MASTER THESIS

HARDWARE ACCELERATOR SHARING WITHIN AN MPSOC WITH A CONNECTIONLESS NOC

Gerben Wevers

DEPARTMENT OF ELECTRICAL ENGINEERING, MATHEMATHICS AND COMPUTER SCIENCE
COMPUTER ARCHITECTURES FOR EMBEDDED SYSTEMS

EXAMINATION COMMITTEE
Prof. dr. ir. M.J.G. Bekooij
Dr. ir. J.F. Broenink
B.H.J. Dekens, M.Sc.

16-09-2014

UNIVERSITY OF TWENTE.

University of Twente

Master Thesis

Hardware Accelerator Sharing within an MPSoC with a connectionless NoC

Author:
G.G.A. Wevers

Student number:
s0144053

Committee:
Prof. dr. ir. M.J.G Bekooij
Dr. ir. J.F. Broenink
B.H.J Dekens, Msc

Research Group Computer Architecture for Embedded Systems,
Department of EEMCS University of Twente, Enschede, The Netherlands

September 16, 2014

UNIVERSITEIT TWENTE.

COMMIT/

Abstract

For the last decades, increasing the computational performance of a microprocessor chip was mainly achieved by scaling transistor sizes. Not only can more transistors be placed in a single die, smaller transistors allow higher clock frequencies. While transistor sizes are still decreasing, designers are facing mayor power consumption issues which prevent further performance improvements by simply increasing clock frequencies. A clear trend is visible where multiple cores are added to the same chip to form so-called multi-core systems.

The same trend is visible in the embedded systems domain where System-on-Chips (SoCs) are transformed into Multi-Processor System-on-Chips (MPSoCs). An MPSoC can either be homogeneous (consisting of identical processing elements) or heterogeneous (consisting of different types of processing elements, e.g. Central Processing Units (CPUs) and weakly programmable hardware accelerators). Communication between the processing elements is taking place via a Network-on-Chip (NoC).

At the University of Twente multiple researchers are working on an MPSoC called Starburst. The main characteristics of this platform are: (1) the Starburst platform is a scalable distributed shared memory many-core system, (2) it targets real-time streaming applications where firm real-time requirements are assumed and (3) the set of applications to be run on the platform (and thus the communication pattern) is unknown at design time, which requires the platform to be flexible.

The Starburst platform was originally designed as a homogeneous MPSoC consisting of multiple identical soft-core CPUs. As a case study a Phase Alternating Line (PAL) video decoding application was mapped onto the platform. The demonstrator produced a video quality far from commercially acceptable and for some operations multiple parallel executing CPUs were required. This case study showed that the computational power was simply limited by the performance of the CPUs in the system. In order to improve the computational performance of the Starburst platform weakly programmable hardware accelerators were added, which transformed Starburst from a homogeneous MPSoC to a heterogeneous MPSoC.

A problem of this approach is the fact that a hardware accelerator can only process a single data stream. In order to process multiple data streams, multiple physical copies of this hardware accelerator have to be added to the platform. This thesis focussed on techniques to share hardware accelerators across multiple data streams, as it was expected that one shareable hardware accelerator has a lower area footprint than multiple unshared accelerators. Additionally, the platform is becoming more flexible in the sense that more applications are able

to execute on the same platform.

We solved this problem by implementing a centralized component called a *gateway*. The gateway is used to buffer multiple incoming data streams, and to push packets of data at high speed sequentially through the accelerators. Most accelerators have a configuration and/or state. Context switches are applied where the configuration and state for a certain data stream is loaded into the accelerator and extracted after a specific number of samples have been processed. A dataflow model of the sharing mechanism is constructed, which allows us to give real-time guarantees such as latency and throughput.

A case study was carried out which focussed on the audio part of the PAL signal. As this signal contains a stereo audio signal, we have used the hardware accelerator sharing mechanism to decode the left and right audio stream on the same set of accelerators, where a continuous (real-time) audio signal is produced.

Acknowledgements

First of all, I would like to thank Marco for the numerous discussions we had and feedback I have received during my research. I highly appreciate his critical view on my ideas and solutions, and the discussions we had on the dataflow models, as it took some time to get them right.

I would like to thank Berend for the daily supervision, as he was always available to answer my questions. More than once I walked into his office with just a single simple question eventually leaving the room an hour later after discussing lots of new ideas for my project. I really liked his enthusiastic approach when brainstorming these new ideas or solving the numerous problems I encountered with the Starburst or Xilinx tooling.

Additionally I would like to thank Jochem Rutgers (who recently received his doctorate degree) as he originally designed and developed the Starburst platform. I could always ask him for help when annoying bugs popped up.

I would like to finish by thanking Lisanne, my girlfriend, for supporting and motivating me throughout this project. Without her finishing my master's thesis would be much, much harder.

Gerben Wevers
Enschede, September 2014

Contents

List of Figures

List of Tables

Acronyms

$\mathcal{LR}$	Latency-Rate.
ADC	Analog to Digital Converter.
ASIC	Application-Specific Integrated Circuit.
BE	Best Effort.
BRAM	Block Ram.
CF	Compact Flash.
CFIFO	C-HEAP FIFO.
CORDIC	Coordinate Rotation Digital Computer.
CPU	Central Processing Unit.
CRC	Cyclic Redundancy Check.
CSDF	Cyclo-Static Dataflow.
DAB	Digital Audio Broadcasting.
DDR3	Double Data Rate type 3 SDRAM.
DMA	Direct Memory Access.
DSP	Digital Signal Processor.
DSP48E1	Digital Signal Processing Element.
EM	Event Models.
FCFS	First-come, First-served.
FIFO	First-in, First-out.
FIR	Finite Impulse Response.
FPGA	Field-Programmable Gate-Array.
FPU	Floating Point Unit.
FSL	Fast Simplex Link.
GAM	Global Accelerator Manager.
GS	Guaranteed Service.
HD	High Definition.
HSDF	Homogeneous Synchronous Dataflow.
IIR	Infinite Impulse Response.
IP	Intellectual Property.
IPIC	Intellectual Property Interconnect.
ISA	Instruction Set Architecture.

ISE	Instruction Set Extension.
L2	Level-2.
LMB	Local Memory Bus.
LUT	Look-Up Table.
LUTRAM	LUT RAM.
MCM	Maximum Cycle Mean.
MMU	Memory Management Unit.
MPMC	Multi-Port Memory-Controller.
MPSoC	Multi-Processor System-on-Chip.
NI	Network Interface.
NoC	Network-on-Chip.
OO	Object Oriented.
OPB	On-chip Peripheral Bus.
PAL	Phase Alternating Line.
PLB	Processor Local Bus.
POSIX	Portable Operating System Interface.
RISC	Reduced Instruction Set Computer.
RNG	Random Number Generator.
ROM	Read-Only Memory.
ROSPS	Rate Optimal Static Periodic Schedule.
RPC	Remote Procedure Call.
RR	Round-Robin.
RTC	Real-time Calculus.
SDF	Synchronous Dataflow.
SDR	Software Defined Radio.
SoC	System-on-Chip.
SPM	Scratchpad Memory.
SPS	Static Periodic Schedule.
SSE	Streaming SIMD Extensions.
STS	Self-Timed Schedule.
TDM	Time-Division Multiplexing.
UART	Universal Asynchronous Receiver/Transmitter.
UHD	Ultra High Definition.
UML	Unified Modeling Language.
USB	Universal Serial Bus.
WCSTS	Worst-Case Self-Timed Schedule.

WCWT Worst-Case Waiting Time.

Chapter 1

Introduction

1.1 Context

When Intel invented the microprocessor in 1971, probably no one would have expected the impact it would have and unimaginable evolution it would go through in the upcoming decennia. Starting with a clock speed of 740 kilohertz and equipped with a mere 2300 transistors, 43 years later the latest Intel Haswell chips have around 1.4 billion transistors, operating on clock-speeds of several gigahertz.

The demand for such increases can be explained by looking at for example the domain of consumer electronics, such as video or audio applications. New standards such as Digital Audio Broadcasting (DAB), its successor DAB+, High Definition (HD) video (at a resolution of 1920×1080 pixels) and its successor Ultra High Definition (UHD) or 4K video (at a resolution of 3840×2160 pixels) clearly explain the demand for embedded systems with enough computing power to run these applications. The fact that a lot of these embedded systems are battery powered (e.g. mobile phones) explains why power efficiency is a second import requirement.

For the last decades, increasing the computational performance of a microprocessor chip was mainly driven by scaling transistor sizes. Reducing the size of a transistor has three important benefits [7]:

1. When reducing the dimension of a transistor, its total area reduces quadratic. E.g. a reduction of 30% ($\times 0.7$) in dimension, corresponds to a 50% shrink in area ($\times 0.7^2$). Or, as Moore's law dictates, allows the number of transistors to double.
2. A smaller transistor allows a higher clock frequency, due to lower internal delays. Increasing the clock-frequency directly improves the computational performance.
3. A smaller transistor requires less power to operate on.

While the dimensions of transistors are still decreasing, improving the performance by increasing the clock frequency stalled some years ago when hardware designers hit the *"power wall"* : the amount of energy required to power all transistors is larger than the amount of energy that can be supplied and can be dissipated. Instead of increasing the clock frequency, designers had to come with other ways to exploit the potential performance out of the increasing number of

transistors. This led to the development of so-called multi-core systems, where multiple processing cores have been placed in the same circuit.

Looking again at the application domain, a lot of the audio- and video applications are integrated in embedded systems. A clear trend is visible where all components (e.g. a general-purpose processor, a Digital Signal Processor (DSP) and memory) that are required to run these applications are integrated in a single die and form a SoC. The trend of adding multiple processing cores in the same circuit is also visible in this domain, resulting in the development of MPSoCs.

MPSoC can be divided in two distinct classes; the first class consists of homogeneous architectures which are based on the replication of identical processing units. The advantages of these types of architectures are mainly the ease of programmability, the flexibility, fault tolerance and scalability [19]. The other class consists of the heterogeneous architectures where next to the processors for example weakly programmable hardware accelerators have been added. Even though in general they are harder to program and are less flexible, they possess better energy efficiency compared to homogeneous architectures. Furthermore, they can improve real-time behavior by reducing conflicts among processing elements and tasks [30].

There are two broad classes of applications: *data-oriented* and *control-oriented* applications [6]. Image processing and audio processing are examples of applications belonging to the domain of data-oriented or streaming based applications. Usually a sequence of computations is applied on a stream of data, with little or no reuse of data. This is why the computations can often be executed in parallel. The other class consist of control-oriented applications such as an ABS controller in a car or a system to control the position of a robotic arm. The code of these applications often contain a lot of conditional branches, need to keep track of a large amount of state and often has a high amount of data reuse. This complicates parallel execution of computations.

Real-time systems are computing systems with a temporal constraint. The correctness of the result does not only depend on the value of the computation, but also at the time at which the result is produced [10]. This means that real-time systems must react within precise time constraints to events in the environment. The maximum time within which it must complete its execution is called a deadline. Based on the consequences of missing a deadline three categories of real-time systems can be distinguished:

- **Hard** - Missing a deadline is highly undesirable and may cause catastrophic consequences.
- **Firm** - Missing a deadline is highly undesirable, but does not cause any damage.
- **Soft** - Deadline misses lead to a graceful performance degradation.

When executing real-time applications, the ability to analyze a system is an

important requirement in order to give real-time guarantees. The analysis of heterogeneous designs is becoming increasingly complex, when multiple types of processing units (with different programming models), connected via some sort of interconnect, are added to the system. It is important to maintain as much analyzability per processing unit as possible. During this research an existing heterogeneous MPSoC will be modified. The most important requirement of these modifications is the fact that this happens while preserving analyzability.

1.2 Research platform

The previous section explained the context of this research. This section will focus on the research platform, in order to understand and formulate the problem description. Afterwards, the platform will be discussed in more detail in Chapter 3.

At the University of Twente multiple researchers are working on an MPSoC called Starburst. The main characteristics of this platform are [23]:

- The platform is a scalable distributed shared memory many-core system.
- The Starburst platform targets real-time streaming applications where *firm* real-time requirements are assumed. Deadline misses are highly undesirable but not catastrophic.
- The set of applications to be run on the platform (and thus the communication pattern) is unknown at design time. For this reason, the architecture must be flexible.

The Starburst MPSoC is being developed on a Xilinx ML605 development board. This development board is equipped with a Virtex-6 Field-Programmable Gate-Array (FPGA). A high level overview of the platform is given in Figure 1.1.

The platform consists of a power-of-two number of *processing tiles*. The processing tiles, also called MicroBlaze tiles, are equipped with a Xilinx MicroBlaze CPU, and are denoted with MB_x in Figure 1.1. The figure displays a 32-core configuration, which is currently also the maximum given the FPGA resources. One additional processor is added to the platform, running embedded Linux. This processor is connected to (and in control of) several on-board peripherals like the Ethernet port or the Universal Serial Bus (USB) controller. The linux core is therefore mainly used for interaction with the environment. The processors are so-called *softcore* CPUs, which can be fully implemented in reconfigurable hardware by the logic synthesis tools. Due to this reconfigurability it is possible to configure the CPU at design time, where different components like a Floating Point Unit (FPU), barrel shifter, hardware multiplier or hardware divider may be added to the system.

The processing tiles and the Linux core are connected to an interconnect which

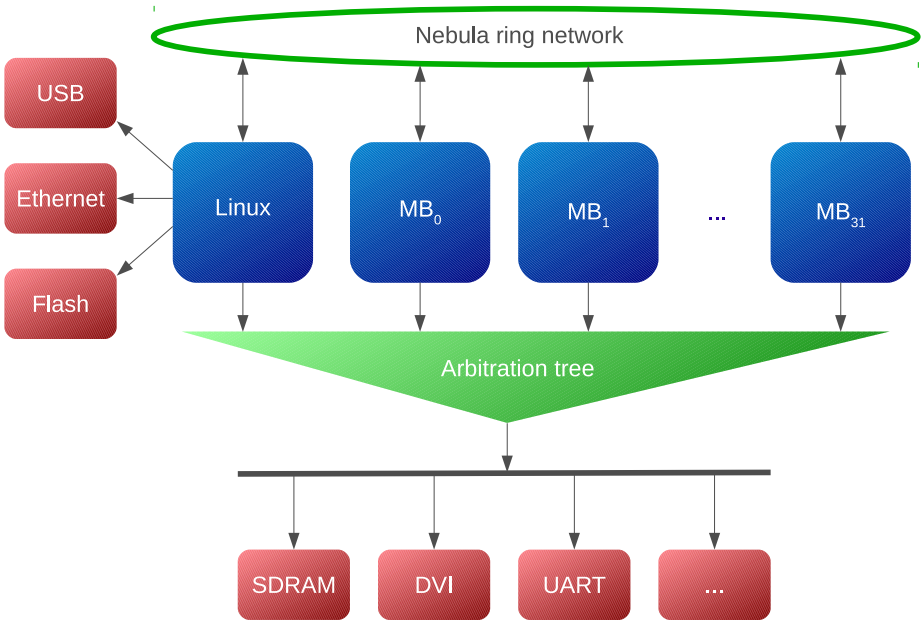

Figure 1.1: High level overview of the Starburst platform (the arrows indicate master/slave relations)

facilitates communication on the platform. The interconnect consists of two separate parts. The first part is the *Warpfield arbitration tree* which makes communication from and to shared resources like external Double Data Rate type 3 SDRAM (DDR3) memory possible. This is a *latency-critical* channel, where the performance of the platform directly degrades with a higher latency. Take for example a read operation from external DDR3 memory. When one of the CPUs issues a read operation, this CPU has to wait until its instruction has propagated through the complete interconnect, data is fetched out of memory and is propagated back through the interconnect before the CPU can continue with its computation. Meanwhile stall cycles are inserted in the pipeline of the CPU which results in a degradation of performance. For this reason the latency introduced by the arbitration tree is kept as low as possible. The arbitration tree provides a slave interface to each Microblaze in the system. When a Microblaze issues a read- or write-request this request is packetized and is given a timestamp and source ID. The packets enter a binary tree where at each step local arbitration based on the timestamp is applied. This way arbitration is applied according to a First-come, First-served (FCFS) policy, where a packet with the lowest timestamp is allowed to proceed first.

The second part is the *Nebula ring network*. This part implements all-to-all communication between the processing tiles. Instead of the more conventional interconnects like busses and mesh networks, this interconnect has a ring topol-

ogy with a very small area footprint [13]. Figure 1.2 gives an overview of the Nebula ring network. This figure shows that each CPU is connected via a Network Interface (NI) to a router, and each router is connected to its two neighbouring routers to create a ring-like structure. In the same figure the principle of executing a streaming application on the platform is depicted. Together the processors are processing a stream of data, by sending the results of their computation to another processor in the system, creating a chain of processors.

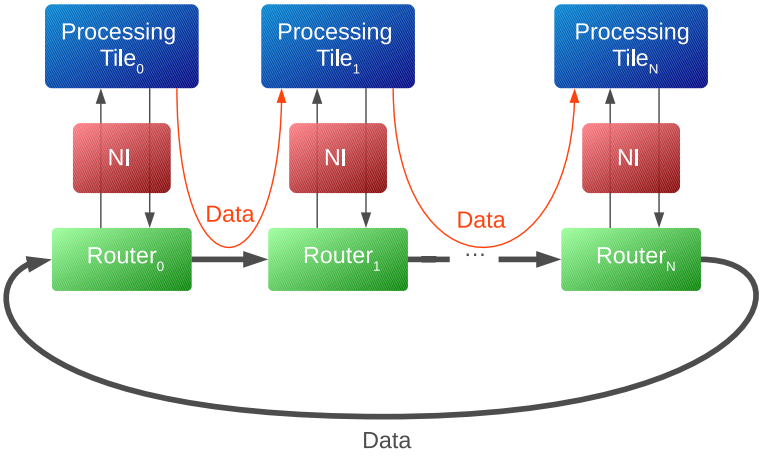

Figure 1.2: Schematic overview of the Nebula ring

As discussed in the previous section, the Starburst platform targets (real-time) streaming applications. A key advantage of streaming applications is the fact that they usually are latency insensitive, which means that the latency introduced by the communication channel has no influence on the performance of the complete system, if the throughput of the communication channel is high enough. Latency is a fundamental problem in digital circuit design because of the limited speed at which a signal by definition can travel. With digital circuits becoming increasingly larger, signals have to travel larger distances. But also when clock frequencies are increasing, the latency (expressed in the number of clock cycles) will increase. The Nebula ring network tries to exploit this *latency insensitive* property by only supporting *posted write* actions; a CPU does not have to wait on data being accepted by a receiver, instead the processor can immediately continue with its computations when data has been placed on the ring interconnect without having to insert stall cycles. Lossless communication, which guarantees that no data will be dropped during communication, is achieved in the software layers running on top of the hardware platform. Basically, after writing a packet to another processing tile, the receiving tile will write a message back, to acknowledge that the sending tile can write a new packet.

In order to demonstrate the performance of the Starburst platform multiple

applications have been mapped onto it. One of these applications is a PAL video decoder [13], which is a good example of a streaming application, where a sequence of computations has to be applied on the incoming stream of data. This case-study showed that the processing power of the homogeneous Starburst platform was simply limited by the computational power of the general purpose CPUs in the system. For example, the video decoder required four (out of 32) fully utilized CPUs to do one specific operation (I/Q amplitude detection).

Hardware acceleration

A commonly used technique to improve the performance of computationally expensive functions is to accelerate them using hardware accelerators. Hardware accelerators are dedicated pieces of hardware designed for one specific operation. By using hard-wired implementations the overhead of general purpose architectures is avoided, which results in increased performance and energy efficiency by orders of magnitude compared to an implementation on a general purpose architecture [17]. In [16] support for hardware accelerators was added to the Starburst platform. With this addition the Starburst platform changed from an homogeneous architecture to a heterogeneous architecture consisting of a mix of identical processors and a set of weakly programmable hardware accelerators.

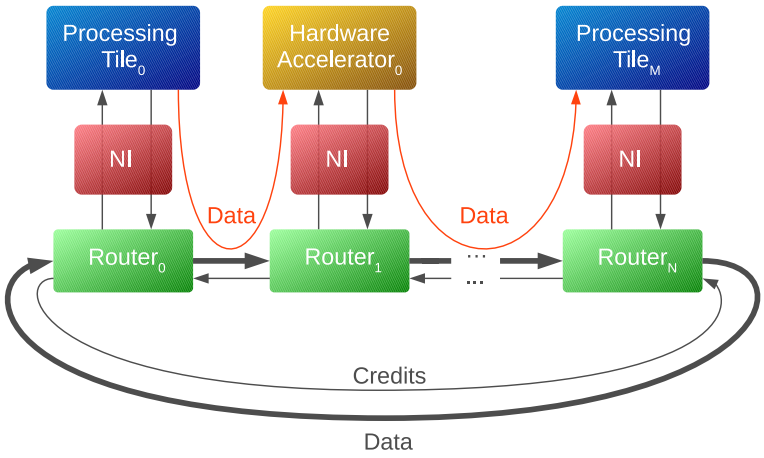

Figure 1.3: Schematic overview with one hardware accelerator, connected to the Nebula ring interconnect

Figure 1.3 gives a schematic overview of how hardware accelerators are integrated into the Starburst platform. As depicted in this figure, hardware accelerators are connected to the Nebula ring interconnect, just as how the processors are being connected; via a NI on a router. The hardware accelerators are again connected in a way optimized for streaming applications; a CPU will feed the hardware accelerator with data, when the hardware accelerator finishes

its computation it will forward the result to a second CPU. As discussed, lossless communication between CPUs is achieved in the software layers running on the processors. For the hardware accelerators another form of flow-control was required, this is because they are unaware of memory addresses and have no software running which manages communication.

Credit-based flow-control was implemented for this purpose. Chapter 3 will focus on this communication protocol in more detail, for now only the basic principle will be explained. Every hardware accelerator is equipped with a hardware-based First-in, First-out (FIFO)-buffer. Credits are denoting the number of free buffer positions in this FIFO-buffer. Processing tiles are only allowed to send data to a hardware accelerator if they have at least one credit, which corresponds to free space in the FIFO-buffer. When the hardware accelerator accepts a data sample, a credit is sent back over an additional *credit-ring* (see Figure 1.3) to the corresponding processing tile.

When a hardware accelerator receives a data sample, it is unable to determine the sender or the recipient; this information is simply not embedded in the data stream. Instead, the NIs to which the hardware accelerators are connected are programmed to give them the required knowledge. One register holds the memory address to which the accelerator has to forward its computed results, a second register holds the memory address to which credits have to be returned, furthermore a counter is implemented to keep track of the number of credits. This is an important property; a processor has to configure the hardware accelerator before the stream of data can be processed by the accelerator.

As a case-study, the PAL video decoder was modified, where the computationally expensive I/Q amplitude detection operation was implemented in hardware. This case-study showed some interesting results; mapping this specific operation on an hardware accelerator freed four CPUs, while the performance of this specific operation increased by 366%.

1.3 Problem Description

The previous two sections described the context of this research, and gave a high level overview of the research platform. This section will describe the problem in more detail.

As discussed before, the Starburst platform is designed to be flexible, because the set of applications to be run on the platform is unknown at compile time. The addition of hardware accelerators increases the performance, however it reduces the flexibility of the platform. It is, for example, currently impossible to process more than one stream by a single hardware accelerator, instead multiple (physical) copies of the same accelerator have to be added to the platform at design time.

Integrating hardware accelerators into the Starburst platform showed another interesting result. While hardware accelerators are able to process data at high speed, measurements (in [16]) indicated that the utilization of their hardware accelerator was around 6%. It turned out that if a CPU is used to feed an accelerator with new data, the throughput of the CPU is the bottleneck and therefore directly limiting the utilization of the hardware accelerator. As a solution one could reduce the computational performance of an accelerator in order to let it better match the computational performance of a CPU. Generally this results in a lower area footprint, as the implemented logic is used multiple times for one computation. Trading area for speed is a common practice when designing (digital) hardware.

The need for a more flexible usage of hardware accelerators, combined with the fact the hardware accelerators are currently most of the time in an idle state waiting for new data, sparked the idea to multiplex multiple data streams over the hardware accelerators. The idea is that by sharing these hardware accelerators across multiple streams, the platform is becoming more flexible in the sense that more applications are able to execute on the same platform. A second advantage of mapping multiple streams over a single hardware accelerator is the fact that this opens the door of increasing the utilization of the hardware accelerator, where the additional logic of the accelerator is used more efficiently.

Sharing one (high performance) accelerator instead of adding multiple (low performance) physical copies is an interesting consideration. The first solution is expected to be more flexible in the sense that more applications can be executed on the same platform. The second solution is expected to be less complicated, because no (complex) sharing mechanisms have to be implemented. The decision was made to focus on the first option where multiple data-streams are multiplexed over a single hardware accelerator. A flexible solution is preferred and eventually high speed data coming from external peripherals such as an Analog to Digital Converter (ADC) daughter board should be processed on the hardware accelerators, which requires high performance accelerators in the first place. It is however unknown what additional hardware is required to facilitate hardware accelerator sharing, and whether this additional hardware can justify the amount of hardware saved by sharing these components. This is something which will be part of this thesis.

When designing a method which implements hardware accelerator sharing, the most important requirement of this method is the fact that it should be predictable. We define predictability as the ability to construct a sufficiently accurate temporal analysis model of the hardware design for which a computational efficient analysis algorithm exists. With this model calculations can be performed and useful numbers can be extracted. One can think of numbers like minimum throughput or maximum latency when processing a data stream over a shared hardware accelerator. For this reason multiple real-time analysis methods are discussed in Section 2.2. Processing multiple data streams over a single hardware accelerator can be seen as a form of *resource sharing*. Capturing the

effects of resource sharing in a real-time analysis model will be discussed in Section 2.3.

A producer can start using a hardware accelerator after it has been configured. As discussed before, a hardware accelerator should be programmed with an address to which data has to be forwarded to and an address to which credits have to be returned. Furthermore, most accelerators operate based on a certain configuration. One can think of the coefficients of an Finite Impulse Response (FIR) filter or the mixing frequency of a digital signal mixer. The configuration can easily be altered by writing new configuration values to specific registers of a hardware accelerator over the Nebula ring interconnect. In the case of hardware accelerator sharing this means that it should be able to handle multiple accelerator configurations. Next to a configuration, a hardware accelerator might have state. One could think of the registers of an FIR filter which hold the intermediate results. Again, this means that the hardware accelerator should be able to handle multiple states in order to process multiple data streams.

There are some preliminary ideas to handle configuration and state. For example by equipping a hardware accelerator with multiple sets of registers which hold the configuration and state for multiple data streams. This way the hardware accelerator will use a set of registers based on the incoming data stream. A second idea solves this problem by constantly updating the state and configuration from a remote CPU, before a data stream is processed by this accelerator. While the first solution raise questions about the increased hardware usage and flexibility, the *context switches* of the second idea will introduce an overhead which will reduce the utilization and throughput of the hardware accelerator. How to handle state and configuration is a problem which will be researched during this thesis.

Currently the hardware accelerators exhibit a relative low utilization, where they are most of the time waiting on new data to be processed. The CPUs seem to be the bottleneck, where they are simply unable to feed the accelerators with data fast enough. Increasing the utilization of the hardware accelerators is something which has to be taken into account when designing a method to share hardware accelerators.

1.4 Research Questions

The goal of this research is to multiplex multiple data streams over a hardware accelerator. This way the Starburst MPSoC is becoming more flexible, more efficient and can have a lower area footprint. The targeted streaming applications have firm real-time requirements, and for this reason we aim for a predictable and analyzable solution. The goal is to come with an initial implementation to give an evaluation of hardware costs and resource utilization. From this summary, the following research questions are extracted:

- How to handle multiple configurations and states for a hardware accelerator?
- Is it possible to increase the currently low utilization of the hardware accelerators via hardware accelerator sharing?
- Which techniques can reduce the overhead introduced by switching between data-streams?
- What additional hardware is required to share hardware accelerators, and can this additional hardware be justified by the amount of hardware saved by sharing hardware accelerators?
- Can a (detailed) dataflow model of this sharing protocol be constructed, and related to that, can an abstraction of this dataflow model be created, which is easier to use at the cost of a less accurate dataflow model?
- Can the real-time behaviour be accurately predicted?
- Can we find an algorithm to calculate proper buffer sizes, and related to that, can we compute at which granularity the switches between data-streams should take place?

1.5 Contributions

As the main contributions of this work we:

- Identify multiple architectural alternatives (Chapter 4).
- Develop a hardware accelerator sharing mechanism which can be used to process multiple data streams on the same set of hardware accelerators (Chapter 5). This implementation includes:
  - A gateway module implemented on a MicroBlaze CPU, used to manage the accelerators and to implement a Round-Robin scheduling policy.
  - An interface from this gateway module to the hardware accelerators, to be able to read and write configuration and state from and to the accelerators.
  - A Direct Memory Access (DMA) controller, used to speed up the transfer of data through the accelerators.
- Construct a dataflow analysis model which can eventually be used to determine proper buffer sizes and to calculate achieved throughput (Chapter 6).

- Create an operation setup where hardware accelerator sharing is used to run a PAL audio application (Chapter 7). This setup is used for the evaluation of hardware costs and performance.

1.6 Outline

The remainder of this thesis starts by presenting related work in Chapter 2. It includes research about hardware accelerator integration methods, real-time analysis models, and modelling run-time arbitration in dataflow models. After that, the research platform is presented in more detail in Chapter 3.

Next we will focus on actual problem of this thesis, starting with presenting multiple architectural alternatives in Chapter 4, where the details of the actual implementation are presented in Chapter 5. Next, a dataflow model of the sharing mechanism is constructed in Chapter 6, and a case study is carried out in Chapter 7 focussing on decoding stereo audio via hardware accelerator sharing.

Finally, conclusions and future work are included in Chapter 8

Chapter 2

Related Work

This chapter will focus on related work concerning hardware accelerator sharing. Section 3.2.5 will focus on different architectures to integrate hardware accelerators. In Section 2.2 different real-time analysis models will be discussed. Capturing the effects of (run-time) arbitration in a Synchronous Dataflow (SDF) model will be discussed in Section 2.3. We end the chapter with a section focussing on state of the art architectures.

2.1 Hardware accelerator architectures

As discussed in Section 1.3 hardware accelerators are being integrated into the Starburst platform in a specific way. This chapter will focus on different architectures to integrate hardware accelerators, in order to find its advantages and disadvantages.

Instruction Set Extension

One traditional way of integrating hardware accelerators is by means of Instruction Set Extension (ISE). With this technique the Instruction Set Architecture (ISA) of a CPU is expanded, by adding pieces of hardware to the pipeline of the original CPU. This principle is depicted in Figure 2.1.

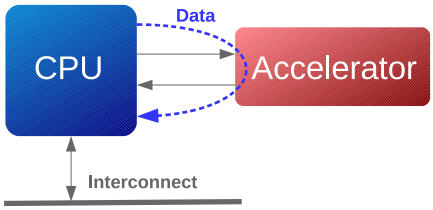

Figure 2.1: Hardware Accelerator integration using Instruction Set Extension

The Starburst MPSoC is equipped with Xilinx MicroBlaze softcore CPUs [37]. The CPUs are configurable at design time, where different components like a FPU, barrel shifter, hardware multiplier or hardware divider may be added to the system. These customizations are good examples of ISE.

Also the well-known x86 ISA, which can be found in most desktop computers, is extended multiple times in the last two decades, examples of these extensions are: MMX introduced with the Intel Pentium CPU, Streaming SIMD Extensions (SSE) (Intel Pentium III), SSE2 and SSE3 (Intel Pentium 4) and SSE4 (Intel Core). Also the switch from 32 bit to 64 bit CPUs in 2003 was also achieved with an ISE (called AMD64 or x86-64).

When executing an operation on a connected hardware accelerator, the issuing CPU has to wait until the result of this operation has been returned. This introduces latency which directly influences the performance of the issuing CPU. An advantage of adding an hardware accelerator via an ISE is the fact that hardware is placed relatively close to the processor, with a relative low latency as a result. A clear disadvantage is the fact that the hardware accelerator is connected directly to a CPU, which makes sharing of the accelerator among multiple CPUs impossible.

Remote Procedure Call

Another way to integrate hardware accelerators is by means of a Remote Procedure Call (RPC). With a RPC the programmer has the ability to execute an instruction or a set of instructions at another place in the system. This technique is depicted in Figure 2.2, where the CPU and hardware accelerator are both connected to the same interconnect.

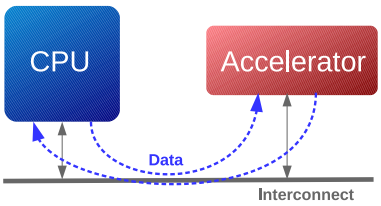

Figure 2.2: Hardware Accelerator integration using a Remote Procedure Call

An example of integrating hardware accelerators by means of a RPC is the ST33F1M CPU developed by STMicroelectronics [24], targeting the domain of security applications (e.g. Pay TV, banking and transit). The micro-processor is based on a Cortex M3 CPU, where several external hardware modules have been added to a local bus. Examples of these modules are a Cyclic Redundancy Check (CRC) module, a Random Number Generator (RNG) and a coprocessor called Nescrypt which is used to execute specific security operations.

Integrating hardware accelerator using a RPC means that this component is connected to some sort of interconnect. An advantage is the fact that this opens the door of sharing this component across multiple CPUs, as these CPUs are all able to access this component. A disadvantage of this approach is the increased latency, and therefore a lowered utilization of the issuing CPU.

Stream Processing Hardware Accelerators

While there are clear differences between integrating hardware accelerators by means of an ISE or RPC, there are some similarities. One of them is the fact that in both cases the issuing CPU is also the CPU to which data has to be returned. Returning data to the same CPU is often not an optimized method for streaming applications. The best way to deal with the data resulting from an operation is send it away to another CPU. This way the issuing CPU can already start working on the next block of data, without having to care about the returning data. Adding hardware accelerators in a stream is a method optimized for streaming applications. This method is depicted in Figure 2.3. Not having to return data back to the issuing processor has an additional advantage; it opens the door of chaining multiple hardware accelerators together, which is also displayed in this figure. In [16] support for hardware accelerators was added to the Starburst platform, where the accelerators have been added in a stream.

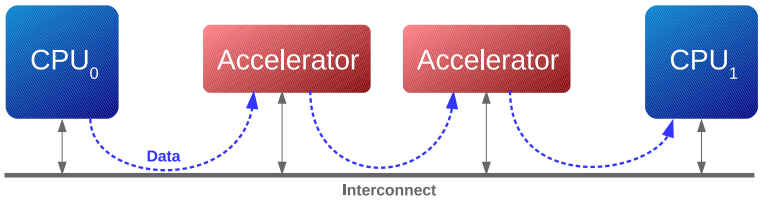

Figure 2.3: Stream processing hardware accelerators

Not having to wait on the data to be returned, means that the issuing CPU can start working on the next packet of data. This follows the concept of SDF models. In these models each task can be mapped on either a CPU or hardware accelerator, constructing a stream of tasks. Furthermore, if the connection between the CPU and hardware accelerator has enough buffering capacity, this interconnect does not have any influence on the utilization of the issuing CPU. When the hardware accelerators and CPUs are connected to the same shared interconnect, this again opens the door to share the hardware accelerators across multiple CPUs.

Another property of a stream processing hardware accelerator is the fact that for a certain stream of data it has to be configured once, after which it will operate completely stand-alone. Looking at the current situation of the Starburst platform only one stream of data can be mapped over a hardware accelerator. This means that the programmer has to configure a hardware accelerator only once, the state of this accelerator is of no concern at all to the programmer. When multiple data streams are mapped over one hardware accelerator, configuration and state are suddenly becoming of utmost importance. The hardware accelerator has to use to correct configuration and state for a specific data stream.

2.2 Real-time analysis models

Real-time analysis techniques are used to guarantee the correct real-time behaviour of a system. In this section three main analysis techniques will be discussed.

Event Models

Event Models (EM) [15] is the underlying analysis technique in the SymTA/S approach. This technique uses propagation of traffic, by characterizing the traffic with a period P and a jitter J. Jitter is the maximum deviation between the arrival time of the k-th event, relative to the k-th period. The method is not suitable for cyclic graphs, where the addition of a feedback loop will result in an infinitely large jitter [2]. Furthermore, this technique can result in a low accuracy because it does not capture the correlation between different streams accurately [14]. This is because the correlation is captured in the time interval domain, which may result in a pessimistic characterization.

Real-time Calculus

Real-time Calculus (RTC) [11] is an analysis technique originating from the Network Calculus domain. It is also based on the characterization of traffic between components, however the correlation between streams is captured in the time domain, instead of the time interval domain. It has been shown that the Event Models technique is a special case of what can be represented with RTC.

A few years ago, this technique has been showed to be applicable for applications modelled as cyclic Homogeneous Synchronous Dataflow (HSDF) graphs [25]. However, it cannot handle a combination of cyclic resource dependencies and data dependencies [14].

Synchronous Dataflow

SDF [20] [3] is a popular and widely studied dataflow modelling language for streaming applications. It computes end-to-end delay not based on the sum of delays, but on the schedule computed given worst-case firing durations. SDF uses graphs consisting of nodes and edges denoting respectively actors and dependencies between these actors. Actors produce and consume tokens, which are containers consisting of a fixed amount of data. The tokens are communicated over the edges in the SDF graph. SDF can handle a combination of resource and data dependencies, and is able to capture the correlation between data streams. Furthermore, analytical analysis is possible in the form of MaxPlus-algebra and

Maximum Cycle Mean (MCM) analysis. These techniques can be used to analyse temporal properties (e.g. deadlock freedom) or resource requirements (e.g. buffer sizes) for a practical implementation of the model.

2.3 Arbitration

In an MPSoC streams of data are processed by tasks, running on the processing elements of the MPSoC. In order to reduce costs, resources are being shared among these tasks, e.g. running multiple tasks on a single processor, letting multiple tasks share a single memory port or sharing the links of an NoC. Arbitration is the key ingredient to share these resources while guaranteeing temporal constraints such as throughput and latency constraints. The same holds when multiplexing multiple data streams over a single hardware accelerator. In order to guarantee temporal constraints, some form of arbitration is required. Furthermore, we want to be able to capture the effects of sharing such a resource in an SDF model. This chapter will focus on arbitration methods which can be captured in an SDF model.

Offline/online scheduling

One specific form of resource sharing is executing multiple tasks on a single CPU, where the CPU itself is a shared resource. In order to be able to meet temporal constraints, a designer has usually two options in a real-time system: calculating an offline schedule or using a run-time scheduler belonging to a specific class of schedulers. Scheduling at run-time instead of determining a fixed schedule at compile time is attractive for e.g. the following reasons [29]:

1. A high resource utilization can be obtained even in cases where there are tasks with a significant variation in their execution time and/or execution rate.
2. There is no need to compute and store a schedule for each combination of jobs that is simultaneously active.
3. Executing an arbitrary combination of tasks is simplified.

(Non-)Starvation free scheduling

In [3, 4, 28] the effects of sharing resources in the case that Time-Division Multiplexing (TDM) or Round-Robin (RR) arbitration is applied is captured in the response time of the actors in an SDF model. Wiggers [29] extends this work, by showing that the effects of run-time scheduling can be captured in an SDF model if the scheduler belongs to the class of Latency-Rate ($\mathcal{LR}$) servers, which is a broader class than only TDM and RR schedulers. The concept $\mathcal{LR}$ server

comes from the Network Calculus domain, where the effects of scheduling traffic passing through routers in a packet-switched network is captured in a simple and elegant model. The theory of this approach is based on a *busy period* of a session. A server belongs to the class of $\mathcal{LR}$ servers if the *average* rate of service offered during a busy session, over every interval starting at time Θ from the beginning of the busy period, is at least equal to its reserved rate ρ. The parameter Θ is called the *latency* of the scheduler.

The next example is taken from [2] and will show how the effects of a TDM scheduler can be captured in a dataflow model. Figure 2.4 gives a schematic overview of a TDM scheduler. The scheduler can be modelled by a large rotating wheel where a task $a(i)$ has a time-slice B in a total period P. When the time-slice B is depleted, the task is pre-empted. Only when a new period P is started, the task will get a new time-slice.

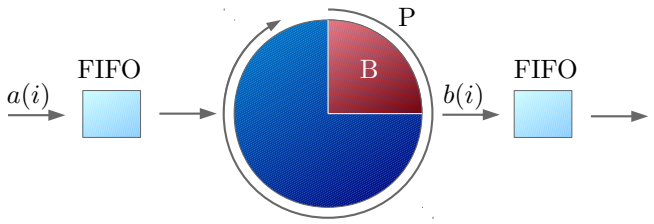

Figure 2.4: Task scheduled under TDM with a fixed budget B (in red) during a period P (in blue)

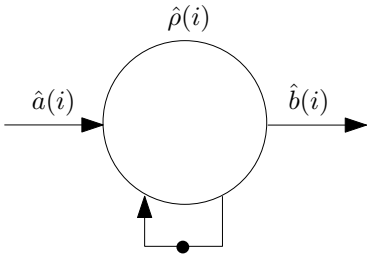

Figure 2.5: Single actor dataflow model

Figure 2.5 shows a single actor SDF-model. The idea is that the effects of scheduling a task under TDM is captured in the execution time $\hat{\rho}(i)$. The first step is determining the total execution time $\rho(j)$, which is the time between the arrival time of a task on edge $a(i)$ and the time at which the processor has finished processing this task. Figure 2.6 shows this principle. The total execution time $\rho(j)$ is determined by the original execution time $x(j)$, plus the total time at which task $a(i)$ is not being serviced, which is equal to $P - B$, multiplied with the maximum number of pre-emptions task $a(i)$ can encounter, which is equal to $\lceil \frac{x(j)}{B} \rceil$. Because the arrival time of task $a(i)$ is unknown at compile time, the worst-case situation has to be taken into account; which

happens when task $a(i)$ arrives just as budget B has been depleted. Therefore the ceiling function is used in this equation.

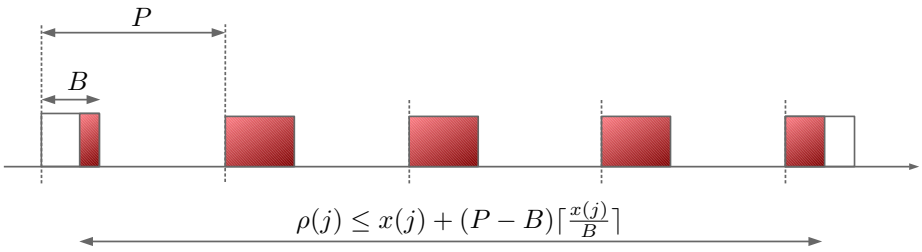

Figure 2.6: Execution of a task during its time slices

The next step is proving that the dataflow model in Figure 2.5 is a valid *abstraction* of the TDM scheduler in Figure 2.4. An abstraction is valid if the arrival times of tokens on the edges of the SDF-model in Figure 2.5 are not earlier than the arrival times on the edges of the model in Figure 2.4, which corresponds to $\forall i \geq 0, (\forall j, a(j) \leq \hat{a}(j)) \Rightarrow b(i) \leq \hat{b}(i)$. While the prove is beyond the scope of this example, mathematical induction can be used for this prove.

In [14] it is shown that also the effects of non-starvation free schedulers such as *Static Priority Pre-emptive* scheduling can be modelled in a dataflow graph. For a certain task τ_i the maximum amount of time it takes to execute q consecutive executions is calculated (denoted by $w_i(q)$), by including the computation time of the set of tasks $hp(i)$ with a higher priority running on the same processor. It requires iterative fixed-point computation to calculate the response time, and therefore no closed-form expression for the throughput like MCM exists. This method requires, next to the execution times of all the tasks, knowledge about the arrival rates of these tasks.

2.4 State of the Art

This chapter will focus on related work considering the sharing of hardware accelerators in an MPSoC.

In [8] the difference between equipping all CPUs in the system with a dedicated (tightly connected) hardware accelerator and sharing a hardware accelerator on a globally shared On-chip Peripheral Bus (OPB) is investigated, by mapping an image compression algorithm on the system. The architecture uses a centralized Synchronization Engine which offers locks and barriers to the CPUs. When a CPU locks the shared accelerator, all subsequent requests will be queued and handled at a later stage. Interrupts are used to inform the requesting accelerator the computation is finished.

Bouthaina et al. [9] propose another architecture to share hardware accelerators in a heterogeneous MPSoC. The architecture consists of a mix of private

and shared hardware accelerators connected to respectively private and shared busses. Furthermore, shared memory between the processors make communication between the processors possible. Mutexes and locks are used to solve concurrency problems, when multiple processors want to use the same shared hardware accelerator.

In [12] Cong et al. research hardware accelerator sharing on a different architecture. The platform in this research consists of a mesh network to connect all components in the system, where the processors share components like Level-2 (L2)-caches, global memory and hardware accelerators. Concurrency problems introduced when sharing hardware accelerators are solved using a hardware based arbiter, called Global Accelerator Manager (GAM). When a processor wants to use a hardware accelerator, the processor has to lock the accelerator via the GAM. The GAM can either give the processor permission to use the accelerator (1), give an instruction to wait (2) or reject access to the accelerator (3). Based on this reply, the requesting processor may decide to continue the computation in software.

While the authors of [8], [9] and [12] are able to improve the performance, reduce energy consumption or hardware costs by implementing and sharing hardware accelerators, they do not consider real-time applications. Mutexes and hardware locks are used to acquire an accelerator, which will make real-time analysis probably complex or even impossible.

Tong et al. [26] try to schedule multiple radio standards (with different throughput constraints on a *multi-standard multi-channel channel decoder*. The decoder basically consists of a CPU and several weakly programmable hardware accelerators with limited buffer sizes, all connected to a local NoC. The CPU in this cluster is in control of all hardware accelerators. It updates the configuration of the modules and is able to chain multiple hardware accelerators together.

On the CPU a RR scheduler per hardware accelerator is implemented. In order to run applications where multiple hardware accelerators are forced to execute simultaneously (due to the limited buffer sizes), they propose a so-called *coupled scheduling policy*. Next, they propose an algorithm to calculate synchronization times and Worst-Case Waiting Times (WCWTs) under this coupled scheduling policy. This way they are able to guarantee throughput constraints, when executing applications on the channel decoder.

A difference compared to the Starburst platform is the fact that one of the hardware accelerators in this system is able to read data from shared external memory directly. This component is always the first accelerator in a chain of accelerators, and is basically a combination of a hardware accelerator and a DMA engine. On Starburst the hardware accelerators are not equipped with a DMA engine, instead they are as small and simple as possible and relying on a producer to feed them with data over the Nebula ring interconnect, they are unable to initiate a read action themselves.

Furthermore, a local NoC with a mesh topology is used to chain multiple hard-

ware accelerators together. The routing of this NoC is determined by the application mapped onto the platform. The Starburst platform is equipped with a ring interconnect, as discussed in Chapter 3. It has, in comparison to mesh-networks in general, relative low hardware costs, which scales linear to the number of connected cores. Using the Nebula ring interconnect instead of a mesh network is an attractive option to reduce hardware costs.

Tong et al. do not consider all overhead sources; scheduling on the CPU introduces an overhead, the same holds for the time it takes to reconfigure the hardware accelerators or to get the state out of the accelerators. Additionally, no mechanism or protocol which makes sure that all data samples have been processed by the hardware accelerators *before* they are being reconfigured is being discussed. The same holds for a mechanism which ensures that enough free space is available in order to store all computed results. Furthermore, they only give a (detailed) Cyclo-Static Dataflow (CSDF) model for their specific case-study, without providing a dataflow model with a higher level of abstraction. Via such models specific details can be hidden.

2.5 Summary

Traditionally hardware accelerators are being integrated via an ISE or a RPC. The biggest disadvantage of these methods is the fact that the results of the computation are being returned to the issuing processor. During this computation the issuing processor has to wait for the computation to finish which results in a degradation of performance. On the Starburst platform hardware accelerators are integrated in a way that is optimized for streaming applications. The processors and hardware accelerators are connected to a ring interconnect, and instead of letting the hardware accelerators return the results of their computations to the issuing processor, the results are forwarded to the next step in the chain. This way the issuing processor can already start working on its next computation.

When executing real-time applications, the ability to analyze a system is an important requirement in order to give real-time guarantees. The Starburst MPSoC targets streaming applications with firm real-time requirements. The following real-time analysis techniques have been discussed; Event Models, Real-time Calculus and Synchronous Dataflow. Of these techniques, SDF analysis turned out to be the only applicable option. In contrast to Event Models and Real-time calculus, SDF can handle cyclic graphs, can handle a combination of data and resource dependencies and can be used to give a closed-form expression for throughput. Furthermore, other parts of the Starburst platform, like the Nebula ring interconnect, are already modelled with SDF models. Sticking to one analysis technique instead of having to use multiple techniques makes real-time analysis more convenient.

Arbitration is the key ingredient when guaranteeing temporal constraints while sharing resources. This section focussed on capturing arbitration effects in a dataflow model. When a scheduler belongs to the class of $\mathcal{LR}$-servers (like Round-Robin or TDM), it can be modelled with an SDF-graph. In contrast to Static Priority Pre-emptive scheduling, no fixed point computation is required to calculate the total waiting time.

Sharing hardware accelerators while providing real-time guarantees is a rather new concept. To the best of our knowledge only one comparable study has been performed in this area. The architecture in this research uses, compared to the Starburst architecture used in our research, an interconnect with a mesh topology which has a much larger area footprint compared to the ring topology used on the Starburst platform. It is unknown whether our ring interconnect is suitable to implement hardware accelerator sharing, and what additional hardware is required to implement this technique.

Chapter 3

Starburst Platform

Chapter 1 gave an introduction to the research platform, this chapter will discuss the platform in more detail.

The Starburst hardware/software MPSoC platform is currently being developed within the University of Twente. The main goal for this project is to research and exploit: [1]

1. MPSoC architectures with means for low power, composability, and reconfigurability.

2. A design flow for MPSoC based systems using high level synthesis.

3. An MPSoC run-time system management. By means of dynamic reconfiguration, the run-time system is capable of dealing with adaptive service requirements and platform variability.

The platform can be divided in three parts. The first part is the physical hardware platform which will be discussed in Section 3.1. Section 3.2 will focus on the reconfigurable hardware, such as the processing tiles, communication network and hardware accelerators. Section 3.3 covers the software layer running on the Starburst platform.

3.1 Hardware Platform

The Starburst MPSoC is being developed on a Xilinx ML605 development board. This development board is equipped with a Virtex-6 FPGA, consisting of 241,152 Logic Cells. Each Logic Cell contains four Look-Up Tables (LUTs) and eight flip-flops. Besides the FPGA, the ML605 development board is equipped with several peripherals which are used throughout the Startburst platform. Figure 3.1 gives an overview of the development board, in which several peripherals are indicated. The most important components are the 512MB DDR3 memory, the Ethernet connection, the DVI output, the CompactFlash storage, the Universal Asynchronous Receiver/Transmitter (UART) interface, and the USB port.

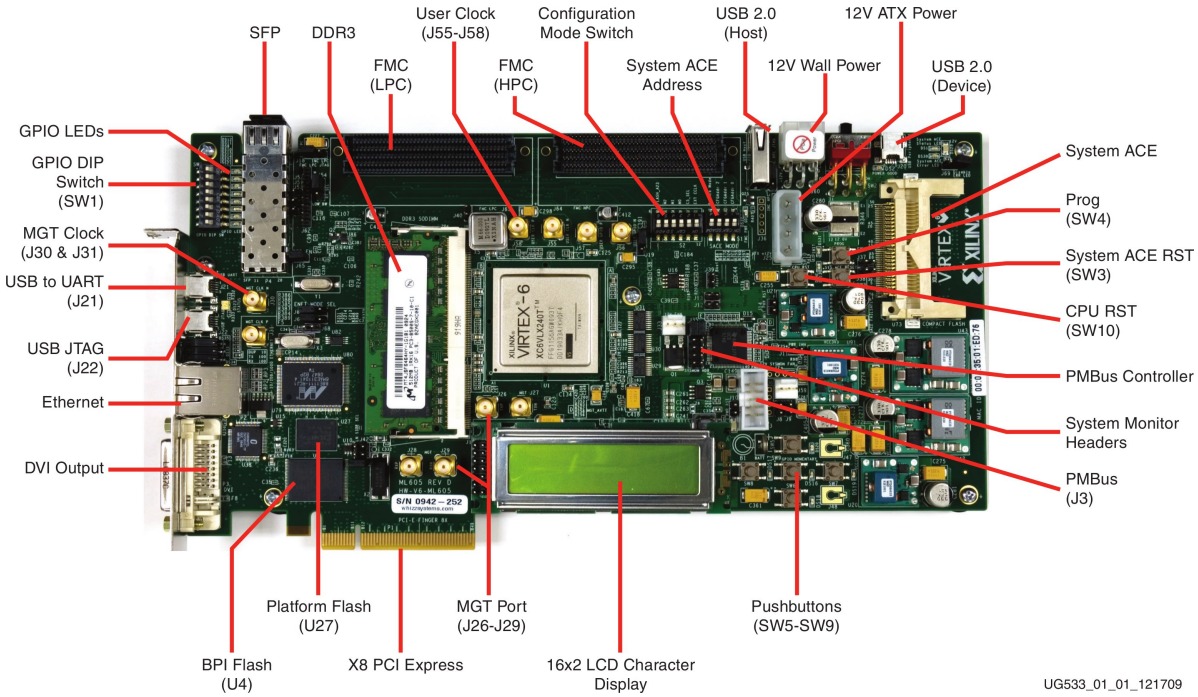

Figure 3.1: ML605 development board

3.2 Starburst

Figure 3.2 depicts a high level overview of the Starburst platform. In this figure the 32-core configuration is displayed, which is (given the available resources of the FPGA) currently the configuration with the maximum number of cores. In this figure different classes of components can be identified. The platform consists of a power-of-two processing tiles, a Linux tile dedicated to I/O capabilities (both depicted in blue), the Nebula ring network for all-to-all communication between the processing tiles and an arbitration tree to access DDR3 memory and several shared peripherals (both depicted in green). The components are red are peripherals available on the Starburst platform.

In the following sections these different components will be discussed in more detail. First, the processing tiles will be discussed in section 3.2.1. The Linux tile, which is actually a processing tile with some additional hardware, will be discussed in section 3.2.2. In Section 3.2.3 the Warpfield arbitration tree will be discussed. The Nebula ring network makes all-to-all communication between the cores possible and will be discussed in Section 3.2.4.

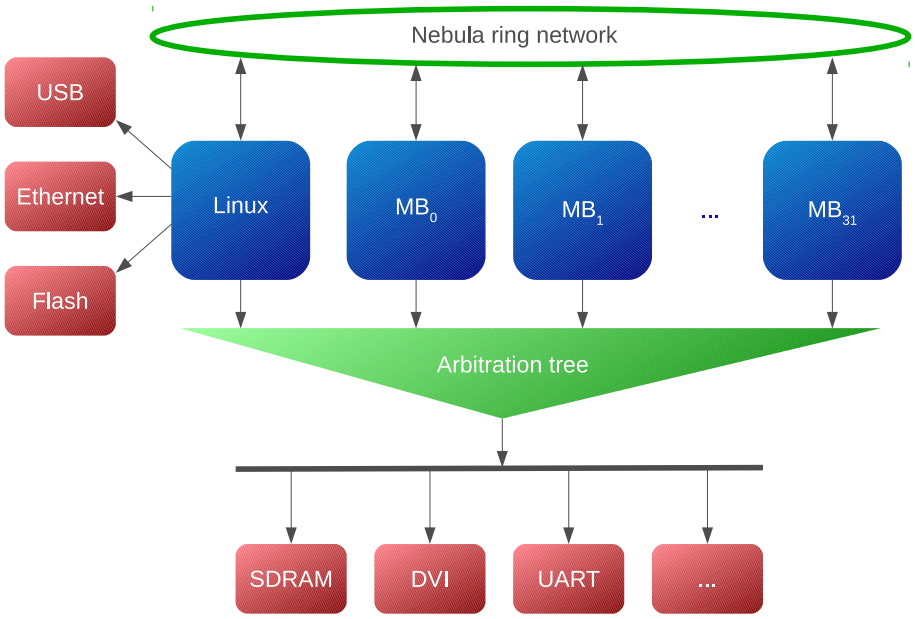

Figure 3.2: High level overview of the Starburst platform. Arrows indicate master-slave relations.

3.2.1 Processing tile

The basis of the Starburst platform is formed by the processing tiles, which are based on Xilinx MicroBlaze CPUs. Figure 3.3 gives an overview of a processing tile. The different components in this figure are discussed below.

The Xilinx MicroBlaze CPU is a Reduced Instruction Set Computer (RISC), organized as a Harvard architecture. This means that the CPU has separate bus interfaces for data and instruction access [37]. The Xilinx MicroBlaze is a so-called softcore which can be fully implemented in reconfigurable hardware by the logic synthesis tools. Due to this reconfigurability it is possible to configure the CPU at design time, where different components like a FPU, barrel shifter, hardware multiplier or hardware divider may be added to the CPU.

The MicroBlaze CPU is equipped with two caches; one instruction cache and one data cache. The caches are implemented without cache coherency between the processing tiles, there is only a write-back policy active. The size of these caches is configurable at design time, by default (and throughout this report) it has a size of 32 kB. When small streaming applications are executed, all instructions should fit into the instruction cache, which greatly reduces the number of requests to the shared memory.

Each processing tile is equipped with local data memory (4 kB by default). This

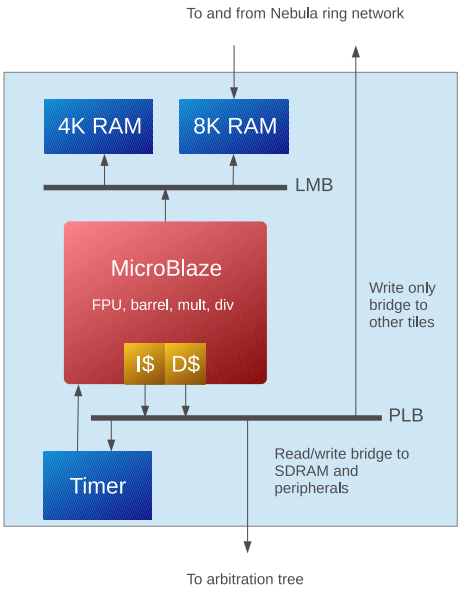

Figure 3.3: Processing tile

memory is connected using a single cycle Local Memory Bus (LMB), and used for local data storage. This memory is also used for local kernel administration, e.g. cache sizes and the number of processing tiles in the system. Scratchpad Memory (SPM) is connected via the same LMB. This SPM is used for all-to-all communication between the processing tiles instead of using the shared memory for this purpose. This communication protocol will be discussed in Section 3.3.2.

TDM is used to implement multi-threading on the processing tile. For this purpose, and in order to keep track of time, a dedicated hardware timer is added to each processing tile.

3.2.2 Linux tile

The Starburst platform contains exactly one Linux tile, which is almost an exact copy of a regular processing tile. The only hardware extension to the Xilinx MicroBlaze is the addition of a Memory Management Unit (MMU), which is necessary to be able to run Linux. The Linux tile is mainly used for I/O purposes, e.g. to communicate with the host PC, or control USB connected peripherals.

Compared to a processing tile, the Linux tile is equipped with additional peripherals connected via its local Processor Local Bus (PLB). Examples of these components are the Ethernet controller, the controller for the Compact Flash

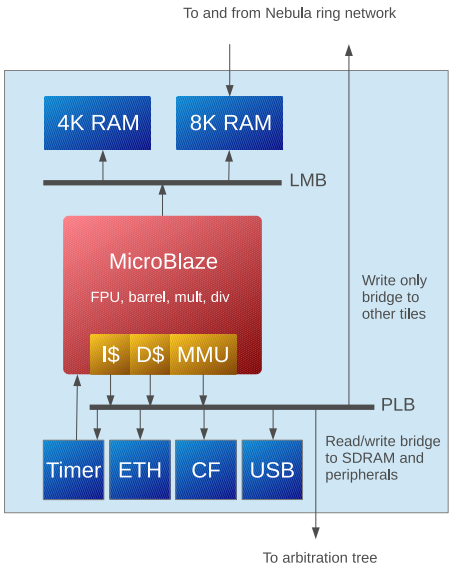

Figure 3.4: Linux tile

(CF) card, an the controller for the USB port.

3.2.3 Warpfield

The processing tiles and the Linux tile are connected to an interconnect which facilitates communication on the platform. The interconnect consists of two separate parts. The first part is the Warpfield arbitration tree. The arbitration tree is mainly used to access the DDR3 memory, but other components like the DVI controller and UART interface are also accessible through this interconnect.

The arbitration tree is developed to solve two traditional problems: super-linear scaling of hardware resources to the number of cores and a high latency for memory reads [23]. In order to overcome these problems an arbitration tree has been developed with the following features:

1. Starvation-free scheduling.
2. Work-conserving to optimize for latency.
3. Scales linearly in hardware costs to the number of cores.
4. Pipelined and decentralized arbitration to avoid long wires for high performance.

The arbitration tree provides a PLB slave interface to each Microblaze in the system. When a Microblaze issues a read- or write-request this request is pack-

etized and is given a timestamp and source ID. The packets enter a binary tree where at each step local arbitration based on the timestamp is applied. This way arbitration is applied according to a First-come, First-served (FCFS) policy, where a packet with the lowest timestamp is allowed to proceed first.

The last step is a demultiplexer, which routes the data to the correct peripheral. An additional demultiplexer is used to send data coming from the peripherals back to the corresponding Microblaze based on the source ID, without the need for any arbitration.

3.2.4 Nebula ring

The second part of the Starburst interconnect is the Nebula ring interconnect. This part of the interconnect provides an all-to-all communication network as a latency-tolerant interprocess channel. This in contrast to the Warpfield arbitration tree, which is a latency-critical channel. In the latter type of channel, the performance degrades immediately with a higher latency. In the case of latency-tolerant channels this degradation depends on the application [23]. The Nebula ring interconnect has a small area footprint compared to other ring topologies. This small area footprint was achieved by: [13]

1. Making the Nebula ring unidirectional, which makes routing decisions trivial.
2. Under the assumption that slaves always accept data, FIFO buffers at the output of the network are omitted.
3. There is no need for FIFO buffers in the routers, because no contention or head-of-line blocking can occur.
4. There is no support for back-pressure. When a packet has entered the ring, it will travel one hop every cycle until it reaches its destination.
5. The ring provides automatic serialization, therefore no arbitration is required when multiple masters want to write to the same slave.

Figure 3.5 gives a schematic overview of the Nebula ring NoC. In this schematic we can identify typical NoC components like routers and NIs. Every processing tile is connected via a NI to a router, and by connecting every router to its two neighbouring routers, a ring-like structure is created. The NIs form the bridges between the (network unaware) processing tiles and the networking part of the NoC.

The basic idea of the ring interconnect is that when a data packet has entered the ring (i.e. has acquired a free slot), it can always proceed to a next router, until it has reached it's destination. Therefore, a data packet has to be buffered in the NI until it is accepted on the ring. When a packet has reached its destination, the corresponding router will remove the packet from the ring and delivers it to

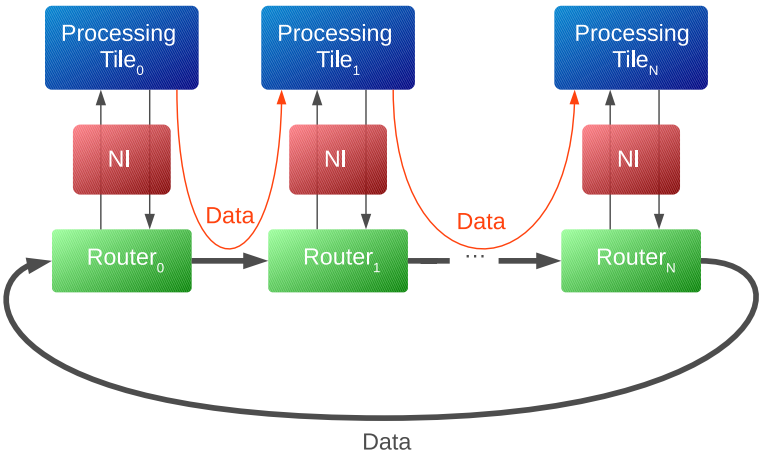

Figure 3.5: Schematic overview of the Nebula ring

the connected NI. This action frees the occupied slot, so it can later be used to transfer a new data packet.

This construction, however, introduces starvation. If a processing tile would occupy all free slots, other processing tiles might never get access to send their data packets. To guarantee a processing tile access to the ring (and therefore eliminating starvation) a fairness protocol is implemented. All slots on the ring are labelled and a processing tile is only allowed to use a slot, when the slot-label equals the ID of the processing tile. This way all processing tiles are guaranteed to have an equal share of the total bandwidth. This traffic is classified as Guaranteed Service (GS) traffic.

The drawback of this fairness protocol is the fact that a lot of empty slots cannot be used by other processing tiles, reducing the available bandwidth of the ring interconnect. A work-conserving principle is introduced, by allowing slots to be hijacked under very strict conditions. The idea is that when an empty slot passes a router, this router may hijack this slot if the destination router is reached before the owner of the slot is reached. An overview of the rule set as implemented in the Nebula ring router, is given in Table 3.1. This traffic is classified as Best Effort (BE) traffic.

3.2.5 Hardware Accelerator integration

As discussed in Section 1.3 support for hardware accelerators was added to the Starburst MPSoC, in order to improve both computational performance and power efficiency [16]. The hardware accelerators are just like the processing tiles directly connected to the Nebula ring interconnect, as depicted in Figure 3.6.

Table 3.1: Rule set Nebula ring router

1.	Incoming packets addressed to the local node are ejected to the local node.
2.	All other incoming packets are passed to the neighbour router if the current slot does not belong to the local node and the local node has a packet to send.
3.	Insert a packet on the ring if the local node has a packet to send and the slot is *available*.
	A slot is available if at least one of the following rules is true:
1.	The slot ID is equal to the local node ID.
2.	The target address of the local packet is reached before the current slot passes its owner and the slot is free.
3.	The target address is equal to owner of the slot and the slot is free.

In order to achieve lossless communication on the Nebula ring, some form of flow-control is required. For the processing tiles flow-control is implemented in the software layer (more about this in Section 3.3.2). Hardware accelerators however are unaware of memory addresses, have no SPM an have no software running to manage communication. For this reason an alternative form of flow-control was required.

Credit-based hardware flow-control is implemented for this purpose. At the consuming side a hardware based FIFO buffer is implemented, which stores incoming data samples, before they are accepted by the hardware accelerator. The number of credits denote the depth of the FIFO, between a producer and a consumer. A producer consumes a credit when data is produced, a consumer will send a credit back when data is accepted. Credits are sent over a dedicated credit-ring (see Figure 1.3), rotating in the opposite direction compared to the data-ring. This means that a credit has to travel the same distance as the data has to travel, which can reduce the total hop delay of data plus credit packet when cores and hardware accelerators are placed close to each other [16].

The idea of these hardware accelerators is that they are operating on a (continuous) stream of data. The accelerators receive their data from one of the processing tiles over the Nebula ring network, and after processing the data is sent to a second processing tile, which will perform further processing on the stream of data. A problem was the fact that hardware accelerators are unaware of memory addresses in the system and are therefore unable to directly communicate with the processing tiles in the system. In order to give the accelerators the required knowledge to make communication possible, the NIs are extended with an additional module called a *ringshell*. A ringshell is equipped with a set of programmable registers; one of these registers stores the address to which credits have to be returned, another register stores the address to which data has to be forwarded to. Furthermore, a ringshell is equipped with a programmable counter which stores the number of credits. An advantage of this set-up is the

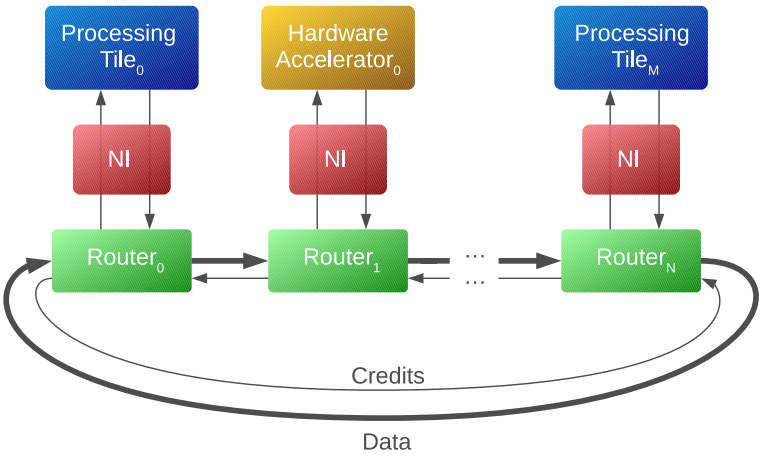

Figure 3.6: Schematic overview of two processing tiles and one hardware accelerator, connected to the Nebula ring interconnect

possibility to easily chain multiple accelerators by letting one accelerator forward its data to a second accelerator and letting this second accelerator sending its credits back to the first one.

The last step is to return from the flow-controlled stream of data to a memory mapped processing tile. Simply writing to the SPM of a processing tile is not supported, due to the absence of any form of flow-control. Furthermore, a hardware accelerator is only able to send data to a fixed address, where the SPMs have to be addressed consecutively. The ideal situation is to be able to emulate the software based FIFO communication (See Chapter 3.3.2), which is used to communicate data between processing tiles. The author from [16] determined this is not a trivial hardware design, and therefore came up with a temporary solution, where a hardware based FIFO buffer is directly connected to a MicroBlaze using a Fast Simplex Link (FSL). The FSL is an extension to the MicroBlaze execution pipeline, with an unidirectional point to point communication interface. With this construction a hardware accelerator is writing data in this hardware FIFO buffer, the consuming MicroBlaze will poll this buffer over the FSL interface to check whether new data is available. Figure 3.7 gives a complete overview of the integration of hardware accelerators including the ringshells and a FSL-link.

3.3 Software

The previous sections discussed the two lower (hardware) layers of the Startburst platform. This section will focus on the software layer running on top of this

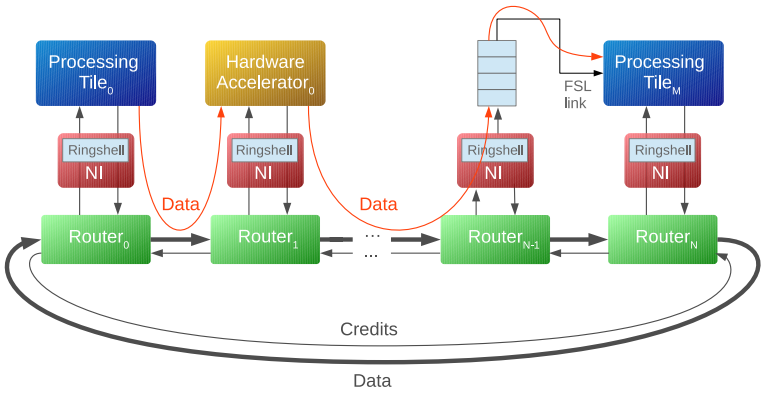

Figure 3.7: Schematic overview of two processing tiles and one hardware accelerator, connected to the Nebula ring interconnect.

hardware.

3.3.1 Helix kernel

Every processing tile in the system runs a small Portable Operating System Interface (POSIX) compatible operating system. This kernel has support for the newlib C library and implements multi-threading based on the Pthread standard. A TDM scheduler is implemented to switch between the threads on a processing tile. For this reason every processing tile is equipped with a dedicated hardware timer, which is the only interrupt source on a processing tile.

The Helix kernel runs several deamons for services like memory management, communication, synchronization, profiling, statistics and scheduling.

3.3.2 CFIFO

One service which deserves special attention is the service which offers communication on the Starburst platform. One specific (and most frequently used) communication protocol is the software based C-HEAP FIFO (CFIFO) [22] implementation. With this implementation core-to-core communication can be set up for arbitrary data-types, with a configurable FIFO-depth and block-size.

The protocol uses a double read- and write-pointer administration (rp, wp) stored in the SPM at both the producing and consuming side, as depicted in Figure 3.8. In this administration, the pointers *rp'* and *wp'* are copies of the pointers *rp* and *wp*. The copies of these pointers are updated after a read- or write action has been completed. Thus, these copies are always a delayed

version of the original pointer, and give therefore a conservative estimation of the amount of free or occupied positions in the software FIFO.

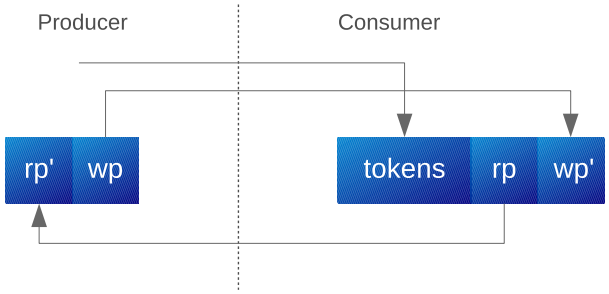

Figure 3.8: CFIFO administration overview

An advantage of this double administration is the fact that the producer and consumer only have to poll their local SPM to check if there is respectively free space or new data available. This does not only reduce the latency, it also makes sure that no shared memory bandwidth is wasted polling for free space or new data. The tokens itself, and the updates of the read- and write-pointers are also sent over the Nebula ring-network, which is a further reduction of the consumed shared memory bandwidth.

Chapter 4

System Architecture

This chapter will focus on a solution for our problem, by discussing several architectural alternatives.

4.1 Architectural Alternatives

While multiple system architectures were considered during this research, two basic principles can be identified. As an example Figure 4.1 illustrates the internals of an accelerator, where a set of arbitrary hardware operations are connected in a pipeline. Data enters the accelerator on the left side, and leaves the systems after it is being processed at the right side. Registers (reg_0, reg_1) are placed in between the operations, and store the intermediate results. Some operations may require a certain configuration, which is stored in an additional register ($conf_1$).

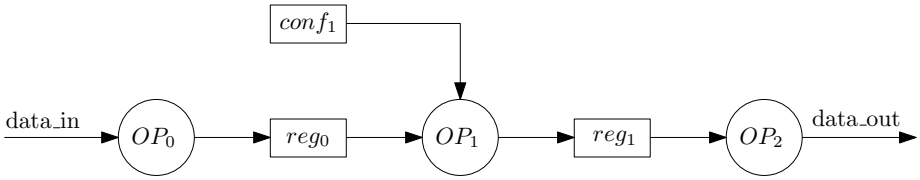

Figure 4.1: Hardware Accelerator pipeline

A problem arises when data samples from multiple data streams enter this system. The state and configuration registers can only store the contents for one data stream, mixing of data streams is simply not allowed. There are two basic solutions for this problem.

The first solution is based on the duplication of registers, which is displayed in Figure 4.2. In this image, the original pipeline is modified, were each register is replaced by a set of registers. Multiplexers and demultiplexer are used to select the correct register, based on the incoming data stream. In this example the registers are duplicated to support three data streams.

The second solution is based on TDM, where a data stream is given exclusive access on a hardware accelerator for a certain amount of time. Instead of duplicating each configuration and state register, the architecture is modified in such

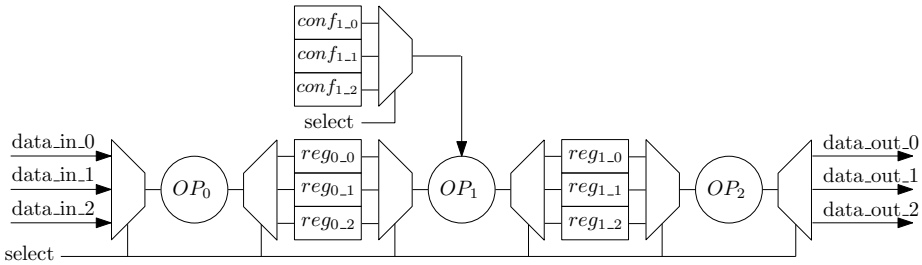

Figure 4.2: Hardware Accelerator pipeline with duplicated registers

a way that these registers are (remotely) readable and writeable. After configuring the registers for a specific data stream, the corresponding data stream can be processed. Afterwards, the state of the accelerator is saved, and the state and configuration for another stream is loaded.

An advantage of the first solution is its simplicity; due to the nature of the Nebula ring network data is automatically serialized. For this reason each clock-cycle only one data-sample will enter the hardware accelerator, which means that no contention of data will occur. Hardware costs are the biggest concern of this solution. With hardware accelerators becoming increasingly complex, an increasing number of registers have to be duplicated to support multiple data streams. This means that more registers and (de)multiplexers have to be added to the system. Additionally, duplication of registers is not always straightforward (e.g. when a design contains Intellectual Property (IP)) or even impossible when specific on-chip components of an FPGA are used (e.g. dedicated multipliers [35], which contain internal registers which are not remotely accessible). For this reason the decision was made to focus on the second solution, based on TDM.

4.2 Time Division Multiplexing

A sharing mechanism based on TDM can be implemented in various ways. Three options were identified; local, distributed and centralized. These options will now briefly be discussed.

The first solution is to apply TDM scheduling *locally* at each hardware accelerator. This requires modifications to the hardware accelerators; FIFO-buffers have to be added to buffer all incoming data streams, logic is required to let the accelerators reconfigure themselves and a scheduler has to be implemented which makes sure that the data streams are processed in the correct order. A clear disadvantage of this approach is the fact that only a fixed number of streams are supported. Additionally, a large number of registers are added to a hardware accelerator which may remain completely unused when not all stream

slots of the accelerator are occupied.

Another option is a more *distributed* solution, where the processors (who are sharing the same hardware accelerator) are required to program the accelerator, feed it with data and save the state afterwards themselves. When a processor is done, it informs the next processor that it can safely start using the accelerator. A clear advantage is the fact that this sharing mechanism could be implemented completely in software, an thus no additional hardware is required. A disadvantage of this approach is the fact that this will not improve the utilization of the accelerators, as the low-speed data streams coming from the processors are processed in a sequential order.

The last option is a *centralized* solution, which is based around components which we call *gateways*. This idea is depicted in Figure 4.3. As discussed, the processors of the Starburst MPSoC are not able to read and write data fast enough over their local PLB. As a result, the utilization of the hardware accelerators remains relatively low. A gateway component (GW_0) will temporary save the incoming low-speed data streams, and pushes the streams one at a time at high speed through the hardware accelerators. A second gateway (GW_1) is used to route the data stream to the correct receivers.

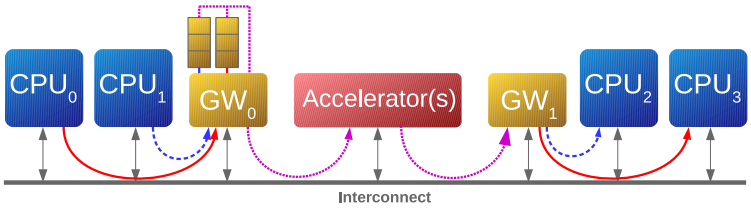

Figure 4.3: Hardware Accelerator sharing via gateways

The centralized solution turned out to be best solution for our problem; additional hardware is required but is has to be implemented only once, the utilization of the hardware accelerators is improved by streaming data streams at high speed over the hardware accelerators and depending on the application it is likely that this solution will also work when the application is mapped on hardware accelerators only.

4.3 Accelerator Gateway

As discussed in the previous section, the hardware accelerators will be shared via a centralized component called a gateway. This section will discuss several important aspects of this component.

4.3.1 Arbitration

When designing a method which implements hardware accelerator sharing, the most important requirement of this method is the fact that it should be predictable. As discussed predictability is the ability to construct a sufficiently accurate temporal analysis model of the hardware design for which a computational efficient analysis algorithm exists. With this model calculations can be performed and useful numbers (e.g. minimum throughput and maximum latency) can be extracted.

Section 2.2 focussed on different predictable scheduling techniques. As discussed in this chapter, it is possible to capture the effects of run-time scheduling in a dataflow model, if the scheduler belongs to the class of $\mathcal{LR}$-servers. Examples of such schedulers are TDM and Round-Robin schedulers.

Furthermore, the scheduler is required to be *non-preemptive*. Preemption is the operation of suspending a running task, where the execution is resumed at a later moment in time [10]. On a CPU this technique is implemented in the operating system, and can for example be used to make sure that all tasks running on a CPU get some amount of processing time. The need for a non-preemptive scheduler can best be explained with an example. Figure 4.4 gives a schematic overview of a gateway, a hardware accelerator and a consumer. The orange arrow indicates the stream of data through the routers, NIs and the hardware accelerator. Actual data is depicted by the blue tokens on these arrows.

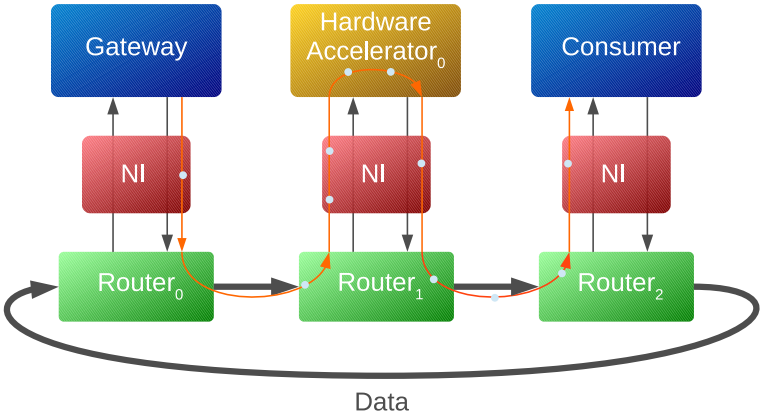

Figure 4.4: Non-preemption example: a gateway and a consumer are communicating data via a Hardware Accelerator

Assume now that preemption is possible. In order to implement preemption, the complete context of this pipeline need to be saved such that this context can be restored at a later moment in time. This means that *all* data (depicted by the blue dots in Figure 4.4) residing in the relevant components like the routers,

NIs and hardware accelerators has to be saved. And that is something which is highly undesirable, as there is currently no way to read the contents out of these components or write this data back at a later moment in time.

For this reason, the scheduler has to be non-preemptive, which means that a predefined amount of data (*a data packet*) has to be completely processed by the gateway before a switch to another stream is allowed. Furthermore, all data has to be received by the consumer, which makes sure that the complete processing pipeline is empty. For this purpose Round-Robin arbitration is implemented. Round-Robin is a simple scheduling policy, where tasks are being processed without priority in a cyclic order. Furthermore, it is a non-preemptive scheduler belonging to the class of $\mathcal{LR}$ servers, allowing us to create a dataflow model of it.

4.3.2 Buffering

As discussed in the previous section, the Round-Robin scheduler operates on packets of data, which have to be processed completely before a data packet from another stream can be processed. As the data packets have an (at design time determined) fixed size and are processed in a fixed cyclic order, this knowledge allows us to determine temporal properties like the maximum latency and minimum throughput at which the data packets are being processed. The fixed size of a data packet allows us to put an upper bound on its *processing time*, which is the time it takes to completely process a data packet.

In order to be able to put an accurate upper bound on the processing time, it is required that a complete data packet is available in the corresponding input buffer of the gateway before it is being serviced. When the gateway would already start processing an incomplete data packet, the time at which the data packet is completely processed depends on the throughput of the preceding actor which is feeding the gateway with data, which is most of the time a low speed data stream coming from one of the CPUs. Taking worst-case situations into account forces us to base the response time calculations on these low throughput data streams, cancelling all benefits of using the gateway to push data at high speed through the accelerators.

In Figure 4.5 a CSDF* graph is depicted, where a gateway (GW) is communicating data via an accelerator (Acc) to some consumer ($Cons$). For simplicity execution times have been omitted, as we are only interested in the consumption and production rates on the edges between the actors. A data packet with a size of S samples is being processed by this chain of actors. Checking the presence of a complete data packet in the gateway module (marked in red) can be modelled in this CSDF graph by consuming all S tokens in the first iteration, in the remaining iterations no tokens are consumed.

*An introduction on (C)SDF is given in Chapter 6

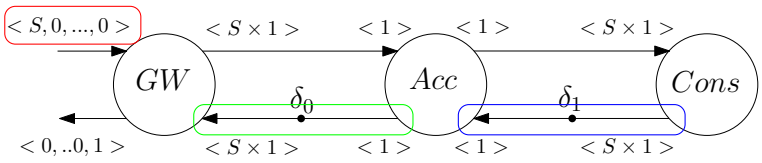

Figure 4.5: CSDF graph of an accelerator pipeline

Communication between the gateway and the hardware accelerator is taking place on a word by word basis, which results in $S \times 1$ single word transactions from the gateway to the accelerator. Besides checking the presences of a complete packet at the input of the gateway, the gateway should also make sure that enough free space at the end of the pipeline is available, which makes sure a complete packet can be stored without stalling. In Figure 4.5 this corresponds to checking the number of tokens on the edge running from the consumer to the accelerator (marked in blue), this edge should at least hold S tokens in order to store a complete packet without stalling. Letting the gateway check buffer capacity at the consumer is problematic as it is currently only able to check the presences of buffer capacity at the accelerator, denoted by the number of tokens on the edge between actors Acc and GW (marked in green). A mechanism to check for free space at the output is required, which will be discussed in Chapter 5.

In theory the gateway could start processing a packet without checking the presence of free space at the output, however in that situation the total processing time will then depend on the throughput at which the consumer in Figure 4.5 is able to process the incoming data words. When the consumer is a task mapped on one of the CPUs in the system, this results in a relative low throughput. Again, taking worst-case situations into account forces us to base the processing time calculations on this low throughput at which the consumer is able to process data, cancelling all benefits of using the gateway to push data at high speed through the accelerators.

Concluding, input and output buffers are required to store complete packets of data, both at the input and output of the pipeline formed by the accelerators between the in- and output gateway. Via these input and output buffers and the requirement that only completely received packets are being processed high throughputs can be guaranteed, which is otherwise hard or even impossible.

4.3.3 Algorithm

The scheduling algorithm implemented in the gateway can best be explained with the pseudocode listed in Algorithm 1. This pseudocode shows how two streams (*stream_a* and *stream_b*) are being serviced.

Algorithm 1 Gateway Algorithm

```
1:  procedure GATEWAY
2:  loop:
3:      if ((stream_a has data) and (output buffer has free space)) then
4:          configure the accelerators
5:          load state into the accelerators
6:          process data for stream_a
7:          wait for all data to be processed
8:          save state of the accelerators
9:      end if
10:     if ((stream_b has data) and (output buffer has free space)) then
11:         configure the accelerators
12:         load state into the accelerators
13:         process data for stream_b
14:         wait for all data to be processed
15:         save state of the accelerators
16:     end if
17:     goto loop.
18: end procedure
```

The algorithm starts by checking whether a stream has data available in its input buffer, and if enough space at the output is available (Line 3). If so, the accelerators for this streams have to be configured, by loading its configuration and state. During the next step the data samples of *stream_a* are pushed through the connected accelerators. As discussed in the previous section, the algorithm can only continue if all data has been received by the consumer. If all data has been received the current state of the accelerators is being saved before the next stream is being serviced. Next, all other streams are processed in a cyclic order.

4.3.4 Data Packets

One interesting property of the scheduling policy is at which granularity the data streams are being multiplexed over the hardware accelerators. As discussed in the previous section, a predefined amount of data (a data packet) is being processed before a next stream can be processed. Changing the size of a data packet has two consequences;

1. Context switches introduce an overhead, mainly caused by the time it takes to reconfigure the accelerators. This overhead will reduce the utilization and therefore the maximum throughput which can be achieved over an accelerator. In this sense, one should minimize the number of context switches in order to reduce the introduced overhead, e.g. by maximizing the size of the data packets.

2. Data packets have to be buffered before they can be processed. In that sense, one should minimize the size of the data packets, as this allows smaller buffer sizes. With smaller buffer sizes, less memory has to be added to the system. Furthermore the latency introduced by the sharing mechanism is minimized.

The calculation of proper buffer sizes is a trade-off between maximizing the throughput over the accelerators and minimizing memory usage. This calculation will be discussed as future work in Section 8.2.

4.4 Summary

This chapter focussed on the system architecture. The decision was made to implement a centralized gateway component. This gateway will buffer incoming (low speed) data streams, and will push them each at high speed through the accelerators. With this technique little to no modifications are required on the hardware accelerators and offers a possibility to improve the currently low utilization of the hardware accelerators. Round-Robin scheduling will be applied in this gateway component, where packets of data are processed in a cyclic order. Buffers to hold complete packets of data will be implemented both at the input of the gateway and after the last accelerator.

Chapter 5

Implementation

The previous chapter gave an overview of the system architecture. As explained a centralized gateway module will be implemented, which will take care of buffering incoming data streams and performing arbitration between the streams. This chapter will focus on the implementation of this module.

5.1 Hardware

A wide variety of tasks has to be supported by the gateway, such as:

- Buffering of incoming data streams.
- Reconfiguring of hardware accelerators.
- Performing arbitration over the data streams.
- Checking for free space at the output.
- Pushing data streams at high speed through the accelerators.

One question directly popped up during the design of this gateway; should the gateway be implemented in software, running on one of the processors in the system, should it be implemented completely in hardware, or is a combination of both worlds the best trade-off. When answering this question, one has to consider flexibility, hardware costs and power efficiency. Eventually the decision was made to start with a solution completely in software, running on one of the processors in the system. The idea behind this choice was to quickly implement a sharing mechanism in software, and then gradually replace parts of this (slow) software solution with a more efficient implementation in hardware. One of the advantages of using a CPU to implement accelerator sharing is the fact that communication between processors and the gateway module is already implemented, as the existing C-HEAP FIFO (CFIFO) protocol (see Section 3.3.2) can be used for this purpose.

In the next sections the hardware components which were added to the gateway are discussed.

5.1.1 Accelerator Interface

The Nebula ring interconnect only supports write operations. A processor is able to write data to specific registers of an accelerator, in order to fill it with configuration and state. Reading the contents of these registers, in order to extract the state of an accelerator, or to debug the contents of an accelerator during code development, turned out to be more problematic. As a solution to this problem a so called read-back mechanism was introduced in [16]. By writing a SPM address to specific addresses of an accelerator, the accelerator will take care of sending the contents of the corresponding register to the supplied SPM address. This way a read operation is replaced by two separate write actions.

This solution is in its current state unusable, because the requesting processor is unable to safely determine when the contents of the requested register has arrived in its local SPM. Another concern is the fact that the returning data has to travel across the complete remaining part of the Nebula ring. This means that the latency introduced by this part of the network has a direct influence on the time it takes to reconfigure an accelerator, and thus also a (negative) influence on the utilization of the hardware accelerators. Because the Starburst MPSoC is becoming increasingly larger in terms of actors (processing tiles and accelerators) connected to the Nebula ring, the size of the Nebula ring is expected to constantly increase which will result in a degradation of (reconfiguring) performance.

As a (temporary) solution a so-called *Accelerator Interface* has been developed. This hardware component is used to connect the accelerators directly to the local PLB of the gateway, giving the gateway the ability to easily read configuration and internal state out of the accelerators. This module is clearly not implemented as a permanent solution; it scales poorly, and has a large area footprint. However, as future modifications on the network are expected (See section 8.2), this solution was preferred as it was easy to prototype and allowed us to quickly focus on the actual multiplexing problem.

An overview of the Accelerator Interface module is depicted in Figure 5.1. A *Xilinx PLB Slave Single* component [33] was used for this purpose. This IP simplifies the connection of custom hardware to the PLB, as it translates the rather complex PLB protocol to the much simpler Intellectual Property Interconnect (IPIC) protocol. This IPIC protocol turned out to be pin-compatible with the configuration interface on the accelerators. By default, the Accelerator Interface is mapped on the address range 0x20000000 - 0x200001FF, where each accelerator has a sub-map of 0x40 bytes. (Acc_0 is mapped on 0x20000000 - 0x2000003F, acc_1 on 0x20000040 - 0x2000007F, etc). Based on this address, custom logic has been added, which will pass on the read or write actions to the correct accelerator.

Figure 5.2 gives an overview of a processing tile, which is now equipped with an Accelerator Interface. Connections are made between the Accelerator Interface

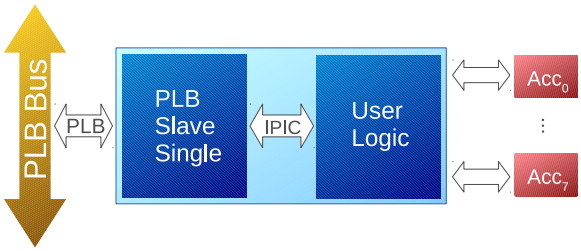

Figure 5.1: Accelerator Interface

and the ringshell of a NI. The ringshell is the modular addition to the NI which implements for example credit based communication. The ringshell was modified where a port was added to let it connect to the Accelerator Interface and logic was added to read and write state and configuration via this interface. By adding the required hardware to this modular extension, there was no need to modify each hardware accelerator separately.

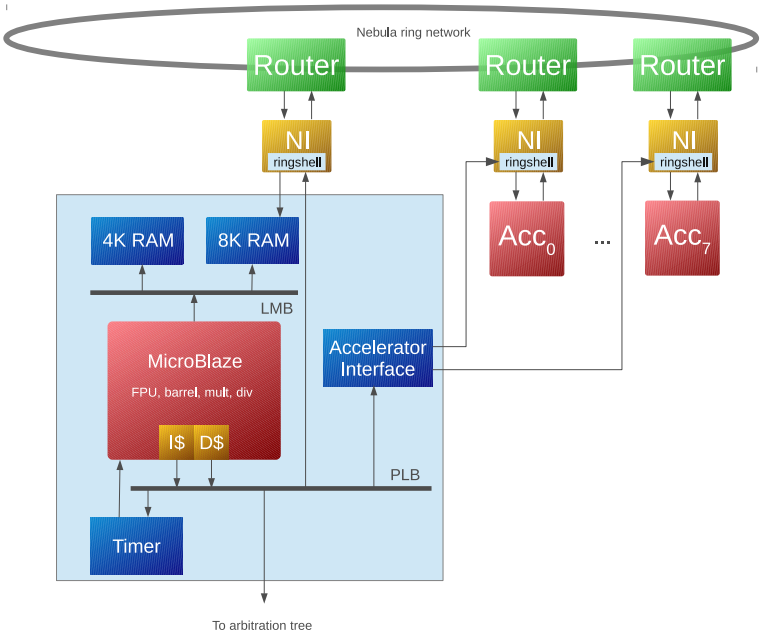

Figure 5.2: Gateway overview, including Accelerator Interface

5.1.2 Direct Memory Access

By implementing the gateway module in software one problem remains unsolved; a CPU is relatively slow and therefore unable to stream data at high speed through the accelerators.

The first step in order to come with a solution was to find the cause of this problem. For this reason performance measurements on the CPUs have been performed. As discussed in Section 3.3.2, the CFIFO communication protocol allows the processors to communicate with each other over the Nebula ring network. Via this protocol the communication of arbitrary data types with arbitrary sizes is possible.

The performance measurements focussed on the throughput at which data is communicated via this CFIFO protocol. By letting two neighbouring CPUs communicate data via a CFIFO buffer with a depth of 1, the impact of the packet size on the achieved throughput is investigated. By using two neighbouring processors we can in theory use the complete bandwidth of the Nebula ring interconnect, under the assumption that no other tasks are executing on the system. Each experiment starts by taking a snapshot of the hardware timer at the producing CPU. After sending multiple packets to the consumer, another timer snapshot is taken. Based on these two snapshots, the size of one packet and the total number of packets, the total throughput can be calculated. Sending a large number of packets (16 million) resulted in a total processing time of around 15 seconds for the smallest packet size. This way we tried to average out the jitter introduced by for example the caching mechanism, the kernel switches or the time it takes to start and stop the hardware timer. The results of these measurements are displayed in Figure 5.3.

The graph in this figure has an interesting S-shape. This shape can be explained by looking at the acknowledgement which is returned after a consumer has finished reading its received data. Sending a packet with a size of 4 bytes means that an equally sized acknowledgement is returned. However, this acknowledgement has to travel a larger distance over the ring interconnect, resulting in a large overhead. Increasing the packet size will clearly reduce this overhead. Based on this graph, there is a turning point around a packet size of 32 bytes. With this size, the time it takes to send 32 bytes equals the amount of time it takes to return the acknowledgement. Further increasing the packet size results in a decreasing overhead and thus an increased throughput. An interesting property of this graph is the fact that throughput seems to be limited at $50 MB/s$. Further increasing the packet size has little to no impact on the achieved throughput.

Further research learned that the throughput of a processing tile is limited by its local PLB. As multiple masters are connected to the same bus, arbitration by a central controller is applied which takes 3 clock cycles [32]. Fetching data from cache or local memory will take in the best case situation 1 clock cycle.

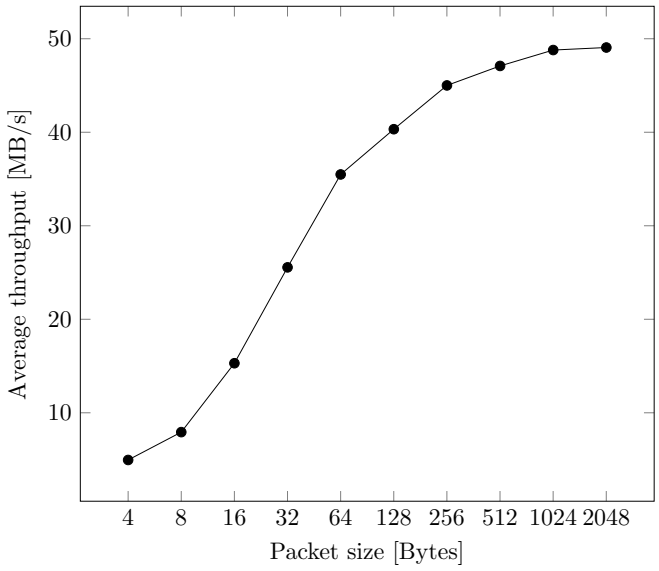

Figure 5.3: CFIFO performance measurements

Sending data to the correct peripheral and routing an acknowledgement back takes another 4 clock cycles, resulting in a total time of 8 clock cycles per transaction. With a clock-speed of $100MHz$ a maximum throughput of $\frac{100}{8} = 12.5MS/s$ or $50MB/s$ can be achieved. This confirms the bottleneck as already identified in Figure 5.3.

Bursting is a technique which can be used to improve data transfer speeds. Instead of communicating one word, multiple words are communicated during one transaction. This reduces the transaction overhead, resulting in higher data rates. The first idea was to implement a so-called DMA controller in the form of the *Xilinx XPS central DMA controller* IP-core [34]. A DMA-controller is a hardware component which can be used to copy a programmable quantity of data from a programmable source address to a programmable destination address. During this transaction no processor intervention is required, and is able (in contrast to a CPU) to burst the data. Figure 5.4 gives an overview of the gateway module equipped with a DMA controller. By default the SPM (denoted "8K RAM") is connected to the much faster LMB. As this bus only supports a single master component, the SPM had to be moved to the PLB to let the DMA-controller communicate with it. The dashed red line in Figure 5.4 displays the communication pattern; the first step is to program the DMA-controller (1), the next step is to read a block of data from the SPM (2) and during the last step this block of data is forwarded via the NI to another processing tile (3).

One disadvantage of this approach is the fact that the NIs don't have support for burst transactions. This means that a block of data can be read from local

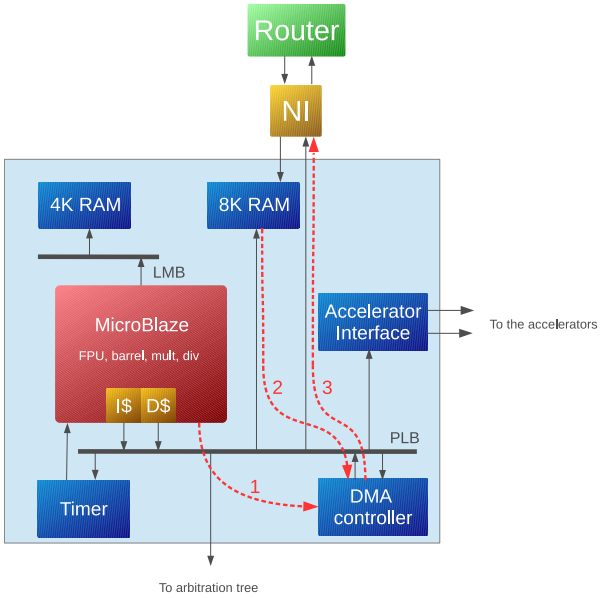

Figure 5.4: Gateway overview, including DMA Controller

memory, but the DMA-controller still has to use single word transactions to forward its data to a NI. As no improvement in throughput is expected this way, two possible solutions for this problem are identified:

1. Modify the NI to support burst transactions.
2. Modify the DMA-controller to directly connect it to a NI, this way completely bypassing the PLB.

Within the Starburst project, a hardware module was already being developed which incorporated a custom written DMA-controller in order to stream data from external memory at high speed over the Nebula ring interconnect and vice versa. This module can be used for benchmarking or debugging purposes. The decision was made to reuse parts of this component in order to make a DMA-controller which directly connected to the ring interconnect. Compared to the Xilinx DMA IP-core a lower area footprint was expected, as less functionality is required (more on this in section Section 7.1).

An overview of this custom written DMA component is given in Figure 5.5. The module consists of a PLB master controller, and a PLB slave controller. Both controllers implement a state machine which handles the PLB communication protocol. Via the PLB slave controller the module can be configured, after which the master controller will start fetching data from memory. This data is stored in a FIFO-buffer, after which it is being streamed to the NI via the Fifo2NI component.

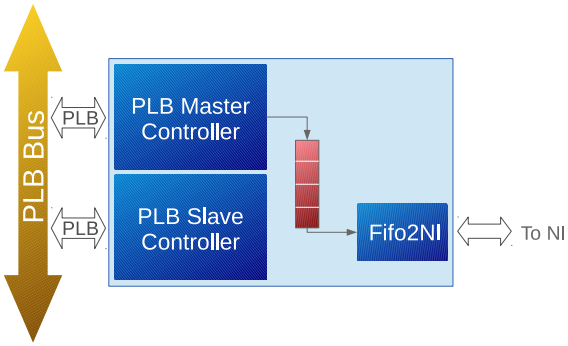

Figure 5.5: Schematic overview of the RingDMA controller

Figure 5.6 gives a new overview of the gateway module with this custom written DMA-controller incorporated. As depicted the DMA-controller is connected to a dedicated NI and router. Even though the additional NI will result in a higher area footprint, there are preliminary ideas to place the hardware accelerators on a dedicated interconnect which requires an additional NI in the first place. This idea will be discussed in more detail as future work in Section 8.2.

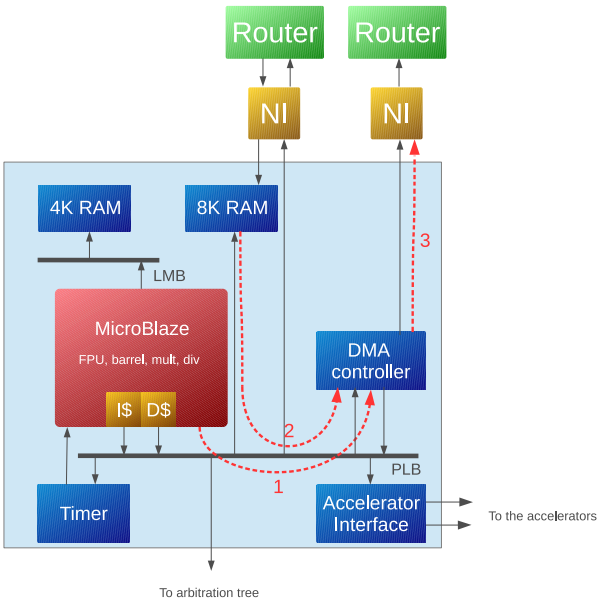

Figure 5.6: Gateway overview, including custom DMA Controller

Currently a CPU has to poll the DMA-controller to check whether it has finished moving the data. Improvements are expected when the polling mechanism is replaced by for example an interrupt mechanism, because polling the

DMA-controller will generate traffic on the PLB resulting in a degradation of performance. An interrupt mechanism will inform the CPU as soon as it has finished moving all data. Implementing a solution based on interrupts would require a lot of modifications to the Helix kernel running on the CPUs. That was not a feasible solution considering the remaining time for this project at that point in time, and can therefore be considered as future work.

With the DMA-controller incorporated in the system, new performance measurements have been performed. The results of these measurements are depicted in Figure 5.7. As the minimal bursting size of this DMA-controller is 32 bytes, experiments with smaller packet sizes have been omitted. Data is communicated according to the CFIFO protocol, were the DMA-controller is used for the actual transfer of data. Overhead is introduced by the acknowledgement which is returned when a complete packet has been received and by configuring the DMA-controller. For the smallest packet size of 32 bytes this overhead caused a slightly lower throughput compared to the software solution. For larger packet sizes a clear improvement is visible, boosting the throughput to $217 MB/s$ for a packet size of 2048 bytes. This is an improvement of 444% compared to the software solution. As the Nebula ring interconnect has a theoretical maximum throughput of $400 MB/s$, 54% of this theoretical maximum throughput was used after implementing the DMA-controller. Overhead sources which prevent a 100% utilization are for example the polling mechanism, the return of a credit over the opposite part of the ring network and overhead introduced by reconfiguring the DMA controller before a transfer can take place.

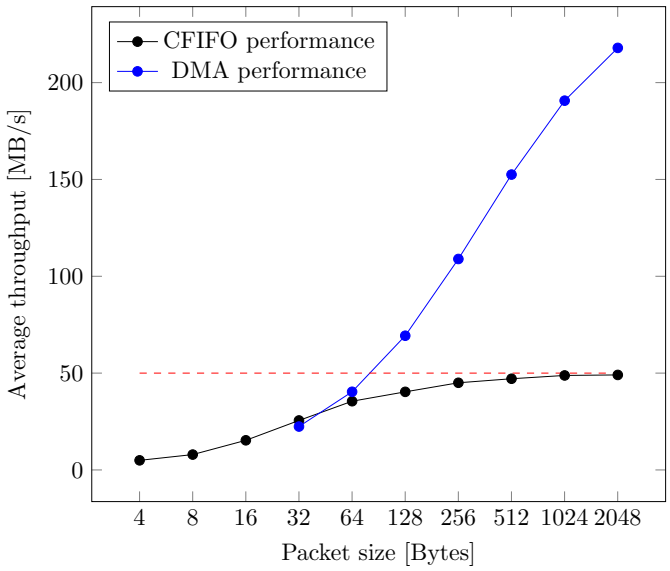

Figure 5.7: DMA performance measurements. The dashed line in red illustrates the theoretical mamximum CFIFO performance

The DMA-controller can be used according to the register map in Figure 5.8. By default the DMA-controller will be placed at address 0x900000. After configuring the source and destination registers, the transaction can be started by writing a '1' to the enable-bit in the control register. Currently the processor has to poll the status register to check whether the DMA-controller has finished the transaction.

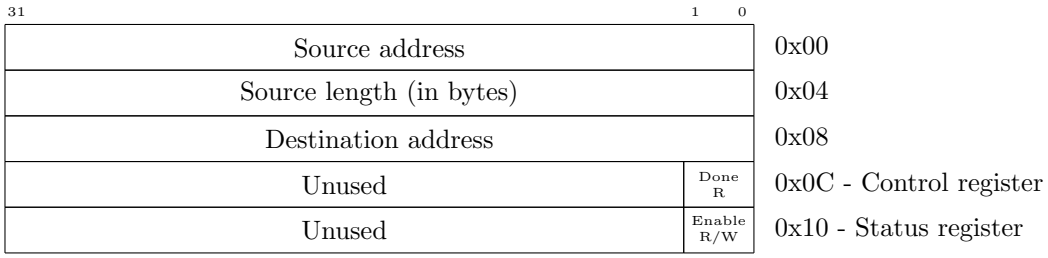

Figure 5.8: DMA-controller address map

5.1.3 Exit Gateway

In the previous two sections two modifications to the gateway module have been discussed; an Accelerator Interface which allows the gateway to easily read and write data from and to the connected accelerators. The second addition is a DMA controller, which makes it possible to stream data at high(er) speeds through the accelerators. This section will focus on one missing feature, which is the ability to check whether enough free space is available at the output in order to store a completely processed data packet. This requirement has been discussed in Section 4.3.2. In the current implementation it is only possible to check the presence of tokens on the connection between the gateway and the first connected accelerator, it is impossible to determine if free space at the output is available.

As explained in Section 3.2.5 returning from a memory unaware hardware accelerator to a memory aware processor is currently achieved with a hardware FIFO-buffer directly connected via a FSL-link to a processor. As there is no way to easily extract the amount of free space available in this FIFO-buffer, this solution is not suitable for our problem. Furthermore, a fundamental problem of this FIFO-buffer is the fact that each processor has to be equipped with this additional hardware in order to let it communicate with a hardware accelerator. For the CPUs in the system this problem has already been solved via the CFIFO communication protocol, as discussed in Section 3.3.2. Via this communication protocol a processor is able to determine if free space for a complete packet is available at the receiving side before a data packet is directly written in the local memory of the receiving CPU.

This gave us the idea to use the CFIFO communication protocol to solve the

missing piece of the puzzle. The idea is to implement a regular CFIFO buffer between a gateway and a consumer. However, instead of writing data directly to the local memory of the receiving CPU, data is streamed through these accelerators into the local memory of the receiving processor. Afterwards the receiving processor is informed that new data is available. Via this protocol the gateway is able to determine the availability of free space which solves our problem. An additional (and highly appreciated) advantage is the fact that the hardware FIFO-buffers and FSL-links can be completely omitted as the results are directly streamed into local memory.

There are however two problems concerning this idea:

1. A hardware accelerator is equipped with a programmable register which stores the address where data has to be forwarded to. This way data can only be forwarded to one fixed address (keyhole addressing), where the SPM of the receiving processor has to be addressed incrementally.

2. The receiving processor should be informed that new packet of data is available. However, this can only be done if *all* data has passed through the chain of accelerators and is stored in the SPM of the receiving processor, as it would otherwise process invalid and incorrect data.

In order to solve those two problems we propose to use an additional component placed directly after the chain of accelerator called an *Exit Gateway*. The purpose of this component is to:

1. Generate incrementing addresses for the incoming data packets, in order to be able stream the incoming data packets directly to the SPM of a receiving processor.

2. Count the number of processed words, in order to determine if a complete packet has been processed.

3. Inform the receiving processor that a new packet has been received, according to the CFIFO protocol.

The positioning of this component is displayed in Figure 5.9. As discussed the exit gateway is positioned right after the last accelerator, and is connected to the accelerator interface. This way the gateway (GW_0) is able to program this component with a start address and a packet size and after sending all data it is able to check if all data has passed this component.

After adding the Exit Gateway to the Starburst platform (see Appendix A.1 for detailed information), the module can be programmed according to the register map as displayed in Figure 5.10. As an accelerator is forwarding data to this component a credit return address has to be programmed in the ringshell of the Exit Gateway (at address 0x00), all other ringshell registers are unused. The second step is to program a start address pointing to the SPM of the receiving processor (at address 0x14). Besides this address the packet size (at address

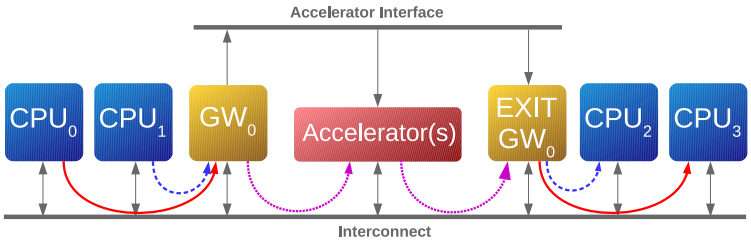

Figure 5.9: Hardware Accelerator sharing overview, including exit gateway

0x18) has to be programmed and the word counter has to be set to zero (at address 0x1C).

After programming the Exit Gateway, the earlier discussed DMA controller can start streaming data. However, instead of polling the DMA controller to check if all data has been processed, the gateway should poll the Exit Gateway (at address 0x10) to determine if the programmed number of words have passed this component. This is still a sub-optimal solution, where additional traffic is generated over the PLB limiting the achievable throughput of the DMA controller. Just as the DMA controller a solution based on interrupts is preferred but not yet implemented. A solution based on interrupts can therefore be considered as future work.

31 … 1	0	
Credit Return Address		0x00 (Ringshell)
Data Forward Address (unused)		0x04 (Ringshell)
Data Snoop Address (unused)		0x08 (Ringshell)
Number of Credits (unused)		0x0C (Ringshell)
Unused	Done R	0x10 - Status register
Start Address		0x14
Packet Size		0x18
Word Counter		0x1C

Figure 5.10: Exit Gateway address map

5.2 Software

An important aspect of hardware accelerator sharing is the fact that the configuration and state for multiple data streams needs to be managed. As discussed in Section 5.1.1 an Accelerator Interface was added to the gateway, which allows the gateway to write and read data from and to the accelerators. This section will focus on the software side, where a mechanism to handle multiple configurations and states will be discussed.

As discussed it is required to manage multiple configurations and state for a single accelerator, however it is also possible to chain multiple accelerators together. This means that for a single data stream the state and configuration for an arbitrary number of accelerators needs to be managed. An efficient data structure was required to store this information. The fact that hardware accelerators can be chained together was the inspiration to store the configuration and state in a *linked list*. A linked list is a data structure consisting of a set of *nodes*, where each node consists of some data and a pointer to the next node in the list. The configuration and state for a single accelerator is stored in such a node. Next, multiple nodes are chained together where the order of nodes represents the required chaining of accelerators. An advantage of this linked list is the fact that an arbitrary number of nodes (and thus hardware accelerators) can efficiently be chained together, where the node in software maps directly to a accelerator in hardware.

Writing software for the Starburst platform requires the programmer to be careful, as the CPUs are implemented without an MMU no memory protection is available. Writing to the memory space of another process will not raise any exceptions and will result in undefined behaviour. Besides that it is required to determine for each accelerator to which physical address data has to be forwarded or to which address credits have to be returned. Using an incorrect address will simply deadlock the complete stream of data. Writing (and thoroughly testing) code once and reusing this code multiple times at other positions in the system is the key to solve these kind of bugs in the code. For this reason the accelerator administration was developed via an Object Oriented (OO) approach (in `C++`), which offered powerful techniques like *polymorphism* and *inheritance*.

An Unified Modeling Language (UML) class diagram is displayed in Figure 5.11, and gives an overview of the most important classes and their relations. The earlier discussed linked list consists of *Accelerator* objects, where each accelerator has a *aggregation* relationship to a next Accelerator object. The Accelerator class contains code which is equal for all accelerators, e.g. code to calculate the address to which an accelerator has to forward data, or to which address credits have to be returned. The Accelerator class contains two functions *writeState()* and *saveState()* which can be used to load or save the configuration and state to or from an accelerator. The idea is that when either of these functions is called,

the objects in the linked list will recursively call this function on the next object in the linked list after executing their own code. As the actual behaviour of these functions depends on the type of accelerator, subclasses are used to implement this specific functionality (e.g. the classes FIR Filter, FM Demodulator and Mixer in Figure 5.11). To be more specific, those two functions are *pure virtual* functions, which makes the Accelerator class *abstract*. This basically means that a programmer is forced to implement the functions in a subclass, it is impossible to directly instantiate the Accelerator class. Furthermore, the configuration and state also depends on the type of accelerator and is therefore also stored in these subclasses. This way accelerator-specific functionality is implemented via a technique called polymorphism, inheritance allows us to reuse code which is common to all accelerators.

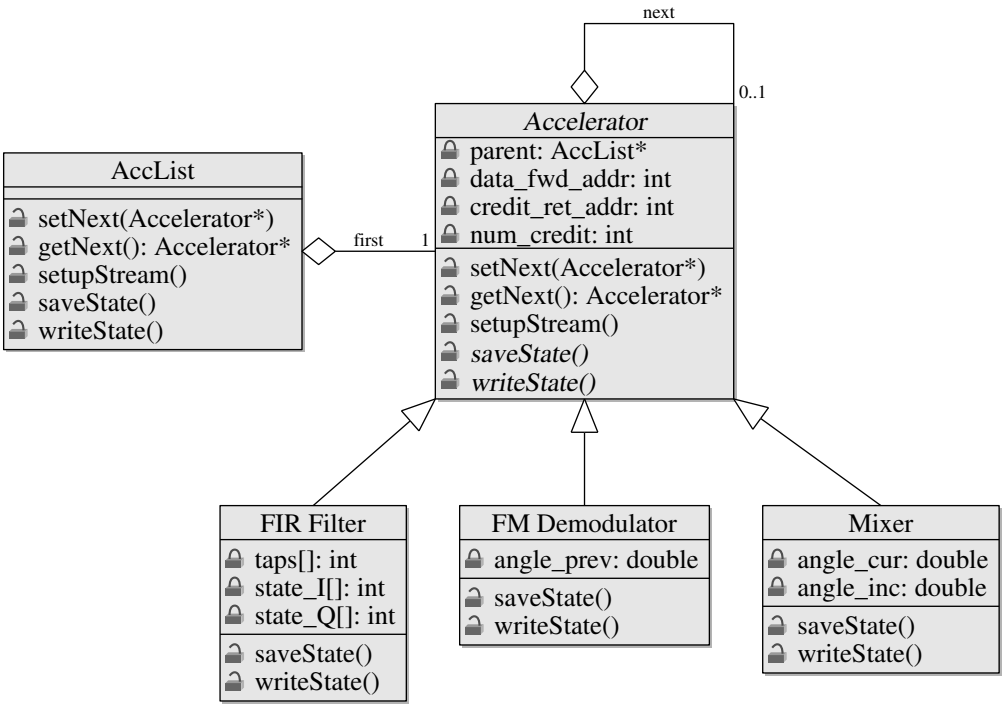

Figure 5.11: UML diagram accelerator administration

After creating a list of accelerator objects in software, a connection with the real hardware has to be made. First, it should be checked whether the requested accelerator is present in the system, whether it is unused and at which physical address the accelerator can be found. For this reason a central administration is maintained, implemented in the *AccList* class. A component called a *ring map* is added to the Starburst platform. This is a simple Read-Only Memory (ROM) which contains a layout of the Starburst platform. Based on this layout the

Acclist class is able to determine if a requested accelerator is present, available and at which address the accelerator can be found. Right after creating the linked list, the programmer has to call the function *setupStream()*. This function will be recursively called by the accelerator objects in order to determine the availability and physical address for all accelerators in the linked list.

In Appendix A.2 an example is given, showing how to instantiate and use this software implementation.

Chapter 6

Dataflow analysis

The previous chapter focussed on the implementation of the sharing mechanism. In this chapter we present the temporal analysis model. We start with a model where just one stream is being serviced by a gateway, next this model is extended to incorporate multiple streams.

6.1 Introduction

A dataflow model is a directed graph $G_S(E, V)$, consisting of actors $v \in V$ and edges $e \in E$, where each edge describes a directed channel between actors: $e_i = (v_i, v_j)$. Edges represent unbounded queues to store tokens. Actors have a firing rule; when at least a specific number of tokens (called quanta) is present at all input edges the actor fires. When this happens all tokens at the input edges are instantly consumed, and after a certain execution time a specific number of tokens is produced at the output edges. When enough tokens are available an actor can fire multiple times simultaneously. Self edges with one token can be added to an actor in order to prevent parallel execution when this is not desired.

A large number of dataflow models exists, with different types of firing rules. In this chapter the following tree dataflow models are used: HSDF, SDF and CSDF. These models are visualized in Figure 6.1, and will now shortly be discussed.

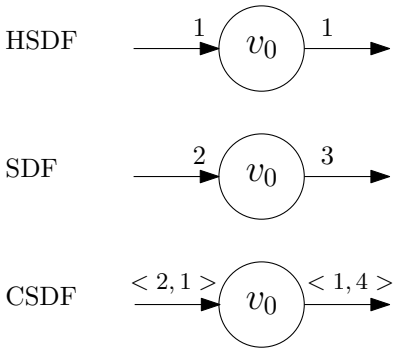

Figure 6.1: Visualization of a number of dataflow models

HSDF is dataflow model with a firing rule where all actors will consume one token on each incoming edge, and produce one token on each outgoing edge [3].

This in contrast to SDF where an arbitrary number of tokens are consumed and produced. CSDF [5] extends SDF by introducing the concept of *phases*. Each actors cycles through a predefined number of phases, where $\rho_v(p)$ is denoting the execution time for phase p. $\pi(e_{ij}, p)$ and $\gamma(e_{ij}, p)$ are respectively denoting the production quanta and consumption quanta on edge e_{ij} during phase p. Furthermore, the number of phases for actor v_i is denoted as $\theta(v_i)$. It is important to note that all actors do not need to have the same number of phases. Overlapping firings are not allowed [5], which means that all actors have an implicit self-edge with a single token on it. One exception to this rule are single-phase actors, we allow them to fire concurrently. Self-edges have to be added to prevent overlapping firings.

Two important properties are being used in this discussion; *refinement* and *abstraction* [2]. A component is said to refine another component in the temporal domain if:

$$\forall i, m \bullet \hat{a'}_m(i) \leq \hat{a}_m(i) \Rightarrow \forall j, n \bullet \hat{b'}_n(j) \leq \hat{b}_n(j) \tag{6.1}$$

Where a_m are the input ports and b_m the output ports of component C, and a'_m the input ports and b'_m the output ports of component C'. In other words, this implies that one component C' refines a component C, then in the worst-case situation tokens produced by C do not arrive earlier than tokens produced by C'. Refinement is denoted as $C' \sqsubseteq C$. The opposite of refinement is abstraction. Abstraction is most of the time used to reduce the number of cases that need to be considered during analysis, this comes however at a cost of a reduced accuracy.

6.2 Nebula Ring Network

An important property of the Nebula ring network is the fact that it can be modelled with a dataflow model. There are two types of traffic on the Nebula ring network (CFIFO communication and credit-based communication), the dataflow models of both communication protocols will be discussed in the next sections.

6.2.1 CFIFO Communication

A CFIFO software buffer with capacity α is modelled as an SDF graph as depicted in Figure 6.2. The model consists of a producer P and consumer C, with execution times $\hat{\rho}_P$ and $\hat{\rho}_C$. Both actors are connected to a ring with a total size of N hops, which means that communication between actors P and C

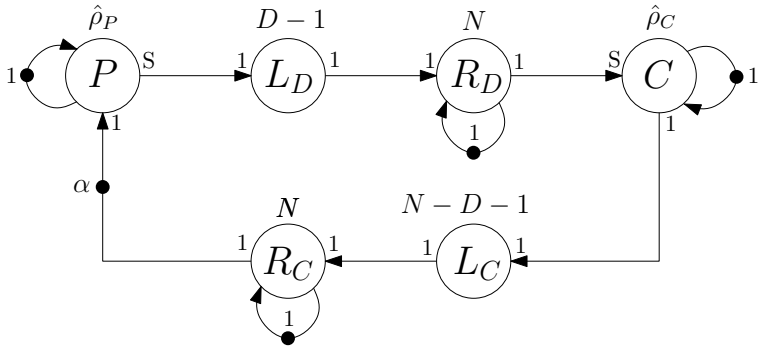

Figure 6.2: SDF model Nebula ring interconnect

is rate limited to $\frac{1}{N}$ as a router has to wait N cycles in the worst case situation before his own slot* will pass again.

Latency is introduced by the network, where the worst-case situation occurs when the own slot has just moved away. This introduces a latency of $N-1$ cycles, after this amount of cycles the slot passes the router again. The number of hops needed to get from P to C is defined as D. This results in a total latency of $N+D-1$ cycles. Two separate actors are used to model latency (L_D) and rate (R_D) between actors P and C, where we need to subtract the execution time of the rate limiter (N) from the execution time of the latency actor. This results in the following execution times:

$$R_D = N$$
$$L_D = (N+D-1) - N = D-1$$

When S tokens have been received and consumed by the consumer, a token will be sent back to the producer. The communication between C and P is again rate limited by $\frac{1}{N}$, which is modelled by actor R_C. The distance between P and C was defined as D. A credit has to travel across the opposite part of the ring, which means that it has to travel $N-D$ hops. As the credit has to wait at most $N-1$ cycles before it is accepted by the ring, this results in a total latency of $(N-D)+(N-1) = 2N-D-1$. Subtracting the execution time of the rate limiter from the execution time of the latency actor results in the following execution times:

$$R_C = N$$

*Every router passes data and addresses to its neighbouring router, both of which may be empty. These pairs are unique and defined as a *slot* on the ring network. By labelling each slot with an unique and incrementing number we can distinguish between slots. By numbering the routers in the same manner we introduce the concept of an *owned slot* when these numbers match.

$$L_C = (2N - D - 1) - N = N - D - 1$$

Note that the presented dataflow model is based on the worst-case (guaranteed) bandwidth of $\frac{1}{N}$. In Section 3.2.4 the work conserving principle of the Nebula ring was explained. The idea is that when an empty slot passes a router, this router may hijack this slot if the destination router is reached before the owner of the slot is reached. The theoretical upper bound decreases linearly with the number of hops data has to travel. When data is addressed to its direct neighbour all slots may be hijacked (under the assumption that all slots are empty), when an actor is sending data to itself no slots may be hijacked, resulting in the already calculated lower bound of $\frac{1}{N}$. The bandwidth B can therefore be described as:

$$\frac{1}{N} \leq B \leq \frac{N - D + 1}{N}$$

Where B is the total bandwidth, N the ring size and D the number of hops data has to travel. Being able to hijack other slots also has consequences on the total latency introduced by the ring interconnect. The worst case situation has already been discussed, which happens when a slot has just passed the router which forces the router to wait $N - 1$ cycles. Additionally, data has to travel D hops resulting in a total latency of $N + D - 1$ cycles. In the best case situation data can directly hijack another slot, which takes 1 cycle. Combining this with the fact that data has to travel D hops results in a latency of $D + 1$ cycles. The bandwidth L can therefore be described as:

$$D + 1 \leq L \leq N + D - 1$$

6.2.2 Credit-based Communication

As discussed in Section 3.2.5 hardware accelerators are integrated into the Starburst platform, where communication is taking place based on credits. An accelerator is equipped with a hardware FIFO buffer, where credits are denoting the number of free positions in this buffer. Accelerators are consuming data word-by-word, where a credit is sent back to the producer after the consumption of each word. A producer is therefore only allowed to send data if it has at least one credit.

Figure 6.3 shows an SDF model of this accelerator communication protocol, where producer P is communicating data with accelerator *Acc*. For simplicity an ideal network is assumed, where the latency and rate actors have been omitted. The model in Figure 6.3 implies that a buffer with size $\alpha \geq S$ is always required. However, hardware accelerators often consume and produce only one word per time unit, and as such it is not desirable to have a large input buffer. A problem

occurs when a packet with a size of S has to be communicated, where $S > \alpha$. In that situation the model will deadlock as none of actors is able to fire.

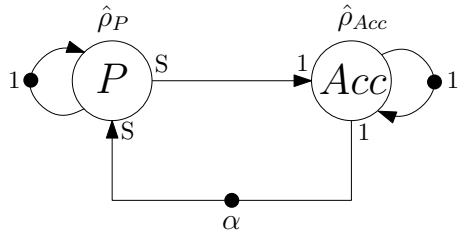

Figure 6.3: Potentially dead-locked accelerator communication model

Figure 6.4 shows an CSDF model of the accelerator communication model. A packet of S words is communicated in S separate phases, where single words are written at the end of each phase. After word has been accepted by accelerator Acc, a credit is sent back to the consumer. In contrast to the previous model, this model is dead-lock free if $\alpha < S$.

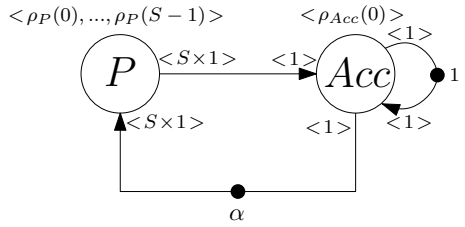

Figure 6.4: CSDF accelerator communication model

Figure 6.5 depicts the same CSDF model, where the earlier discussed latency and rate actors have been added. In contrast to the CFIFO communication protocol, credits are travelling on a dedicated credit-ring, rotating in the opposite direction. This has consequences for the latency actor L_C, as the credits have to travel the same distance $(D-1)$ as the actual data.

6.3 Single data stream

With the dataflow models of the communications protocols introduced, we can now focus on the dataflow model of the sharing mechanism. As discussed in the previous sections (Section 4.3.2 and Section 5.1.3) the gateway and a consumer are communicating data via a CFIFO software buffer. However instead of streaming the data directly to this consumer, data is streamed through one or more hardware accelerators in order to process the data stream. For this reason we start with a SDF model of a CFIFO software buffer, which is depicted in

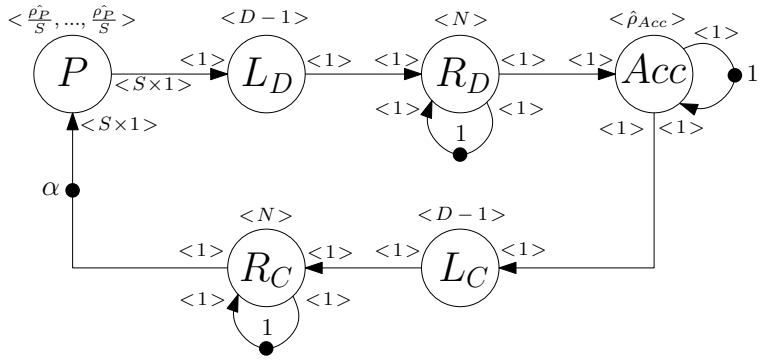

Figure 6.5: CSDF accelerator communication model

Figure 6.6. The software buffer has a depth of α, which is modelled by the back-edge (with α initial tokens) running from the consumer C_0 to the gateway GW_0. A packet of S words is being communicated. An interesting property of this model is that while a packet of S words is being communicated, $S+1$ tokens are being produced and consumed. This extra token models the update of the write-pointer (See Section 3.3.2) just after sending all words to the consumer. It is important to note that normally the sum of tokens one these edges reflects the total buffer capacity. Due to the return of the single token this is not true in our situation, as the total buffer capacity equals $\alpha \times S$. For simplicity an ideal network is assumed which allows us to remove the latency and rate actors modelling the network characteristics.

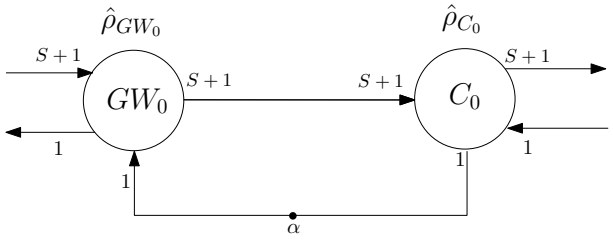

Figure 6.6: SDF model: CFIFO communication protocol

In the previous section it was explained why a CSDF model is required to accurately model credit-based communication with a hardware accelerator. For this reason the SDF model in Figure 6.6 is modified to a CSDF model which is depicted in Figure 6.7. Instead of producing $S+1$ tokens at once in the case of the SDF model, tokens are produced during $S+1$ separate phases in the CSDF model.

Instead of communicating data directly to consumer C_0, data is streamed through one or more accelerators. For this reason edge $e(GW_0, C_0)$, is removed from the

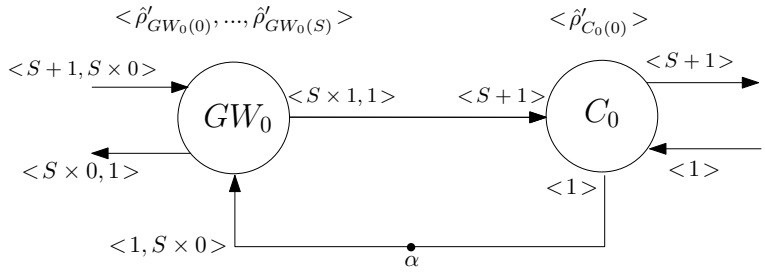

Figure 6.7: CSDF model: CFIFO communication protocol

model in Figure 6.7, and an accelerator is added. The modified model is displayed in Figure 6.8. The gateway and accelerator are communicating data credit-based, where the accelerator has a hardware FIFO buffer with a depth of δ_0. This depth is modelled via a back-edge with δ_0 initial tokens running from the accelerator to the gateway.

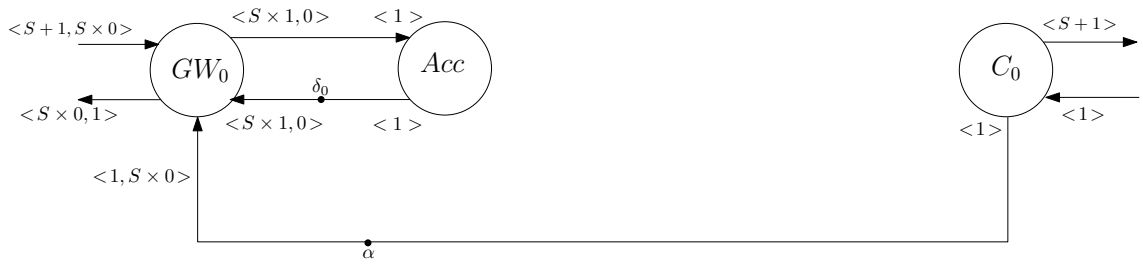

Figure 6.8: CSDF model: Hardware Accelerator added

The next step is adding the Exit Gateway which was discussed in detail in Section 5.1.3. The purpose of this component is counting the number of words which have been processed, which allows the gateway to determine if the complete accelerator pipeline is empty. It also generates addresses in order to correctly address the consumer. The addition of the Exit Gateway is depicted in Figure 6.9. The accelerator is communicating data with the Exit Gateway credit-based, where this component is equipped with a hardware FIFO buffer with a depth of δ_1. This is modelled via a back edge running from actor GW_1 to actor *Acc*. After receiving and producing S tokens, the Exit Gateway will generate the pointer update for the CFIFO protocol itself, which means that in total $S+1$ tokens are produced by this actor.*

The last step is adding an edge running from actor GW_1 to GW_0 without initial tokens. This models the requirement that S words have been processed by the

*The functionality to let the Exit Gateway generate the pointer update is not yet implemented, instead this is currently solved in software where the gateway (GW_0) is sending the pointer update.

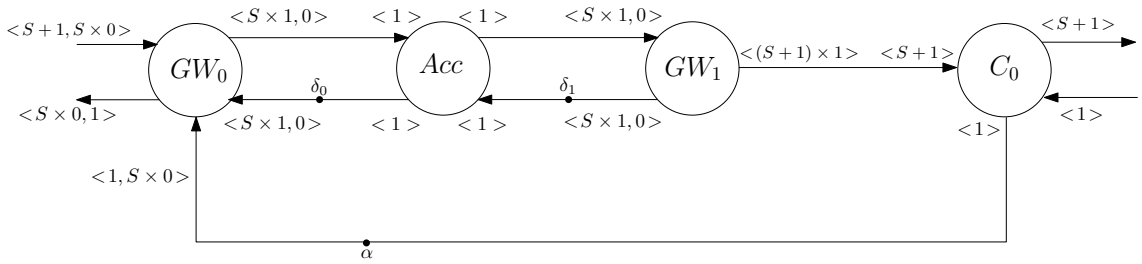

Figure 6.9: CSDF model: Exit Gateway added

Exit Gateway, which makes sure that the complete accelerator pipeline is empty. A token will appear on this edge right after S tokens have been received. Actor GW_0 will wait for this token before it continues. The addition of this edge is depicted in Figure 6.10.

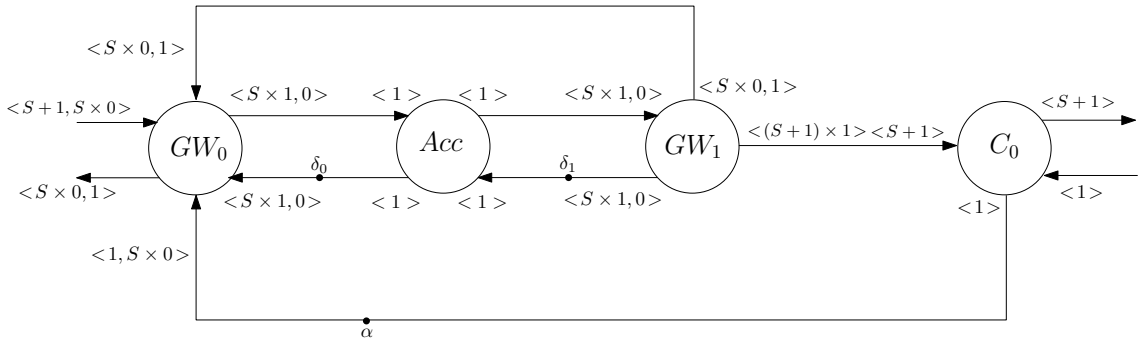

Figure 6.10: CSDF model: Back-edge added

6.3.1 Abstraction

The constructed CSDF model is, apart from the missing network actors, rather detailed. While the amount of detail is not a real problem when the gateway is processing just one data stream, it is becoming a problem when multiple data streams are being processed. What we are actually (only) interested in is the total amount of time (in the worst-case situation) it takes to process a data packet of S samples. This amount of time is defined as the difference between the moment of time GW_0 consumes all input tokens and the moment the last sample has left the exit gateway GW_1. If this amount of time is calculated for all data streams which are processed by the gateway, the influence of the data streams on each other can be determined.

For this reason an abstraction of our CSDF model will be constructed which allows us to hide specific details. This principle is depicted in Figure 6.11. As

we are only interested in the moment the last gateway produces the last token of a packet, the idea is to represent the gateway, the accelerators, the exit gateway and all network actors (all covered by the dashed red box) in a single actor GW'. As discussed, this abstraction is valid if component GW' never produces tokens earlier than the original set of components which are covered by the red dashed box.

The consumption of tokens happens in both situations at the same moment of time; in the original CSDF model all $S+1$ tokens are consumed in the first phase, the same holds for a tokens from edge $e_{(C_0,GW_0)}$. In the abstraction this is also true, as actor GW' will instantly consume the same amount of tokens. The production of tokens by actor GW_1 happens in $S+1$ independent phases, actor GW' will produce all $S+1$ tokens after an execution time of $\hat{\rho}'_{GW}$. This basically means that the abstraction is valid if actor GW' will not produce tokens earlier than the last token production of actor GW_1. This forces us to calculate the worst-case production time of the last token on actor GW_1, according to the refinement property this production time can then be used to determine $\hat{\rho}'_{GW}$.

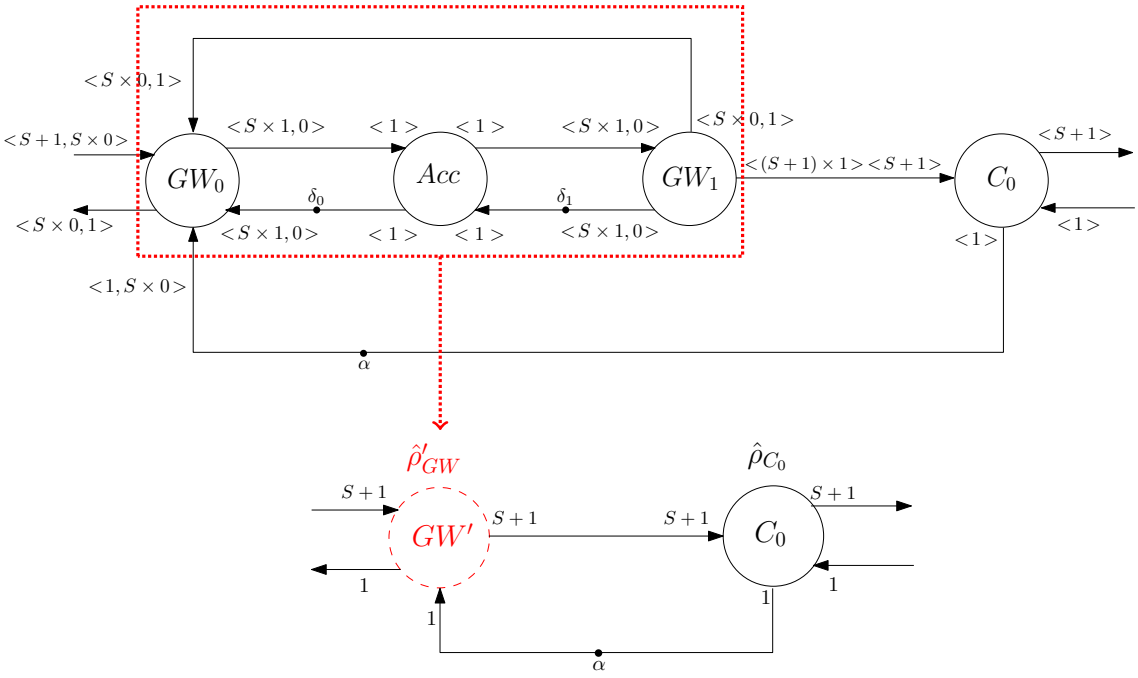

Figure 6.11: CSDF model: Abstraction

6.3.2 HSDF conversion

In the introduction of this chapter HSDF was already introduced. HSDF is a restricted version of SDF where all actors will consume one token on each incoming edge, and produce one token on each outgoing edge [3]. HSDF graphs can be executed in a self-timed manner. During self-timed execution all actors will execute as soon there is at least one token on each incoming edge. If the graph is strongly connected (from each actor all other actors are reachable) and the order of tokens is maintained (by giving each actor a self-edge with one initial token, or by giving the actor a constant response time) the HSDF graph has strong analytical properties.

The first property is the fact that a graph is deadlock-free if each cycle in the graph has at least one token. Secondly, the execution of HSDF graphs is *monotonic*, which means that decreasing actor firing times or increasing initial tokens will result in non-increasing actor start times. The reason for this behaviour is the fact that an earlier token arrival time cannot result in a later actor start time. And third, an HSDF graph will always enter a periodic regime. Or more precise, after an initial phase $K \in \mathbb{N}$ the execution enters an periodic regime with a period $\mu.N$, with $N \in \mathbb{N}$ describing the number of produced tokens during one period (the cyclicity) and $\mu \in \mathbb{R}$ the inverse of the average throughput. This means that for all actors $v \in V$, $k > K$ the start time $s(v, k+N)$ of actor v in iteration $k+N$ is described by:

$$s(v, k+N) = s(v, k) + \mu.N \tag{6.2}$$

The Maximum Cycle Mean (MCM) of a HSDF graph is equal to μ and defined as:

$$MCM(G) = \max_{c \in C_G} CM(c) \tag{6.3}$$

$$CM(c) = \sum_{v \ on \ c} WCRT(v)/d(c) \tag{6.4}$$

Basically the Cycle Mean (CM) of a simple cycle c is calculated by summing the response times for all actors v on cycle c, and dividing it by the total number of tokens on cycle c (denoted by $d(c)$). The maximum of all Cycle Means will then determine μ, hence the name Maximum Cycle Mean.

With HSDF now shortly introduced, we can return to our original problem. The goal is to convert our rather detailed model to a less detailed abstraction, as was displayed in Figure 6.11. The missing piece of the puzzle is the worst-case execution time of actor GW'. The idea is now to calculate this execution time based on the MCM of our original CSDF model. For this reason the original CSDF graph is transformed into an equivalent HSDF graph [5]. A detailed

explanation of the algorithm to transform a CSDF graph to an HSDF graph is beyond the scope of this thesis. It is sufficient to know that this transformation is applied by mapping all separate phases of a CSDF actor to individual HSDF actors and constructing new edges between the HSDF actors, with the correct number of initial tokens. This introduces some serious problems in our situation; the size and structure of the equivalent HSDF graph depends both on the packet size S, and the number of initial tokens δ_0 and δ_1. (See Figure 6.11)

We could solve this problem by using specific tools which are able to automate the transformation, like Hebe [27]. This tool was used during this thesis, and was able to successfully calculate an equivalent HSDF graph and extract its MCM. There was however one large drawback of this approach. It is virtually impossible to predict the impact on the transformed HSDF graph if the original CSDF graph is (slightly) modified. A different packet size S or different number of initial tokens δ_0 or δ_1 forced us to do a completely new transformation and MCM calculation. Eventually, it is required to calculate an appropriate packet size S. A parametric solution is preferred in this situation, where the impact of tuning certain parameters is directly visible.

6.3.3 HSDF model

The previous section learned that an (automatic) CSDF to HSDF conversion is not preferable. The size and structure of the resulting model depends on multiple parameters such as the packet size S, as the gateway actors with $S+1$ phases are mapped onto $S+1$ separate HSDF actors. Instead of applying transformations we have manually constructed an HSDF model, which will be discussed in this section.

One of the reasons which forced us to use CSDF to model the accelerator sharing mechanism is the fact that after communicating S tokens a single token is returned from GW_1 to GW_0 (See Figure 6.12). It is hard to model this behaviour accurately in a (compact) HSDF model. However, the token from GW_1 to GW_0 is communicated only during the last iteration, when all other tokens have already been communicated. This allows us to distinguish two independent phases: during the first phase S tokens are communicated from GW_0 to GW_1, during the second phase a single token is returned. The total processing time can therefore be determined by calculating and summing the processing times for both separate phases.

Figure 6.13 shows a new and detailed HSDF model, with two gateways (GW_0 and GW_1), an accelerator and network actors (L_* to model latency, R_* to model bandwidth). As this is an HSDF model, the actors have a fixed (production and consumption) quanta of 1. Instead of having S duplicated actors to model the fact that S tokens are communicated (what happened with the CSDF to HSDF transformation), the actors in this model have to fire S times in order to communicate S tokens.

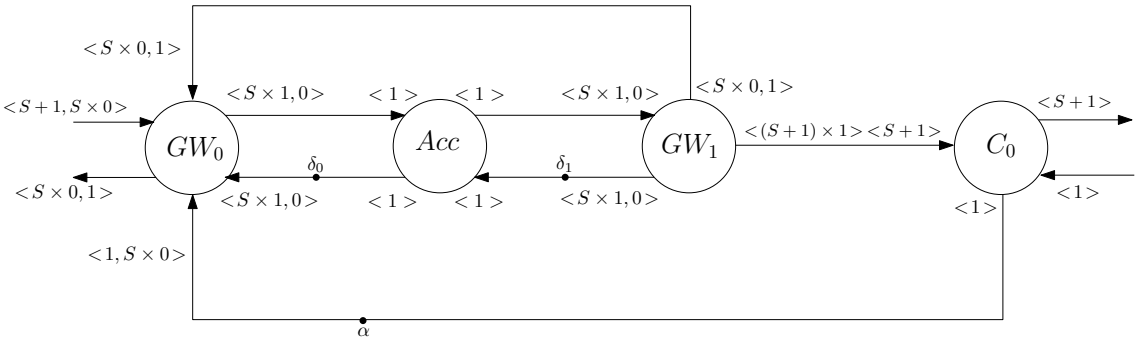

Figure 6.12: CSDF model

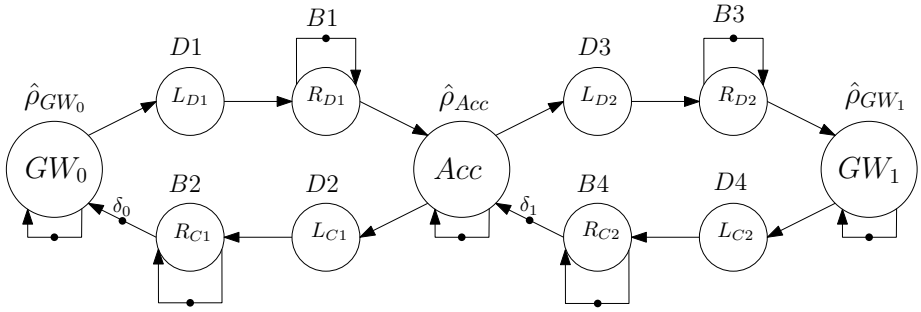

Figure 6.13: HSDF model of two gateways and one hardware accelerator

6.3.4 Schedule Definitions

The total processing time will be calculated by means of an execution schedule. The schedule function $s(v, k)$ represent the time at which the instance k of actor v is fired [21]. As firings are counted from 0, instance k corresponds to the (k+1)-th firing. The finishing time of the k-th firing of actor v is denoted as $f(v, k)$. It always holds that $f(v, k) = s(v, k) + \rho(v, k)$. If $\hat{\rho}(v)$ is the worst-case firing duration, then $\rho(v, k) \leq \hat{\rho}(v)$, for all $k \in \mathbb{N}_0$.

A Self-Timed Schedule (STS) is a schedule where each actor fires as soon as all its input edges hold enough tokens. In the case of a HSDF graph this means that the input edges hold at least one token. When worst-case execution times are assumed ($\hat{\rho}(v)$ instead of $\rho(v, k)$) the schedule becomes a Worst-Case Self-Timed Schedule (WCSTS). An interesting property of WCSTS has already been discussed; after a transition phase K, the schedule will enter a periodic regime:

$$s(v, k + N) = s(v, k) + \mu.N \tag{6.5}$$

Instead of letting a graph execute self-timed, we could delay the enabling of the

actors to force a graph to execute with a certain period* T. The schedule is then called a Static Periodic Schedule (SPS) and is defined as:

$$s(v,k) = s(v,0) + T.k \tag{6.6}$$

In [21] it is proven that for any HSDF graph G, it is possible to find a SPS schedule as long $T \geq \mu(G)$. If $T < \mu(G)$ no SPS exists. If the period T is chosen such that it is equal to the MCM $\mu(G)$, the schedule is called a Rate Optimal Static Periodic Schedule (ROSPS):

$$s(v,k) = s(v,0) + \mu(G).k \tag{6.7}$$

Concluding, when worst-case execution times are assumed during the self-timed execution of graph G, the graph will fire according to the WCSTS. Eventually (after a transition phase K) the schedule will settle into a periodic behaviour, with a period $\mu(G)$. On the other hand it is always possible to enforce a graph G to execute with a period $T \geq \mu(G)$. When $T = \mu(G)$ the schedule is called a ROSPS. It is important to note that because of monotonicity in any admissible SPS schedule actors can never fire earlier than the WCSTS. This way you can see an SPS as an upper bound; under the WCSTS actors will never fire later than this SPS.

6.3.5 Schedule Calculation

In the previous section we showed that the ROSPS schedule can be used as an upper-bound, as the actors of the corresponding graph G will never fire later during self-timed execution. Via this periodic schedule we are able to give a parametric solution for the total processing time, which will be discussed in this section. In Figure 6.14 our constructed HSDF graph is displayed. What we are interested in, is the production time of the S-th token on actor GW_1, defined as

$$f(GW_1,(S-1)) = s(GW_1,(S-1)) + \hat{\rho}(GW_1) \tag{6.8}$$

As discussed in the previous section, it is sufficient to calculate this finishing time based on the ROSPS, during STS the firings will never happen later. From Equation 6.7 and Equation 6.8 it follows that

$$f(GW_1,(S-1)) = s(GW_1,0) + (S-1)\cdot\mu(G) + \hat{\rho}(GW_1) \tag{6.9}$$

From Equation 6.9 it follows that it is sufficient to calculate the first enabling time of actor GW_1 (defined as $s(GW_1,0)$) which depends on the positioning of

*The schedule becomes periodic after a transition phase $s(v,0)$, which depends on the position of the initial tokens in graph G.

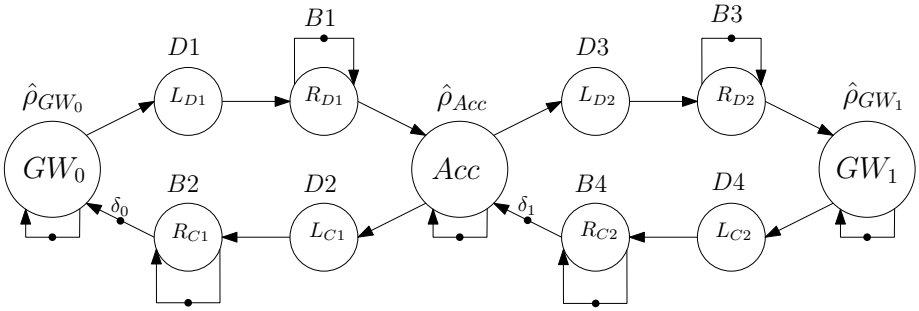

Figure 6.14: HSDF model of two gateways and one hardware accelerator

initial tokens in the dataflow graph, and the MCM of graph G ($\mu(G)$). Simulation (or manually construction an admissible schedule) allows us to determine $s(GW_1, 0)$. Under the assumption that the initial tokens δ_0 and δ_1 are placed as depicted in Figure 6.14 we have constructed a schedule as depicted in Figure 6.15.

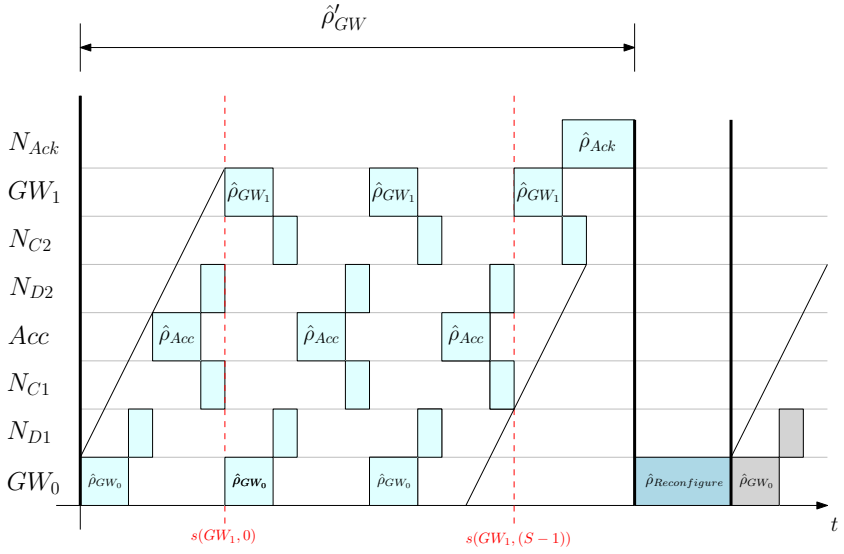

Figure 6.15: Execution schedule ($\delta_0 = 1, \delta_1 = 1, S = 3$). For simplicity each pair of network-actors (R_* and L_*) is replaced by a single actor N_*.

Based on the positioning of initial tokens actor GW_0 is the first actor to fire at $t = 0$. Next we follow the chain of actors leading to eventually the first enabling of actor GW_1 at moment $t = s(GW_1, 0)$ (marked in red). After the first execution we can enforce the ROSPS with period $\mu(G)$ eventually leading to the $S-1$-th enabling of actor GW_1 at moment $t = s(GW_1, (S-1))$ (marked in red).

When the $S-1$-th firing finishes, a token is sent from actor GW_1 to actor GW_0 to inform that all data packets have been processed. In the actual implementation this acknowledgement is communicated through the Accelerator Interface, over the local data bus of the gateway. The worst-case effects introduced by this interconnect are modelled via actor N_{ack}. When the acknowledgement reaches actor GW_0, it will start reconfiguring the accelerators (with a worst-case execution of $\hat{\rho}_{Reconfigure}$). Afterwards the gateway will start processing the next data packet, which is marked in grey.

Next, we investigate the effects of using larger data buffers on the accelerator and the Exit Gateway. A new (self-timed) schedule is constructed where $\delta_0 = \delta_1 = 2$. We start with the WCSTS, which is displayed in Figure 6.16. The influence of the larger buffer size is directly visible in the utilization of the hardware accelerator. The schedule in Figure 6.15 shows that the accelerator is processing data during 2 out of 6 time units. The schedule in Figure 6.16 shows that after increasing the buffer sizes, the accelerator is processing data during 4 out of 6 time units, doubling its utilization.

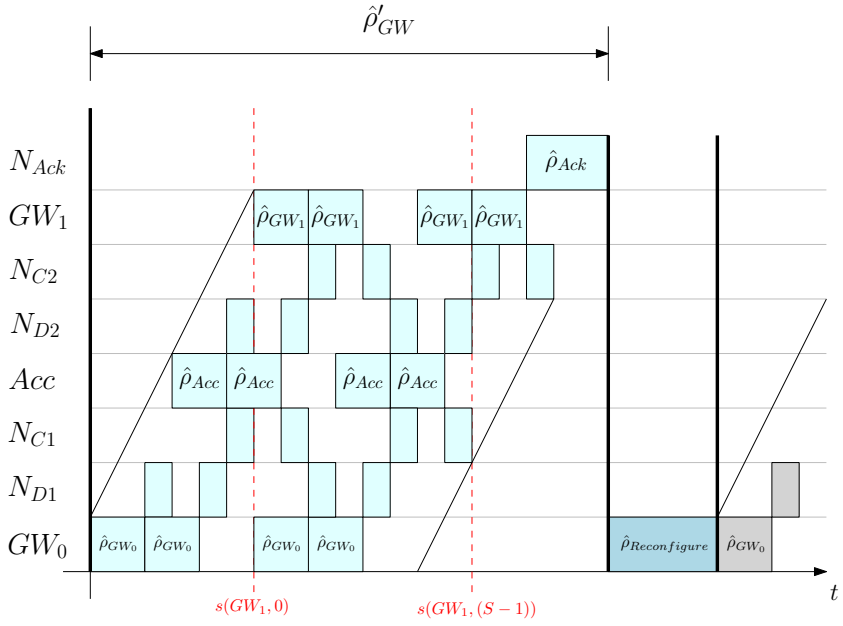

Figure 6.16: Self-timed execution schedule ($\delta_0 = 2, \delta_1 = 2, S = 4$). For simplicity each pair of network-actors (R_* and L_*) is replaced by a single actor N_*.

Next, for the same configuration with $\delta_0 = \delta_1 = 2$ a new schedule is constructed. However in contrast to the previous schedule we will construct the ROSPS schedule, where we enforce a strict periodic firing of actors. In this specific

schedule it turns out that each even firing (2 and 4) of the actors is delayed with one time unit. Based on the constructed schedule we can conclude that the utilization remains unchanged; the accelerator is processing data during 2 out of 3 time units. This specific example also shows the pessimistic approach of the ROSPS schedule where each odd firing (1 and 3) takes place at the same time as the WCSTS, and each even firing (2 and 4) 1 time unit later.

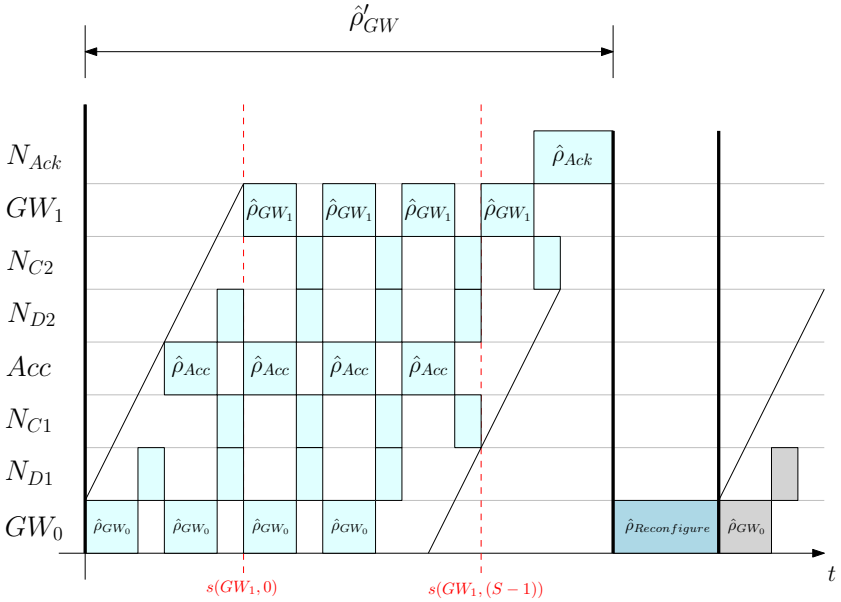

Figure 6.17: ROSPS execution schedule ($\delta_0 = 2, \delta_2 = 1, S = 4$). For simplicity each pair of network-actors (R_* and L_*) is replaced by a single actor N_*.

Concluding, we use the ROSPS in order to determine the total processing time. First we determine the first (worst-case) execution of actor GW_1, by constructing the WCSTS. This is the time it takes to fill the complete processing pipeline, which we denote as λ_{GW_1}

$$\lambda_{GW_1} = s(GW_1, 0) = \hat{\rho}_{GW_0} + D1 + B1 + \hat{\rho}_{Acc} + D3 + B3 \tag{6.10}$$

After the first enabling of actor GW_1 ($s(GW_1, 0)$) we can fire actor GW_1 periodically with a period equal to $\mu(G)$. For this reason we have to calculate the MCM of our HSDF graph, according to Equation 6.4. All Cycle Means of our graph are listed in Equation 6.11.

$$\mu(G) = \max \left(\begin{array}{c} \mu(c_1) = \dfrac{\hat{\rho}_{GW_0} + D1 + B1 + \hat{\rho}_{Acc} + D2 + B2}{\delta_0} \\ \mu(c_2) = \dfrac{\hat{\rho}_{Acc} + D3 + B3 + \hat{\rho}_{GW_1} + D4 + B4}{\delta_1} \\ \mu(c_3) = \dfrac{\hat{\rho}_{GW_0}}{1} \\ \mu(c_4) = \dfrac{\hat{\rho}_{Acc}}{1} \\ \mu(c_5) = \dfrac{\hat{\rho}_{GW_1}}{1} \\ \mu(c_6) = \dfrac{B1}{1} \\ \mu(c_7) = \dfrac{B2}{1} \\ \mu(c_8) = \dfrac{B3}{1} \\ \mu(c_9) = \dfrac{B4}{1} \end{array} \right) \tag{6.11}$$

From Equation 6.11 it follows that when the when c_1 or c_2 are the largest cycles, increasing the number of tokens δ_0 or δ_1 directly lowers the MCM. In that situation the MCM will be determined by either the bandwidth of the interconnect (via cycles c_6, c_7, c_8 or c_9) or by the execution time of the gateway actors (cycles c_3 or c_5) or the accelerator (cycle c_4).

Based on Equation 6.9, Equation 6.10, Equation 6.11 and the constructed schedules, we are able to give a parametric solution for the total time it takes to process a packet of S tokens:

$$\hat{\rho}'_{GW}(G, S) = \lambda_{GW_1} + (S-1) \cdot \mu(G) + \hat{\rho}_{GW_1} + \hat{\rho}_{Ack} \tag{6.12}$$

6.4 Multiple data streams

The previous section focussed on processing a single data stream, where a parametric equation for the total processing time was given. However, the purpose of the gateway is to process multiple data streams, which will be the focus of this section.

As discussed in Section 4.3.3 Round-Robin scheduling will be applied, where the gateway is processing the data streams in a cyclic order. Before a data stream can be processed the set of accelerators has to have their state loaded, afterwards the state has to be saved. This principle can be drawn as a dataflow graph, an example with two data streams is depicted in Figure 6.18.

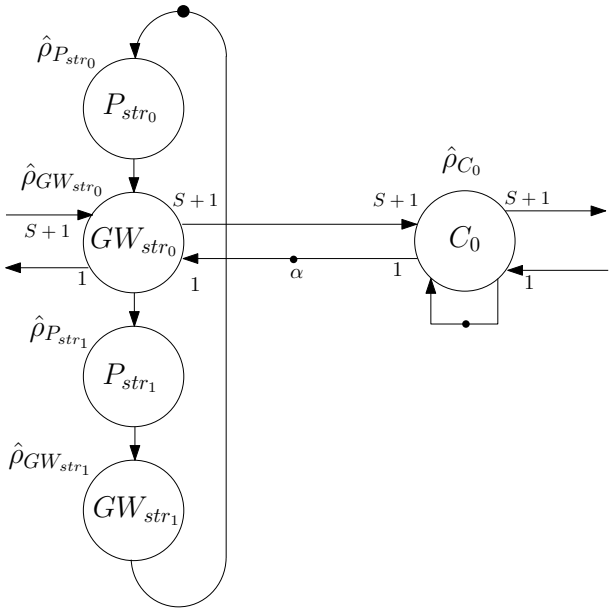

Figure 6.18: SDF model multiple data streams

In this figure two data streams are being processed which is modelled via actors GW_{str_0} and GW_{str_1}. We deliberately left out edges on actor GW_{str_1}, as our algorithm will proceed with a next data stream when insufficient tokens $(S+1)$ are present. By leaving out these edges we assume that this actor will always fire even when insufficient tokens are available, which models the worst-case situation. Before data can be processed, accelerators need to be programmed and afterwards state need to be saved. Both actions are modelled via a single actor P_{str_i}. The firing duration of this actor depends on the number of registers which have to be save and/or loaded. The actors are connected in a cyclic order with one initial token, which models the Round-Robin scheduling mechanism.

Assume now that on the input edge of actor GW_{str_0} a complete data packet is available. The worst case situation occurs when this packet arrives right after actor GW_{str_0} finishes its execution. This means that all remaining actors have to fire first before actor GW_{str_0} is able to fire again. Or, when a complete packet is available in the worst-case situation *all* actors have to fire (including actor GW_{str_0}) before this packet is processed. We could draw the same graph for actor GW_{str_1} where the edges on GW_{str_0} are removed. Again we can conclude that in the worst-case situation all actors have to fire before a complete packet is processed.

This knowledge allows us to draw a new graph as depicted in Figure 6.19, where an expression for $\hat{\rho}_{GW}$ is given by Equation 6.13.

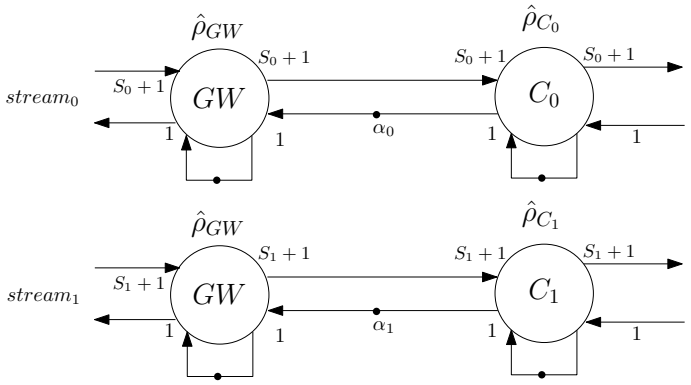

Figure 6.19: SDF model multiple data streams

$$\hat{\rho}_{GW} = \sum_{i=0}^{n-1} \hat{\rho}_{GW_{str_i}} + \sum_{i=0}^{n-1} \hat{\rho}_{P_{str_i}} \tag{6.13}$$

n = number of data streams

Concluding, in this chapter the temporal analysis model of our sharing mechanism is presented. Based on a detailed HSDF graph, a parametric solution for the total processing time for a data packet is given. When multiple streams are being processed by the same gateway we showed that in the worst case situation *all* actors have to fire before a complete packet is being processed.

Chapter 7

Evaluation

7.1 Hardware Costs

We start this evaluation with an overview of the hardware (area) costs of the gateway component and the other components which were modified during this project.

The Xilinx ML605 evaluation board is used as hardware platform for the Starburst MPSoC. The most important hardware resources of this board are: slice registers, LUTs, LUTRAM, Block Ram (BRAM) and Digital Signal Processing Element (DSP48E1). Each slice in the FPGA contains eight registers and four LUTs. A LUT is in fact a logical function generator with six inputs and two outputs. A LUT can also be used as a 256 bit RAM and is then called a LUTRAM. Additionally, the FPGA is equipped with dedicated RAM called BRAM. A BRAM can store up to 36 KB of data with a much smaller area footprint than a equivalent implemented in LUTRAM. Finally, a DSP48E1 is a dedicated DSP component, used for multiplications and additions. Designed to operate on high frequencies with a area footprint much smaller than the equivalent LUT implementation.

As a reference the hardware costs of the larger components of Starburst platform are presented in Table 7.1: a Microblaze processor, a Linux processor (equipped with a MMU) and a Multi-Port Memory-Controller (MPMC). As the Linux processor is equipped with a MMU, this component is larger than normal Microblaze processors.

	Starburst MicroBlaze	Linux MicroBlaze	MPMC
Slice reg	3123	3417	4759
LUT	3929	4476	3260
LUTRAM	316	236	188
BRAM or FIFO	18	19	17
DSP48E1	6	6	0

Table 7.1: Hardware usage of reference components

Table 7.2 shows the hardware usage of a router and a NI. These components have not been modified during this project and are only added as a reference. Clearly visible is the small size of the components. They both have a size of

about 2% of a MicroBlaze CPU. The largest part of the NI are located in the FIFO buffer inside this component which is implemented in LUTRAM.

	Router	NI
Slice reg	83 (2,7%)	6 (< 0,1%)
LUT	96 (2,4%)	75 (1,9%)
LUTRAM	0	56 (17,7%)

Table 7.2: Hardware usage Networking Components. The percentages show the relative size compared to a Starburst MicroBlaze

The first hardware component added to the gateway is the Accelerator Interface as discussed in Section 5.1.1, which allows a gateway to read data from and write data to an accelerator over its local PLB. As this component can connect up to 7 accelerators the hardware usage for the minimal and maximal occupation has been measured, and displayed in Table 7.3. The synthesis tooling is clearly able to optimize the design by removing unused logic.

	1 accelerator	7 accelerators
Slice reg	152 (4,9%)	152 (4,9%)
LUT	53 (1,3%)	345 (8,7%)

Table 7.3: Hardware usage Accelerator Interface. The percentages show the relative size compared to a Starburst MicroBlaze

The hardware accelerators are connected to the Accelerator Interface via the ringshell as explained in Section 5.1.1. As the ringshell had to be modified to make this connection possible, the hardware usage of this component is measured and presented in Table 7.4. Even though the ringshells for a Microblaze and hardware accelerator are based on the same VHDL-code, a different hardware usage is measured in both the old and the new situation as the synthesis tooling is able to optimize the designs by removing unused logic which depends on the connected component.

	MicroBlaze		Accelerator		
	Old	New	Old	New (FIFO depth = 1)	New (FIFO depth = 4)
Slice reg	203 (6,5%)	237 (7,6%)	234 (7,5%)	310 (9,9%)	330 (10,6%)
LUT	133 (3,3%)	177 (4,5%)	199 (5,0%)	230 (5,8%)	268 (6,8%)
LUTRAM	0	0	0	0 (0%)	32 (10,1%)

Table 7.4: Hardware usage Ringshell. The percentages show the relative size compared to a Starburst MicroBlaze

When we only focus on the old situation, the ringshell connected to a MicroBlaze

is smaller than a ringshell connected to an accelerator. This difference can be explained by the fact that a MicroBlaze-ringshell has less functionality; it doesn't need to return credits and doesn't have a configuration interface as this interface is only used to configure a hardware accelerator over the Nebula ring interconnect.

In the new situation we have to note that the ringshell was not only modified to let it connect to the Accelerator Interface. During the first weeks of this research the ringshell was also modified to support multiple credits when communicating with a hardware accelerator. The old situation only supported a single credit. It required the implementation of a programmable credit counter at the producing side and a hardware FIFO-buffer at the receiving side. This explains the increased size of a MicroBlaze-ringshell, as it is now equipped with a programmable credit-counter. Compared to the old situation the number of slice registers increased with 16%, the number of LUTs with 33%.

This programmable credit-counter is also present in the ringshell of a hardware accelerator, as this counter is required to chain multiple hardware accelerators together. Next to this counter the ringshell is also equipped with a hardware FIFO buffer. For this reason two measurements have been performed; one with a FIFO depth of 1 word (comparable to the old situation) and a measurement with a FIFO depth of 4 words.

The second addition to the gateway module is the DMA-controller, which is explained in Section 5.1.2. We first based our design on IP in the form of the Xilinx DMA controller, eventually a custom implementation was written. Table 7.5 presents the hardware usage of both custom implementation and the Xilinx implementation. The difference between the two components can be explained with the fact that our custom implementation has less functionality. The Xilinx DMA controller is able to move data aligned to the byte, whereas our custom DMA controller requires the addresses to be word aligned. Furthermore, the Xilinx DMA controller has support for incremental addressing and keyhole addressing, whereas our implementation always uses keyhole addressing. The Xilinx DMA controller uses LUTRAM to store blocks of data, the custom implementation stores this in a BRAM.

	custom DMA controller	Xilinx DMA controller
Slice reg	436 (13,9%)	538 (17,2%)
LUT	371 (9,4%)	593 (15,1%)
LUTRAM	0	18 (5,7%)
BRAM or FIFO	1 (16,6%)	0

Table 7.5: Hardware usage DMA-controller(s). The percentages show the relative size compared to a Starburst MicroBlaze

The last component which is required to implement hardware accelerator sharing is the Exit Gateway which translates credit-based data to address-based data, as

explained in Section 5.1.3. The hardware usage of this component is presented in Table 7.6.

	Exit Gateway
Slice reg	229 (7,3%)
LUT	145 (3,6%)

Table 7.6: Hardware usage Exit Gateway. The percentages show the relative size compared to a Starburst MicroBlaze

Concluding, this means that, apart from the networking components like the NI, router and ringshell, a dedicated MicroBlaze, DMA-controller, Accelerator Interface and Exit Gateway is required to implement hardware accelerator sharing. The total combined (minimal) hardware usage of these components is presented in Table 7.7. The numbers show that there currently is a large overhead in terms of hardware usage. There are some ideas to reduce these hardware costs, which are discussed in Section 8.2.

	Total HW usage
Slice reg	3940 (126,2%)
LUT	4498 (114,5%)
LUTRAM	316 (100%)
BRAM or FIFO	19 (105,6%)
DSP48E1	6 (100%)

Table 7.7: Combined hardware usage of all required components

7.2 Case study: PAL audio decoding

In the final section of this evaluation chapter the accelerator sharing mechanism will be evaluated using a realistic application. During a recent study a PAL video application has been developed for the Starburst MPSoC. In this demo a PAL signal is decoded and displayed on a VGA screen, where the analogue signal is completely processed in the digital domain on the processing tiles. Later hardware accelerators have been added to the platform to speed up specific parts of the application. This case study will focus on the audio part of the PAL standard, as this part is not yet implemented on the Starburst platform. As the standard supports stereo audio this case study will use the hardware accelerator sharing mechanism in order to decode the two audio channels (the left and right audio channel) with the same set of hardware accelerators.

Figure 7.1 gives an overview of the PAL standard. The spectrum consists of a luminance carrier which creates basically a black and white image. Years later color was added to the standard, in the form of a chrominance signal which is modulated over the chrominance carrier at $4,43MHz$. The PAL standard supports two FM-modulated audio carriers; the first carrier positioned at $5,5MHz$ and a second carrier at $5,5MHz + 242KHz$ [18]. The two streams can either be used to embed two different mono audio channels or to embed one single stereo audio channel. In the latter situation the first carrier contains a mix of the left and right channels ($\frac{L+R}{2}$, for backwards compatibility purposes) and the second carrier only the right stream.

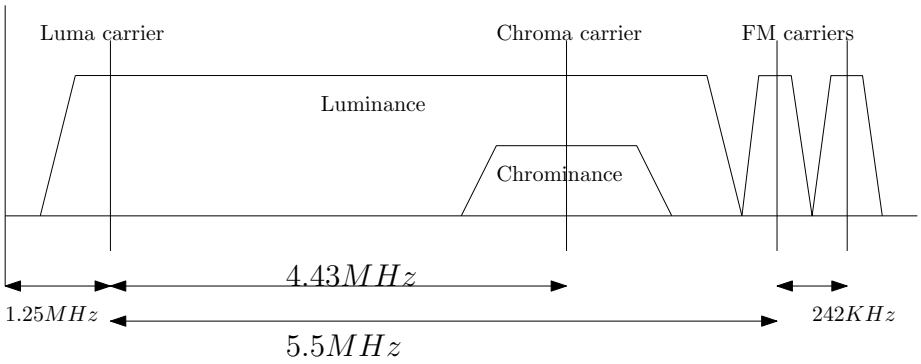

Figure 7.1: PAL spectrum

Figure 7.2 shows the set of operations which have to be applied on the incoming data stream y_{in}. The first operation is mixing the left and right audio stream, resulting in a complex-valued data stream at baseband. The next step involves

a low-pass filter and a down-sample operation, to obtain a data stream with a lower bit-rate. In the next step the FM signal is demodulated which results in a real-valued data stream. After this operation another low-pass filter and a down-sample operation is applied. As discussed the first stream contains both the left and right audio track $\frac{L+R}{2}$. This means that subtracting $\frac{R}{2}$ leaves $\frac{L}{2}$.

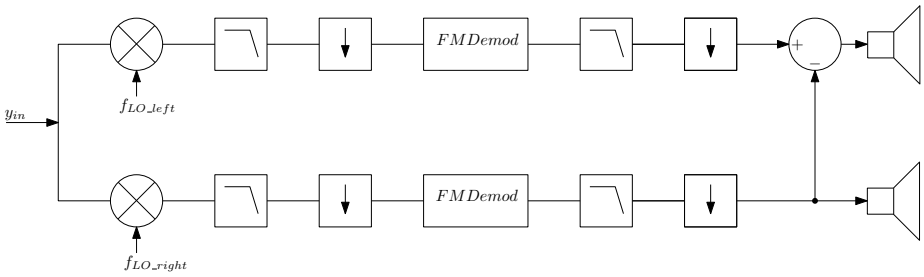

Figure 7.2: FM decoding pipeline

An interesting observation is the fact that the two audio channels require the same processing steps. That means that if these operations are implemented in hardware, the two data streams could possible be mapped on the same set of hardware accelerators. Additionally, one data stream requires two low-pass filter and down-sample operations and two vector operations (a signal mixer and demodulator). This makes the PAL audio decoding application an ideal candidate to demonstrate hardware accelerator sharing, as it turned out that it is possible to map the complete application on two hardware accelerators.

In the next sections these two hardware accelerators will be discussed.

7.2.1 CORDIC

Both the signal mixer and the FM decoder are vector operations. One way to perform vector operations is using the Coordinate Rotation Digital Computer (CORDIC) algorithm, which was first published in 1959. The algorithm allows efficient vector operations, and can efficiently be implemented in both hardware and software. The CORDIC algorithm will now briefly be explained.*

Consider the iterative rotation of vector $(x^i, y^i)^T$ by an angle a_i to obtain $(x^{(i+1)}, y^{(i+1)})^T$:

$$\begin{bmatrix} x^{(i+1)} \\ y^{(i+1)} \end{bmatrix} = \begin{bmatrix} \cos a_i & -\sin a_i \\ \sin a_i & \cos a_i \end{bmatrix} \begin{bmatrix} x^i \\ y^i \end{bmatrix}$$

*Source: Implementation of Digital Signal Processing, course by S.H. Gerez, Twente University

This can be rewritten to:

$$\begin{bmatrix} x^{(i+1)} \\ y^{(i+1)} \end{bmatrix} = \cos a_i \begin{bmatrix} 1 & -\tan a_i \\ \tan a_i & 1 \end{bmatrix} \begin{bmatrix} x^i \\ y^i \end{bmatrix}$$

Suppose now that $\tan a_i$ is chosen such that $\tan a_i = d_i 2^{-i}$, with $d_i = \pm 1$, as this allows the rotation to be calculated without multiplications as a multiplication with 2^{-i} can be implemented with only shift operations. The only exception is the multiplication with the first factor $\cos a_i = \frac{1}{\sqrt{1+2^{-i}}}$. This factor is called the K factor, and can be calculated at design time via:

$$K = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1+2^{-i}}}$$

The principle behind the CORDIC algorithm is depicted in Figure 7.3. Suppose that vector v_0 is rotated by angle α. What the CORDIC algorithm will do is rotating vector v_0 in multiple iterations (3 iterations in this specific example) to end up with vector v_3. By storing the corresponding angles $\arctan 2^{-i}$ $(45.0°, 26.6°, 14.0°$, etc.) in a Look-Up Table (LUT), the CORDIC algorithm can keep track of the total rotation via an *angle accumulator* z^i.

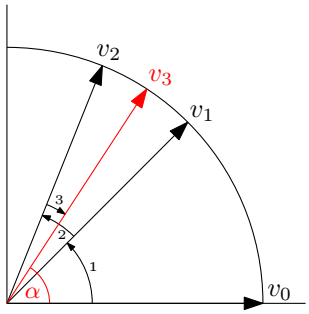

Figure 7.3: CORDIC algorithm illustration

The first step is to initialize the angle accumulator z^0 with this angle α. Based on the value in the angle accumulator z^i, the values d^{i+1} are chosen such that the angle accumulator will converge to 0. At the same time, the values d^i are used to perform the actual rotation, which can be implemented with just shift operations (apart from the multiplication with the K factor):

$$\begin{bmatrix} x^{(i+1)} \\ y^{(i+1)} \end{bmatrix} = K_i \begin{bmatrix} 1 & -d_i 2^{-i} \\ d_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x^i \\ y^i \end{bmatrix}$$

The mode of operation in this example is called *rotation mode*, where a vector can be rotated by an arbitrary angle α. Another mode of operation is called

vectoring mode. In vectoring mode the angle of a vector can by calculated. After initializing the angle accumulator z^0 at zero, the algorithm tries to rotate the vector towards the x-axis by converging y^i towards zero. Meanwhile the angle accumulator stores all subsequent rotations. When the algorithm has finished, the angle accumulator z^n stores the angle of the original vector.

The idea is to implement one CORDIC module, which can be used in both rotation and vectoring mode. In rotation mode, the cordic module can be used as a signal mixer. By equipping the module with an *angle register* and an *angle increment register*, the module will automatically increment the angle register when a new data sample has been received. Based on the value in the angle increment register, the mixing frequency can be programmed. In vectoring mode, the module can be used as an FM demodulator. FM demodulation equals determining the angle between subsequent input vectors. By placing the CORDIC module in vectoring mode, the angle of a vector can be calculated. By storing the last calculated angle, the angle between subsequent samples can be determined.

Instead of developing this module from scratch, IP in the form of the Xilinx CORDIC LogiCORE was used [36]. This decision was made to reduce development and testing time. A problem of this approach is the fact that the Xilinx CORDIC module can only be implemented in either rotation mode or vectoring mode. A combination of those two modes was not possible. First two separate hardware accelerators have been developed, one accelerator with a CORDIC in rotation mode, and one accelerator with a CORDIC in vectoring mode. Later, the two IP cores were embedded in one hardware accelerator, in order to demonstrate that one CORDIC module could be used to implement the complete PAL audio decoding application. Designing a custom CORDIC module with support for both modes remains future work. The hardware usage of the three hardware accelerators are presented in Table 7.8.

	CORDIC (Mixer)	CORDIC (FM demodulator)	CORDIC (Combined)
Slice reg	1714 (54,9%)	1008 (32,2%)	2658 (85,1%)
LUT	1882 (47,9%)	1053 (26,8%)	2906 (73,9%)
LUTRAM	162 (51,2%)	2 (0,6%)	164 (51,8%)
BRAM or FIFO	2 (11,1%)	0	0

Table 7.8: Hardware usage CORDIC. The percentages show the relative size compared to a Starburst MicroBlaze

After adding the CORDIC accelerator to the Starburst platform, the module can be programmed according to the register map as displayed in Figure 7.4. The module can be programmed as a signal mixer by writing a 0 to the mode bit in the control register (address 0x10). A 1 will program it as a FM decoder.

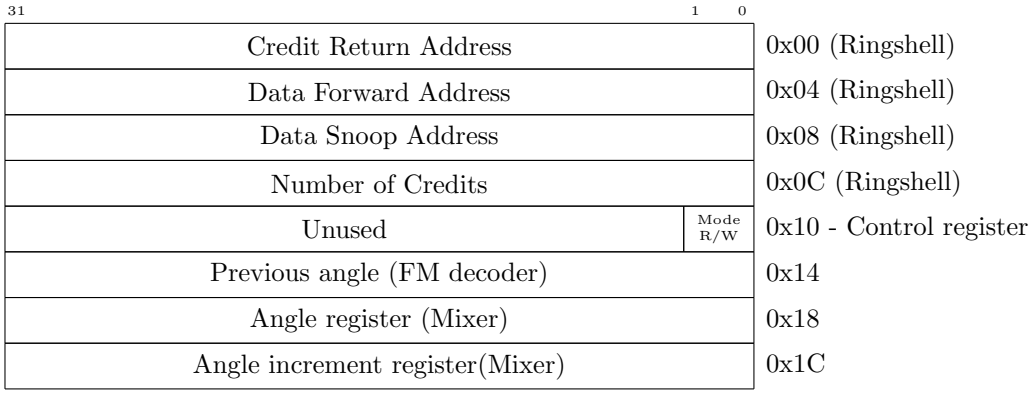

Figure 7.4: CORDIC address map

7.2.2 FIR Filter

As depicted in Figure 7.2 the decoding pipeline requires multiple low-pass filters. After the first operation (the signal mixer) the (complex valued) data signal is at baseband. The low-pass filter is then used to remove all signals not belonging to our data signal. Additionally the bit-rate is reduced via a down-sample operation. A second low-pass filter (and down-sample operation) is implemented after the FM decoder.

In the digital domain signal filters can typically be considered in two categories: Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters. The output of a FIR filter is computed as a weighted, finite term sum of past and present filter inputs. (See Equation 7.1) A FIR filter has no feedback, i.e. the output is independent of previous output values, it only depends on previous input values. This makes the impulse response of the filter finite, hence the name Finite Impulse Response. This in contrast to IIR filters, where feedback is present. (See Equation 7.2) Due to this feedback the impulse response of a IIR filter is infinite.

$$y[n] = \sum_{k=0}^{M} b_k x[n-k] \tag{7.1}$$

$$y[n] = \sum_{l=1}^{N} a_l y[n-l] + \sum_{k=0}^{M} b_k x[n-k] \tag{7.2}$$

Advantages of a FIR filter over an IIR filter is the fact that they are in general easier to design to meet certain specifications, can easily be designed to have a linear phase response and are (due to the missing feedback) always stable. The main disadvantage is the fact that in general a FIR filter requires more

computational power compared to IIR. The decision was made to use a FIR filter to implement the low-pass filtering operations of the PAL audio decoder, where the advantages of a FIR filter outweighed its disadvantages, i.e. we prefer an easier to design, always stable and a linear phase response filter over a higher required computational power. Again, the idea is to implement a single FIR filter in the form of a hardware accelerator, and share this accelerator across multiple streams.

A FIR filter has two important properties; it has certain scaling factors or coefficients (b_k in Equation 7.1) and delay elements which store delayed input values ($x[n-k]$ in Equation 7.1). This means that when a FIR filter is shared across multiple data streams, it should be possible to load a different set of coefficients. Additionally, it should have the functionality to extract and load the contents of the delay elements which is the state of the filter. Implementing the FIR filter via Xilinx IP turned out to be impossible, as there is simply no way to extract the contents of the delay elements. This forced us to build a filter ourselves, with the discussed functionality implemented.

FIR filters can be implemented according to different structures. Discussing these structures separately is far beyond the scope of this thesis, instead we will only focus on one type of structures, namely Direct-Form structures. Direct-Form means that the transfer function (See Equation 7.1) is directly evaluated. Figure 7.5 shows the so-called Direct Form I structure. In this structure the delay elements are placed before the multipliers. A disadvantage of this approach is the fact that large combinatorial paths are created, i.e. starting from the input of the filter ($x[n]$) the signal has to travel (in the worst-case situation) across the first multiplier (with coefficient b_0) and *all* adders to eventually reach the output of the filter ($y[n]$). A large combinatorial path means that a signal has to travel a longer distance, which in general means that lower clock-speeds are achievable.

$$y[n] = b_0x[n] + b_1x[n-1] + b_2x[n-2] + b_3x[n-3]$$

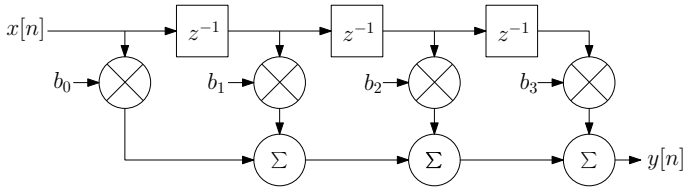

Figure 7.5: FIR filter structure (Direct Form I, $N = 3$)

An improved structure is the Direct Form II structure, which is depicted in Figure 7.6. This structure is also called a *transposed* form filter. Compared

to the former structure, the delay elements are placed after each multiplier and adder. This structure will lead to exactly the same output, however the combinatorial path is reduced to only one multiplier and one adder for all taps. This means that higher clock-speeds are achievable, or at least the synthesis tooling has a less difficult job mapping the hardware. For this reason the FIR filter is implemented according to this structure.

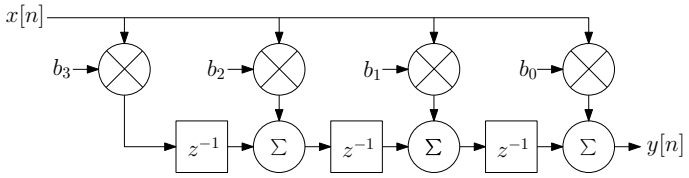

Figure 7.6: FIR filter structure (Direct Form II (Transposed form), $N = 3$)

The FIR filter is designed to support complex-valued data signals. Data samples are 32 bit wide, and contain a 16 bit I (In-phase) and a 16 bit Q (Quadrature) field. Both the I and Q signals are fixed point signals with 15 fractional bits. For this reason the filter is equipped with two identical processing pipelines; one for the I signal and one for the Q signal, where the coefficients at both pipelines are identical. Additionally, the number of taps is configurable at design time which allows us to easily alter the filter size. The structure of this filter is depicted in Figure 7.7, where the two separate pipelines are clearly visible.

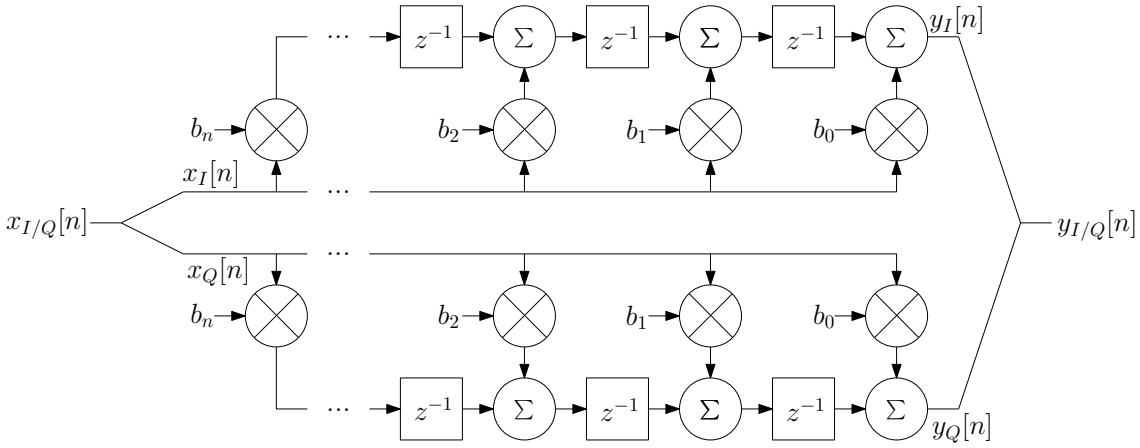

Figure 7.7: FIR filter structure with two identical pipelines for the I and Q signals

As depicted in Figure 7.2, after the low-pass filters down-sample operations are applied in order to reduce the bit-rates of the data streams. Instead of implementing a down-sampler in a separate hardware accelerator, this functionality is implemented in the same module as the FIR filter. Down-sampling with a

factor N means that only every N-th sample is passed through, all other samples are dropped. The down-sampler is therefore implemented as programmable counter which increments at every data sample coming out of the filter. After N samples a reset of the counter is applied and a single data sample is passed through.

After adding the FIR filter accelerator to the Starburst platform, the module can be programmed according to the register map as displayed in Figure 7.8. With a limited address-range for an accelerator filter coefficients are sequentially loaded via a single coefficient register (address $0x14$) starting with coefficient b_0 and ending with coefficient b_n. The same holds for the contents of the delay elements, which can be sequentially saved and loaded via the registers at addresses 0x18 and 0x1C. Registers 0x20 and 0x24 are used to program the down-sampler.

31 … 1 0	
Credit Return Address	0x00 (Ringshell)
Data Forward Address	0x04 (Ringshell)
Data Snoop Address	0x08 (Ringshell)
Number of Credits	0x0C (Ringshell)
Reserved for future usage	0x10 - Control register
Tap coefficients	0x14
Intermediate results (I samples)	0x18
Intermediate results (Q samples)	0x1C
Decimation factor	0x20
Decimation counter	0x24

Figure 7.8: FIR filter address map

Table 7.9 presents the hardware usage of the FIR filter. The first column shows the hardware usage for a 2×33 taps filter (33 taps for both the I and Q signal). Additionally the average hardware usage for a single tap is presented. From Table 7.9 we can conclude that the hardware usage of our 2×33 taps filter is substantial. Compared to a MicroBlaze CPU it has twice the number of slice registers, is nearly three times as large in the number of LUTs and uses five times as much LUTRAM. Additionally, for each multiplication and addition a dedicated DSP48E1 element is used*, where a MicroBlaze uses 6 of these elements.

*Both operations (a multiplication and addition) can be implemented via a single DSP48E1 element but require modifications to the VHDL source code. Basically the code should be written according to a specific coding template (See [31], page 168) which allows the synthesis tooling to place *both* operations on a single DSP48E1 element.

	FIR Filter (2×33 taps)	FIR Filter (Average tap size)
Slice reg	6512 (208,5%)	99 (3,1%)
LUT	10837 (275,8%)	164 (4,1%)
LUTRAM	1650 (522,2%)	25 (7,9%)
DSP48E1	132 (2200,0%)	2 (33,3%)

Table 7.9: Hardware usage FIR Filter. The percentages show the relative size compared to a Starburst MicroBlaze

We have to note that the FIR filter is implemented fully pipelined and able to process data samples each clock-cycle. This clearly comes at a cost where a lot of hardware is required to implement the FIR filter. We could reduce the degree of parallelism in order to save hardware at a cost of a lower throughput. However, a lower throughput means a longer processing time, and requires more memory to buffer the incoming data streams in the gateway.

7.2.3 Application

With the CORDIC and FIR modules implemented, we can now focus on the software application. At the base of this software application is a FM encoding and decoding application developed in GNU Radio. GNU Radio is a free and open-source Software Defined Radio (SDR) development toolkit that provides signal processing blocks which can be used to implement software radios, with or without the use of external RF hardware.

The big advantage of the GNU Radio application was the fact that it could be used as a reference design. Instead of equipping the Starburst platform with an ADC daughter-board to capture and process an actual PAL signal, we have used the GNU Radio application to synthesise an artificial PAL audio signal*. This signal could then be decoded on the Starburst platform and compared to the results of the GNU Radio application to check it for correctness.

The first step is dividing the decoding pipeline in separate partitions, as displayed in Figure 7.9. A partition is a set of operations which can concurrently be mapped on the set of shared accelerators which are managed by the gateway. Partition 1 and 2 consists of a signal mixer, a FIR filter and a down-sample operation. As presented in Figure 7.1 the spectrum has two separate audio signals. By programming the CORDIC module as a signal mixer and tuning the mixing frequencies ($f_{LO_{left}}$ and $f_{LO_{right}}$) the audio signals are mixed to baseband. The incoming data stream y_{in} has a sample-rate of $2.8MHz$, which is down-sampled after the FIR filter with a factor 8 to reduce it to $352.8KHz$.

*The synthesized test signal is not yet completely according to the PAL specification as presented in [18]. Specific parameters such as mixing frequency, impulse response of the FIR filters and down-sample rates have to be tuned to fit the specification.

The gateway will start with partition 1, where the results are streamed to the SPM of the gateway. This way the results can be processed further (partition 3) at a later point in time. After processing partition 1, the gateway will switch to partition 2. The same incoming stream y_{in} is processed with another mixing frequency, where the results are again stored in the SPM of the gateway to be able to process the data during partition 4.

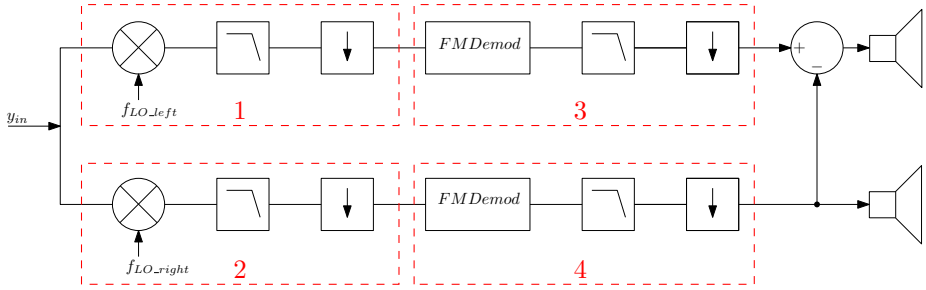

Figure 7.9: Partitioned FM decoding pipeline

Partition 3 and 4 consists of a FM demodulator, a FIR filter and a down-sample operation. For this reason the CORDIC module is programmed as an FM demodulator, a new impulse response is loaded into the FIR filter and the sample-rate is again reduced with a factor 8, resulting in a sample-rate of $44.1KHz$. The results of both partitions are streamed to a MicroBlaze processor which is applying the subtraction as displayed in Figure 7.9. After the left and right audio streams are extracted, the result can be streamed to a play-back device such as an USB sound-card or to the connected host computer.

7.2.4 Results

The PAL audio decoder application is constantly switching between the four partitions presented in Figure 7.9. As discussed in Chapter 6 the total processing time for these four partitions depends on the actual streaming of data and the reconfiguring time. First we have measured the (constant) reconfiguring time. Measurements showed that programming the FIR filter and cordic module takes 25 microseconds in total (where 110 configuration and state registers are written). Extracting the state out of these accelerators takes 16 microseconds (where 68 state registers are read). In both situations this means that reading and writing a configuration or state register takes on average 23 clock cycles.

The PAL audio decoder is applying two down-sample operations, both with a factor 8. This means that for every $8 \times 8 = 64$ input samples 1 sample is produced at the output of the processing pipeline. As we have to count the number of processed data samples in the Exit Gateway (See Section 5.1.3), we are forced to use data packets with a multiple of 64 samples, as this results

in the production of an integer number of data samples at the output of the processing pipeline.

The goal of this application is to reach a minimal throughput of $44.1KB/s$ at the output of the processing pipeline, in order to get a continuous (real-time) audio stream. This throughput has been measured by starting a timer when the first sample leaves the processing pipeline, after a specific number of samples have been counted the timer is stopped which allows us to calculate the achieved throughput. For the minimal packet size of 64 samples a throughput of $5.51KB/s$ was achieved. For this reason the packet size was increased as a larger packet reduces the reconfiguring overhead which increases the achieved throughput. For different packet sizes throughput measurements have been performed, which are presented in Figure 7.10.

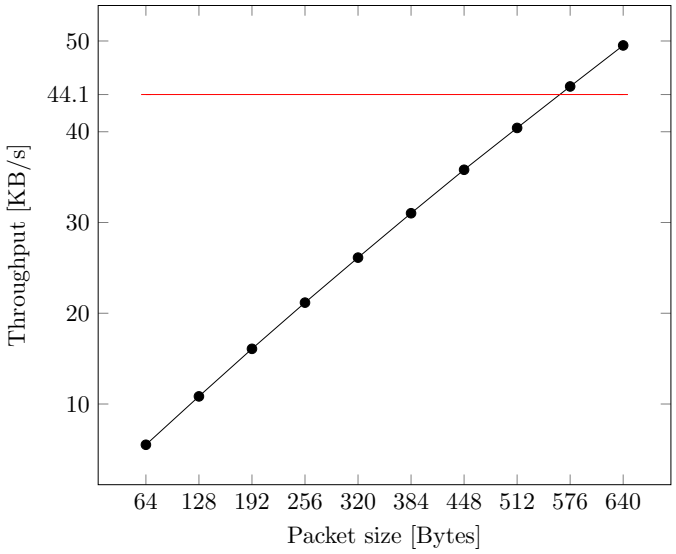

Figure 7.10: PAL audio decoder performance measurements. The red line at $44.1KB/s$ depicts the minimal throughput which results in a continuous (real-time) audio stream.

From Figure 5.3 we can conclude that minimal packet size of 576 samples is required. With a total down-sampling factor of 64 this results in the production of 9 data samples at the output. Interesting is to note the total processing times for the four partitions with this specific packet size, which are presented in Table 7.10. What we can conclude is that the four partitions are serviced in 194 microseconds, where in total 9 audio samples are produced. This means that the actual streaming of data takes $\frac{12+12+3+3}{194} = 15\%$ of the total processing time, during the remaining $\frac{4*(25+16)}{194} = 85\%(!)$ the gateway is reconfiguring the accelerators.

Action	**Execution time** (in microseconds)
Load state 1	25
Stream 576 samples	12
Save state 1	16
Load state 2	25
Stream 576 samples	12
Save state 2	16
Load state 3	25
Stream 72 samples	3
Save state 3	16
Load state 4	25
Stream 72 samples	3
Save state 4	16
Total	**194**

Table 7.10: PAL audio decoder processing times

Chapter 8

Conclusion and Future Work

In this thesis a hardware accelerator sharing mechanism was added to the Starburst platform. In this chapter the conclusions are summarized and future work is presented.

8.1 Conclusion

The implemented accelerator sharing mechanism is based on a centralized component called a gateway. This centralized component will buffer multiple incoming data streams and will sequentially push them through the shared accelerators. The gateway is currently implemented on a dedicated MicroBlaze processor equipped with several hardware extensions.

An important property of the sharing mechanism is the fact that most accelerators have a certain configuration and/or state for a specific data stream. When multiple streams are being processed by a hardware accelerator multiple configurations and/or states needs to be managed. Basically a context switch need to take place where the current state is saved and a the configuration and state for the new data stream is restored. This bring us to the first research question:

1. How to handle multiple configurations and states for a hardware accelerator?

The MicroBlaze CPUs and hardware accelerators are connected to the Nebula ring network. This network has, as discussed in Section 3.2.4, only support for write operations. This property turned out to be problematic for the sharing mechanism as it is required to *read* the state out of an accelerator before a new state could be loaded. For this reason an Accelerator Interface was added to the gateway (see Section 5.1.1) which allows the gateway to read and write the contents in the hardware accelerator over its local data-bus. This way the write-only Nebula ring network could be bypassed. The gateway manages the configuration and state for multiple data streams in software. (see Section 5.2)

Managing the state and configuration on a CPU in software turned out to be the most flexible and easiest to prototype solution. This solution is flexible as we are (under the assumption that enough memory is available to store configuration and state) not limited to a maximum number of data-streams. The downside of this approach is the fact that this is not the best solution in terms of performance.

In a previous master's thesis support for hardware accelerators was added to the Starburst platform. Afterwards a case study was carried out, which showed that the utilization of their hardware accelerator was around 6% when a MicroBlaze CPU. For this reason the following research question was formulated:

2. Is it possible to increase the currently low utilization of the hardware accelerators via hardware accelerator sharing?

In Section 5.1.2 we explained that a MicroBlaze CPU is unable to feed an accelerator with data fast enough. It turned out that the local data-bus of a MicroBlaze CPU is the limiting factor; communicating one data word takes (at least) eight clock cycles. When a hardware accelerator is implemented fully pipelined (and thus able to process a data sample each clock-cycle) its utilization will never exceed 12.5% when a MicroBlaze CPU is feeding it with data.

The gateway is implemented on a MicroBlaze CPU which means that its performance was also limited by its local data bus. For this reason the gateway is equipped with a DMA controller. A DMA controller is a hardware component used to copy a programmable amount of data from one location to another location in the system without processor intervention. Additionally, a DMA controller is able to burst blocks of data, drastically reducing the communication overhead.

As a case study a PAL audio decoder was implemented on the Starburst platform, where the left and right audio channel are processed on the same set of hardware accelerators, doubling its utilization. Therefore we can conclude that hardware accelerator sharing is a suitable method to increase the utilization of a hardware accelerator. We have to note however that the context switching overhead in this case study is substantial; around 85% of the time the accelerators are being (re-)programmed, the actual streaming of data takes around 15% of the total time. These numbers bring us to the next research question:

3. Which techniques can reduce the overhead introduced by switching between data streams?

An answer to this question is not given in this thesis. The case study showed that the hardware accelerator mechanism was working correctly, even with the substantial context switching overhead. Time limitation prevented us to focus on reducing this overhead. There are some preliminary ideas to reduce this context switching overhead, but the details are not yet worked out. The ideas are discussed as future work in Section 8.2.

4. What additional hardware is required to share hardware accelerators, and can this additional hardware be justified by the amount of hardware saved by sharing hardware accelerators?

Section 7.1 focussed on the hardware usage of the implemented hardware components. The sharing mechanism uses a dedicated MicroBlaze CPU, extended with an Accelerator Interface and a DMA controller. Measurements showed

that the Accelerator Interface has a size of roughly 5% of a MicroBlaze CPU, the DMA controller has a size of roughly 12%.

Answering the quantitative question if this additional hardware can be justified by the amount of hardware saved by sharing hardware accelerators is difficult to answer, as it depends on the hardware accelerators which are being shared. For the hardware accelerators implemented during our case study we can conclude that it is better to share the accelerators than implementing multiple accelerators of the same type. The CORDIC modules have a size (depending on the mode of operation) of 30% or 50% of a MicroBlaze CPU. The implemented FIR filter has a size of multiple MicroBlazes.

The size of other commonly used hardware accelerators in the DSP domain (e.g. Viterbi, Reed-Solomon or FFT) is expected to be substantial. Sharing these large components to process multiple channels or multiple standard is preferred. For small hardware accelerators we have to conclude that it might be better to simply duplicate them, as the hardware used for the sharing mechanism introduces a large overhead.

Even when hardware costs is of no concern we think that the sharing mechanism is a valuable addition, as it drastically increases the flexibility of the Starburst platform. This platform is currently being developed on a FPGA allowing us to change the accelerator configuration to fit a specific application. When Starburst would be implemented in an Application-Specific Integrated Circuit (ASIC), the freedom to change the accelerator configuration is lost, which clearly shows the advantages of the sharing mechanism. With the sharing mechanism implemented we are able to run a wide variety of applications on the same (fixed) accelerator configuration.

An important requirement of the sharing mechanism was the fact that it had to be predictable. Where predictability is defined as the ability to construct a sufficiently accurate temporal analysis model of the hardware design for which a computational efficient analysis algorithm exists. With this model calculations can be performed and useful numbers can be extracted. One can think of numbers like minimum throughput or maximum latency when processing a data stream over a shared hardware accelerator. For this reason the following research question was formulated:

5. Can a (detailed) dataflow model of this sharing protocol be constructed, and related to that, can an abstraction of this dataflow model be created, which is easier to use at the cost of a less accurate dataflow model?

In Chapter 6 we have showed that we are able to create a dataflow model of the sharing mechanism. The gateway module implements a Round-Robin scheduler, which is sequentially processing a predefined amount of data (a data packet) from each data stream. A detailed dataflow model is used to calculate the total processing time for a single data packet from a specific data stream. Next, an abstraction of this dataflow model is created which can then be used to

calculate the worst-case servicing time under the Round-Robin scheduler, where the processing times for all other data packets are included.

6. Can the real-time behaviour be accurately predicted?

While we are able to capture the effects of the hardware accelerator sharing mechanism in a dataflow model, the question remains if this dataflow model accurately predicts the actual behaviour in a practical setting. Time limitations prevented us to fully answer this question, there is for example no dataflow model of the PAL audio decoding application which could help us answering this question.

In [2] accuracy is defined as the difference between the average and a reliable estimate of the worst-case situation. An advantage of using hardware accelerators is the fact that their temporal behaviour can be accurately determined. In contrast to the MicroBlaze CPUs they are not equipped with caches, not connected to shared DDR3 memory or running a task scheduler. This allows us to determine parameters (in terms of clock-cycles) such as latency and throughput. This in contrast to e.g. the MicroBlaze on which the gateway application is executing.

The gateway itself is implemented on a MicroBlaze CPU where the previously discussed disadvantages apply. As a simple test we have disabled both the instruction and data cache, which resulted in an average throughput of $3.62 KS/s$, where we would normally achieve the required $44.1 KS/s$. This clearly shows why caches in a real-time context are problematic; taking worst-case situations in account (disabling the caches) results in a performance degradation with a factor of 12.

The Nebula ring network is another interesting component which has an influence on the accuracy. The gateway is equipped with a DMA-controller, in order to speed up the transfer of data. While we are able to guarantee a minimum throughput, a work-conserving principle was added to the Nebula ring network, where we are able to hijack unused slots on the ring. Even if the partitioning of tasks (and thus the data streams on the Nebula ring) is known at design-time, we are unable to exactly predict slots which may be hijacked, which influences the accuracy.

7. Can we find an algorithm to calculate proper buffer sizes, and related to that, can we compute at which granularity the switches between data-streams should take place?

This research question remains unanswered in this thesis and can therefore be seen as future work. We have a rough idea of the steps we need to take to calculate the buffer sizes, but this has not yet resulted in a complete algorithm. Appendix A.3 shows an example where most steps are explained.

8.2 Future Work

During this research a number of research question have been answered related to the sharing hardware accelerators. However, there are still many improvements we can think of. A list of ideas will now be presented, where some of these improvements are discussed.

- The sharing mechanism has, in its current state, a large context switching overhead. In Section 7.2.4 we showed that it takes on average 23 clock cycles to read or write a word from and to a hardware accelerator. With a reduced context switching overhead higher throughputs can be achieved. Different solutions are possible, including but not limited to:
  - Using a DMA controller to speed up the transfer of configuration data from and to the accelerators. As the gateway module is already equipped with a DMA controller, we could reuse this component for this purpose limiting the additional required hardware.
  - Equipping the hardware accelerators with additional hardware to store multiple configurations and states. This way the gateway will just inform the accelerators to switch to a specific configuration. In theory we could reduce the context switching time to several clock cycles, drastically reducing the context switching overhead. On the other hand, via this construction only a limited number of configurations is supported.
- In the current situation, the hardware accelerators are directly connected to the Nebula ring interconnect. While this is a rather simple solution, there are some drawbacks. When data is processed via our sharing mechanism, we want the data to be processed as fast as possible with the lowest amount of latency. This way data packets (and buffer sizes) can be kept as small as possible.

  The Nebula ring interconnect does not really fit these requirements; only a part of the total throughput can be guaranteed (GS traffic), the rest of the traffic has to be claimed by hijacking other slots (BE traffic). The amount of slots which may be hijacked depends on the partitioning of tasks on the Starburst platform. When two processors are communicating data, where the samples are passing a set of hardware accelerators, only a specific amount of slots my be hijacked by these accelerators resulting in larger packet sizes and larger buffer sizes. Furthermore, when multiple accelerators are chained together, a lot of intermediate results are communicated over the Nebula ring interconnect.

  One idea to solve this problem is by introducing an additional interconnect, where the hardware accelerators are connected to. The advantage of this approach is the fact that the two types of traffic (CFIFO and credit-based) are moved to separate interconnects, greatly reducing the amount of

interference. This idea is depicted in Figure 8.1. In the image two separate ring networks are visible, where the gateway is the connecting link. While just one additional network is depicted, we could create multiple smaller ring networks, creating clusters of accelerators.

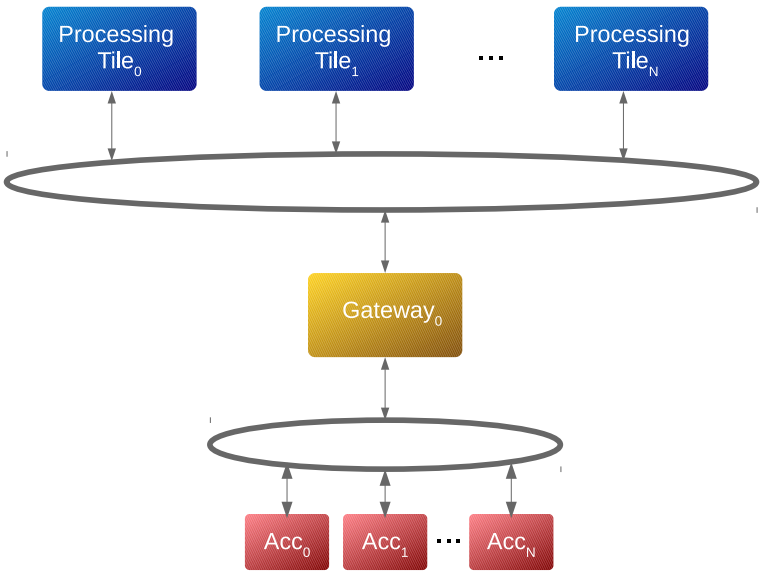

Figure 8.1: Separate Accelerator Network

For the processing tiles in the system nothing will change; currently data is streamed to the accelerators via the gateways, the same is true when an additional interconnect is added. When this separate interconnect will use the same ring-like structure no increase in hardware is expected as the same number of routers and network interfaces is required. On the other hand, support for credit-based flow-control can be completely removed from the Nebula ring interconnect, as all accelerators are no longer connected to this network. However, it is unknown if the ring-like structure is the best solution for this additional network. The Starburst platform is focussing on *streaming* applications, which are latency tolerant. Traffic between the two gateways is clearly not latency tolerant as an increased latency will result in a increased processing time and therefore a lower throughput.

- An algorithm to calculate proper buffer sizes is still missing. We have some rough ideas how to solve this problem, an example is given in Appendix A.3.
- The gateway is constructed around a regular MicroBlaze CPU. By default this CPU is equipped with several additional hardware components like a hardware multiplier, hardware divider, barrel shifter and FPU. As

the gateway is not using these kind of operations, they could simply be removed which results in a reduction of hardware area.

- The idea of placing the accelerators on an additional interconnect is already discussed. If this idea would be implemented, this opens the door to merging the two separate gateways (the processor and the Exit Gateway) in one single component, greatly simplifying the design.
- In the current implementation, the gateway processor will send an updated write pointer to the consumer, when all data samples have been processed. This functionality should be implemented in the Exit Gateway, as was discussed in Chapter 6.
- A lot of research is done into automatic partitioning and mapping of tasks on the Starburst platform. Integrating the hardware sharing mechanism into these tools (called Omphale) would increase the usability if the Starburst platform, as the programmer is currently required to manually program the gateway (scheduler) and the accelerators.

Appendix A

How to use Hardware Accelerator Sharing

A.1 Hardware Modifications

The Starburst platform is generated based on a XML-file. This section will focus on the modifications to this XML-file to implement hardware accelerator sharing.

The first modification is the enabling of the Accelerator Interface and the DMA-controller on the MicroBlaze where the gateway will be mapped on. Additionally we need to make sure that the SPM is connected to the PLB which makes sure that the DMA controller is able to reach this memory. For these reason we need to set the following parameters:

```
1 <parameter id="accinterface" type="boolean" value="true"/>
2 <parameter id="fifo_on_plb" type="boolean" value="true"/>
3 <parameter id="ringdma" type="boolean" value="true"/>
```

Next, we need to reserve a NI and router for the DMA-controller. This can be done by adding the following *external* accelerator:

```
1 <accelerator id="dma0" type="ringdma" parent="mb2" extern="true">
2   <parameter id="plb_addr" type="boolean" value="true"/>
3 </accelerator>
```

The last step is adding an Exit Gateway. This can be done by adding the following accelerator:

```
1 <accelerator id="c2a0" type="credit2addr" parent="mb2">
2  <parameter id="buffer" type="int" value="4"/>
3  <parameter id="plb_addr" type="boolean" value="true"/>
4 </accelerator>
```

A.2 Software Example: Accelerator Management

The following code shows how to manage the configuration and state for multiple accelerators, as explained in Section 5.2.

```
1
2  //Instantiate an AccList class, and add accelerator (
       sub)classes to the linked list.
3  AccList *foo = new AccList();
4  foo->setNext(new RingDma());  //Always first component
       in the linked list
5  foo->setNext(new CordicMix(mixing_frequency));
6  foo->setNext(new FirFilter(coefficients,
       num_coefficients, downsample_factor));
7  foo->setNext(new CreditToAddr());  //Exit gateway,
       always last component in the linked list
8  foo->setupStream();
9
10 //Instantiate a second AccList class, and add
       accelerator (sub)classes to the linked list.
11 AccList *bar = new AccList();
12 bar->setNext(new RingDma());    //Always first
       component in the linked list
13 bar->setNext(new CordicMix(mixing_frequency));
14 bar->setNext(new FirFilter(coefficients,
       num_coefficients, downsample_factor));
15 bar->setNext(new CreditToAddr()); //Exit gateway,
       always last component in the linked list
16 bar->setupStream();
17
18 while(true) {
19
20 {
21 //Stream 1: Check for data at the input and free space
       at the output. Configure DMA controller and C2A
       component
22 }
23
24 foo->writeState();
25
26 {
27 //Stream 1: Stream input data, poll exit gateway to
       see if all data has been processed and inform
       receiver all data has been received
28 }
```

```
29
30 foo->saveState();
31
32 {
33 //Stream 2: Check for data at the input and free space
       at the output. Configure DMA controller en C2A
      component
34 }
35
36 bar->writeState();
37
38 {
39 //Stream 2: Stream input data, wait until all data has
       been received and inform receiver all data has
      been received
40 }
41
42 bar->saveState();
43 }
```

A.3 Buffer Size Calculation

A (complete) algorithm to calculate proper data packet and buffer sizes is currently missing, and can be seen as future work. However we think it should be possible to calculate these sizes via (integer) linear programming which we will show via an example.

Suppose that we have two data streams α_0 and α_1 which are being processed by a gateway, with throughput constraints x_0 and x_1. The first stream is executed on a set of accelerators which is modelled by dataflow graph G_0, the second data stream is modelled via dataflow graph G_1. As the switching between data streams introduces an overhead (mainly caused by reconfiguring the accelerators), the gateway is processing packets of data of these data streams. What we are interested in is the minimal size of the data packets S_0 and S_1, in order to meet the given throughput constraints.

In Chapter 6 we showed that we are able to give a parametric solution for the time it takes to process a complete data packet, where the total processing time depends on some constant latency λ (e.g. the time it takes to fill the pipeline and to return an acknowledgement from the Exit Gateway to the first gateway) and on the packet size S. Which gives:

$$\hat{\rho}'(G, S) = \lambda(G) + (S - 1) \cdot \mu(G)$$

The reconfiguring time can be modelled via two actors P_0 and P_1. As the gateway is processing the streams in a cyclic order via a Round-Robin scheduler, this means that the total worst-case processing time for one scheduling iteration of this example equals:

$$\hat{\rho}_{total} = (\lambda(G_0) + (S_0 - 1) \cdot \mu(G_0)) + P_0 + (\lambda(G_1) + (S_1 - 1) \cdot \mu(G1)) + P_1$$

As during one iteration S_0 and S_1 tokens are produced respectively for streams α_0 and α_1, the following throughput is achieved for stream α_0:

$$\frac{S_0}{(\lambda(G_0) + (S_0 - 1) \cdot \mu(G_0)) + P_0 + (\lambda(G_1) + (S_1 - 1) \cdot \mu(G1)) + P_1}$$

And for stream α_1:

$$\frac{S_1}{(\lambda(G_0) + (S_0 - 1) \cdot \mu(G_0)) + P_0 + (\lambda(G_1) + (S_1 - 1) \cdot \mu(G1)) + P_1}$$

The equations above clearly state an optimization problem; increasing the packet size of one data stream to increase the achieved throughput negatively influences the achieved throughput for the other data stream. For this reason we define the following LP:

$$\begin{aligned}
&\text{minimize } S_0 + S_1 \\
&\text{subject to } \frac{S_0}{(\lambda(G_0) + (S_0 - 1) \cdot \mu(G_0)) + P_0 + (\lambda(G_1) + (S_1 - 1) \cdot \mu(G1)) + P_1} \geq x_0 \\
&\qquad \frac{S_1}{(\lambda(G_0) + (S_0 - 1) \cdot \mu(G_0)) + P_0 + (\lambda(G_1) + (S_1 - 1) \cdot \mu(G1)) + P_1} \geq x_1 \\
&\qquad S_0 \geq 0, S_1 \geq 0
\end{aligned}$$

It is important to note that the first two constraints can be rewritten into two simple linear equations, as the terms $\lambda(G_0), \mu(G_0), \lambda(G_1), \mu(G_1), x_0$ and x_1 are constants. Time limitations prevented us to fully implement and test the above LP, and can therefore be seen as future work.

A.4 FIR Filter Design

The impulse response for the FIR filter can be designed via the fdatool from Matlab. After designing the filter we need to quantize the coefficients via the option 'Set quantization parameters', where we can use the following parameters.

- Filter arithmetic: Fixed point
- Filter precision: Specify

- Numerator word length: 32 (Best-precision fraction lengths: uncheck)
- Numerator frac. length: 31

- Use unsigned representation: uncheck
- Scale the numerator coefficients to fully utilize dynamic range: uncheck

- Input word length: 16
- Input fraction length: 15

- Output word length: 16
- Output fraction length: 15

- Rounding mode: Nearest
- Overflow mode: Saturate

- Product word length: 32
- Product fraction length: 31

- Accum. word length: 32
- Accum. fraction length: 31

Via 'Targets' -> 'Generate C header ...' we can generate a C-header file, containing the filter coefficients, which can directly be loaded into the FIR filter.

Bibliography

[1] Netherlands streaming (nest) consortium. research summaries. http://www.nest-consortium.nl/projectSummary.php.

[2] M. J. G. Bekooij. Lecture Notes for the Real-Time Systems 2 Course, 2013.

[3] M. J. G. Bekooij *et al.* Dataflow analysis for real-time embedded multiprocessor system design. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, pages 81–108. Springer, 2005.

[4] M. J. G. Bekooij, S. Parmar and J. van Meerbergen. Performance Guarantees by Simulation of Process. In *Proc. Int'l Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2005.

[5] G. Bilsen *et al.* Cvclo-Static Dataflow. *IEEE Transactions on Signal Processing*, 44(2), 1996.

[6] G. Blake, R. G. Dreslinski and T. Mudge. A survey of multicore processors. *IEEE Processing Magazine*, pages 26–37, 2009.

[7] S. Bokar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54:67–77, 2011.

[8] S. Borgio *et al.* Hardware DWT accelerator for Multiprocessor System-on-Chip on FPGA. *ICSAMOS*, pages 107–114, 2006.

[9] D. Bouthaina *et al.* Shared hardware accelerator architectures for heterogeneous MPSoCs. In *Proc. of the Int'l Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, July 2013.

[10] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, 3th edition, 2011.

[11] S. Chakraborty, S. Künzli and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Design. In *Proc. of the Design,Automation and Test in Europe Conference and Exhibition (DATE)*, 2003.

[12] J. Cong *et al.* AXR-CMP: Architecture support in Accelerator-Rich CMPs, 2011.

[13] B. H. J. Dekens *et al.* Low-Cost Guaranteed-Throughput Communication Ring for Real-Time Streaming MPSoCs. In *Design and Architectures for Signal and Image Processing (DASIP)*, 2013.

[14] J. P. H. M. Hausmans *et al.* Dataflow Analysis for Multiprocessor Systems with Non-Starvation-Free Schedulers. *Proc. of the 16th Int'l Workshop on*

Software and Compilers for Embedded Systems (SCOPES), pages 13–22, 2013.

[15] R. Henia *et al.* System level performance analysis - the SymTA/S approach. *Computers and Digital Techniques*, 152(2), 2005.

[16] G. Hoekstra. Hardware Accelerator Integration in a Connectionless Network-on-Chip. Technical report, University of Twente, 2013.

[17] R. Hou *et al.* Efficient Data Streaming with On-chip Accelerators: Opportunities and Challenges. In *Proc. of the Int'l Symposium on High Performance Computer Architecture (HPCA)*, pages 312 – 320, 2011.

[18] ITU-R. Transmission of multisound in terrestrial television systems - BS.707-4, 2002.

[19] C. Jalier *et al.* Heterogeneous vs homogeneous MPSoC approaches for a Mobile LTE modem. In *Proceedings of the Design,Automation and Test in Europe Conference and Exhibition (DATE)*, 2010.

[20] E. A. Lee and D. G. Messerschmit. Synchronous Data Flow. In *Proc. of the IEEE*, pages 1235 – 1245, 1987.

[21] O. M. Moreira and M. J. Bekooij. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing*, 2007(1):083710, 2007.

[22] A. Nieuwland *et al.* C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, 2002.

[23] J. H. Rutgers, M. J. G. Bekooij and G. J. M. Smit. Evaluation of a Connectionless NoC for a Real-Time Distributed Shared Memory Many-Core System. In *Proc. of the Design Automation Conference (DAC)*, 2012.

[24] STMicroelectronics. ARM SC300 datasheet, 2013.

[25] L. Thiele and N. Stoimenov. Modular Performance Analysis of Cyclic Dataflow Graphs. In *Proc. of the ACM Int'l Conf. on Embedded Software (EMSOFT)*, pages 127–136, 2009.

[26] W. Tong *et al.* Hard-Real-Time Scheduling on a Weakly Programmable Multi-core Processor with Application to Multi-standard Channel Decoding. In *Proc. of the Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 151–160, April 2012.

[27] M. Wiggers *et al.* Efficient Computation of Buffer Capacities for Multi-Rate Real-Time Systems with Back-Pressure. 2006.

[28] M. H. Wiggers *et al.* Efficient Computation of Buffer Capacities for Cyclo-Static Real-Time Systems with Back-Pressure. In *Proc. of the Real-Time*

and Embedded Technology and Applications Symposium (RTAS), pages 281 – 292, 2007.

[29] M. H. Wiggers, M. J. G. Bekooij and G. J. M. Smit. Modelling run-time arbitration by latency-rate servers in dataflow graphs. In *Proc. of the 10th Int'l workshop on Software & compilers for embedded systems (SCOPES)*, 2007.

[30] W. Wolf, A. A. Jerraya and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008.

[31] Xilinx. Xilinx XST User Guide for Virtex-6 and Spartan-6 Devices, 2009.

[32] Xilinx. LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a), 2010.

[33] Xilinx. PLBv46 Slave Single (DS561 v1.01a). pages 1–29, 2010.

[34] Xilinx. Xilinx DS579 LogiCORE IP XPS Central DMA Controller (v2.03a), Data Sheet, 2010.

[35] Xilinx. DSP48E1 Slice (UG369 v1.3). 369:1–52, 2011.

[36] Xilinx. LogiCORE IP CORDIC (DS249 v4.0), 2011.

[37] Xilinx. MicroBlaze Processor Reference Guide (UG081, v13.4), 2012.