MASTER THESIS

STATE-SAVE OVERHEAD REDUCTION TECHNIQUES FOR SHARED ACCELERATORS IN AN MPSOC WITH A RING NOC

Oscar Starink

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

EXAMINATION COMMITTEE
Prof.dr.ir. Marco Bekooij
Dr.ir. Jan Broenink
Ir. Guus Kuiper

UNIVERSITY OF TWENTE.

2-10-2015

University of Twente

Master Thesis

State-Save Overhead Reduction Techniques for Shared Accelerators in an MPSoC with a Ring NoC

Author:
Oscar Starink

Student number:
S1378694

Committee:
Prof. dr. ir. Marco Bekooij
Dr. ir. Jan Broenink
Ir. Guus Kuiper

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

October 2, 2015

Abstract

In the last decade chip manufactures moved from single core designs to multi-core designs. This trend is a result of the increasing demand for performance, and the increasing availability of chip area. The same trend is visible in the embedded domain. As a result, System-on-Chips (SoCs) are becoming Multi-Processor System-on-Chips (MPSoCs). These multi-processor systems can be homogeneous or heterogeneous. In a homogeneous system all processors are identical, while a heterogeneous system contains multiple different processing elements. A processing element can perform a general or a specific task. A Central Processing Unit (CPU) is an example of a general purpose processing element, while a hardware accelerator is an application specific processing element. The MPSoCs also contain a Network-on-Chip (NoC) that is connecting all the processing elements within an MPSoC.

Researchers at the University of Twente have developed an MPSoC, that targets real-time streaming applications. It started as a homogeneous MPSoC with an NoC that can give real-time guarantees about the traffic, and has grown into a heterogeneous MPSoC. By adding hardware accelerators the architecture could deliver more performance.

The hardware accelerators have enough performance to process multiple data streams, but the architecture was not capable of sharing a hardware accelerator over multiple streams. So multiple hardware accelerators were needed, one for each stream. This was solved by introducing a centralized component called the gateway. The gateway orchestrates the sharing of an accelerator by multiple data streams. This is done by processing a block of data from one stream, and then a block of data from an other stream. A case study showed that the gateway could correctly enable sharing of an accelerator by multiple streams, but the utilization of the accelerators was low, due to the so called state-save overhead. Because the accelerators contain state, the gateway must save and restore this state when a switch is made between data streams. This thesis is focused on determining the causes of the high state-save overhead and the definition and evaluation of techniques that reduce this overhead.

We identified multiple causes for the high state-save overhead. A new gateway architecture is proposed that reduces this overhead by extending the high-speed ring network to support state being streamed from and to the accelerators. The proposed architecture is implemented and a dataflow model is proposed that corresponds to the new architecture. With this dataflow model it is possible to determine some real-time properties of the system.

A case study is performed in order to evaluate the proposed architecture. The evaluation results show that the proposed architecture reduces the state-save overhead up to 65%, while the hardware cost only increases with 10%. The reduction in state-save overhead resulted in a 2.5 times higher utilisation of the accelerators.

Acknowledgements

First of all I would like to thank Marco for his supervision and feedback during my thesis. I really admire his determination to explore new areas, and his enthusiasm that is an inspiration for others to become explorers.

I would also like to thank Guus, Berend and Gerben. They helped me get started with the Starburst architecture and were always willing to help or answer the questions that I had.

Additionally I would like to pay my tribute to all the CAES member, for several memorable discussions during the coffee breaks and Friday afternoon drinks.

Finally I would like to thank Tristia, for her support and motivation during my thesis.

Oscar Starink
Enschede, October 2015

Contents

List of Figures

List of Tables

List of Acronyms

ACC	ACCelerator.
ADS	Application Domain Specific.
AXI	Advanced eXtensible Interface.
CORDIC	COordinate Rotation DIgital Computer.
CPU	Central Processing Unit.
CSDF	Cyclo-Static DataFlow.
DDR3	Double Data Rate type 3.
DMA	Direct Memory Access.
EGW	Exit GateWay.
FIFO	First In, First Out.
FIR	Finite Impulse Response.
GW	entry GateWay.
ISE	Instruction Set Extension.
LUT	Look-Up Table.
LUTRAM	LUT used as RAM.
MPSoC	Multi-Processor System-on-Chip.
NoC	Network-on-Chip.
OS	Operating System.
PLB	Processor Local Bus.
RAM	Random Access Memory.
RPC	Remote Procedure Call.

RWI	Read/Write Interface.
SDF	Synchronous DataFlow.
SoC	System-on-Chip.
USB	Universal Serial Bus.
WCET	Worst Case Execution Time.

Chapter 1

Introduction

1.1 Context

For the last decades the number of transistors on a chip grew exponentially. This made enormous advances in CPUs possible. Where one of the first CPUs had several thousand transistors, the latest CPUs consist of several billion transistors.

These developments are driven by the demand for more computational performance. There is not only a demand for powerful personal computers in the consumer market, also embedded systems found in video and radio applications continue their demand for more computational performance. Advances in decoding, decompression and software defined radio algorithms are big contributors to this demand.

In order to cope with the demand, chip manufacturers started to create MPSoCs. These architectures can make use of the many transistors that are available by placing and connecting multiple processors. Performance is delivered by the parallel capabilities of the architecture. One can divide MPSoCs into two categories, homogeneous and heterogeneous. A homogeneous architecture consists of multiple identical processors, while heterogeneous architecture consists of multiple different processors. These heterogeneous architectures often add hardware accelerators that can only perform one specific computation. The advantage of these accelerators is that they can perform these computations effectively in terms of speed and energy.

The hardware accelerators can even be so fast that sharing becomes a possibility. Accelerator sharing is a technique where two or more independent computations are mapped on the same accelerator. This means that the accelerator becomes a shared resource and therefore needs more explicit synchronisation between its users and also needs a scheduling policy. Hardware architectures capable of sharing accelerators is a relatively new research area.

Mapping applications on parallel hardware is not always an easy task, and a lot of research is done is this area. This thesis will look at streaming applications. Streaming applications typically operate on an input stream and compute the output stream. These applications can often be mapped elegantly on parallel hardware.

Streaming applications usually have real-time requirements. This means that correctness not only depends on the computed result, but also on the time when results are produced. In order to guarantee correct temporal behaviour, models can be used to check temporal correctness. Developing models and modelling techniques is an actively researched area.

1.2 Research Platform

This section will give the background needed to understand and formulate the problem description. The first subsection contains a high level overview of an architecture that can share its accelerators. We will call an architecture capable of this an accelerator sharing architecture. The second subsection will introduce the models that are used to analyse the accelerator sharing architecture.

1.2.1 Accelerator Sharing Architecture

In this subsection we will present a heterogeneous MPSoC architecture that is suitable for streaming applications and capable of accelerator sharing. This architecture is first introduced in [1].

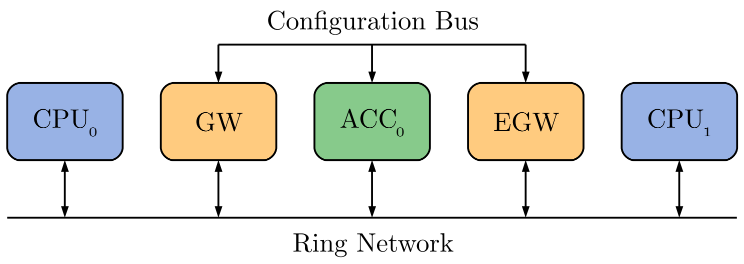


Figure 1.1: Global system overview

Figure 1.1 shows a global overview of the important components in the system. The system consists of CPU, accelerator and gateway tiles. Figure 1.1 shows only a small number of CPU and accelerator tiles but typical systems consist out of multiple CPUs and accelerators. It can even have multiple gateway pairs.

A CPU tile is connected to the ring network in order to communicate with other tiles. CPU tiles can be used for general computations.

An accelerator tile is a specialized component that can perform a specific operation efficiently. For example a Finite Impulse Response (FIR) filter. The accelerator tiles are also connected to the ring network. Additionally they have a configuration interface, which is connected to the configuration bus. An accelerator has an internal state that must be saved and restored when switching between two streams, the configuration bus is used for this.

Then there are the gateway tiles that have a specific function. The entry GateWay (GW) is responsible for the coordination of the sharing of the accelerators. When a block of data is received from a producer by the entry gateway, it will configure and restore the state of the accelerators using the configuration bus. Then it will send the data block to the accelerators, via the ring network. When the accelerators have processed the data the entry gateway will save the state of the accelerators, again using the configuration bus.

After the last accelerator in the accelerator chain comes the Exit GateWay (EGW). This tile is used to write the data to the correct consumer CPU. It is also used to check if all the data elements are processed by the accelerators. Because only then a state-save can be performed.

Placing an entry and exit gateway around an accelerator chain, makes it possible to share the accelerators in a transparent way. The gateway pair will "hide" the accelerator sharing for the producer and consumer.

1.2.2 Dataflow Modelling

In order to guarantee temporal constraints of an architecture, models are used that capture the temporal behaviour of the architecture. In this subsection we will discuss two dataflow models that capture the temporal behaviour of the accelerator sharing architecture, that were introduced in [2]. The first model is directly derived from the architecture. The second model is an abstraction of the first model.

For now only simplified dataflow models are presented. These are only used to illustrate some basic concepts. This subsection should be readable without expert knowledge about dataflow modelling.

Figure 1.2 shows the first model. It shows 5 actors connected with each other by edges. The actors have a one-to-one relation with the architecture. The actor v_P is the producing CPU, v_G is the entry gateway, v_A is an accelerator, v_{EG} is the exit gateway and v_C is the consuming CPU.

The topology of the dataflow graph in Figure 1.2 is a chain. In this dataflow model the chain implies a pipeline. Pipelines offer great performance if they can be utilised constantly.

The edge from v_{EG} to v_E represents the signal from the exit gateway to the entry gateway that indicates that all data is processed and the accelerators can be reconfigured. So reconfiguration is postponed until all data is out of the accelerator chain. This is called a pipeline flush. Flushing a pipeline has a performance penalty, because the pipeline is no longer constantly utilised.

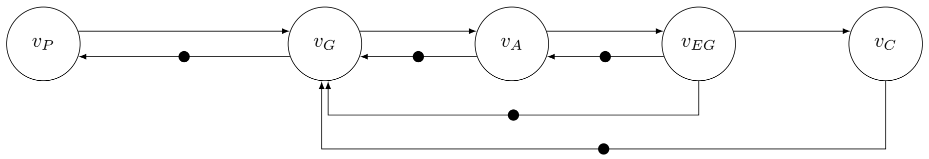


Figure 1.2: Dataflow model of a shared accelerator

Another basic concept is abstraction of dataflow models. This is a useful technique that can be applied on a complex model, in order to reduce its complexity. Figure 1.3 shows an abstraction of the model in Figure 1.2. Actor v_G, v_A and v_{EG} and the edges between them are abstracted into v_S.

An abstraction can be used to reduce the complexity which can make reasoning about the model simpler. However by removing details, the model can become less accurate and will likely be an over approximation of the original model. An abstraction is done in such a way that some properties that hold for the abstraction also hold for the original model.

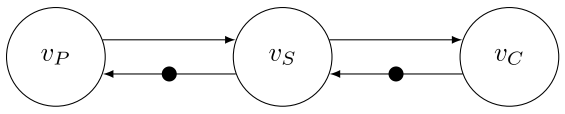


Figure 1.3: Abstracted dataflow graph of Figure 1.2

1.3 Problem Description

Sharing accelerators has its advantages, but it also introduces some problems. This section will highlight those problems.

As mentioned in the previous sections accelerator sharing has its advantages. The accelerators will be better utilized and sharing an accelerator reduces the hardware. However there are disadvantages as well. Sharing an accelerator requires synchronisation of its users, and it can result in all kinds of synchronisation problems, like deadlock and race conditions. Another disadvantage is that these synchronisation have some overhead. However this overhead is relatively small, and thus acceptable.

The architecture described in section 1.2 solves synchronisation problems by introducing gateways. The gateway is responsible for managing the accelerator sharing, it handles all synchronisation and schedules the requests. This makes development of applications easier, because producers and consumers only connect to the gateway and are not aware of other computations on the accelerators.

The biggest problem is state. Most of the accelerators have an internal state that is needed for the computation. For instance the previous samples are needed in the computation of an FIR filter. As two different computations most likely have different states, the state must be saved and restored between computations. This will introduce some overhead, which is called state-save overhead. The size of the state-save overhead depends on how the state is saved. The state-save overhead prevents full utilization of the accelerators.

In order to reduce a large state-save overhead ratio, computations are made longer. So context switches will occur less often. This will increase the throughput of an application. But when a task processes more data, it will delay the start of the other tasks because they are sharing the accelerator. So when more data is processed between switches, it results in a higher latency and larger data bursts. In order to cope with larger data burst, larger buffers are necessary. So there exist a trade off with on one hand throughput and on the other buffer size and latency.

State-saves in the architecture described in Section 1.2 are done by copying the state to the memory in the gateway. This means that the state-save overhead is linearly related to the state size. Measurements on the architecture indicate that the state-save overhead is relative high, 23 clock cycles per 32-bit word transfer [1]. In this thesis we will describe a technique to reduce the state-save overhead.

In order to analyse the temporal behaviour, there must be a dataflow model of the proposed architecture. It is likely that a different dataflow model is needed when the hardware is modified to reduce the state-save overhead. If this is the case also an abstraction should be made, in order to simplify the new dataflow model. Furthermore the improvements of the new architecture should be reflected in the new dataflow models.

1.4 Research Questions

The goal of this research is to answer the following research question:

How to reduce state-save overhead for shared accelerators in an MP-SoC with a ring NoC?

In this thesis we will find the cause of the state-save overhead, and present an architecture that will reduce this overhead. The presented architecture should be performant, analysable and should have a low area footprint. These properties should be quantified by means of measured results. From these objectives the following sub-questions are derived:

- What is the cause for the large state-save overhead?
- Is it possible to introduce an architecture that reduces the state-save overhead?
- What is the performance of this architecture?
- What is the hardware cost of this architecture?
- Is it possible to model the temporal behaviour of the architecture?
- Do the models capture the gain in performance?

1.5 Contributions

In this thesis we describe improvements of an architecture capable of sharing accelerators [1] and their corresponding dataflow models [2]. The main contributions described in this thesis are:

- Pinpointing the cause of the large state-save overhead in the existing accelerator sharing archtecture [1].
- The proposal of an architecture capable of sharing accelerators that has lower state-save overhead.
- Description of the implementation of the proposed architecture.
- Proposing dataflow models for the architecture.
- Evaluation of the architecture and the dataflow models.

1.6 Outline

The outline of this thesis is as follows. First we discuss related work in Chapter 2. In this chapter we will position our work within the research area, and we will compare our work with alternative approaches. In Chapter 3 we describe the details of the existing accelerator sharing architecture. We pinpoint the causes of the high state-save overhead of this architecture and propose a new architecture that reduces the state-save overhead. In order to guarantee real-time constraints, dataflow models can be used. We will discuss these models in Chapter 4. We describe the existing dataflow models of the previous architecture. Then a new dataflow model for the proposed architecture is presented, and we make an abstraction of this model. Evaluation is done in Chapter 5. We will evaluate the proposed architecture and its dataflow models. Finally we will make a conclusion and propose future work that can be the basis for new research. This is done is Chapter 6.

Chapter 2

Related Work

In this chapter we will discuss work related to state-saving for shared accelerators. First we will discuss some methods to include accelerators in an architecture. Next we discuss a number of accelerator sharing architectures. In the third part we will discuss multiple real-time analysis techniques. Finally we mention a technique to model shared accelerators.

2.1 Hardware accelerators

In this section we will discuss the advantages and disadvantages of hardware accelerators and how they can be integrated into a system.

In systems that perform calculations there is typically a trade-off between efficiency and flexibility. It is often the case that flexible systems are not as efficient as their static counter part.

Hardware accelerators can often perform only one specific calculation, but can therefore be specialised in this calculation, which results in a better efficiency. So typically hardware accelerators are faster, use less energy and/or use less hardware resources than flexible solutions. As a result hardware accelerators are often used in situations where efficiency is more important than flexibility. And this is the reason that hardware accelerators are used in our design.

When a system needs to be flexible but also efficient, a combination of the two can be made. Examples are heterogeneous systems with CPUs and hardware accelerators. The CPUs are flexible and can perform general-purpose calculations, while the accelerators can efficiently perform specific calculations. There are several ways to combine CPUs and hardware accelerators, we will discuss some of them.

2.1.1 Instruction set extension

One technique to combine CPUs and hardware accelerators is via an Instruction Set Extension (ISE). The CPU is designed to have some additional instructions that are used to control the hardware accelerator. This results in a tight coupling of the CPU and accelerator. Examples of these ISEs are the MMX and SSE extensions in the x86 processor architectures.

Our architecture does not use ISEs to control the hardware accelerators, because this technique prevents sharing of the accelerator, due to the tight coupling of the CPU and accelerator.

2.1.2 Remote procedure call

Another way to control hardware accelerators is via Remote Procedure Calls (RPCs). RPCs enable a CPU to start a calculation somewhere else in the system. Typically the CPU and accelerators are connected via a bus and the RPCs are performed by reads and writes. An example of this technique is the IBM 4764 PCI-X Cryptographic Coprocessor [3]. This is a hardware accelerator that can be used for cryptographic calculation. It is connected via the PCI bus which is a common bus in computers.

While it is possible to share the accelerators with this technique, we do not use RPCs in our architecture because it is not possible to cascade the result of a calculation to another accelerator. This is a disadvantage because our architecture targets streaming applications, where the ability to chain accelerators can be a real advantage. Also when a RPC is performed the CPU has to wait for the result. This time is lost, since the CPU cannot perform any useful calculations while it wait. When stream based hardware accelerators are used the CPU only has to write a data stream to the accelerator. The CPU does not have to wait on the result, because this stream will typically go to an other consumer. Because the CPU does not have to wait, it can perform more useful calculations.

2.1.3 Stream based hardware accelerator

The stream based hardware accelerators perform calculations on streams. The CPUs are used to produce and consume data streams that can be processed by accelerators. A good example of this technique is the Starburst architecture [4]. CPUs and accelerators are connected via the Nebula ring network which has support for stream based communication [5].

Our architecture uses this technique to combine CPUs and accelerators, and is based on the Starburst architecture. This architecture allows efficient mapping of streaming applications and it is possible to chain accelerators. In [1] the Starburst platform is extended to support the sharing of accelerators.

2.2 Accelerator sharing architectures

In this section we will discuss several accelerator sharing architectures that are related to the accelerator sharing techniques presented in this thesis. First we will describe context switches that are used by Operating Systems (OSs) to share the CPU over multiple programs. After this we will describe the PROPHID and Eclipse architectures that are both capable of accelerator sharing.

2.2.1 Context switch

The accelerator sharing techniques in this thesis have a lot in common with the context switches performed by modern OSs such as Windows and Linux. These context switches make it possible for multiple programs to share the same CPU. This looks a lot like multiple data streams that share the same accelerator. Just as an accelerator the CPU has an internal state, due to the general-purpose registers and status flags. The state needs to be saved, so it can be restored later. The saving and restoring of the state is done within a context switch. During a context switch the state of the CPU is stored into memory. Then it will determine which program will be continued. The CPU state corresponding to that program is loaded from memory and restored. Now the program can continue its execution. Typically context switching is done periodically by a timer interrupt. This will interleave the executions of the different programmes, giving the illusion that they are running in parallel.

Similarities with our techniques are that the state is saved into memory and restored on a later moment. Another similarity is that both approaches result in interleaving. Furthermore they both have some overhead that is due to the saving and restoring. A difference is that the context switch is performed by the CPU itself, while we initiate the saving and restoring from the gateway. Another difference is that the CPU performing the context switch is directly connected to memory. This is in contrast with the accelerator, which has no access to memory. Instead the state is retrieved by the gateway and saved into memory of the gateway. Lastly the context switching is done periodically, while the gateway switches streams after a fixed number of data samples, this is called the packet size. We can say that the task that share an accelerators are cooperatively scheduled, because after the packet size the running task allows other task to run. So there are aware that the accelerator is shared, and give the other tasks also a chance to use the accelerator. While tasks that share a CPU are typically pre-emptively scheduled, this means that the tasks are not aware that the CPU is shared. The context switch interrupts a running task and pauses it, while a task that was paused will be continued.

With both techniques you can control the granularity of interleaving, by changing the period or the packet size. By increasing the packet size the overhead of the switches becomes smaller, but the responsiveness decreases. In contrary, the overhead becomes bigger and the responsiveness increases if the packet size is decreased. An appropriate packet size depends on the application.

2.2.2 PROPHID

PROPHID [6] is a heterogeneous multiprocessor architecture that is designed to deliver guaranteed real-time processing for multimedia applications. The architecture consists of two main parts. The first is one CPU that is primarily used for control oriented tasks and the second part consists of multiple Application Domain Specific (ADS) processors that perform the high performance and time critical operations. The CPU and ADS processors are connected to a central bus. The ADS processors are also connected to a programmable high bandwidth communication network. There is a main memory which can be accessed from the central bus and from the communication network via an arbiter.

In order to improve the utilization of the ADS processors, they are capable to process between 1 and 5 data steams in a time interleaved fashion. The ADS processors have multiple input and output First In, First Outs (FIFOs), equal to the number of streams it supports. Context switches are done at a fine granularity, in order to keep the FIFOs sizes small, because FIFOs can typically hold only 32 samples. The ADS processors also have multiple state banks, equal to the number of streams it supports. These state banks make context switches almost instant, this is the reason that fine granularity is possible without a large state-save overhead.

There are a lot of similarities between PROPHID and our architecture. Both allow hardware accelerators to be shared over multiple streams and consequently, both perform state-saves. A difference is that ADS processors have additional hardware and local memory to perform the state saving and restoring. This means that the maximum number of streams is determined by the number of states the local memory can hold. Another difference is it that the CPU is not connected to the high-throughput network and cannot be used to process data streams. While in our architecture the CPUs are connected to the ring network and they can be used to process data streams.

2.2.3 Eclipse

Eclipse [7] is a heterogeneous multiprocessor architecture for stream processing. The computations are done by CPUs and coprocessors. The CPUs and coprocessors are connected to the communication network via so called shells. These shells hide the underlying communication network, and provide 5 primitives that enable stream based communication via FIFOs and a way to perform a task switch. The coprocessors use these primitives to get the input stream(s) and store the output stream(s). The task switches are also initiated by the coprocessor.

The 5 primitives are `GetTask`, `Read`, `Write`, `GetSpace` and `PutSpace` [7]. A coprocessor gets a task ID together with optional configuration data with the `GetTask` primitive. The task ID is used as identifier for the different task, and is used as an argument for all other primitives. Then it needs to reserve free space in the destination FIFO and check for data in the source FIFO. This is done with the `GetSpace` primitive. If there is no data in the source FIFO or no free space in the destination FIFO, it cannot perform the current task and it will request a new task. If it is possible to continue it will read the source FIFO with the `Read` primitive. This is followed by the `PutSpace` primitive to indicate that the data is read. The coprocessor can now start its computation. The results are stored with the `Write` primitive and followed with the `PutSpace` primitive to indicate that there is new data. Now the coprocessor can start with a new task.

Eclipse and our architecture have some similarities. Computations can be done by a mix of processors and coprocessors and communication between tasks is done with FIFOs. A difference is that Eclipse provides a uniform interface to the network for processors and coprocessors via its shell primitives. However these high level primitives result in large hardware costs of the shells. In our architecture the processors and accelerators have two separate communication mechanisms, resulting in a lower hardware cost. Another difference is that each coprocessor needs a shell, in our architecture multiple accelerators can be managed by one entry gateway and exit gateway pair. While the Eclipse is capable of sharing coprocessors, state-saving mechanisms for coprocessors are not described.

2.2.4 Starburst

The Starburst platform is an MPSoC with CPUs and hardware accelerators, that targets streaming applications. The accelerators in this system are stream oriented. The CPUs and accelerators in the system are connected via a ring NoC, that has support for data streams. In [1] the Starburst platform is extended to support sharing of accelerators by introducing an entry and exit gateway to the system. These gateways can schedule multiple different data streams over the accelerators and it will save and restore the state of the accelerator when this is needed. The state saving and restoring of the accelerators is done via a configuration bus that connects all accelerators to the entry gateway.

Our proposed architecture is based on [1]. We also used the gateways to enable the sharing of accelerators. However, measurements in [1] show there is a large state-save overhead. In our proposed architecture we present a new technique to save and restore the state of the accelerator that reduces the state-save overhead.

2.3 Real-time analysis techniques

Analysis of real-time systems is used to guarantee temporal constrains. There are three major frameworks that can be used to perform real-time analysis. These frameworks are suited to model concurrent application and pipelined execution. We will briefly mention these three techniques.

2.3.1 SymTA/S

The SymTA/S [8] framework is based on event models. In event models the traffic is characterised with a period and a jitter. The traffic characterisation can have low accuracy, because the correlation between different streams is not captured. It does not support cyclic data dependency in the general case because the analysis technique will report that the latency is infinite [9].

2.3.2 Real-time calculus

Real-time calculus [10] is an analysis technique based on network calculus. It characterises the traffic between components in the time domain. Attempts are made to handle cyclic dependencies [11], [12]. Both approaches only consider cyclic data dependencies or cyclic resource dependencies, but not a combination of both.

2.3.3 Synchronous dataflow

Synchronous DataFlow (SDF) [13] [14] is closely related to Kahn networks. SDF operates on directed graphs, where cycles are allowed. Tokens are transported over the edges, and can be used for example to model data or free space in a buffer. The actors have firing durations. When all input edges contain enough tokens, the actor is enabled and after the firing duration the actor will fire. When an actor fires it consumes tokens for the input edges and will produce tokens on the output edges. The number of tokens that is produced or consumed is indicated by the production and consumption quanta. Analysis is done by creating a schedule. This schedule is used to determine minimal buffer sizes and guarantee throughput constrains. Executions of tasks can be specified in multiple ways, for instance using Worst Case Execution Time (WCET) or with a (σ,ρ)-characterisation [15]. Resources can be shared with different schedulers, starvation-free schedulers such as round robin and budget scheduling, and recently also non-starvation-free schedulers can be used such as static-priority scheduling [15].

In this thesis we will use synchronous dataflow to model temporal behaviour, because it is the only technique that can deal with cyclic data and resource dependencies. Cyclic data dependencies are used to model finite buffers and the sharing of the accelerators result in a cyclic resource dependency. Another motivation is that parts of the proposed architecture already have been modelled with data flow, such as the NoC in [16], and the accelerator sharing architecture presented in [1] has been modelled with dataflow in [2].

2.4 Modelling accelerator sharing

In this section we will discuss related work relevant to the modelling of accelerators that are shared.

In [17] SDF and Cyclo-Static DataFlow (CSDF) are used to model the sharing of accelerators. The data streams are scheduled with a round robin scheduler. The models can be used to satisfy minimum throughput constrains for multiple data streams.

While the techniques presented in [17] can deal with shared accelerators, they do not include state saving. So the accelerators that are shared can not have state. Furthermore, they do not discuss methods to determine optimal packet sizes. We will propose a dataflow model that does include state saving and can be used to determine optimal packet sizes. We will make use of the methods presented in [2].

Chapter 3

Implementation

This chapter will describe the architectures used for accelerator sharing. The first section will describe the previous architecture and its limitations. Then the basic idea is presented that will deal with the problems. Finally the proposed architecture is described in detail.

3.1 Previous accelerator sharing implementation

In this section the previous accelerator sharing implementation will be shown that was described in [1]. First the hardware of the architecture is presented. After this the software structure will be explained.

3.1.1 Hardware

Section 1.2.1 already gave a global overview of the architecture, this subsection will recap Section 1.2.1 and give a more detailed overview of the architecture.

The Starburst [4] is heterogeneous MPSoC and consists out of CPUs and accelerators. We will use Figure 3.1 to explain all the components of the system. Figure 3.1 shows only a small number of CPUs and accelerators, but a typical system consists out of multiple CPUs and accelerators.

The Nebula ring network [5] connects all the components in the system. The Nebula is an unidirectional, guaranteed-throughput ring network. It uses slots to guarantee that all components that are connected to the ring have a guaranteed-throughput.

The CPUs are Xilinx MicroBlazes running the Helix OS [4]. All CPUs have a scratch pad memory that can be written via the Nebula ring network.

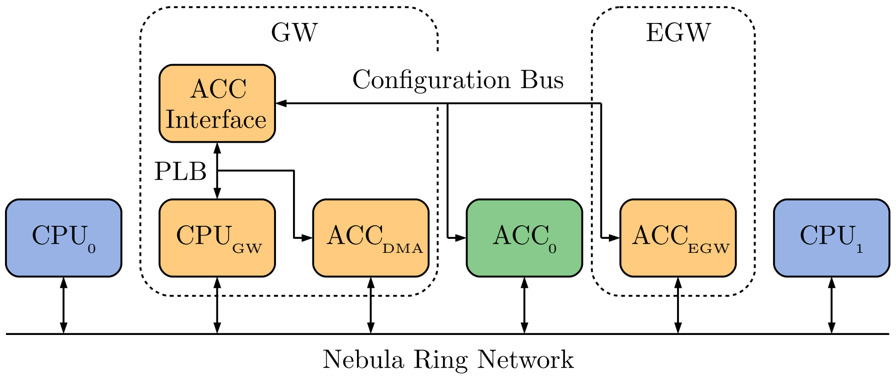


Figure 3.1: High level system overview

The accelerators are also connected to the ring network. Additionally they have a configuration interface, which is connected to the ACCelerator (ACC) Interface. The ACC Interface is used to perform the state-saves.

Then there are some components that together have a specific function. The gateway CPU (CPU$_{GW}$) together with the ACC Interface and accelerator Direct Memory Access (DMA) (ACC$_{DMA}$) form the entry gateway. These components are connected via the Xilinx Processor Local Bus (PLB) bus. The entry gateway is responsible for the coordination of the sharing of the accelerators. When a block of data is received by the entry gateway, it will configure and restore the state of the accelerators using the ACC Interface. Then it will send the data block via the Nebula ring network to the accelerators. This is done with the accelerator DMA, also called Ring DMA. When the accelerators have processed the data the entry gateway will save the state of the accelerators, again using the ACC Interface.

When the design contains multiple accelerators, it is possible to use the output of an accelerator as input for another accelerator. When this is done the accelerators form a chain, and we will call this an accelerator chain.

The last accelerator in the accelerator chain is the exit gateway accelerator (ACC$_{EGW}$), and this forms the exit gateway. This accelerator is used to write the data to a scratch pad memory of a CPU. It is also used to check if the last data element is processed by the accelerators.

There are components that are not shown because they are of no relevance for the discussion in this thesis. However they will be mentioned here for completeness. Every CPU is connected to one global Double Data Rate type 3 (DDR3) memory, with an arbitration tree. There also is one CPU that runs Linux. This CPU has additional peripherals, such as an Universal Serial Bus (USB) and an Ethernet controller.

There are two types of communication over the Nebula ring network. One is write-only, address based communication to the scratch pad memories. This communication is used by the CFIFOs. CFIFOs logical FIFOs that are implemented in software that are based on the C-HEAP [18] algorithm. The CFIFOs are used for communication between CPUs. The other type of communication is credit based. Communication between accelerators is of this type. The credit based communication is designed to have low hardware cost and supports back pressure. The back pressure is needed prevents buffer overflows in the accelerator [5]. The sender of data to the accelerator is keeping a local counter of free space in the buffer of the accelerator. When the sender sends one data word it will subtract one from the counter. By preventing the sender to send while the counter is zero, a buffer overflow can not occur. When the accelerator consumes a data word from its buffer, a credit is sent to the sender. When the sender receives the credit the counter will be incremented by one. In order to support the credit based communication the Nebula ring network contains additional hardware. Figure 3.2 shows a detailed overview of the additional hardware.

Figure 3.2 shows how an accelerator is connected to the ring network. The link performs the most basic operation, it schedules data transfers and credit transfers in the available slots. The credit control down block will buffer received data and when the data is consumed by the accelerator it will generate an acknowledgement credit. The credit control up block keeps a local counter of free space of the next accelerator in the chain, and prevents data to be sent when the local counter is zero. The ring shell is used to configure the destination addresses of the credit and the data. These addresses must correspond with the previous and next accelerator addresses in the chain of accelerators.

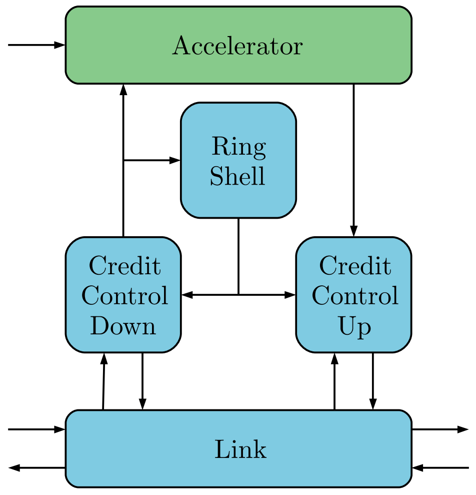


Figure 3.2: Nebula ring network connection

3.1.2 Software

The software that runs on the gateway CPU is responsible for the state-saves and state restores. The code is written in C++ and uses an object orientated approach. For every accelerator there is a corresponding class that can save and restore the state of that accelerator. The class is also responsible for the storage of the state data. Every specific accelerator class inherits from the accelerator base class. This base class contains the data forward and the credit return address. These addresses correspond with the addresses needed in the ring shell in order to have correct credit based communication.

Then there is an AccList object that is used to create accelerator chains. Multiple accelerators can be added to the AccList. The AccList is used to process a block of data with the accelerators, this function is called process and the following will happen. First the data forward and the credit return address are configured for every accelerator in the list. Then for every accelerator the state is restored, by calling the restore function of every accelerator object. Now the data can be sent to the accelerators. The gateway waits until all data is processed by the accelerators. The last thing it does is saving the state of the accelerators, by calling the save function of every accelerator object in the list.

The AccList object uses the decorator pattern [19] to represent the structure of the accelerator chain. The template method pattern [19] is used to generate the correct behaviour.

When the accelerators are shared there is more than one AccList object and these objects need to be scheduled. The scheduler will check that there is data available from the producer and that the consumer is ready for new data. If these conditions are satisfied it will call the corresponding AccList process function.

3.1.3 Pinpointing the problem

This subsection pinpoints the cause for the high state-save overhead. As stated before the average time to read or write a 32-bit word of state from an accelerator is 23 processor cycles [1]. After looking into the implementation the following causes for the high state-save overhead were identified.

- Slow state data access over the ACC Interface. The ACC Interface is connected to the CPU via the PLB bus. This bus has a high transfer overhead, when a single transfer is done. This is likely a significant part of the state-save overhead.
- A lot of code in the critical loop. The code that is responsible for saving and restoring the state is located in multiple C++ objects. So the gateway CPU needs to perform a lot of memory accesses and this can result in cache misses, which will degrade the performance.

3.2 Basic idea

This section presents approaches to solve the problems that where identified in Section 3.1.3.

- The proposed design will use the Nebula ring network to perform state-saves, instead of the PLB bus. The Nebula ring network is capable to transfer one word each cycle, and thus fast. This will help to reduce the state-save overhead. It would even be possible to use the Ring-DMA to send state to the accelerators. An other advantage is that the ACC Interface can be removed, reducing the hardware cost of the design. In order to be able to send state to the accelerators via the ring network it is necessary to be able to differentiate between data and state. So some additional control logic is needed in order to support this. By placing this control logic between the accelerators and the network, the control logic has full control over the accelerator and is immediately connected to the Nebula ring network. So no additional network connections are needed, and this will keep the hardware cost down. Another advantage of this approach is that most of the hardware already exists and can be reused. This will keep the development time within limits. Reuse of the accelerators is achieved by connecting the control logic to the existing configuration interface of an accelerator. In order to differentiate between state and data, commands will be used to indicate what is being send, state or data. Sending these commands has some overhead, but they are needed to be able to use the Nebula ring network. We expect that the performance gain will outweigh the extra overhead. A disadvantage of this approach is that when the state of an accelerator is saved to the memory in the entry gateway, it will traverse the whole ring network, because the Nebula ring network is unidirectional. For now we will accept this, but in future work section we will present a possible solution to prevent this.

- In order to improve the software that performs the state-saves, we will try to minimize the number function calls and the number of instructions that are executed in the critical loop. Due to the template method pattern the code that performs a restore-save cycle, is located in different C++ objects, one for each accelerator. These objects contain multiple functions that perform operations needed in the different stages of a restore-save cycle. This results in a lot of function calls. Because most of the time the entry gateway is moving data when performing a restore-save cycle, so we will try to optimize this. The following is proposed; the decorator and template method patterns are combined with the builder pattern [19]. This allows us to use the decorator pattern to define the structure of the accelerator chain. Next the template method pattern is used to build a list of data transfers. The list will represent the transfers needed to perform a restore-save cycle. When needed the list can be processed using the interpreter pattern [19]. This will generate the behaviour that will perform the restore-save cycle. With this approach it will be possible to have a low code size and only a few function calls in the critical loop, because only the interpreter is executed. This is expected to result in a better performance. Another advantage is that the list of transfers can be optimized. Multiple single transfers to the same address can be optimized to a burst transfer. This allows us to make use of DMA burst transfers, resulting in faster transfers. Because we still use the decorator pattern it is still easy to define the structure of the accelerator chain. A disadvantage of this method is that it uses more memory, because it needs to store the list of transfers. However the embedded system has a large DDR3 memory that can be used so this is not a real problem. Another disadvantage is that once the list of transfers is build, it is no longer possible to add additional accelerators to the processing chain. But most applications do not require dynamic addition of accelerators, so typically this will not be an issue.

3.3 Proposed implementation

This section will present the proposed architecture for accelerator sharing. This section is divided in three subsections. First the hardware is presented. The second subsection will discus the changes made in the software. The last subsection will discus what the steps are to perform a restore-save cycle.

3.3.1 Hardware

Figure 3.3 shows the top level changes to the architecture shown in Figure 3.1. The ACC Interface is completely removed, and all accelerators that were connected to the ACC Interface are now connected to the Nebula ring network via the Read/Write Interface (RWI). The RWI makes it possible to configure the accelerators via the Nebula ring network, this is indicated by the configuration bus connected to the sides of the accelerators and the RWI. Note that the DMA accelerator does not have an RWI. This is because it is still directly connected to the gateway CPU via the PLB bus.

In Figure 3.4 can be seen that the RWI is placed in between the accelerator and the Nebula ring network logic. So using the RWI does not require rigorous changes in the existing hardware.

In the following subsections we will describe the additional hardware and the changes to the existing hardware.

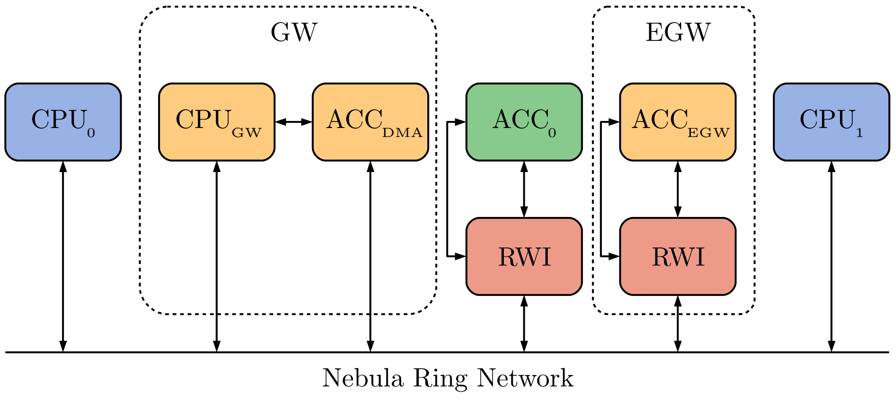


Figure 3.3: High level overview of the proposed architecture

Read/Write Interface

Because the RWI receives both data and state, it needs a way to differentiate between the two. This is done by introducing modes. The RWI has internal modes and the mode dictates if the received words are data or state. The RWI has the following modes: normal, read, write and bypass. What these modes exactly do, will be described in the section RWI modes. First it is discussed how modes are changed. The modes can be altered by sending commands. So now the RWI needs to differentiate between data and commands. This is done by using the address of the data transfer. Table 3.1 shows the meaning of the bits in the address space of the Nebula ring network. Bit 15 is removed from the free address space and is now used to indicate a RWI command.

Bit	Meaning
31:28	0x6 is the address space of the Nebula ring network
27:20	CPU ID
19:17	Sub ID, ID 0 is the CPU, other ID's are accelerators
16	0 for credit based data, 1 for Ringshell access
15	0 for data, 1 for RWI commando
14:0	Free address space

Table 3.1: Meaning of Nebula ring network address bits

When data is sent to the RWI and address bit 15 is set, the data should be interpreted as a command. Table 3.2 shows the meaning of the commando data bits. The ID field is used to indicate for which accelerator the command is. When the ID of a command does not match the accelerator ID, it is sent to the next accelerator in the chain. This is done with the data forward address provided by the ringshell. Bit 15 is set to indicate that it is a command. When

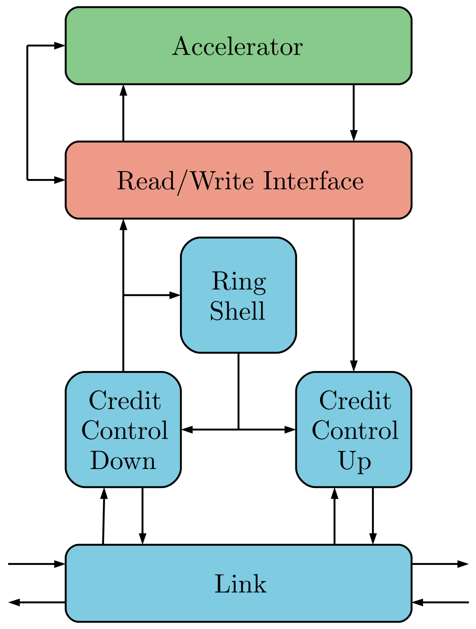


Figure 3.4: Nebula ring network connection with RWI

the ID field (bits 31:20) is a match, the mode field (bits 19:18) is used to switch modes. When the mode becomes reading or writing, the address field (bits 11:0) is used to determine the start address, and the number of reads/writes field (bits 17:12) indicates the number of reads/writes.

Bit	Meaning
31:20	ID
19:18	Mode, '00' normal, '10' read, '01' write, '11' bypass
17:12	Number of reads/writes
11:0	Read/write address

Table 3.2: Meaning of the RWI command data bits

RWI modes

The RWI has 4 modes and Figure 3.5 shows how each mode can be reached. After a reset the RWI will start in bypass mode. Each mode has it own behaviour and these will be explained below.

When in normal mode, the RWI will be transparent for data to and from the accelerator. It will appear as if the accelerator is directly connected to the network. So this is the mode the RWI should be in, when the accelerator is used for processing.

When the RWI is switched to read mode, a start address and the number of reads are set. This mode will read a word via the accelerator configuration bus from the start address and will sent this word to the next accelerator via the Nebula ring network. The number of reads is decreased with one and the start address is incremented to the next word. While the number of reads is not zero it will perform another read cycle. When the number of reads reaches zero, the RWI will switch to the bypass mode.

When the RWI is switched to write mode, a start address and the number of writes are set. This mode will read a word from the Nebula ring network and will write that word to the accelerator via the configuration bus at the start address. The number of writes is decreased with one and the start address is incremented to the next word. While the number of writes is not zero it will perform another write cycle. When the number of writes reaches zero, the RWI will switch to the bypass mode.

When in bypass mode the RWI will send data from the Nebula ring network to the next accelerator via the Nebula ring network. In this mode the data is not altered and the accelerator is bypassed. This mode allows state information for a later accelerator to be sent as data without it getting altered.

The RWI only accepts commands when it is in normal or bypass mode and when the accelerator indicates that it is done. An accelerator is done when all input is processed and the results are produced. This will guarantee that the data and commands keep their order and that the accelerator is in a known state when a command is performed. If these preconditions are not enforced, it would be possible to interleave state and data. For instance if a read command is performed, while the accelerator is still producing data.

Every accelerator gets its own RWI and later we will describe how the RWI and their different modes can be used to perform a state-save.

Accelerator

The proposed architecture makes it possible to reuse the existing accelerators. However some additional functions need to be added to the accelerators. In order to indicate whether an accelerator is done a signal is added to the accelerator configuration bus. This signal is used by the RWI to correctly switch between modes. This signal did not exist in the previous hardware, so all the accelerators are altered so that they can generate this signal.

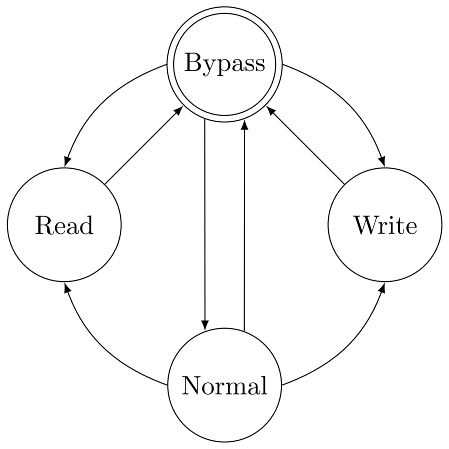


Figure 3.5: Modes of the RWI and the possible transitions

Ring DMA

The ring DMA is also reused but needed some adjustments to correctly deal with the RWI commands. Where before the DMA would only send to one address (because it was always data), now there also is a command address. So in the previous design the DMA only buffered the data, and the destination address was set in a separate register. The altered DMA, buffers the data together with the destination address, so now the buffer can contain a mix of data and commands for the accelerators.

The following things were not changed. The DMA still uses the PLB bus to read the data from memory, and only when 8 consecutive reads are done, burst reading is used. This means that writing a few words is slow.

Configuring the DMA is still done via 4 registers. The 4 registers are: source address, destination address, number of bytes to transfer and a start/status register. These registers are accessible via the PLB bus.

Commands and data are always written to the accelerators with the ring DMA, because only then the order of the writes are preserved and the proposed state-save mechanism needs the preservation of order. So the CPU must never write to the accelerators directly because the order of direct writes and DMA writes is unknown and can interleave because both have separate transfer buffer.

So to send one commando to the RWI, 4 PLB writes are done to setup the DMA and the DMA will perform one PLB read. This is a noticeable overhead, especially when a lot of small transfers are done. In the next section an optimization is presented that tries to minimise this overhead by grouping multiple small transfers.

```
Chain *chain_a2 = new Chain();

Cfifo_source<interm_fifo_r> *source_a2 = new
    Cfifo_source<interm_fifo_r>(rd_data_interm[0]);
CordicQuad *quad_a2 = new CordicQuad();
FirFilter *fir_a2 = new FirFilter(B1, BL, 8);
Cfifo_sink<output_fifo_w> *sink_a2 = new
    Cfifo_sink<output_fifo_w>(wr_data_acc_a);

chain_a2->addShackle(source_a2);    //decorate
chain_a2->addShackle(quad_a2);
chain_a2->addShackle(fir_a2);
chain_a2->addShackle(sink_a2);

chain_a2->init();                   //build & optimize
```

Listing 3.1: Code snippet showing easy composition

3.3.2 Software

In this subsection we will explain the software components, and what is done to improve performance.

As mentioned earlier the software on the gateway CPU plays a critical role in the sharing of the accelerators. The scheduling of multiple streams and the state saving and restoring are all done by the software. So in order to reduce the sharing overhead the software needs to be fast.

Chain composing

In this section we demonstrate how easy an accelerator chain can be defined.

Listing 3.1 is taken from the evaluation application and shows the easy composition due to the decorator pattern. First a `Chain` is instantiated. Then the following components, called shackles, are instantiated and added. The first shackle is `Cfifo_source`, this will manage the CFIFO communication from the producing CPU. CFIFO is the type of FIFO that is used to communicate between CPUs. Next are the `CordicQuad` and `FirFilter`. Note that the filter parameters are given at instantiation. The last shackle is the `Cfifo_sink`, this will maintain the CFIFO communication with the consuming CPU. In the end the `Chain` is initialized. This will build and optimize the `Chain`. These steps are described in the next sections. After these steps the `Chain` is ready to be used by the interpreter. The following sections will describe these last steps in more detail.

Chain building

In this section we will describe how the existing code is changed from performing actions into code that creates a list of actions.

First the code is analysed in order to see what kind of actions are performed. It was found that most of the time the CPU is configuring the DMA in order to perform a DMA transfer. So three actions are defined: a single word DMA transfer, a multiple word DMA transfer, and a call function action that can be used to do something else than a DMA transfer.

Next all accelerator functions are altered to generate a list of actions instead of performing the actions. Generating a list of all actions is done by calling all the accelerator functions in the same way as the critical loop did and concatenating the actions into one list. So the builder still uses the template method pattern to generate the list. However this is not done in the critical loop so it is not a problem that there are a lot of function calls.

Chain optimization

Another benefit of creating an action list is that it can be optimized to group DMA transfers. So after generating the list, an optimizer rewrites the list. The optimizer looks for consecutive single word DMA transfers to the same address. If the optimizer finds these, it will reserve memory for the data of the consecutive writes and will place the data in this memory. This is done so the DMA can access this data. The optimizer will replace the consecutive single word DMA transfer actions with one multiple word DMA transfer action in the list. This reduces the number of DMA configurations and therefore improves performance.

Chain interpreter

The list of actions can now be used to perform a restore-save cycle, by interpreting the list. This allows for a tight critical loop, because only interpreter code is run. This code is small because there are only three different actions. The call function action can still be used to execute other code in the critical loop. However, in this proposed implementation the function call actions are used sporadic and the called functions are designed to be small.

Scheduling chains

When accelerators are shared the gateway must schedule multiple chains. This is done with a round robin scheduler. Listing 3.2 shows a part of the critical loop and the round robin scheduler. Before a restore-save cycle is done by the `go()` method, the scheduler checks if the pre-conditions are satisfied with the

```
while(1){
    if(chain_a1->ready())    //check pre-conditions
        chain_a1->go();      //interpreter

    if(chain_b1->ready())
        chain_b1->go();

    ...
}
```

Listing 3.2: Code snippet showing critical loop and round robin scheduler

`ready()` method. This method will check if there is data to be processed and if there is space in the buffer that accepts this data.

For now the scheduler always performs a state-save between every packet. Even if the packets came from the same stream. However it should be possible to process multiple packets of one stream without state-saves in between. Also note that this can only occur when the other streams do not yet satisfy the pre-conditions.

3.3.3 Restore-save sequence

In this subsection we will explain how the RWIs are used to perform a restore-save cycle. There are three major stages. These are the restore, process and save stages.

In the Figures that follow we will use letter to indicate the mode of the RWIs and the type of the commands that are sent. The modes are denoted with N, B, R and W which are normal, bypass, read and write respectively. The commands are denoted with the same letter and will change the mode accordingly. The commands have a subscript that indicates for which RWI the command is. An A is used to mark commands for the accelerator RWI and an E is used to mark commands for the exit gateway RWI.

Restore stage

This is the first stage and we assume that the RWIs of all accelerators are in bypass mode. In order to restore state of the first accelerator, the gateway sends a write command to the RWI of the accelerator followed with the state data. When an accelerator is restored, its RWI will go into bypass mode. This allows us to send state to accelerators further in the chain. If there were more accelerators the state would be restored in the same way. Next the exit gateway accelerator is configured in such a way that it will sent the processed data to the right place. This is done by sending a write commando to the exit gateway followed with some state data that contains the correct configuration. Finally all accelerators are put in normal mode by sending the normal command to each of them. Now the chain is ready to process data. Figure 3.6 shows all the described transfers and mode changes of this stage.

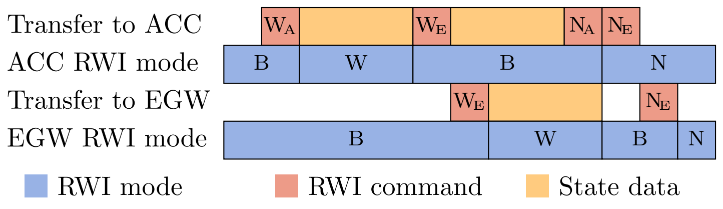


Figure 3.6: Transfers and RWI modes during restore stage

Process stage

When the state of the accelerators is restored and the exit gateway is configured, the accelerator can be used to process data. Figure 3.7 shows that the entry gateway will send the data to the accelerator, it gets processed and the accelerator sends it to the exit gateway. The exit gateway knows who the consumer is and will send it the processed data.

When all the data send by the gateway, it can immediately start with the commands for the state-save. Because the accelerators will finish their calculation, before the RWI accepts the command, because the RWI waits for the done signal from the accelerator. This means that the exit gateway no longer needs to signal the entry gateway that all data is processed and that it may perform a state-save.

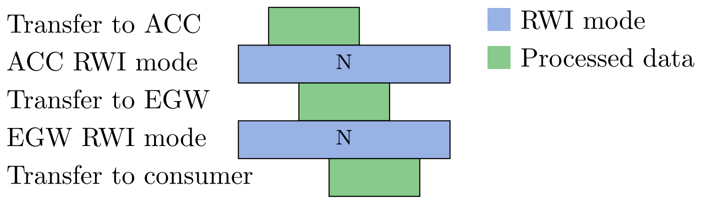


Figure 3.7: Transfers and RWI modes during process stage

Save stage

Finally the state of the accelerators is saved. This is shown in Figure 3.8. We will use the exit gateway to send the state to the entry gateway. So the entry gateway needs to be configured. In order to do this we put the accelerator in bypass by sending the bypass command. Next we send a write command to the exit gateway followed by the configuration data. We put the exit gateway in normal mode by sending the normal command. Now the exit gateway is configured and ready to receive state and send it to the entry gateway. So we send a read command to the accelerator, and the accelerator will start sending its state to the exit gateway. The exit gateway will then send the state to the entry gateway. The entry gateway will store the state and will use this in the next restore-save sequence.

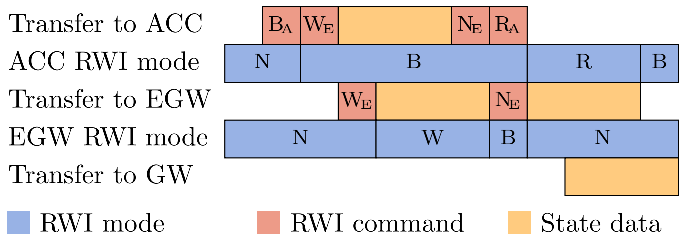


Figure 3.8: Transfers and RWI modes during save stage

Chapter 4

Dataflow Analysis

In this chapter we will describe the dataflow model of the previous architecture and we will propose a dataflow model for the proposed architecture. The first section in this chapter will introduce the dataflow model of the previous architecture. In the second section a new dataflow model is proposed that incorporates the changes in the proposed hardware architecture. The last section presents an abstraction of the proposed dataflow model. This chapter will require a basic knowledge of dataflow modelling.

Temporal analysis is an important part of real-time system design. It is used to give temporal guaranties over the design. Dataflow is one of the exiting analysis techniques. It is typically used to calculate worse case throughput, or to find optimal buffer sizes under throughput constraints. In [2] a new use of dataflow analysis is presented, the calculation of optimal packet sizes. The paper gives techniques to find optimal packet sizes under throughput constraints. The packet sizes dictate the granularity at which state-saves are performed. Small packet sizes result in high state-save overhead, while large packet sizes will increase latency and require bigger buffers. The ability to find optimum packet sizes is a nice addition to dataflow modelling, which will lead to better optimized designs.

4.1 Previous Accelerator Sharing Models

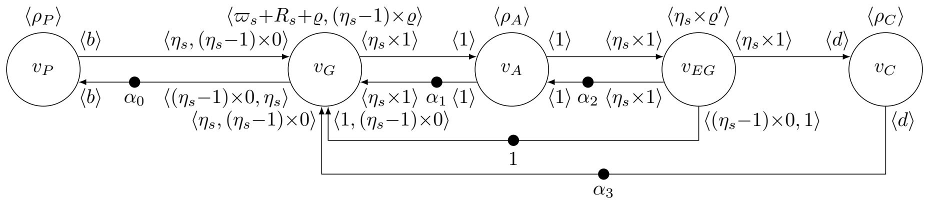
In this section we will explain the models that are used with the previous architecture. These models will later form the basis of the proposed models.

Figure 4.1 shows the CSDF model of the previous architecture. This model was introduced in [2]. The model contains 5 actors, corresponding to the producer, the entry gateway, an accelerator, the exit gateway and the consumer.

This model only has one accelerator, but multiple accelerators can be added in between v_G and v_{EG}.

First we explain what all the edges represent, and after this we explain the meaning of all symbols. The edge and the back edge between v_P and v_G represent the CFIFO buffer between the producing CPU and the entry gateway CPU. The variable α_0 indicates the number of initial tokens on edge (v_G,v_P) which corresponds with the capacity of the CFIFO between v_P and v_G. The edge and back edge from v_G to v_A and from v_A to v_{EG} form the accelerator pipeline. The edges represent the credit based communication between the components, that prevents the overflow of the local accelerator FIFOs. The edge from v_{EG} to v_G represents the synchronisation between the two that occurs when the entry gateway waits until all the data is processed by the accelerator chain. The edge from v_{EG} to v_C shows that the CFIFO of the consumer is filled by the exit gateway. However the back edge is from v_C to v_G, because the entry gateway has to reserve space in the CFIFO before the exit gateway can write to this CFIFO.

Symbols b, η_s and d are packet sizes of the producer, gateway and consumer respectively. The buffer size of the CFIFO between the producer and the gateway is α_0, and between the gateway and the consumer is α_3. The size of the hardware buffers between the entry gateway and the accelerator is α_1, and between de accelerator and the exit gateway is α_2. The firing durations of the producer, accelerator and consumer are ρ_P, ρ_A and ρ_C respectively, which correspond to the execution times of these components in the implementation. The variable ϱ represents the time that the gateway needs to send a sample to the accelerator. The variable ϱ' represents the time that the exit gateway needs to send a sample to the consumer. R_s represents the time that the gateway needs in order to reconfigure the accelerators. Lastly ϖ_s represents the maximum delay that the other streams introduce when the accelerator is shared.



$\langle\rho_P\rangle$
$\langle\varpi_s{+}R_s{+}\varrho, (\eta_s{-}1){\times}\varrho\rangle$
$\langle\rho_A\rangle$
$\langle\eta_s{\times}\varrho'\rangle$
$\langle\rho_C\rangle$
$\langle b\rangle$
$\langle\eta_s, (\eta_s{-}1){\times}0\rangle$
$\langle\eta_s{\times}1\rangle$
$\langle 1\rangle$
$\langle 1\rangle$
$\langle\eta_s{\times}1\rangle$
$\langle\eta_s{\times}1\rangle$
$\langle d\rangle$
v_P
v_G
v_A
v_{EG}
v_C
$\langle b\rangle$
α_0
$\langle(\eta_s{-}1){\times}0, \eta_s\rangle$
$\langle\eta_s{\times}1\rangle$
α_1
$\langle 1\rangle$
$\langle 1\rangle$
α_2
$\langle\eta_s{\times}1\rangle$
$\langle\eta_s, (\eta_s{-}1){\times}0\rangle$
$\langle 1, (\eta_s{-}1){\times}0\rangle$
$\langle(\eta_s{-}1){\times}0, 1\rangle$
$\langle d\rangle$
1
α_3

Figure 4.1: Previous CSDF model

In [2] an abstraction of the CSDF model is made. The next paragraph will describe how this is done, later the same methods are used to make the abstraction.

Figure 4.2 shows the SDF model that is an abstraction of the CSDF model. The differences are that the model is no longer a CSDF model and that actors v_G, v_A and v_{EG}, and the edges between them are combined into v_S. This is done by creating an execution schedule for the gateways and the accelerator. This execution schedule is then used to bound the execution time of v_S.

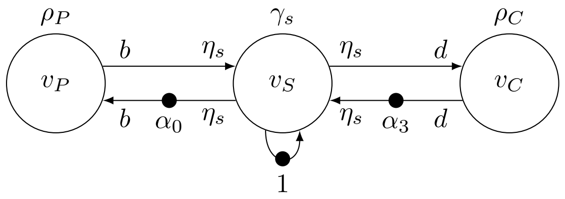


Figure 4.2: Previous SDF model

The following equations are used in [2] to prove that the abstraction holds, Equation (4.1), Equation (4.2) and Equation (4.3). The multiple streams that managed by the gateway are denoted with the set S, where s is one of those streams. Where τ_s is the time it takes to process η_s elements from stream s on the accelerators, including state-save overhead, R_s. The delay an other stream can introduce due to sharing is ϖ_s. The total processing time of a stream is γ_s, including the processing times from other streams that share the accelerator.

$$\tau_s \leq \hat{\tau}_s = R_s + (\eta_s + 2) \cdot \max(\varrho, \rho_A, \varrho') \tag{4.1}$$

$$\varpi_s \leq \hat{\varpi}_s = \sum_{i \in S \setminus s} \hat{\tau}_i \tag{4.2}$$

$$\gamma_s = \hat{\tau}_s + \hat{\varpi}_s = \sum_{i \in S} \hat{\tau}_i \tag{4.3}$$

Equation (4.1) shows that τ_s is dependent on the state-save overhead (R_s) and the packet size, η_s. The minimum processing time of one element in this pipeline can be over-approximated by $\max(\varrho, \rho_A, \varrho')$. The +2 comes from the fact that v_G, v_A, v_{EG}, and the edges between them form a pipeline. So when v_G produces a token, it still needs to pass v_A and v_{EG} resulting in a delay equal to two additional firings. Equation (4.2) shows that when sharing an accelerator, the maximum time that a stream has to wait is the sum of all other streams. Finally Equation (4.3) concludes that the total processing time is the sum of all individual stream processing times.

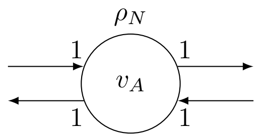
4.2 Proposed CSDF model

In this section we present a CSDF model for the proposed architecture.

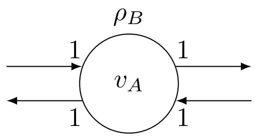
A new dataflow model is proposed because the changes to the hardware architecture made the existing model invalid. The proposed model respects the changes made in the hardware. The two key differences between de previous and the proposed architecture are: state is transferred via the Nebula ring network and the entry gateway no longer checks if all the data is processed by the accelerators. The next paragraphs describe how these differences are included in the dataflow model.

Because the state and the data are now going over the same network it makes sense to model this as well. As the figures in Section 3.3.3 show, performing a state-save involves sending a complex sequence of data and commands to the accelerators and thus constantly changing the mode of the RWI.

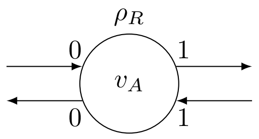
Figure 4.3 shows the firing duration and, production and consumption quanta for all the different situations an accelerator and its RWI can be in.



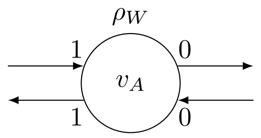
(a) Data in normal mode



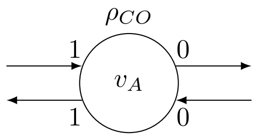
(b) Data in bypass mode



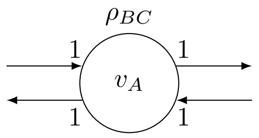
(c) Data in read mode



(d) Data in write mode



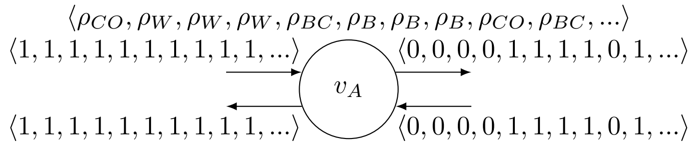
(e) Command for this accelerator



(f) Command not for this accelerator

Figure 4.3: SDF models of the different accelerator behaviours

Note that the behaviour is dependent on the mode, which is changed with the commands. So it may look like we are trying to model data dependant behaviour in CSDF, which is impossible because a firing of an actor is only dependant on the arrival of tokens on its edges. However, because the exact sequence of commands and thus mode changes are known, it can be modelled as a CSDF. This is done by setting the firing duration and quanta vectors so that they mimic the behaviour of a state-save cycle. An example of this is given in Figure 4.4. It shows the first 10 situations the accelerator is in. These correspond with the actions described in Section 3.3.3.

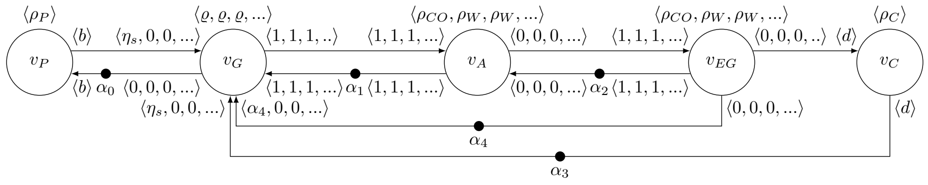

$\langle \rho_{CO}, \rho_W, \rho_W, \rho_W, \rho_{BC}, \rho_B, \rho_B, \rho_B, \rho_{CO}, \rho_{BC}, ... \rangle$
$\langle 1,1,1,1,1,1,1,1,1,1,... \rangle$
$\langle 0,0,0,0,1,1,1,1,0,1,... \rangle$
v_A
$\langle 1,1,1,1,1,1,1,1,1,1,... \rangle$
$\langle 0,0,0,0,1,1,1,1,0,1,... \rangle$
Figure 4.4: CSDF model of an accelerators restore-save cycle

Not only the accelerator has a different behaviour in specific situations, also the entry gateway and exit gateway have changing behaviours. These are also deterministic and can be modelled with long firing duration and quanta vectors. Because the data and state are both send via the Nebula ring network, we assume that both will take ϱ time. We assume the same for the exit gateway, sending data or state will both take ϱ' time.

The entry gateway no longer checks if all data is processed by the accelerators before it performs a state-save. Therefore the exit gateway no longer needs to signal the entry gateway. The signalling was represented by an edge from v_{EG} to v_G in the previous model, so that edge can be removed.

However because the state is now transferred via the ring network, an accelerator will send its state to the exit gateway. The exit gateway will send it to the entry gateway. Because the state is needed when the state is restored by the entry gateway, therefore the entry gateway is dependent on the exit gateway. So the entry gateway not only needs to check if there is data from the producer and reserve space in the buffer of the consumer, it also needs to check if the previous state is received. This is represented by an edge between v_{EG} and v_G, where α_4 represents the number of transfers needed in order to save the state.

The changes result in the proposed CSDF model, see Figure 4.5. It is a complex model due to the exact modelling of the restore-save cycle. While this will model the restore-save cycle correct, it does not model delays due to sharing the accelerator with other streams. In Section 4.3 we propose an additional model that is an abstraction of the CSDF model and does include the delays due to accelerator sharing.


$\langle \rho_P \rangle$
$\langle \varrho, \varrho, \varrho, ... \rangle$
$\langle \rho_{CO}, \rho_W, \rho_W, ... \rangle$
$\langle \rho_{CO}, \rho_W, \rho_W, ... \rangle$
$\langle \rho_C \rangle$
$\langle b \rangle$
$\langle \eta_s, 0, 0, ... \rangle$
$\langle 1, 1, 1, .. \rangle$
$\langle 1, 1, 1, ... \rangle$
$\langle 0, 0, 0, ... \rangle$
$\langle 1, 1, 1, ... \rangle$
$\langle 0, 0, 0, .. \rangle$
$\langle d \rangle$
v_P
v_G
v_A
v_{EG}
v_C
$\langle b \rangle$
α_0
$\langle 0, 0, 0, ... \rangle$
$\langle 1, 1, 1, ... \rangle$
α_1
$\langle 1, 1, 1, ... \rangle$
$\langle 0, 0, 0, ... \rangle$
α_2
$\langle 1, 1, 1, ... \rangle$
$\langle \eta_s, 0, 0, ... \rangle$
$\langle \alpha_4, 0, 0, ... \rangle$
$\langle 0, 0, 0, ... \rangle$
$\langle d \rangle$
α_4
α_3
Figure 4.5: Proposed CSDF model

4.3 Proposed SDF model

In this section we will present an SDF model which is an abstract version of the CSDF model proposed in Section 4.2, which also includes the delay that is a result of the sharing of the accelerators. First we will create an SDF from the CSDF model by means of abstraction. Then we will add the delay that is the result of sharing of the accelerators.

The abstraction made in the section is based on the refinement theory presented in [15].

In order to simplify the CSDF model, we will combine actors v_G, v_A and v_{EG}, as well as the edges between these actors into one actor. We will also limit the number of phases to one, which results in a SDF model. In order to perform the abstraction, a schedule is created to visualize the firings. Figure 4.6 shows the schedule of the state-save sequence described in Section 3.3.3. As shown in Figure 4.6 the duration of a restore-save cycle depends on the number of transfers needed to restore the accelerator state, the number of data sample that are processed, the number of transfers needed to save the state and the number of commands. The total number of restore transfers is L, where l_i is the number of transfers for a partial restore. The number of data samples that are processed is denoted with η_s. This is also called the packet size. The total number of save transfers will be called J, where j_i is the number of transfers for a partial save. The total number of commands is K. The firing durations in Figure 4.6 are all shown as equal, this is probably not true. However later we will over-approximate all the different firing durations into one uniform time step, and this will correspond with the uniform firing durations in Figure 4.6. This over-approximation is an valid abstraction, base on the the-earlier-the-better refinement. This refinement states that shorter firing duration can never result in a worse schedule.

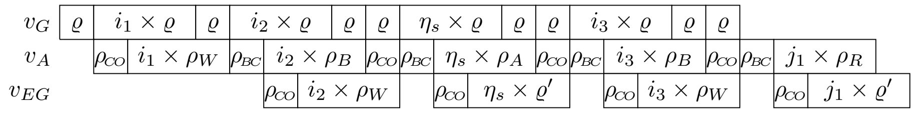


Figure 4.6: Typical schedule of the CSDF model

We can now derive an upper bound for the time it takes to perform a restore-save cycle. Equation (4.4) gives an upper bound on the duration of the restore-save cycle. The maximum possible firing duration of every actor is used to create an uniform time step. This time step is the worse case firing duration of every possible phase of actors v_G, v_A and v_{EG}. With this uniform time step we can create a schedule just like Figure 4.6. The number of uniform time steps needed to perform the restore-save sequence is equal to the number of transfers and the number of transfers is equal to the sum of L, J, K and η_s. Because production by v_G still needs to go through v_A and v_{EG}, two additional time steps are necessary, which result in the +2 in Equation (4.4).

$$\tau_s \leq \hat{\tau}_s = (L + J + K + \eta_s + 2) \cdot \max(\varrho, \rho_A, \rho_B, \rho_W, \rho_R, \rho_{CO}, \rho_{BC}, \varrho') \quad (4.4)$$

Now we can also include the delay introduced by the sharing of the accelerator, which is equal to the sum of the processing time of a packet of each of the other streams. This results in Equation (4.2) which is the same formula used in [2]. The total firing duration can be calculated with Equation (4.3), as is done in [2].

The resulting SDF model is also equal to the model presented in [2], it only differs in the way how $\hat{\tau}_s$ is calculated, and will thus result in another γ_s. The SDF model is shown in Figure 4.7.

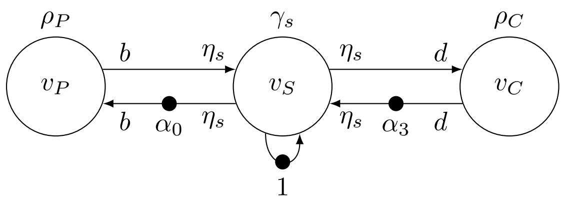


Figure 4.7: Proposed SDF model

The methods to determine the optimal packet size that are described in [2] can still be used. Because the topology of the proposed SDF model is equal to the topology of the SDF model in [2] and γ_s is still a function of η_s.

Chapter 5

Evaluation

This chapter presents an evaluation of the work that has been presented in the previous chapter. The first section will describe the test case used for this evaluation. The following section presents the performance of the implementation in terms of speed and its cost in area. The results are compared with the previous implementation. The last section evaluates the proposed dataflow models. We will compare the throughput and buffer size with the previous models.

5.1 The benchmark application

In this section we will describe the application that is used to evaluate the implementation. The same application is used as in [1]. This will allow us to compare the results of the proposed implementation with the results published in [1]. By using the same application a fair comparison can be made.

The application used in [1], is a stereo FM demodulator. This application is chosen because it can be split in 4 parts that perform similar operations on a data steam. This means the 4 parts can utilize the same accelerators, and therefore these accelerators can be shared. This makes it a suitable application to demonstrate the sharing of accelerator. Figure 5.1 shows a block diagram of the stereo FM demodulation. The radio signal is split into two streams, one for the left and one for the right audio channel. There are two steps performed for both channels. First a channel is mixed with the appropriate carrier frequency. After mixing, the signal is low-pass filtered and down sampled. The second step is FM demodulation, low-pass filtering and down sampling. So in total there are 4 parts, two steps for two channels. We will not go into detail how the different blocks work exactly. It is sufficient to know that the operations done in one part, can be performed by an FIR filter and COordinate Rotation DIgital Computer (CORDIC) accelerator. The CORDIC is used to do the mixing or the FM demodulation. The FIR filter is used to perform the low-pass filtering and the down-sampling.

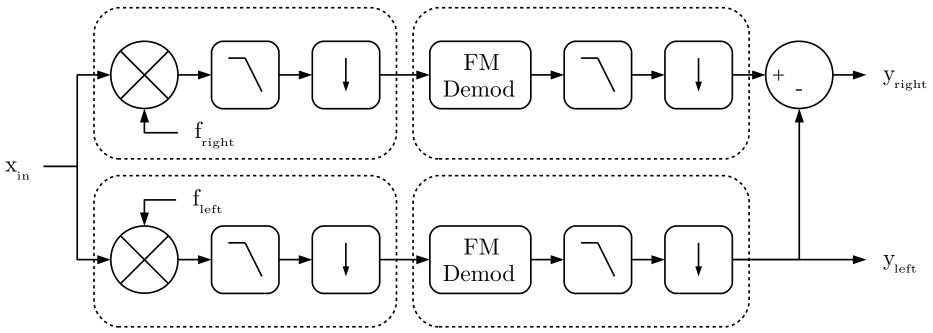


Figure 5.1: Stereo FM demodulation block diagram

Figure 5.2 shows the architecture that is used to evaluate the proposed accelerator sharing architecture. There are 4 CPUs. The first is the producer ($\mathrm{CPU_P}$), it will send a generated radio signal to the gateway. The second CPU is used as entry gateway ($\mathrm{CPU_{GW}}$). The $\mathrm{CPU_M}$ is performing the subtraction and is interleaving the two audio channels. The consumer ($\mathrm{CPU_C}$) is used to save the produced audio signal. The gateway manages two accelerators, a CORDIC accelerator ($\mathrm{ACC_{COR}}$) and an FIR accelerator ($\mathrm{ACC_{FIR}}$). There are multiple RWIs that are used to configure the accelerators.

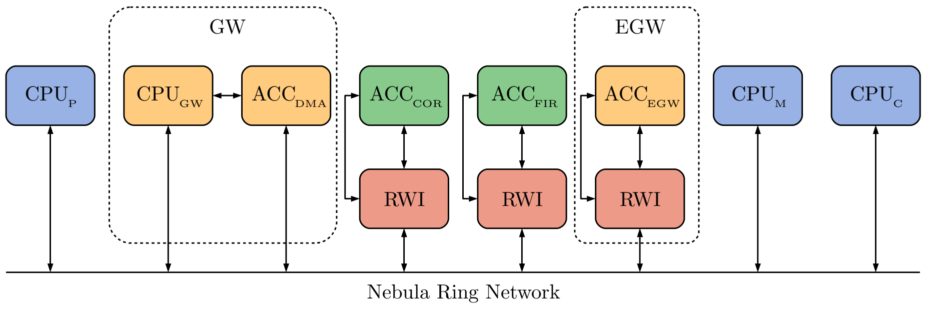


Figure 5.2: High level system overview used for evaluation

The speed of the application is measured by processing 100000 packets and by measuring the time it took to process these packets. So the producer will start a timer and sends 100000 packets to the gateway. The consumer counts the number of packets received and after 100000 packets it will stop the timer. This will result in a measurement in seconds per 100000 packets. These measurements are used to derive the least squares approximation. The measurements and approximation are transformed to 100000 packets per second by taking the multiplicative inverse. Finally we multiply with the number of samples per packet to get the 100000 samples per second. Because the number of samples per packet, also called packet size, is variable, the results will show the throughput for multiple different packet sizes.

The size of the implementation is taken from the hardware synthesis reports. These reports list the number of used slice registers, Look-Up Tables (LUTs) and LUTs used as RAM (LUTRAMs). These are the basic components that are used to create the architecture. The report shows the statistics of the individual subsystems within the system. We include in our comparison only the subsystems that have been adapted by us.

5.2 Evaluation of the implementation

In this section we will present the measured speed and size. We will compare the result with the previous implementation.

5.2.1 Speed

Figure 5.3 shows the measured throughput and the approximation, for the previous and proposed architecture. The measurements are indicated with the dots, while the approximation is shown with the line. The proposed architecture is approximate 2.5 times faster.

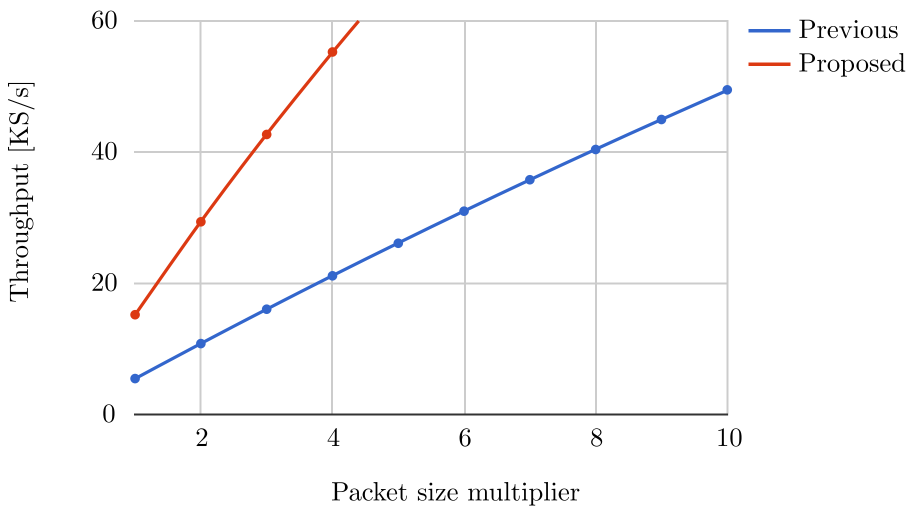


Figure 5.3: Throughput comparison

The approximations of the throughput are used to create Figure 5.4. It shows the utilisation of the accelerators. This is the percentage of time that the accelerators are processing data. The remaining time is used to perform state saves and restores. The best case utilization is also shown. This is the ratio between the data that is processed and the total data sent to the accelerators, consisting of processed data and state data. The best case utilization is shown in Equation (5.1). Note that the number of data transfers is dependent on the packet size multiplier. The best case utilization is made under the assumption that transferring state is as fast as transferring data.

$$\text{BestCaseUtilisation} = \frac{\text{DataTransfers}(\text{PacketSizeMultiplier})}{\text{DataTransfers}(\text{PacketSizeMultiplier}) + \text{StateTransfers}} \tag{5.1}$$

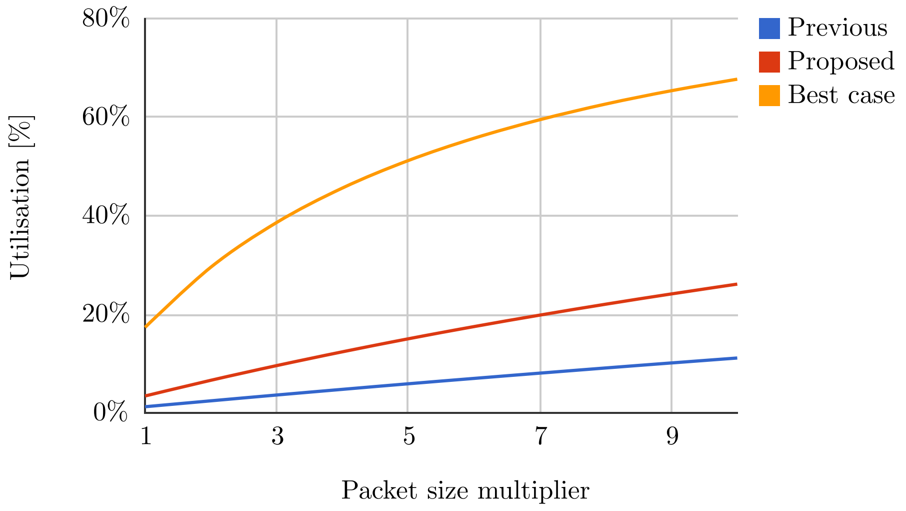


Figure 5.4: Utilisation comparison

Figure 5.4 shows that the proposed architecture is better utilizing the accelerators. However, the proposed utilization is still not close the the best case utilization.

The approximations of the throughput can also be used to give an approximation of the time needed to transfer data and to transfer state. Table 5.1 shows the cycle times for different transfers. The data transfer time did not change. This is as expected, because the way data is transferred did not change. The approximated time to transfer one 32-bit state word in the previous architecture is 26.21 cycles. This is in the same ballpark as the 23 cycles measured in [1]. The proposed architecture needs 9.25 cycles to transfer one 32-bit state word. This is a reduction of 65% compared to the approximated cycles per transfer of the previous architecture. While this is a significant reduction it is still not close to the data transfer time. This is due to the overhead introduced by small commands to the RWIs, the fact that small transfers do not get burst transferred by the DMA and the large DMA configure overhead for small transfers.

Cycles per transfer	Data transfer	State transfer
Previous	1.56	26.21
Proposed	1.56	9.25

Table 5.1: Approximated transfer durations

5.2.2 Size

Figure 5.5 show the hardware cost of the subsystems that have been changed by us. The proposed components also show how they compare relative to the previous components. The costs named CPU represents the hardware cost for a CPU to connect to the Nebula ring network. The cost of a CPU are not shown because the CPU itself did not change. The costs named ACC is the cost for an accelerator to connect to the Nebula ring network. This is without the cost of the accelerator itself. The cost named ACC Interface is only shown once, because the proposed architecture does not have an ACC Interface.

In the proposed architecture the ACC LUTRAM usage is almost doubled. This is because the RWI uses the address to determine if the data is a command or not. So now the address is buffered together with the data. The previous architecture did not use the address so the synthesise optimizer could remove the address buffer. This is why the previous architecture uses less LUTRAMs.

The LUT usage of the ACC is increased due to the additional RWI logic. The LUT and LUTRAM increase of the proposed CPU should not have happened. A possible explanation can be the usage of the new tool chain used to build the proposed architecture.

The overall slice reg decrease is probably because the measurements in [1] had some debug features enabled in the ring shell. These features are no longer enabled in the ring shell version used in the proposed architecture.

Figure 5.6 shows the total cost of the changed parts of the previous and proposed architecture. The total cost of the previous architecture is the sum of 4 CPUs, 4 ACCs and an ACC Interface, while the total cost of the proposed architecture is the sum of 4 CPUs and 4 ACCs. Again there is a decrease in slice reg usage, and an increase in LUT and LUTRAM usage.

Until now we only compared the proposed architecture to the previous architecture, and while the usage of some hardware components is increased, and others decreased, the overall there is a decrease of 91%. This is based on the total number of components in the evaluation application. The previous architecture had 6524 components in total while the proposed architecture has a total of 5915 components.

If we try to compensate for the fact that the measurements of previous architecture included debug features, then we estimate that the proposed architecture

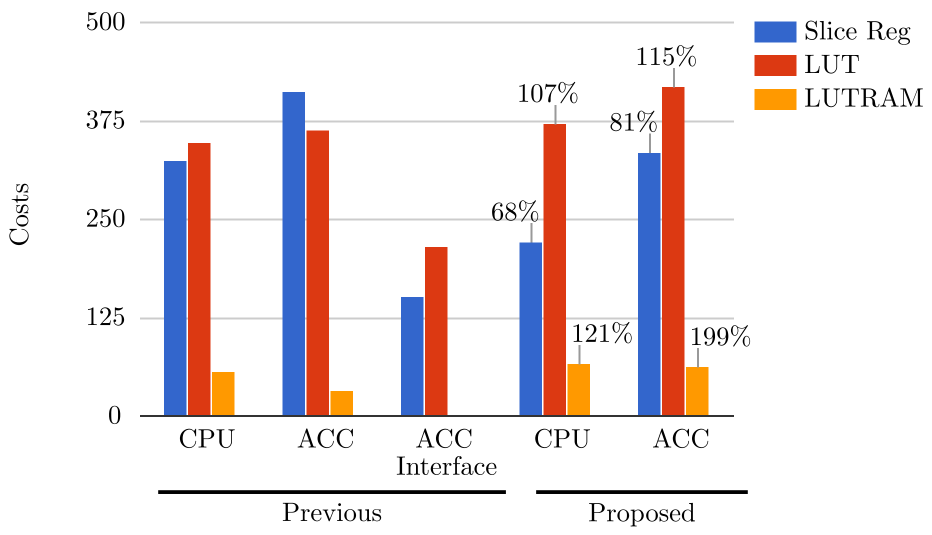


Figure 5.5: Hardware cost for individual components

requires 10% more hardware than the previous architecture. We can therefore conclude that the utilisation of hardware accelerators is increased with 150% at the cost of an 10% increase of the required hardware.

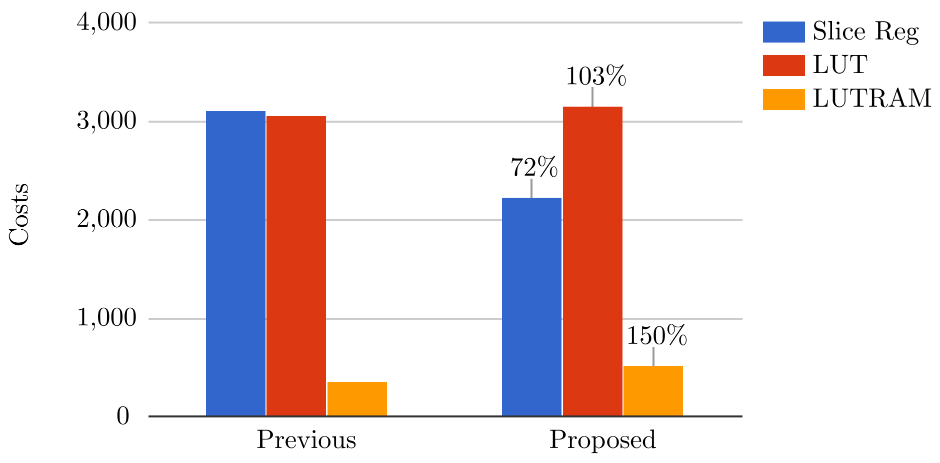


Figure 5.6: Hardware cost comparison

5.3 Evaluation of the dataflow models

In this section we will compare the proposed SDF model with the SDF model from [2]. We will compare throughput, buffer sizes and packet sizes. Both models have the same topology, but a different γ_s. This will make the comparisons straightforward.

5.3.1 Throughput

The throughput of an application can be determined with a dataflow model. When an application has hardware accelerators that are shared, the dataflow model will have multiple instantiations of the proposed SDF accelerator sharing model. The throughput of an application will be dependent on the whole model, and not only on the proposed SDF accelerator sharing model.

However due to the monotonicity of dataflow models, a shorter firing duration can never result in a later production. This also means that a shorter firing duration can never result in a worse throughput.

And because the proposed SDF model only differers in firing duration, the following can be concluded: if the γ_s of the proposed model is smaller than the γ_s of the previous model, then the throughput of the proposed model cannot be worse.

Without the value of γ_s, we cannot give a definitive answer. In order to calculate γ_s, we need to know multiple parameters of the system that could be measured or estimated. However these experiments have not been carried out. We do recognise the importance of evaluation of the dataflow models, so we suggested this as future work, Section 6.2.

5.3.2 Buffer size

Just as the throughput the minimal buffer sizes can be determined with a dataflow model. The minimal buffer sizes of an application depend on the whole model, and not only on the proposed SDF accelerator sharing model.

Because a shorter firing duration cannot lead to worse throughput, the buffer sizes do not need to be increased because the throughput did not decrease. So if the firing duration γ_s decreased, the buffer sizes do not need to be increased and might even decrease.

5.3.3 Packet size

Methods to derive the packet size of the shared streams are described in [2]. These methods are again dependent on the whole dataflow graph, and not only on the proposed SDF model for shared accelerators. However it is still possible to say something about packet sizes of the proposed model.

When firing durations decrease, the previous schedule is still a valid schedule. So the previous packet sizes are also still valid. So shorter firing durations cannot result in worse packet sizes, because the previous packet sizes will always be valid.

So if the firing duration γ_s is decreased, the packet sizes did not increase and might even decrease.

Chapter 6

Conclusion and Future Work

In this chapter we will present the conclusions and future work.

6.1 Conclusion

In this thesis we tried to find an answer for the following research question: **How to reduce state-save overhead for shared accelerators in an MPSoC with a ring NoC?** We will answer this question by answering the following sub-questions.

What is the cause for the large state-save overhead?
In Section 3.1.3 we identified that the large state-save overhead was primary due to two limiting factors. One factor was the slow configuration interface that was used to access the state of the accelerators. The slowness is caused by the high transfer overhead of the PLB. The other factor was the software overhead introduced by the template method programming model. In this programming model the code for a state-save is split across many small functions, which all have their calling cost, but which also prevent global optimization.

Is it possible to introduce an architecture that reduces the state-save overhead?
In Section 3.3 we propose an architecture that reduces the effects of the state-save overhead issues that were identified in Section 3.1.3, while keeping the hardware cost low. The presented architecture makes use of the existing ring network to access state in the accelerators. This allows the removal of the slow configuration interface, resulting in a hardware cost reduction. The existing DMA can be used to write the state to the accelerators, which increases the transfer speed and thus lowers the state-save overhead. The existing ring network is extended with RWIs to support state transfers from and to the accel-

erators. The state-save overhead caused by the software is reduced by changing the template method pattern to an interpreter pattern. By creating an intermediate representation of the action performed during a state-save, it is possible to reduce the amount of code that is executed to perform a state-save and allows us to make global optimizations that were not possible before.

What is the performance of this architecture?
In Section 5.2.1 we perform a number of measurements that capture the performance of the architecture. The proposed architecture is 2.5 times faster than the previous architecture. While this is a decent increase, measurements also indicate that there is still a significant overhead when transferring state. This is caused by small commands for the RWIs that are interleaved with the state data, resulting in multiple small transfers. These small transfer are not handled well by the DMA because only blocks of 8 words are transmitted as a burst and starting a DMA for small transfers has enormous overhead. The DMA was originally added to achieve better PLB read access, but a better solution would be to look for a PLB alternative. The newer Advanced eXtensible Interface (AXI) might be a faster alternative.

What is the hardware cost of this architecture?
In Section 5.2.2 we measured the hardware cost of the proposed architecture and compared it with the previous architecture. The results showed that the hardware cost was decreased for some components while for others it was increased. Overall we estimated a 10% increase of hardware usage. This 10% increase is easily justified by the performance increase of 150%.

Is it possible to model the temporal behaviour of the architecture?
In Chapter 4 we show that is possible to model the temporal behaviour of the proposed architecture with dataflow model. A CSDF model of the accelerator sharing architecture is presented in Section 4.2. An abstraction of the CSDF model is made in Section 4.3, the resulting SDF model is very similar to the SDF model of the previous architecture. The presented models can be used to derive throughput, buffer sizes and packet sizes.

Do the models capture the gain in performance?
In Section 5.3 we compared the proposed model with the previous model. Only the abstracted models were compared. The abstracted models only differ in one parameter, so a comparison was straightforward. We concluded that when this parameter is decreased, the system cannot have worse throughput, worse buffer sizes and worse packet sizes. So decreasing this parameter can possibly result in better throughput, better buffer sizes and better packet sizes.

6.2 Future Work

In this section we describe a number of techniques that could improve the proposed architecture.

6.2.1 Stack machine entry gateway

The proposed architecture for the entry gateway contains a MicroBlaze CPU. This CPU was chosen because the architecture already contained them. So they could be added easily. A disadvantage of this CPU is its size and its slow connection to the ring network. The Ring DMA was added to improve the connection to the Nebula ring network, but this made the design even bigger.

Figure 6.1 shows a global overview for an alternative entry gateway architecture that could be smaller and faster than the proposed gateway architecture. The core of the new entry gateway is a stack based CPU called J1 [20]. The J1 is a simple and small CPU design, less then 200 lines of Verilog [20]. Its simple design makes it easy to customize. With some modification it would be possible to connect it to the Nebula ring network. Because this CPU is not connected to the memory and ring network via PLB, it will have faster memory access than a MicroBlaze. It might even be possible to let the CPU do the memory transfers, so that the DMA can be removed, reducing the hardware cost even further. It might be necessary to add hardware loop support, so that writing multiple words to the Nebula ring network is as fast as DMA. The code can be loaded into the code memory via the Nebula ring network. The existing CFIFO protocol can be used to communicate with other CPU cores in the system.

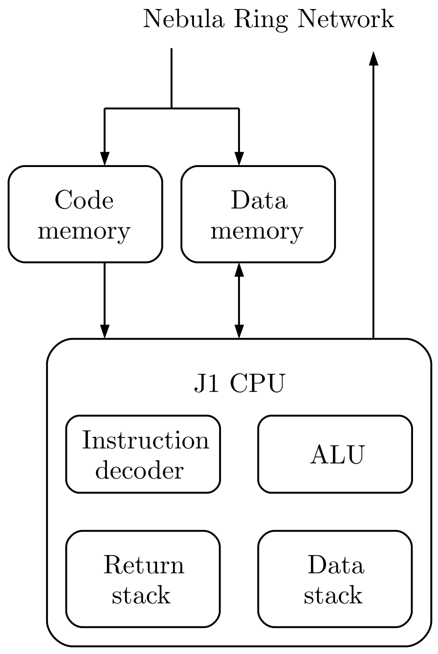


Figure 6.1: Alternative entry gateway architecture with a J1 CPU

6.2.2 Nebula ring topology

As mentioned in Section 3.2, a disadvantage of the proposed state-save mechanism is that the state is transferred around the ring network when it is saved into the memory of the entry gateway. This will reduce the remaining bandwidth of the ring network, which will affect all the CPUs and accelerators between the exit gateway and the entry gateway.

One possible solution is to create a separate ring for accelerators. Figure 6.2 shows such an architecture. The entry gateway and exit gateway are merged into a bridge and the bridge provides access to the accelerators. This will eliminate the bandwidth penalty of state saving, because state saving will no longer use the primary ring network.

However because the bridge is now connected to two ring networks it can of receiving twice as much data. This is a problem because both ring networks might want to write to the local memory of the bridge. Remember that the ring network is designed in such a way that it guarantees that transfers are never lost. So data that is being sent to the local memory will always be written to the local memory. This means that problems occur when both ring networks try to write to the local memory. Because the memory can only store one transfer, while two could arrive. It is simply not possible to connect both ring networks to one memory and guarantee that all transfers are stored. A possible solution is to add a second local memory, so local memory one for each ring network interface, however this would increase the hardware cost, due to the extra local memory. An other disadvantage of this architecture is that the data that is processed by the accelerators is first stored in local memory the bridge, and then copied by the bridge to the consumer. This means that some local memory of the bridge is used as a temporary buffer, and that the bridge spends some time to copying data form its local memory to the memory of the consumer. This is a wast of memory and processor time.

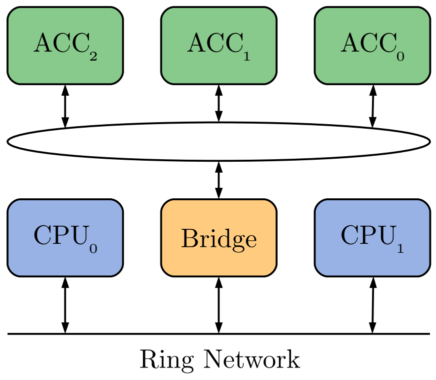


Figure 6.2: Alternative ring network topology

An alternative architecture is shown in Figure 6.3, which is capable of directly writing the data from the accelerators to the consumer.

In this architecture the accelerator ring is limited to token based communication. So transfers over accelerator ring will never try to write to the local memory of the gateway directly, and thus it is possible to use only one local memory.

A new block is introduced, the injector and extractor. An extractor is added to the accelerator ring, and the injector is added to the primary ring, just before the entry gateway.

In order to get data from the accelerator ring to the primary ring we will use the extractor and injector. Because the entry gateway is in control of the accelerators, it knows when an accelerator produces data that needs to be sent to a consumer on the primary ring network. So the entry gateway will configure the accelerator to send the data to extractor. The gateway will also configure the extractor and injector to accept a number of words and send it to the location of the consumer. So now an accelerator can send its data to the extractor and the injector injects this into the primary ring network to the consumer. The the extractor and injector can also be used to transfer state to the entry gateway, by making the entry gateway the consumer.

This approach is possible because the accelerators use credit based communication, so when there is for example a lot of traffic between CPU_0 and CPU_1, the injector does not have the full ring network bandwidth. As a result the injector will probably block the accelerators due to back pressure, because of the credit based communication.

The existing exit gateway is already very similar to the extractor and injector so a realisation of this idea should not be difficult. A disadvantage of this approach is that the CPUs can no longer use the accelerators directly, only via the entry gateway. However if an accelerator is used directly it means that it is not shared so there is no need for a gateway, and the accelerator could be placed on the primary ring network. This would make it directly accessible again. An additional advantage of this approach is that when there are no accelerators on the primary ring network, it can remove support for credit based communication reducing the hardware cost.

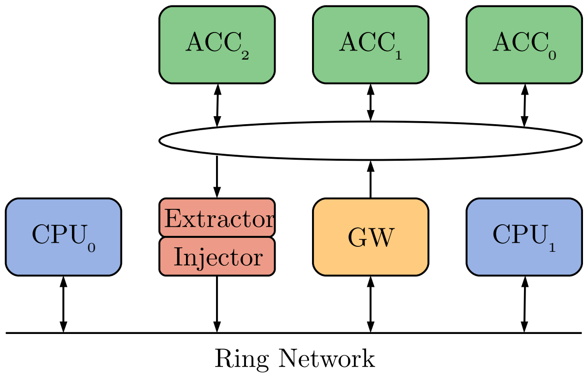


Figure 6.3: Alternative ring network topology using an injector and extractor

6.2.3 Evaluation of dataflow models

In this thesis we proposed new dataflow models for the proposed accelerator sharing architecture. While these models are correct, the accuracy is unknown. Knowing the accuracy of the model is important because it will be reflected in the accuracy of the derived buffer and packet sizes.

In order to determine the accuracy of the dataflow models a case study can be performed. This case study should try to create a model of an application by using estimated or measured values for the model parameters. The application could be the same as in Section 5.1. When a model is created, it can be used to derive maximum throughput, minimum buffer sizes and minimum packet sizes. These values can then be compared with measurements of the application.

Discrepancies in the measured and modelled results, should be explained. These discrepancies can also be used to identify problems in the models that could be improved. We expect that improvements can be made by better modelling the latency of the accelerator pipeline. It might also be possible to create a better abstraction then the proposed SDF model, because a better abstraction will be less pessimistic, and will thus have better accuracy.

Bibliography

[1] G. G. Wevers, "Hardware accelerator sharing within an mpsoc with a connectionless noc," Master's thesis, University of Twente, September 2014.

[2] B. H. J. Dekens, M. J. G. Bekooij, and G. J. M. Smit, "Real-time multiprocessor architecture for sharing stream processing accelerators," in *22nd Reconfigurable Architectures Workshop (RAW 2015), Hyderabad, India*, (USA), pp. 81–89, IEEE Computer Society, May 2015.

[3] "IBM 4764 PCI-X Cryptographic Coprocessor." Online at `http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml`, 2007.

[4] J. H. Rutgers, *Programming models for many-core architectures: a co-design approach.* PhD thesis, University of Twente, Enschede, May 2014.

[5] B. H. J. Dekens, P. S. Wilmanns, G. J. M. Smit, and M. J. G. Bekooij, "Low-cost guaranteed-throughput dual-ring communication infrastructure for heterogeneous mpsocs," in *2014 Conference on Design and Architectures for Signal and Image Processing, DASIP 2014*, (France), pp. 157–164, ECSI Media, October 2014.

[6] J. A. J. Leijten, J. L. V. Meerbergen, A. H. Timmer, and J. A. G. Jess, "Stream communication between real-time tasks in a high-performance multiprocessor," *Design, Automation & Test in Europe Conference & Exhibition*, vol. 0, p. 125, 1998.

[7] M. J. Rutten, J. T. J. van Eijndhoven, E. D. Pol Egbert, G. T. Jaspers, P. van der Wolf, O. P. Gangwal, and A. Timmer, "Eclipse: heterogeneous multiprocessor architecture for flexible media processing," in *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pp. 8 pp–, April 2002.

[8] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *Computers and Digital Techniques, IEE Proceedings* -, vol. 152, pp. 148–166, Mar 2005.

[9] J. P. H. M. Hausmans, S. J. Geuns, M. H. Wiggers, and M. J. G. Bekooij, "Dataflow analysis for multiprocessor systems with non-starvation-free

schedulers," in *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems*, (New York), pp. 13–22, ACM, June 2013.

[10] S. Chakraborty, S. Kunzli, and L. Thiele, "A general framework for analysing system properties in platform-based embedded system designs," in *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pp. 190–195, 2003.

[11] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi, "Cyclic dependencies in modular performance analysis," in *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, (New York, NY, USA), pp. 179–188, ACM, 2008.

[12] L. Thiele and N. Stoimenov, "Modular performance analysis of cyclic dataflow graphs," in *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, (New York, NY, USA), pp. 127–136, ACM, 2009.

[13] E. A. Lee and et al., "Synchronous data flow," 1987.

[14] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, B. Mesman, J. D. Mol, E. Stuijk, and J. V. Meerbergen, "Chapter 4 dataflow analysis for real-time embedded multiprocessor system design."

[15] J. P. H. M. Hausmans, *Abstractions for aperiodic multiprocessor scheduling of real-time stream processing applications.* PhD thesis, University of Twente, Enschede, the Netherlands, April 2015.

[16] G. Hoekstra, "Hardware accelerator integration in a connectionless network-on-chip," Master's thesis, University of Twente, 2013.

[17] W. Tong, O. Moreira, R. Nas, and K. van Berkel, "Hard-real-time scheduling on a weakly programmable multi-core processor with application to multi-standard channel decoding," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pp. 151–160, April 2012.

[18] A. Nieuwland, J. Kang, O. Gangwal, R. Sethuraman, N. Bus, K. Goossens, R. Peset Llopis, and P. Lippens, "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*, vol. 7, no. 3, pp. 233–270, 2002.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[20] J. Bowman, "J1: a small Forth CPU core for FPGAs," in *26th euroForth conference*, pp. 43–46, 2010.