * 16 * float
THRESHOLD_DET_512 = 512 * float
HITS_512 = 512 * ( float )
QUANT_VIDEO = 2 * float
LOGMOD_OFFSET = float
LSB = float
LM_MEAN_CORRECTION = float
THRESHOLD_DET = float
HITS = ( float )
CORNER_TURN_VIDEO = 2 * float
WEIGHTED_VIDEO = 2 * float
FFT_VIDEO = 2 * float
LOGMOD_VIDEO = float
LN = float

BASIC_FUNCTIONS

M_BEGIN
(
    OUTPUTS:
        QUANT_VIDEO Squant,
        LOGMOD_OFFSET LMOffset,
        LSB Lsb,
        LM_MEAN_CORRECTION LMMeanCorr,
        THRESHOLD_DET Tdet
) : 0 OPS

TC_1
(
   INPUTS:
        QUANT_VIDEO_1024_16 quant
    OUTPUTS:
        CORNER_TURN_VIDEO_16_1024 turn
) : 65536 OPS

WV_1
(
   INPUTS:
```

```
        CORNER_TURN_VIDEO_512_16 turn
    OUTPUTS:
        WEIGHTED_VIDEO_512_16 weight
) : 8192 OPS

FFT_1
(
   INPUTS:
        WEIGHTED_VIDEO_512_16 weight
    OUTPUTS:
        FFT_VIDEO_512_16 fft
) : 16384 OPS

LM_1
(
   INPUTS:
        FFT_VIDEO_4096 fft,
        LSB_4096 lsb,
        LOGMOD_OFFSET_4096 lmoffset
    OUTPUTS:
        LOGMOD_VIDEO_4096 logmod
) : 409600 OPS

MN_1
(
   INPUTS:
        LOGMOD_VIDEO_512_16 logmod,
        LM_MEAN_CORRECTION_512 meancorr
    OUTPUTS:
        LN_512 ln
) : 8192 OPS

DT_1
(
   INPUTS:
        LN_512 ln,
        LOGMOD_VIDEO_512_16 logmod,
        THRESHOLD_DET_512 tdet
    OUTPUTS:
        HITS_512 hits
) : 51200 OPS

WV_2
(
   INPUTS:
        CORNER_TURN_VIDEO_512_16 turn
    OUTPUTS:
        WEIGHTED_VIDEO_512_16 weight
) : 8192 OPS

FFT_2
(
   INPUTS:
        WEIGHTED_VIDEO_512_16 weight
    OUTPUTS:
        FFT_VIDEO_512_16 fft
) : 16384 OPS

LM_2
(
   INPUTS:
        FFT_VIDEO_4096 fft,
        LSB_4096 lsb,
        LOGMOD_OFFSET_4096 lmoffset
    OUTPUTS:
        LOGMOD_VIDEO_4096 logmod
) : 409600 OPS

MN_2
(
   INPUTS:
        LOGMOD_VIDEO_512_16 logmod,
        LM_MEAN_CORRECTION_512 meancorr
    OUTPUTS:
```

```
        LN_512 ln
) : 8192 OPS

DT_2
(
   INPUTS:
        LN_512 ln,
        LOGMOD_VIDEO_512_16 logmod,
        THRESHOLD_DET_512 tdet
    OUTPUTS:
        HITS_512 hits
) : 51200 OPS

M_END
(
    INPUTS:
        HITS Hits
) : 0 OPS

Squant
(
    INPUTS:
        QUANT_VIDEO M_BEGIN_Squant
    OUTPUTS:
        QUANT_VIDEO_1024_16 TC_1_quant
) : 0 OPS

LMOffset
(
    INPUTS:
        LOGMOD_OFFSET M_BEGIN_LMOffset
    OUTPUTS:
        LOGMOD_OFFSET_4096 LM_1_lmoffset,
        LOGMOD_OFFSET_4096 LM_2_lmoffset
) : 0 OPS

Lsb
(
    INPUTS:
        LSB M_BEGIN_Lsb
    OUTPUTS:
        LSB_4096 LM_1_lsb,
        LSB_4096 LM_2_lsb
) : 0 OPS

LMMeanCorr
(
    INPUTS:
        LM_MEAN_CORRECTION M_BEGIN_LMMeanCorr
    OUTPUTS:
        LM_MEAN_CORRECTION_512 MN_1_meancorr,
        LM_MEAN_CORRECTION_512 MN_2_meancorr
) : 0 OPS

Tdet
(
    INPUTS:
        THRESHOLD_DET M_BEGIN_Tdet
    OUTPUTS:
        THRESHOLD_DET_512 DT_1_tdet,
        THRESHOLD_DET_512 DT_2_tdet
) : 0 OPS

Hits
(
    INPUTS:
        HITS_512 DT_1_hits,
        HITS_512 DT_2_hits
    OUTPUTS:
        HITS M_END_Hits
) : 0 OPS

Sturn
(
```

```
    INPUTS:
        CORNER_TURN_VIDEO_16_1024 TC_1_turn
    OUTPUTS:
        CORNER_TURN_VIDEO_512_16 WV_1_turn,
        CORNER_TURN_VIDEO_512_16 WV_2_turn
) : 0 OPS

Sweight
(
    INPUTS:
        WEIGHTED_VIDEO_512_16 WV_1_weight,
        WEIGHTED_VIDEO_512_16 WV_2_weight
    OUTPUTS:
        WEIGHTED_VIDEO_512_16 FFT_1_weight,
        WEIGHTED_VIDEO_512_16 FFT_2_weight
) : 0 OPS

Sfft
(
    INPUTS:
        FFT_VIDEO_512_16 FFT_1_fft,
        FFT_VIDEO_512_16 FFT_2_fft
    OUTPUTS:
        FFT_VIDEO_4096 LM_1_fft,
        FFT_VIDEO_4096 LM_2_fft
) : 0 OPS

Slogmod
(
    INPUTS:
        LOGMOD_VIDEO_4096 LM_1_logmod,
        LOGMOD_VIDEO_4096 LM_2_logmod
    OUTPUTS:
        LOGMOD_VIDEO_512_16 MN_1_logmod,
        LOGMOD_VIDEO_512_16 DT_1_logmod,
        LOGMOD_VIDEO_512_16 MN_2_logmod,
        LOGMOD_VIDEO_512_16 DT_2_logmod
) : 0 OPS

Ln
(
    INPUTS:
        LN_512 MN_1_ln,
        LN_512 MN_2_ln
    OUTPUTS:
        LN_512 DT_1_ln,
        LN_512 DT_2_ln
) : 0 OPS

CONNECTIONS

TC_1 -> [ Squant[TC_1_quant] ]
WV_1 -> [ Sturn[WV_1_turn] ]
FFT_1 -> [ Sweight[FFT_1_weight] ]
LM_1 -> [ Sfft[LM_1_fft], Lsb[LM_1_lsb], LMOffset[LM_1_lmoffset] ]
MN_1 -> [ Slogmod[MN_1_logmod], LMMeanCorr[MN_1_meancorr] ]
DT_1 -> [ Ln[DT_1_ln], Slogmod[DT_1_logmod], Tdet[DT_1_tdet] ]
WV_2 -> [ Sturn[WV_2_turn] ]
FFT_2 -> [ Sweight[FFT_2_weight] ]
LM_2 -> [ Sfft[LM_2_fft], Lsb[LM_2_lsb], LMOffset[LM_2_lmoffset] ]
MN_2 -> [ Slogmod[MN_2_logmod], LMMeanCorr[MN_2_meancorr] ]
DT_2 -> [ Ln[DT_2_ln], Slogmod[DT_2_logmod], Tdet[DT_2_tdet] ]
Squant -> [ M_BEGIN[Squant] ]
LMOffset -> [ M_BEGIN[LMOffset] ]
Lsb -> [ M_BEGIN[Lsb] ]
LMMeanCorr -> [ M_BEGIN[LMMeanCorr] ]
Tdet -> [ M_BEGIN[Tdet] ]
Hits -> [ DT_1[hits], DT_2[hits] ]
Sturn -> [ TC_1[turn] ]
Sweight -> [ WV_1[weight], WV_2[weight] ]
Sfft -> [ FFT_1[fft], FFT_2[fft] ]
Slogmod -> [ LM_1[logmod], LM_2[logmod] ]
Ln -> [ MN_1[ln], MN_2[ln] ]
M_END -> [ Hits[DT_1_hits] ]
```

A.5 ADF radar application demo

```
PROCESSORSPEED = 500000000 -- NOPS/S

TYPES
   int = 4
   float = 4

COMPONENTS

HexC44 : [ processor(1 MB): A,B,C,D,E,F ]
{
   A <-> B = 160  -- bi-directional, 15 operations send_delay
   A <-> D = 160  -- bi-directional, 15 operations send_delay
   A <-> F = 160  -- bi-directional, 15 operations send_delay

   B <-> C = 160  -- bi-directional, 15 operations send_delay
   B <-> E = 160  -- bi-directional, 15 operations send_delay

   C <-> D = 160  -- bi-directional, 15 operations send_delay
   C <-> F = 160  -- bi-directional, 15 operations send_delay

   D <-> E = 160  -- bi-directional, 15 operations send_delay

   E <-> F = 160  -- bi-directional, 15 operations send_delay

   -- without these, mapping will not find a solution
   B <-> D = 160
   B <-> F = 160
}
```

This ADF example represents a processor architecture as shown in Figure 9-1.

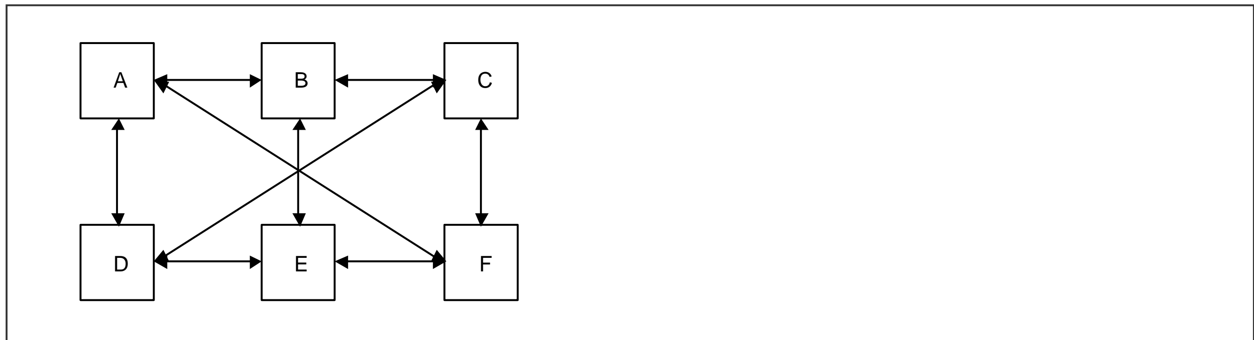


Figure 9-1 Hexagonal processor architecture

The degree of every node is 3; the diameter of this hexagonal network is 2.