NFComms: A synchronous communication framework for the CPU-NFP heterogeneous system

Submitted in fulfilment
of the requirements of the degree of

Doctor of Philosophy of Science

of Rhodes University

Sean Pennefather

Grahamstown, South Africa
June 2019

Abstract

This work explores the viability of using a Network Flow Processor (NFP), developed by Netronome, as a coprocessor for the construction of a CPU-NFP heterogeneous platform in the domain of general processing. When considering heterogeneous platforms involving architectures like the NFP, the communication framework provided is typically represented as virtual network interfaces and is thus not suitable for generic communication. To enable a CPU-NFP heterogeneous platform for use in the domain of general computing, a suitable generic communication framework is required.

A feasibility study for a suitable communication medium between the two candidate architectures showed that a generic framework that conforms to the mechanisms dictated by Communicating Sequential Processes is achievable. The resulting NFComms framework, which facilitates inter- and intra-architecture communication through the use of synchronous message passing, supports up to 16 unidirectional channels and includes queuing mechanisms for transparently supporting concurrent streams exceeding the channel count. The framework has a minimum latency of between 15.5 μs and 18 μs per synchronous transaction and can sustain a peak throughput of up to 30 Gbit/s. The framework also supports a runtime for interacting with the Go programming language, allowing user-space processes to subscribe channels to the framework for interacting with processes executing on the NFP.

The viability of utilising a heterogeneous CPU-NFP system for use in the domain of general and network computing was explored by introducing a set of problems or applications spanning general computing, and network processing. These were implemented on the heterogeneous architecture and benchmarked against equivalent CPU-only and CPU/GPU solutions. The results recorded were used to form an opinion on the viability of using an NFP for general processing.

It is the author's opinion that, beyond very specific use cases, it appears that the NFP-400 is not currently a viable solution as a coprocessor in the field of general computing. This does not mean that the proposed framework or the concept of a heterogeneous CPU-NFP system should be discarded as such a system does have acceptable use in the fields of network and stream processing. Additionally, when comparing the recorded limitations to those seen during the early stages of general purpose GPU development, it is clear that general processing on the NFP is currently in a similar state.

Acknowledgements

I would like to acknowledge the support received during this research. I would first like to give thanks to my supervisors Professors Karen Bradshaw and Barry Irwin as their guidance and support has been essential to the success of this research.

I am deeply indebted to my family for their continued love and support throughout the course of this degree, their care and aid has allowed me to focus on the completion of this thesis.

I am grateful to Netronome for their financial assistance towards this research as well as supplying the hardware necessary for the realisation of this work. Opinions expressed in this thesis and the conclusions arrived at, are my own, and are not necessarily to be attributed to Netronome.

Finally I would like to acknowledge the financial support of the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Easttel, Bright Ideas 39, THRIP and NRF SA (UID 75107). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the author(s) and that none of the above mentioned sponsors accept liability whatsoever in this regard.

Contents

List of Figures

Listings

List of Tables

List of Acronyms

ADSM Asymmetric Distributed Shared Memory

ALU Arithmetic Logic Unit

API Application Programming Interface

BAR0 Base Address Register 0

CCL Collective Communication Library

CCS Calculus of Computing Systems

CLS Cluster Local Scratch

CMD Command

CPP Command Push Pull

CPU Central Processing Unit

CRC Cyclic Redundancy Check

CSP Communicating Sequential Processes

CT Cluster Target

CTM Cluster Target Memory

CUDA Compute Unified Device Architecture

DDLP Data Link Layer Packet

DMA Direct Memory Access

DPI Deep Packet Inspection

DSL Domain Specific Language

DSM Distributed Shared Memory

EMEM External Memory

FPC Flow Processing Core

GCC GNU Compiler Collection

GPR General Purpose Register

GPGPU General Purpose GPU

GPU Graphics Processing Unit

HDSM Heterogeneous DSM

HPC High Performance Computing

HPU Handler Processing Unit

IHC Iterative Hill Climbing

IMEM Internal Memory

IOCTL Input-Output Control

IOMMU IO Memory Managment Unit

IPC Interprocess Communication

IXP Internet eXchange Processor

KPN Kahn Process Network

LTS Label Transition Diagram

MoC Model of Computation

MPI Message Passing Interface

MSI Message Signalled Interrupt

MSIX eXtended Message Signalled Interrupt

NBI Network Block Interface

NFP Network Flow Processor

NN Next Neighbour

NPU Network Processing Unit

NNUS Non-uniform Node Uniform System

OpenCL Open Computing Language

PCSA Probabilistic Counting with Stochastic Averaging

PF Physical Function

PISC Packet Instruction Set Computer

PPE Packet Processor Engine

PPS Packet Processing Stage

RIFFA Reusable Integration Framework for FPGA Accelerators

sPIN Streaming Processing In the Network

STD Standard Deviation

TLP Transaction Layer Packet

UVA Unified Virtual Addressing

UNNS Uniform Node Non-uniform System

1

Introduction

With the rise in the volume of network traffic and the increased complexity of network-based applications, the resource requirements associated with network processing have grown accordingly. To help facilitate the increasing demand for higher network throughput while supporting more complex and adaptive network protocols, server-based systems have begun to rely on dedicated Network Processing Units (NPUs) to handle network specific computations.

NPUs have been developed almost explicitly to target the domain of network processing with architectures designed to handle the high throughput and stream-based nature of network traffic (Wheeler, 2013). Such architectures have been tailored to handle high bandwidth while minimising latency throughout the processing pipeline, from the reception of network frames to their retransmission. Techniques such as memory partitioning and a high degree of concurrency in processing have been introduced to allow NPUs to reach throughput speeds of up to 400 Gbit/s (Netronome, 2014; Mellanox, 2017a).

Due to the advances in parallel and heterogeneous computing, NPU architectures are now also being explored for their use in alternative domains such as general purpose computing (Pugmire *et al.*, 2007). In such situations, these devices are often coupled

with a more traditional generic architecture such as a Central Processing Unit (CPU) to produce a heterogeneous computing platform. This is similar to what has happened in the development of the Graphics Processing Unit (GPU) in the last two decades.

When considering heterogeneous platforms involving architectures like the NPU, the communication framework provided is typically designed for explicit use with domain specific applications, often being represented as virtual network interfaces (Ahmadi and Wong, 2006). Such devices usually execute applications that are developed and run independently of the host, acting as a pre- or post-processing stage to reduce the workload of host-based applications.

1.1 Problem Statement

As described by Lastovetsky and Dongarra (2009) in the taxonomy of heterogeneous platforms, such systems usually consist of multiple processing elements and a communication framework interconnecting these elements. Research into the latter has been extensively performed due to the rise in popularity of CPU-GPU heterogeneous systems (Mittal and Vetter, 2015). However, similar studies in the context of the NPU architecture are limited to the communication of network traffic rather than providing a communication medium for generic transfers (Shah *et al.*, 2004; Kaufmann *et al.*, 2016).

In order to realise a heterogeneous platform utilising an NPU as a coprocessor for the domain of general computing, further exploration into the feasibility of providing a reliable communication medium between the two architectures is required. To clarify, communication mediums in such systems do exist but are intended for a specific use such as transferring network traffic for uploading/flashing application firmware to the NPU. Other systems for interacting with the executing processes are provided, but these are largely for debugging.

1.2 Research Objectives

As presented in the problem statement, the overarching objective of this research is to implement a CPU-NPU heterogeneous system. More specifically, this involves evaluating the viability of introducing a generic communication framework between a host process and an NPU. In order to achieve this aim, four sub-objectives were identified as outlined below.

1. Since to the best of the author's knowledge there was no evidence of a general CPU-

NPU communication framework in the literature, the first objective of this research was set as an exploratory evaluation of the two architectures to determine whether a communication medium could be established. The result of this preliminary evaluation would heavily dictate the remainder of the research as it would determine whether the research was feasible and if so, provide a set of requirements that the proceeding development should adhere to.

2. The second objective followed from the exploratory evaluation by attempting to implement a prototype communication framework, called the NFComms, to allow processes executing on the disparate architectures to communicate during runtime. The type of communication used would largely depend on the result of the exploratory evaluation which in turn would depend on the communication model best suited to the candidate architectures. The intent of this prototype is to provide a reliable communication module. Owing to the limited resources commonly associated with NPU architectures, the resource footprint introduced by such a framework should be kept to a minimum.

3. As the intent of the NFComms framework is to interact with processes operating on the host as well as the NPU, a runtime or application programming interface (API) must be included through which communication can occur. To facilitate this, the third objective of this research was to provide an interface to allow applications, written in the Go programming language, to interact with the communication framework. The primary motivation for selecting Go as the target language is due to its native support for concurrency and communication.

4. Finally, if the NFComms framework was shown to be feasible and a prototype implementation realised, a preliminary evaluation of using the implemented framework to establish a heterogeneous CPU-NPU compute platform could be undertaken. The objective for this aspect of the research was an evaluation of the feasibility of using an NPU as a coprocessor in the domains of general computation as well as network computation. To achieve this, a set of problems spanning the domains of general computing and network processing would be tested against the heterogeneous platform. The results of these tests could subsequently be used to formulate an opinion on the viability of using such a platform for general computing.

1.3 Scope and Limitations

As the primary aim of this research is to evaluate the viability of introducing a generic communication framework between a host process and an NPU, the following restrictions have been introduced to limit scope. Firstly, the type of NPUs on which the research is intended to interface with, are those that act as PCIe endpoints within the host. This restriction allows us to focus on developing a framework to operate over the PCIe bus rather than having to account for multiple communication mediums. The second restriction is that the resulting prototype framework is only expected to support Debian Linux. As the majority of development associated with the framework is expected to reside within a driver, only one kernel is supported for this version. Extending the framework to operate on alternative operation systems such as Windows is not considered within the scope of this study and is instead left for future work.

Another restriction is the type of NPU that the framework can interface with. This research is considered to be an exploration into determining the feasibility of introducing a generic communications framework and the viability of the resulting heterogeneous system for general computing. With this in mind, supporting a large range of NPUs is not required to realise these objectives. For this research the prototype is implemented on a Network Flow Processor (NFP) 4000, provided by Netronome. Further details relating to the device can be found in Section 2.1.1. Throughout this thesis, when a discussion includes technical details specific to the selected NPU architecture, we will discus these in terms of the NFP. Should a discussion not include such details, the more generic term 'NPU' will be used.

Finally, as noted in the research objectives, the prototype framework is intended to interact with the Go programming language. To allow a runtime to be developed in Go that can interface with the framework driver, support for the C programming language would also need to be considered. Support for C however, is very limited, only providing a basic API that can be used to construct a runtime. This is done to simplify the design of the interface for the Go programming language which utilises this API to handle all driver interactions. Implementing an interface to the framework in this manner simplifies porting the framework to other languages; however, doing so is considered outside the scope of this research and is reserved for future work.

1.4 Document Conventions

To improve readability, a set of conventions is employed throughout this document. Firstly, certain keywords within the text are represented using a different font to make them more easily distinguishable. When referring to a Flow Processing Core (FPC)[1] for example, the keyword microengine is used.

Listings and Algorithms

Listings are presented in a single-lined block using a monospaced font. Each line is numbered for ease of reference. Code given in a listing is presented as it is found in the original source unless otherwise stated. Ellipses are used to represent where some information has been omitted. Listing numbers are provided for reference as part of the heading below the listing. An example is given in Listing 1.1.

```
1 for(int i = 0; i < 100; i++) {
2     ....
3     printf("Hello world");
4     //error checking omitted
5 }
```

Listing 1.1: Extended NFP library functions.

To help discuss how actors within the communication framework operate, algorithms are often used in place of listings. These algorithms use pseudo-code to describe the sequence of operations associated with an actor, providing a more concise representation of functionality while obviating the need for unnecessary detail that would be associated with code listings.

Owing to how the development of the NFComms framework is presented however, both algorithms and numbered diagrams are regularly used in conjunction to better explain the transactional elements of different communication events. To help distinguish between references to each, elements pertaining to an algorithm are represented as steps with the line number in brackets, while diagrams are discussed in terms of events with the element circled. An example instance utilising both would be:

> Considering event ④ in Figure 1.1, the equivalent transaction is presented by step **(2)** of Algorithm 1.

When referring to an inline function in the text, the function name is written using a

[1] The processing element associated with the NPU, further discussed in Section 2.1.1

different font and is immediately followed by a pair of parentheses (). When discussing variables associated with an inline listing or algorithm, the variable name is italicised.

Equations

Mathematical equations are indicated by the use of italics. Major equations are centred and given equation numbers in the right-hand margin. For example:

$$f(x) = mx + c \tag{1.1}$$

Number Formatting

Numbers in this document are rounded to two significant digits after the decimal point. The thousands separator is the comma (,) and decimals are given after a period (.). For example:

$123,456,789.12$

The unit of measurement relating to a number is separated from the numeric component with a space. For example:

123.45 KB

1.5 Document Structure

The remainder of the document adheres to the following structure. An overview of background literature necessary to facilitate understanding of the research is presented in Chapter 2. This chapter begins with a review of existing NPU architectures before discussing the current state-of-the-art in terms of communication frameworks for heterogeneous systems. An overview of the Go programming language is also included for reference in subsequent chapters.

As noted in Section 1.2, before design and implementation of the NFComms framework could be undertaken, the feasibility of such a medium was first evaluated. This evaluation is the focus of Chapter 3, which begins by presenting additional background literature in the fields of communication theory and models. This material is used to select an existing notation[2] to act as the underlying model of the communication framework. Communica-

[2]For this research the notation selected is Communicating Sequential Processes (CSP) (See Section 3.1.2)

tion mechanisms conforming to this model are shown to be compatible with both Go and the NFP architecture, providing a basis for how the communication framework should be designed.

Following the feasibility study, Chapter 4 presents the first iteration of the NFComms communication framework. This chapter begins by introducing additional background literature, discussing the concept of naming and symmetry in terms of communicating systems. This is followed by an overview of how existing communication is handled between the host and NFP, identifying how key communication components can be introduced to allow for synchronous message passing. An equivalent discussion on achieving basic communication between the kernel and user-space applications is included. The body of this chapter presents the first iteration of the NFComms framework. As a reactivate approach was taken in the development, the design of the framework is presented through an evaluation of its implementation. The chapter closes with preliminary testing of the framework, where it is concluded that, although functional, the throughput performance of the prototype is not suitable for meaningful applications.

To improve the poor performance of the initial prototype, Chapter 5 presents a revision to the NFComms framework that improves bulk transfer throughput. This chapter begins with an identification of artefacts in the existing framework that contribute to the poor performance observed and is followed by the proposed solution. The majority of the chapter is focused on presenting revisions to the NFComms framework that mitigate the performance limitations observed in Chapter 4. A set of preliminary tests are introduced to determine the functionality of the revised framework with a focus on throughput performance.

The last outstanding objective of this research, as presented in Section 1.2, is to evaluate the viability of applying a heterogeneous CPU-NPU system in the domain of general and network computing. Chapter 6 explores this objective by explaining how the NFComms framework is used to produce the heterogeneous system, and then presents a set of problems and applications that can be mapped to it. The results of testing associated with these applications is used to form an opinion on the feasibility of utilising such a platform for general and network computing.

Finally, the document concludes in Chapter 7 by reviewing the results collected from prior chapters to present the closing statements on utilising an NPU in the field of general and network computing. A reflection on the design and implementation of the NFComms framework is also presented, before a discussion on potential avenues for additional research are presented as candidates for future work.

1.6 Published Work

During the course of this research, certain aspects of the work have been published separately. Although referred to within the relevant chapters, a brief overview of the research output is presented here. Firstly, the work on naming and symmetry presented in Chapter 4 was initially presented in (Pennefather *et al.*, 2017) while the body of the chapter was later published in (Pennefather *et al.*, 2018a). As the research progressed, this work was refined and an extension (Pennefather *et al.*, 2019b) including work presented in Section 6.2 is currently under review. The work presented in Chapter 5 has largely been published in (Pennefather *et al.*, 2018b) with an invited extension being submitted as (Pennefather *et al.*, 2019a). The latter extension includes the work presented in Section 6.1 and is also currently under review. Finally, the work presented in Section 6.5 has been published in (Pennefather *et al.*, 2018c).

2

Literature Review

This chapter serves to provide the reader with the necessary literature to better understand the research presented in this thesis. As the research focuses on providing a communication framework between a host platform and NPU, the current chapter begins with a review of existing NPU architectures considered relevant in Section 2.1. This section also refers the reader to supporting literature including summaries of additional NPU architectures. Given that the NFP-400 was selected as the target architecture for this research, Section 2.1.1 presents a detailed overview of how processing and communication is handled within this architecture.

Section 2.2 follows with a review of the state-of-the-art in terms of communication frameworks for heterogeneous systems. This section presents existing work in this field, both to identify alternative solutions to the research presented in this thesis, and to highlight the key differences between these works and our research.

Given that the architecture selected for implementation of the proposed framework is the NFP, it is important to further review its parent architecture, the Internet eXchange Processor (IXP). Section 2.3 thus provides a summary of relevant research contributions using the IXP architecture. These works help to further contextualise the strengths of

the NFP architecture and identify common approaches to handling communication.

Section 2.4 provides an overview of the Go programming language which has been selected as the host language for the framework implementation. This chapter closes with a reflection on presented literature.

2.1 NPU Architectures

The proposed framework is intended to provide a suitably generic interface that could be adapted to suit a range of network processors. However, for the prototype implementation, the NFP architecture was selected as the target NPU. This decision was based solely on the availability of a suitable card.

This section provides a review of the NFP architecture as well as a brief summary of five other relevant network processing architectures. These additional architectures have been selected, firstly to help highlight how the target architecture has evolved from the original IXP architecture in Section 2.1.2, and how it compares to other currently relevant architectures in Sections 2.1.3 through 2.1.6. For further comparisons, the reader is directed to works by Li and Wan (2003) as well as Jain and Jain (2014), both of which provide summaries and reviews of additional network processor families.

2.1.1 NFP Architecture

Netronome[1] is a fabless semiconductor company founded in 2003 that specialises in the design and production of NPUs. Netronome holds a licensing agreement with Intel to extend the IXP product line with a new generation of NPUs. This new generation is referred to as the NFP architecture and is intended to be a low power high speed network processing solution. As of 2019, there are three revisions of the NFP architecture available, namely, the NFP-32, NFP-400, and NFP-600 range.

The NFP-400 effectively replaces the original NFP-32 range and provides a high performance solution capable of supporting network traffic throughput of 148 Mpps[2], or up to 100 Gb/s (Netronome, 2018b). As the goal of the NFP architecture is to help offload network specific processing from traditional server-based solutions, a number of accelerators are included to help support network specific operations. Such functionality includes packet forwarding and scheduling was well as routing policies, queue management, and network cryptography. As the hardware solution used in this research falls under the

[1] `https://www.netronome.com`
[2] Million packets per second.

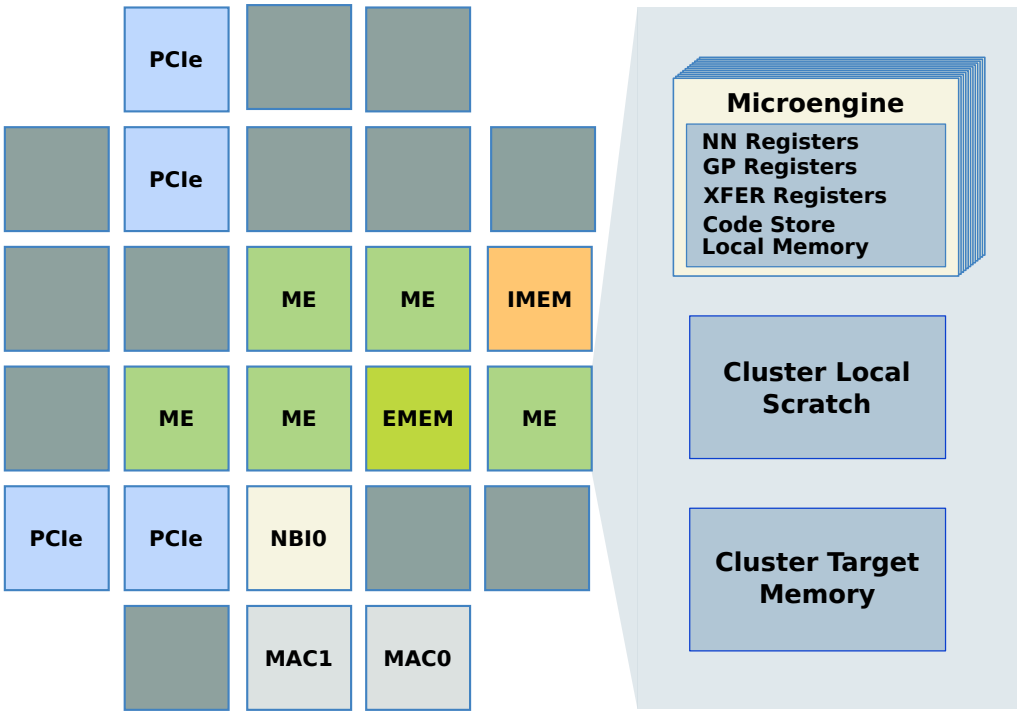


Figure 2.1: NFP microengine resource components, adapted from Netronome (2016).

NFP-400 family, the following summary and associated diagrams correlate with that architecture range.

According to the manufacturer Netronome (2018b), the NFP architecture was designed to achieve high throughput in the domain of network processing. To achieve this, it is composed of multiple processing units, similar to other NPU architectures (Ahmadi and Wong, 2006; Wheeler, 2013). These processing elements are grouped into clusters or islands distributed over the NFP integrated circuit (IC). An interpretation of the resulting topology is shown in Figure 2.1 which depicts some of the processing elements involved in the execution of applications on the platform.

Microengine Topology

For general processing, the NFP-400 consists of 60 32-bit FPCs which are also referred to as microengines. These microengines are divided into five general processing islands with each island consisting of 12 microengines and island specific memory. Each microengine executes as an independent processor with its own instruction store[3], 4 KB local memory, an ALU, and register banks as depicted in Figure 2.2. Furthermore, each microengine

[3]In certain configurations it is possible for two microengines to share a single instruction store (Netronome, 2016)

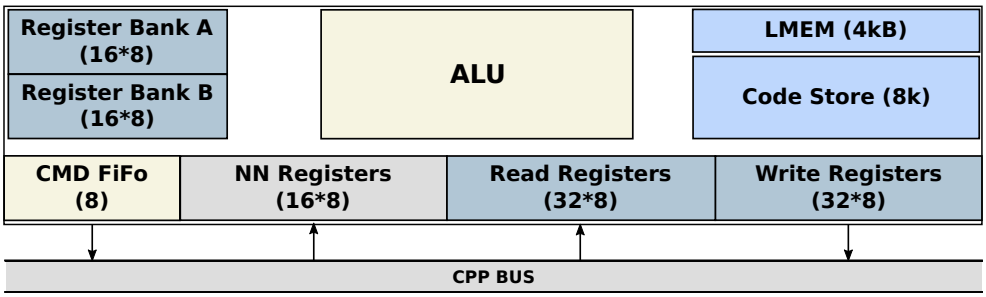


Figure 2.2: Microengine resource components.

can support up to eight contexts or threads. The goal of allowing each microengine to support multiple contexts is to help hide the inherent latencies associated with memory access operations. As a result, management of the threads is performed through cooperative scheduling where the currently active context must actively relinquish control of the arithmetic logic unit (ALU) to allow another context to execute. This approach gives the programmer fine-grain control as to when a context should relinquish control of the ALU. Such points often coincide with a high latency operation such as a memory access event (Netronome, 2018b). The code store and local memory are shared between the eight contexts that can be run on a single microengine, while the register banks are divided such that each context is allocated a dedicated set of registers.

Each of the register banks within a single microengine is used with a specific range of operations. For general processing, each context has a reserved set of 32 general purpose registers (GPRs) that are available for use by the relevant context. This bank is partitioned equally into two groups: A and B. Any instruction performed by a context that uses two input registers requires that one register be from bank A and one from bank B. Fortunately, when developing applications for the NFP, the compiler handles the allocation of variables to register banks in a manner that supports this requirement.

Microengines residing in general processing islands are grouped into sets of 12 units. Within this set, the microengines are ordered such that they form a sequential chain. Each microengine within this chain is numerically distinguishable by its position, with microengine 0 being the first unit and microengine 11 being the last unit in the chain. Each microengine can be considered adjacent to the microengine with the preceding and following numerical IDs. In the context of a single island, microengine 0 does not have a preceding microengine and microengine 11 does not have a following microengine. This configuration is depicted in Figure 2.3.

This ordering is relevant to the next type of register bank, the next neighbour (NN)

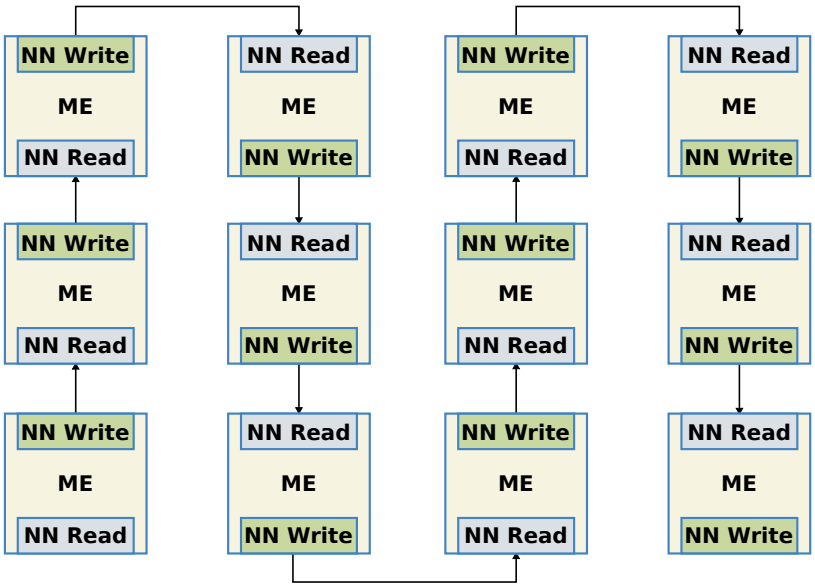


Figure 2.3: Microengine layout for NN registers.

registers. These registers can be used in one of two modes: as a connection to an adjacent microengine, or as additional GPRs. When used for communication with other microengines, data written to an NN register bank by a context operating on a particular microengine (ME 1), can then be read by a context operating on the next sequential microengine (ME 2). In the alternative mode, data written to an NN register can only be read by a context operating on the same microengine on which the write operation occurred, efficiently making the NN registers act as GPRs. It is important to note however, that in this configuration there is a 16 cycle delay[4] between when a NN register was written to and when it can be read.

Owing to how the microengine is structured, it is a closed entity without dedicated access to external resources such as larger capacity RAM or network traffic. Fortunately resolving this is the responsibility of the remaining two register types: the transfer registers and the command (CMD) registers. The transfer registers are further divided into two banks, namely, read registers and write registers.

With the exception of the NN registers, all communication between a microengine and any external resources must occur through the Command Push Pull (CPP) bus. This bus interconnects the microengine with all other external resources such as a dedicated crypto engine, PCIe Direct Memory Access (DMA) engine, and various memory regions which are discussed later in this section. To read the data into the microengine from an

[4]At a clock speed of 600 MHz for the NFP-32 architecture

external entity, that entity must have the data pushed through the CPP bus into the relevant read registers of the target microengine. Any data that the current microengine wishes to submit to an external entity must be moved into the write register bank before it can be read by an external actor.

Furthermore, a microengine cannot initiate an operation involving the transfer registers itself and must instead rely on an external actor to perform any actual move operations on its behalf. To submit such a request, the CMD registers must be used. These registers allow the current microengine to issue a message to an external actor, requesting that it perform the transfer operation for the issuing microengine. The external actor can then move data out of the relevant write registers and push the data into the read registers of the intended recipient. Details relating to the direction of transfer, the target ID (including register or memory location), and the volume to transfer are all included in the request issued.

Intra-Architecture Communication

Netronome (2016) describes the situation where data is moved between two microengines as a subset of the reflect operation. A reflect operation enables a value placed in an output transfer register to be reflected into a specified input transfer register where the output and input transfer registers are not restricted to being on the same microengine or even the same island (depending on the memory manager handling the request) (Netronome, 2006). Reflects between microengines are achieved by making the relevant registers on a remote microengine visible to other microengines by using the `__declspec(visible read_reg/write_reg)` command. The microengine performing the operation (either reflect read or reflect write) must include a remote equivalent of the register made visible by using the `__declspec(visible read_reg/write_reg)` command and declaring a variable of the same name and length as the one declared visible. Operations involving the transfer register made visible on the remote microengine can now be performed as either read or write (depending on register type).

The reflect operation itself is not actually performed by the calling microengine, instead it is a functionality advertised by the Misc Engine of the Cluster Target (CT) Memory Unit (MU). As noted in the FPC Programmers Reference Manual (Netronome, 2006), the Misc Engine is also responsible for interthread signalling and circular rings. The Misc Engine performs reflect operations by first reading the data to be transferred in from a set of registers on one engine so that it can be transferred to the destination engine on behalf of the requesting entity. The buffer into which the CT Misc engine reads the data

for transfer is limited to either 14 words (56 bytes)[5] or 16 words (64 bytes) depending on the silicon revision of the NFP card. Attempting to read more than this limit in a single transaction causes the buffer to overflow and puts the system into a potentially unstable state.

Remote signalling can be performed by the CT MU Misc Engine in a similar fashion. Two types of signalling are used for this communication structure. The first includes the message being passed, allowing the remote microengine to be signalled when the transfer relating to its registers is complete. The second type is a signal relay that does not involve the transfer of any data and is useful for synchronisation. This signal request is generated by issuing the command `interthread_signal` along with the appropriate address and signal register (Netronome, 2008).

Inter-Architecture Communication

To allow for communication between the NFP card and associated host, the NFP includes a PCIe module responsible for providing an interface which conforms to the PCI Express Gen3 standard (National Instruments, 2014). As described in the NFP-6 databook, this module can support up to eight lanes, each capable of up to 8 GT/s [6], allowing for up to 64 GT/s. The module can also act as either a PCIe endpoint or a Root Complex. When configured as an endpoint, the PCIe controller allows access to the internal system bus and its connected targets such as memory units and the internal microengines. When configured as a Root Complex, the PCIe controller can generate transactions necessary to perform endpoint discovery and configuration (Netronome, 2016).

From the context of applications operating on the microengines, it is possible to use the PCIe module to submit MSIX[7] interrupts to the host provided the PCIe module is in endpoint mode. With appropriate modifications to the NFP driver, the host operating system can be equipped to listen for such interrupts and generate a corresponding software interrupt for user space applications.

Memory Topology

In a standard CPU architecture, the memory hierarchy follows a pyramid structure described by Null and Lobur (2015) with CPU registers boasting the shortest access latency and long term storage supporting the largest storage capacity (Stallings, 2010). In the

[5] The NFP 400 architecture used in this research has a 14 word buffer size
[6] Giga transfers per second
[7] An extended version of Message Signalled Interrupts (MSI) as part of the PCIe 3.0 specification.

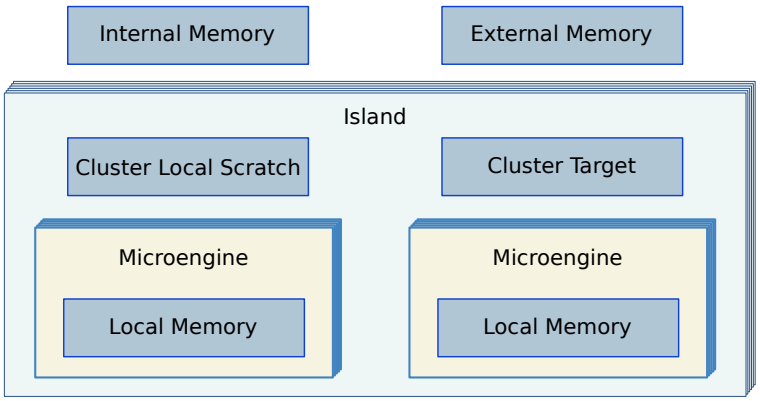


Figure 2.4: Hierarchical layout of memory regions present on the NFP device, adapted from Stuart (2014).

Table 2.1: Memory regions and their respective characteristics.

Memory Type	**Capacity**	**Latency (in execution cycles)**
Local Memory	4 KB	0-3
Cluster Local Scratch (CLS)	64 KB	20-50
Cluster Target Memory (CTM)	256 KB	50-100
Internal Memory (IMEM)	4 MB	150-200
External Memory (EMEM)	Up to 8 GB	150-500

context of generic application development, this hierarchy is abstracted away from the user and is optimised so that data accessed most regularly is stored in memory regions with the shortest access latencies.

The memory landscape on the NPU architecture is also designed as a hierarchy according to the volume of storage available in each region and time taken to access the aforementioned storage. Stark (2015) divides these regions as presented in Table 2.1.

Associated with each memory region is a latency, measured in operation cycles, that the requesting context must wait for the requested memory to be returned. Owing to the layout of the hierarchy, memory regions with greater associated latencies also boast greater storage capacity. Figure 2.4 represents this hierarchy of memory regions along with the storage capacity and associated latency incremented by memory accesses. Aside from the local memory, which is specific to each microengine, accessing memory is done by effectively submitting a request to a memory manager and performing a transfer operation.

To allow for optimal use of the different memory regions, applications developed to operate on the NPU must be explicit when indicating in which memory region each instantiated object must reside. This approach assumes that the developer has a better knowledge of how application objects are used and therefore is able to assign memory in an optimal

manner.

Atomic Memory Operations

An atomic operation is one that cannot be subdivided into further operations and thus observing systems cannot view the effects of such an operation during its execution. Atomic operations are important in concurrent systems because of this indivisibility. From the context of any external system, an atomic operation (or operations) will occur instantly and thus cannot be interrupted until execution is complete (Andrews, 1991; Burns and Davies, 1993).

Though most basic arithmetic and logical operations are atomic, the NFP also supports atomic operations for accessing and modifying memory and interfacing with queues. The Cluster Local Scratch (CLS) memory supports atomic operations such as `add`, `add with saturation`, and `test and clear immediate`, amongst many others. These operations allow calling contexts to guarantee that the memory cannot be modified by other processes until the operation is complete. In the context of concurrent systems, such atomic memory modifications provide the required underlying operations for the implementation of semaphores, barriers, and locks (Netronome, 2016, 2006).

As a final note, the processing capabilities of a microengine have been optimally designed for the domain of network processing and as a result, the processor has a limited set of processing features when compared with more general processing units. Capabilities such as floating point operations and the division operator are, for example, omitted in the NFP architecture (Netronome, 2018b) while other functions such as the modulus operator are present, but are not supported by any dedicated hardware (Netronome, 2016). This reduced instruction set does not severely limit the performance of a microengine in the domain of network processing and in turn, allows for a single IC to support dozens of microengines.

2.1.2 IXP Processor

Though the NFP architecture can be seen as an evolution of the IXP2800 architecture (Intel, 2005), the latter is an important architecture in itself and warrants a brief summary. Furthermore, due to its similarity to the NFP architecture, there can be an assumed level of compatibility between the architectural layout of the processor families allowing some supporting literature of the IXP (discussed in Section 2.3) to be relevant to the NFP as well.

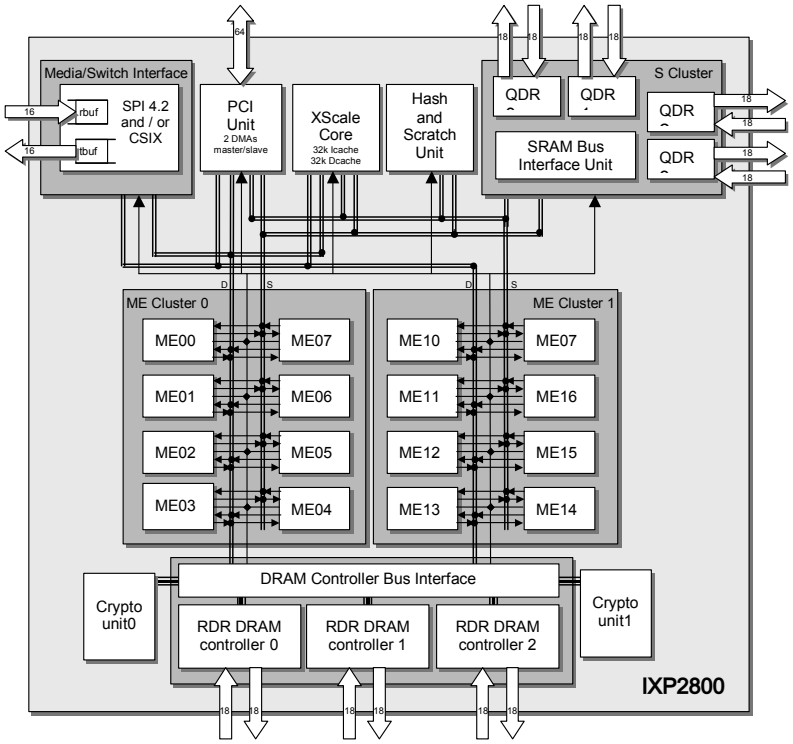


Figure 2.5: Basic architecture layout (Adiletta *et al.*, 2002).

The IXP2800 network processors enable deployment of complete network processing systems on a single, dedicated controller. Intel (2005) explains that these processors are capable of performing network processing at traffic speeds up to 10 Gbit/s and include dedicated cryptographic units, providing support for DES, 3DES, AES, and SHA-1 hashing algorithms (Intel, 2002).

Johnson and Kunze (2003) note that computation on the IXP2800 processors is performed by 14 to 16 microengines, distributed across two islands as indicated in Figure 2.5. Each of these microengines is capable of running eight contexts, each with its own set of registers (depending on the configuration), and includes a round robin non pre-emptive scheduler that determines which context should be executed next. As with the NFP architecture, concurrency is handled through cooperative scheduling where the next context can only start operating once the current context explicitly releases control of the ALU. All contexts within a microengine execute the same body of code with the architecture logic focusing on hiding memory access latencies by enabling pipelining of an application.

This layout allows a context to continue operating until it requires a resource, at which point, it can relinquish execution control and effectively sleep until the resource request is complete. In the meantime, a different context can start executing the same code but at

a different stage until it too requires a resource with an associated latency. This approach allows for better use of the microengine ALU by taking advantage of would-be idle time to execute code. The currently active context can be identified by the application, allowing for code branching. This implies that it is possible for each context to execute separate code, provided all branches reside in the code store of the microengine (Johnson and Kunze, 2003).

Data that cannot be stored in the registers of a microengine must be stored in one of the memory locations available, each of which has a different size and associated latency. Memory access is performed through remote memory management units that handle the acquisition and storage of memory associated with locations specified by the microengines. The management units can also signal microengines when such operations are complete, allowing contexts to sleep until data are locally available for use.

2.1.3 HX300 Processor Family

Of the additional NPU architectures summarised in this review, the Xelerated HX300 architecture by Marvell is the first to be evaluated. As presented by Marvell (2013), this processor family is a single chip NPU that provides packet processing and traffic management for the Marvell programmable Ethernet switches. This architecture is capable of operating at wire speeds from 10 Gbit/s to 100 Gbit/s. The integrated traffic manager utilises a hierarchical scheduler to perform service and subscriber shaping on a per port basis, providing fair usage and QOS. Internal TCAM and SRAM lookup engines reduce forwarding latencies and allow for a high number of classifications to be performed on each packet processed (Marvell, 2013).

Packet processing in the Xelerated HX300 processor family is achieved through an array of Packet Instruction Set Computer (PISC) processor cores as detailed by Marvell (2012). These PISC cores are connected to form a packet processing pipeline through which all received network traffic flows. The resulting pipeline also includes I/O processors referred to as 'engine access points' that allow additional on-chip engines and external memory regions to interact with the pipeline. The application executing on a single PISC is treated as a single sequential thread and the programming model used is a single threaded application targeting a general purpose CPU. Due to the enforced data-flow model, no synchronisation or interprocess communication between PISC cores is needed which further simplifies development.

Due to the wire speeds that this NPU is capable of operating at, it is considered both relevant and current in terms of NPU architecture. By enforcing a data-flow model where

traffic is passed sequentially between the PISC cores greatly simplifies coding complexities. This simplification could however also become a limiting factor for the architecture, as no support for communication between the PISC cores could limit the range of applications that could be deployed to such a device. This platform excels in the target environment of providing intelligence to high speed network switches but would not be a suitable candidate for supporting a heterogeneous communication framework.

2.1.4 Cavium LiquidIO II Smart NIC

The second additional architecture presented is the Cavium LiquidIO II Smart NIC[8], which is also developed by Marvell. Unlike the Xelerated HX300 architecture (see Section 2.1.3), which is more commonly found in intelligent switches, the LiquidIO II Smart NIC is provided as a PCIe daughter card and designed for targeting wire speeds of 10 Gbit/s to 40 Gbit/s.

Network processing is achieved through the use of OCTEON Multi-Core MIPS64 processors with the latest variant introduced by Marvell (2019) being the OCTEON III series depicted in Figure 2.6. The OCTEON III series supports up to 48 quad core cnMIPS[9] processors operating at 1.2 GHz and supporting floating point operations (Cavium, 2014, 2017b).

Along with the cnMIPS processors, this processor family also supports Deep Packet Inspection (DPI) accelerators which include hyper-finite automata engines to assist in the analysis of generic traffic. The DPI accelerators are also capable of accelerating regular expression rules through a hyper-non-deterministic finite automata block (Cavium, 2017a). Another important feature of this device is support for remote direct memory access operations. This allows the LiquidIO II Smart NIC to perform DMA operations with another PCIe endpoint device without requiring the CPU to orchestrate the event, improving transfer rates and reducing CPU load.

2.1.5 TILEncore-Gx36

Another network processing platform of interest to this research is the TILEncore-Gx36 by Mellanox (2017b). As with the other network processing solutions discussed in this section, this application adapter is provided as a PCIe daughter card. The card contains four 10 Gbit/s SFP interfaces and up to 16 GB of on-board DDR3 RAM. All processing

[8] `https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics/index.jsp`

[9] A microarchitecture developed by Cavium that implements the MIPS64 instruction set

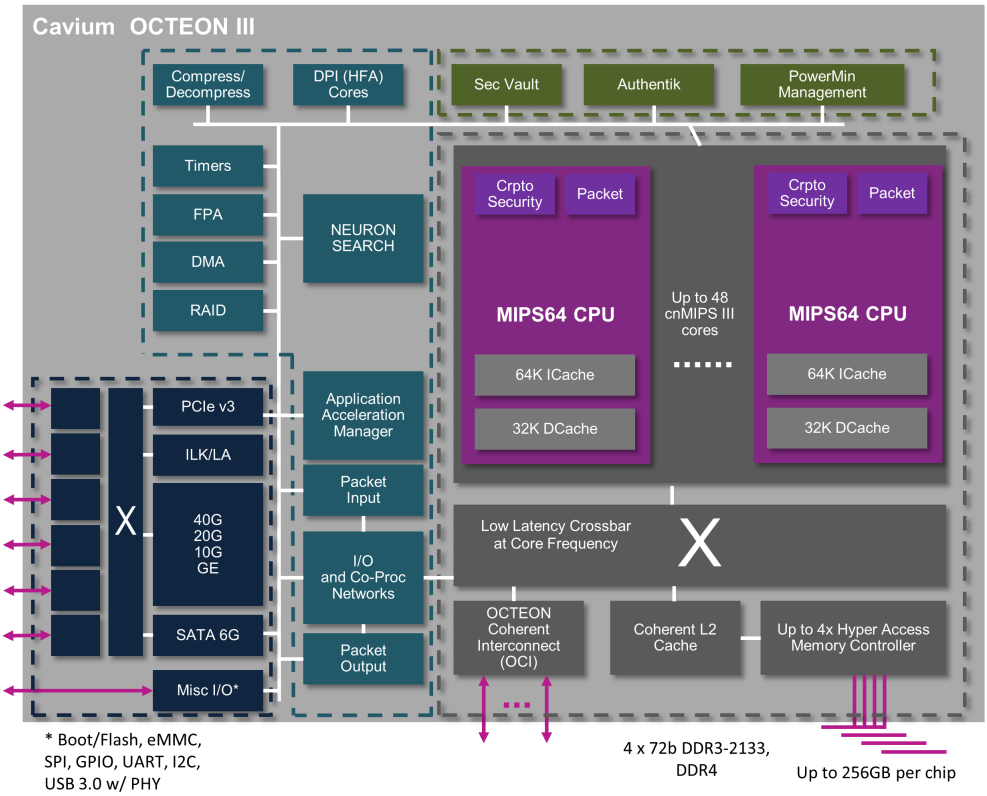


Figure 2.6: Block diagram of the OCTEON III series processor (Marvell, 2019).

is handled through a TILE-Gx36 processor which acts as the interface for both the PCIe and the SFP modules (Mellanox, 2017b).

The TILE-Gx36 is a multicore processor which consists of 36 identical processing elements (or tiles) all interconnected using a mesh framework capable of up to 200 Tbit/s throughput. Mellanox (2017b) states that each element is a fully fledged 64-bit processor operating at 1.2 GHz with three execution pipelines. Each element also includes 12 MB of cache divided into three levels of cache: 32 KB of L1 cache, 256 KB of L2 cache, and 9 MB of L3 cache. Developing applications to deploy to the TILE-Gx36 can be done using standard C and C++ with the TSHMEM library. A cross-compiler is then used to produce a TSHMEM executable which can be deployed to the processor array (Mellanox, 2017b).

2.1.6 Cisco Flow Processor

The final processing architecture reviewed is the Cisco flow processor family. This architecture family has been developed by Cisco to target network speeds ranging from 10 Gbit/s to beyond 100 Gbit/s. As noted by Cisco (2014), the term “Cisco flow processor"

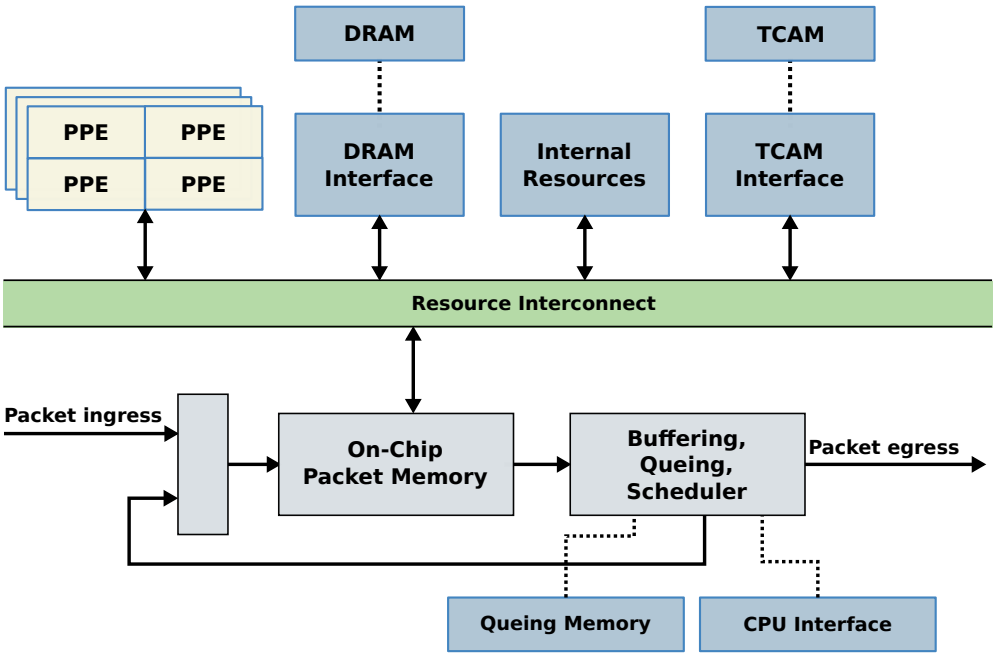


Figure 2.7: An overview of the Cisco flow processor, adapted from Cisco (2014).

is used to describe both the software and hardware architecture that defines the network processor. This family is designed to be multi-generational with the first generation being silicon based; later generations are expected to be single chip solutions that will conform to the software architecture which defines the Cisco flow processor family architecture.

The current generation supports 40 packet processor engines (PPEs) which operate between 900 MHz and 1.2 GHz (Cisco, 2014). These processing units are responsible for handling the on-chip processing network traffic. The PPEs are programmed using C, support up to four threads, and do not have a fixed-feature pipeline. The allocation of traffic to the individual cores is handled by a set of dedicated engines which provide offloading, queuing, and scheduling.

The Cisco flow processor provides a high speed interface for interacting with the host system for control purposes. This allows for runtime interaction between processes operating on the CPU and the PPE cores. This interface is also used to transfer large data structures such as tables or databases between the two architectures.

The handling of network traffic is illustrated in Figure 2.7 where network frames are collected off the wire and stored in the on-chip packet memory. Depending on the application, either the entire packet can be used for processing[10], or just the header information. Once collected, the frame is assigned to the next available PPE thread. As each of the

[10]Such as solutions requiring Deep Packet Inspection (DPI)

40 PPEs is capable of supporting four threads, this results in a total of 160 threads being available to process traffic. Once processed, the active PPE thread hands the packet to the traffic manager which buffers the frame for transmission or recycling back into the inbound packet queue. The packet is assigned to the appropriate queue and scheduled for transmission. By allowing a packet to be recycled into the inbound queue it is possible to have a single packet go through multiple processing passes before finally being forwarded or dropped (Cisco, 2014).

2.2 Existing Heterogeneous Implementations

This section summarises some of the more prominent heterogeneous solutions considered relevant to the research presented in this thesis. This summary aims to provide the reader with the current state-of-the-art by reviewing competing solutions with a focus on communication. Owing to its dominance in the field of heterogeneous systems, an overview of communication between the GPU and CPU is briefly discussed in Section 2.2.1 before being followed by a similar discussion on OpenCL in Section 2.2.2.

Another field of research that is being explored (Pugmire *et al.*, 2007) is CPU-FPGA heterogeneous computation. Considering the obvious similarities between such implementations and the communication framework proposed in this thesis, relevant frameworks targeting a heterogeneous CPU-FPGA systems are explored. As a requirement, all communication between the CPU and FPGA must be performed over the PCIe bus as heterogeneous implementations using alternative mediums such as network communication are not considered relevant enough to warrant discussing here.

Notable CPU-FPGA heterogeneous systems are reviewed in Sections 2.2.3 through 2.2.5 with CPU-NPU heterogeneous solutions being presented from Section 2.2.6 onwards. Although not formally a CPU-FPGA or CPU-NPU heterogeneous system, the Axel communication framework by Tsoi *et al.* (2010) is presented in Section 2.2.8. This framework is considered relevant due to how it approaches some aspects of communication.

2.2.1 CUDA

Although the focus of this research is on providing a framework for communication between a CPU and NPU, relevant work in the domain of CPU-GPU computing is briefly discussed due to similarities in the resulting heterogeneous platforms. For this investigation, only communication between the CPU and GPU via PCIe is considered.

Both the CUDA runtime API[11] and CUDA driver API[12] provide support for transferring memory between the host and the GPU. This was initially achieved by declaring regions explicit to each architecture and later improved upon in CUDA 4 with the introduction of Unified Virtual Addressing (UVA) (Schroeder, 2011). UVA allowed copying without specifying the source or destination architecture type, instead having the API determine the locations of the memory from the pointer value (Schroeder, 2011; NVIDIA, 2018).

In addition to normal memory transfers and UVA, the runtime API supports the ability to pin memory which can have a better throughput performance when compared to transferring from unpinned memory. As noted by NVIDIA (2018) however, pinned memory should be used sparingly as it is a computationally expensive operation and pinning an excessive amount of memory can hamper performance.

Although UVA provided a unified address space, the physical location of memory was still disjoint until the introduction of Unified Memory in CUDA 6 (Harris, 2017). Unified Memory improved on UVA by creating a pool of managed memory between the host and GPUs, accessible through a single pointer. This helps simplify the programming and memory model by allowing objects to be shared between the GPU and CPU. The physical memory associated with an object can then automatically be migrated between architectures on demand without the need to explicitly copy it. The memory, however, still needs to be synchronised between uses on the different architectures using `cudaDeviceSynchronize()` to maintain consistency (Landaverde *et al.*, 2014; Harris, 2017).

[11] `https://docs.nvidia.com/cuda/cuda-runtime-api/index.html`
[12] `https://docs.nvidia.com/cuda/cuda-driver-api/index.html`

2.2.2 OpenCL

Another framework to consider is the Open Computing Language (OpenCL), described by Gaster *et al.* (2013) as a heterogeneous programming framework for developing applications spanning multiple device types supplied by different vendors. The OpenCL standard is managed by the Khronos Group[13] who provide a generic API for the core functionality. By following the OpenCL standard, implementations of the API can be designed for alternative architectures, allowing them to interact with applications designed to use the OpenCL framework.

OpenCL is divided into four models, each of which covers a different aspect of the specification, namely; platform, execution, memory, and programming (Gaster *et al.*, 2013). The actors within OpenCL can be separated into two primary components: the host and the devices. The goal of the host is to orchestrate execution of the application, managing the devices and workflow, while the devices are the component architectures on which work elements are executed. These work elements are described by the platform model of the OpenCL specification and are referred to as OpenCL functions or kernels.

According to Gaster *et al.* (2013), the execution model defines how the OpenCL environment is configured as well as how kernels are executed on the device. More importantly however, this model is also responsible for providing the mechanisms for host-device communication that occurs through mechanisms such as the command queue. The Khronos OpenCL Working Group (2012) state that a command queue is a data structure used to coordinate the execution of kernels on devices. The host performs this by first creating a command queue and associating it with a device. Each device must have its own command queue and only one command queue can be associated with any device at a time (Gaster *et al.*, 2013). Interacting with processing elements operating on that device is achieved by placing a command into the queue which acts as a request to all elements associated with the queue. These commands can executed either in-order or out-of-order. In-order requires that all commands supplied to the command queue be executed and resolved in the order that they were inserted, enforcing serialisation. Out-of-order allows the commands to be issued in the order that they were supplied to the command queue, but does not require them to complete in the same order.

The types of commands that can be supplied to the command queue include (Khronos OpenCL Working Group, 2012):

- Kernel execution commands

[13] `https://www.khronos.org/opengl/`

- Memory commands
- Synchronisation commands

Kernel execution commands allow the host to submit a request to the device for the execution of a kernel across its processing elements (also referred to as compute units). These compute units are the physical elements within the device that implement the virtual scalar processors required to execute the kernel (Chin, 2012). Next, memory commands allow for the transfer of data between memory objects associated with either the device or the host. Such commands include the mapping and unmapping of memory residing in the host address space. Finally, synchronisation commands are used to define the order in which commands are executed (Khronos OpenCL Working Group, 2012).

As commands issued to the command queue are not resolved immediately, OpenCL supports the ability for the host to query the status of an issued command. The returned state indicates whether the command is being processed, has been completed, or is simply queued and waiting. Furthermore, OpenCL provides support for waiting on a set of events to complete, providing a point of synchronisation between the host and devices (Sellitto and Schaa, 2012).

The memory model of OpenCL describes the memory hierarchy used by the kernels. It is not required that this hierarchy map to that of the target architecture, provided the architecture memory can be suitably abstracted to mimic the OpenCL memory hierarchy (Gaster *et al.*, 2013). This allows a unified hierarchy to be described so that kernels can span multiple architectures that could each potentially support a different memory hierarchy. An important aspect of the memory model is the definition and management of shared memory. Compute units are grouped into work-groups which have access to shared memory as well as barriers and synchronisation elements to help manage execution.

The Khronos OpenCL Working Group (2012) describes the memory model associated with compute units as being partitioned into four regions. The first region is private memory, which as the name implies, is a memory region reserved exclusively for the relevant compute unit. All variables or data defined in this region are not accessible by any other compute unit. Next is local memory, a memory region shared between compute units within a specific work-group. This region allows for the establishment of work-group-local data elements and synchronisation primitives, allowing for coordination within the group. The third memory region is constant memory, a region globally accessible but only the host has write access. The last region is global memory, a memory region modifiable by all work-groups as well as the host.

The final model to review is the programming model. As discussed by the Khronos OpenCL Working Group (2012), OpenCL supports two types of programming models: data parallel and task parallel, although a combination of the two models is allowed. The data parallel model is commonly used for GPU-based devices (AMD, 2010) and operates by having a body of work submitted to the device for execution. The work is then divided amongst the work-groups and processed in parallel (Hwu and Stone, 2010). The task parallel model is described by the Khronos OpenCL Working Group (2012) as a model where computations are defined as tasks and only a single instance of a task can be executed in a given index space. In terms of work-groups, this is equivalent to restricting a work-group to contain a single compute unit.

Lastly, Gaster *et al.* (2013) described the communication between compute units operating on devices. This can be achieved through either a shared address space or message-passing. As noted in the memory model, the memory hierarchy contains four tiers with the first tier being accessible only by the relevant compute unit. Higher memory tiers however are shared between compute units allowing for communication. To support this, OpenCL provides a memory consistency model to help facilitate the order of memory operations. This in conjunction with synchronisation elements such as semaphores and barriers, allows for structured communication to be achieved through shared memory.

Message-passing on OpenCL provides a more abstract approach to facilitating communication by enabling explicit intercommunication between compute units regardless of physical location. The implementation of message-passing routines is not defined and can be implemented in shared memory, or using device specific capabilities.

2.2.3 RIFFA

An example of a CPU-FPGA heterogeneous implementation is the Reusable Integration Framework for FPGA Accelerators (RIFFA) by Jacobsen *et al.* (2015). This solution provides an open source communication and synchronisation library for CPU-FPGA interaction. RIFFA allows up to five FPGA PCIe endpoints to be integrated with a single host CPU with communication occurring through synchronous channels. Setup and endpoint discovery is handled by a driver that also provides an interface via Input Output Control (IOCTL) for interacting with the FPGA endpoints. RIFFA has gone through a number of revisions and as of version 2.1, the host software library has been simplified to support only a minimal set of functions for interacting with the communication framework. By supporting only a minimal set of functions, utilising the library in an application is simplified which is the goal behind the most recent revision.

The manner in which communication between the host and FPGA is implemented in RIFFA is very relevant to the research presented as both aim to achieve a similar goal albeit for a different endpoint architecture. For the RIFFA v2.1 implementation, communication occurs through DMA operations, utilising a shared memory location which is made accessible to the endpoint device. Communication can be initiated by either the host or the endpoint device but as it is synchronous, the initiating party blocks until the other party is ready to participate in the event. RIFFA v2.1 utilises DMA scatter gather lists to collect references to the physical pages where memory must either be read from or stored. For downstream operations, this list of references is used by the *RIFFA channel* to read the associated data and push it into a FIFO buffer where it can be read by the user application operating on the FPGA. For upstream operations, the scatter gather list is used by the *RIFFA channel* to issue write requests to transfer data to the supplied addresses.

Synchronisation in RIFFA v2.1 is handled through notification messages that are generated through interrupts either by the FPGA or read/write operations to the FPGA base address register by the host. When the host initiates a message transfer, it calls a read request associated with the RIFFA driver via IOCTL. This call blocks until an interrupt from the FPGA is received or the supplied time-out has elapsed. When the FPGA reaches the equivalent state in its operation, it generates an interrupt, informing the host driver that it is ready to engage in the transfer. At this point, the data transfer can be performed. Once complete, the user IP core signals the *RIFFA channel* that the operation is complete. This signal is propagated to the RIFFA host driver which then requests the number of bytes transferred from the *RIFFA channel*. This last step seems largely to be a sanity check to confirm the correct volume of data was transferred. Once received, the number of bytes is returned to the calling application, unblocking the read request. An overview of this interaction is depicted in Figure 2.8.

Communication downstream follows a similar sequence of events with two notable deviations in how the synchronisation is performed. The first difference is that the user IP core does not initiate a read request, but instead must wait for a signal from the *RIFFA channel*, indicating that a write operation from the host has been received. This signal blocks the operation until consumed by the user IP core, indicating that it is ready to engage in the transfer event. The second difference is that the user IP core is informed that the transfer operation is complete by the *RIFFA channel* and is not expected to generate any signal of its own. This sequence of events follows the logic that all data from the read operations are stored in a FIFO buffer for the user IP core to access and it simply needs to acknowledge the start and end of the operation. From the perspective

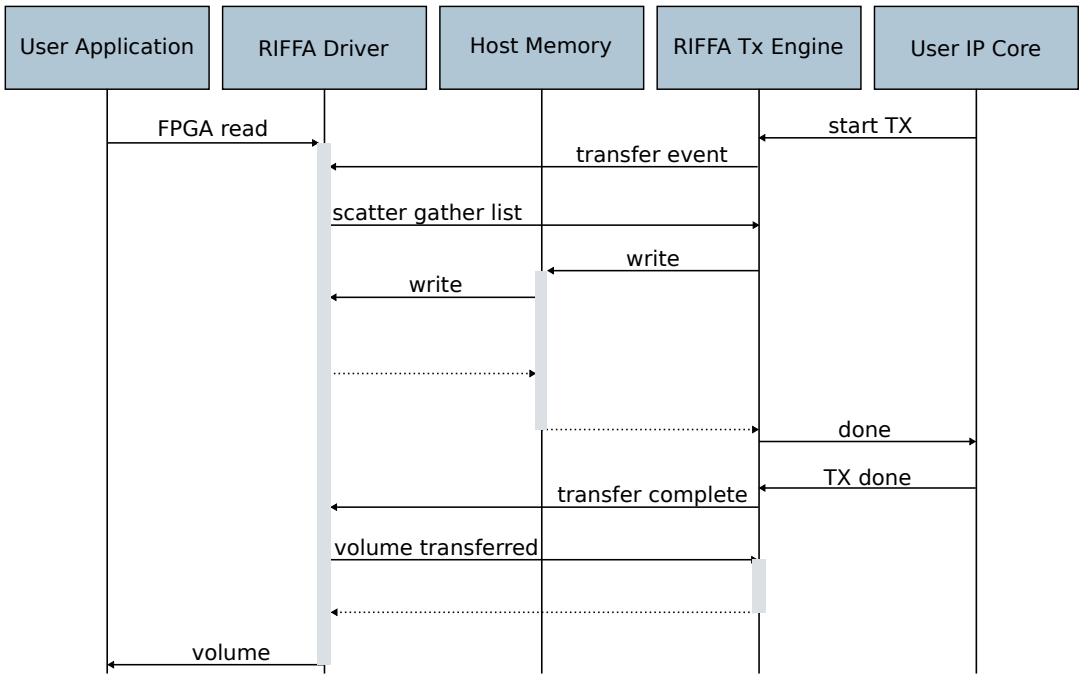


Figure 2.8: RIFFA upstream transfer sequence diagram, adapted from Jacobsen *et al.* (2015).

of the host, the event resolves in the same manner as the upstream communication event with the requesting function via IOCTL blocking until the transaction is complete. Upon completion the number of words written is returned.

2.2.4 ClickNP

Another significant contribution to the field of heterogeneous network processing arises from extensions to an existing software architecture, the Click Modular Router[14]. Kohler *et al.* (1999) describe this software architecture as having been designed to allow for the development of flexible and configurable routers. This flexibility is achieved through the introduction of modules or elements that can be chained together to form a graph through which network traffic can flow. Each element is responsible for a single router function leading to the modularity of the resulting application.

Click also provides a Domain Specific Language (DSL) to assist in the implementation of elements; however, this language was designed exclusively for declaring and connecting elements where the implementation of said solutions must still be performed using C++ supporting a collection of libraries to simplify the process. The resulting applications developed using Click are designed to run on the Linux kernel with only packet references

[14]https://github.com/kohler/click/

being transmitted between elements. The actual packet is stored separately until it is either dropped or transmitted off the platform, reducing communication overheads (Kohler *et al.*, 1999).

The modular approach taken by the Click Modular Router is both novel and effective in providing scalability to a network processing platform. This functionality is coupled with a DSL designed to help describe the elements and their connections. The implementation has produced a software architecture that is receiving interest in the heterogeneous field. Two further contributions which have arisen due to the Click Modular Router are NP-Click (discussed in Section 2.3.3) and later ClickNP (Li *et al.*, 2016).

Li *et al.* (2016) argue that deploying software solutions to FPGAs with dedicated network interfaces would be equally as effective as utilising network processors for network processing. This decision seems to stem largely from FPGAs being a more mature architecture, providing a wider range of functionality. One of the complexities associated with FPGAs that ClickNP attempts to eliminate is coding complexity.

Although ClickNP focuses on deploying network applications to an FPGA, it does support a feature that is both noteworthy, and differentiates it from similar research like NP-Click (Shah *et al.*, 2004). ClickNP provides a generation routing graph that spans both the target FPGA and the host CPU, allowing applications written in ClickNP to be deployed to a truly heterogeneous system.

To develop applications for this heterogeneous system, ClickNP has retained the idea of using a simple DSL to help declare elements and handle the connections which form the application graph. ClickNP has also opted to allow elements to be written in C and introduce extensions for the declaration of element objects. Declared elements are then supported by a set of built-in functions that handle the input and output of associated ports. During the declaration of an element, the user can also specify for which architecture an element should be compiled by prefixing the element with a keyword such as `host` (Li *et al.*, 2016).

To allow application components on each architecture to communicate, a high performance PCIe I/O channel was included in the software architecture. ClickNP uses the Catapult shell architecture to handle communication between the CPU and FPGA. Putnam *et al.* (2014) state that the catapult provides communication by allocating a single non-paged buffer in user-land memory and acquiring a reference to the underlying physical memory for the FPGA. This buffer is divided into 64 units or slots, 33 of which are used by Catapult itself (Putnam *et al.*, 2014). The remaining 31 are available for use by ClickNP.

One of the available slots is used by ClickNP for signalling, while the rest are used for communication. The approach taken by ClickNP and the underlying Catapult shell to communication is considered relevant and efficient, and may provide a potential basis for the framework associated with this research.

2.2.5 sPIN

Hoefler *et al.* (2017) present the programming model for Streaming Processing in the Network (sPIN) which is designed to allow for the offloading of short user-defined functions to processing units operating in the networking domain. The goal of sPIN is to improve bandwidth and latency for simple processing tasks that are dominated by data movement. Though targeting smart NICs, this model could have uses in general computing. sPIN follows some design principles seen in CUDA (Nickolls *et al.*, 2008) and OpenCL (Stone *et al.*, 2010) by having developers specify kernels (also referred to as handlers in the context of this research) that can be deployed to the network specific architectures.

Currently three types of handlers seem to be supported, namely, header, payload and completion. These handlers are processed in a 'streaming way' by the handler processing units. Each kernel or handler is intended to contain only user defined functions not exceeding a few hundred instructions in length. The functions are written in a variant of C which includes additional keywords for indicating handler functions. Once defined, handler functions are allocated to connections which accept one of each type of handler. Though written in C, the handlers are limited to a subset of operations, omitting functionality present on some networking architectures such as dynamic memory management.

The authors state that communication between HPUs occurs through shared memory but no further details are given. As sPIN is largely focused on deploying functions to the networking architecture which resolve without host interaction, communication between the host and the smart NICs is not a focal point of this research. For moving network traffic to the host, sPIN relies on existing work such as Portals 4 (Sandia National Laboratories, 2019).

Though promising, sPIN is largely a theoretical model with all (albeit impressive) benchmarks being performed in a 'cycle accurate' simulation using a PCIe Gen 4 32× interface. At the time of writing, no implementations of sPIN executing on existing hardware could be found. Furthermore, sPIN is largely focused on providing a generic model for describing short functions and their deployment to the networking hardware with little to no focus on communication between such handlers or the host. Should our research expand into transparently offloading components of a heterogeneous application to the NFP,

sPIN would become more relevant; however, as it stands, the existing NFP development environment provided by Netronome is effectively equivalent to what sPIN proposes.

2.2.6 FlexNIC

In a similar fashion as ClickNP, FlexNIC (Kaufmann *et al.*, 2016) attempts to reduce server load due to increased networking demands by offloading some of the network processing to dedicated hardware. The key differences between ClickNP and FlexNIC are that FlexNIC targets network processors for offloading instead of FPGAs, and uses previously established methods of describing network functions such as Click.

The goal of FlexNIC is very similar to our research; it attempts to produce a flexible DMA programming interface for I/O between the host and the target NIC. The key difference is the type of data being transferred. The communication fabric proposed by FlexNIC is designed to optimise the transfer of network traffic between the two architectures, allowing for fine-grained control over what is transferred and where. By providing this support, FlexNIC is able to better utilise the network processor for networking applications, improving packet processing speeds while also reducing the memory footprint on the host. Our research does not focus on the transfer of network traffic, instead leaving that to the existing network data path present in the NFP. We focus instead on providing a medium for runtime communication between processes operating on each architecture.

2.2.7 iPipe

A similar framework to the one described in this research is iPipe (Lin *et al.*, 2018). The goal of the iPipe framework is to couple the NPU more closely with the host device and to allow data-center services to take advantage of a programmable NIC while addressing concerns relating to programmability and resource constraints. The framework allows data-center applications to offload lightweight routines that are regularly invoked onto the programmable NIC, while leaving the more complex application components on the CPU host. Interactions with the framework occur through a set of APIs.

One aspect of iPipe that is of particular interest to this research is how communication over the PCIe is facilitated. IO communication between the two architectures occurs through a set of IO channels that are established between the host and the NPU component of the application. Each channel comprises two unidirectional buffers established in the host memory. One buffer is reserved for messages being transmitted from the host to the NPU while the other contains messages from the NPU that are destined for the host.

Communication is largely driven by the host component of the framework with both buffers residing in host memory. Messages from the NPU are inserted into the receive buffer for the host to process. For detecting when a new message has been received from the NPU, two common approaches can be adopted. The host can either poll for a response from the NPU, or the NPU can generate and submit an MSIX interrupt to the host. The approach taken by the authors is to have the host poll the receive buffer rather than waiting for an interrupt from the NPU itself. This approach reduces the number of interrupts that the host kernel would need to handle but at a potential cost of increased latency. This approach to communication seems to be tailored for the Cavium LiquidIO NIC, the architecture used in the development of iPipe (see Section 2.1.4).

An interesting feature of the iPipe framework is its ability to manage and balance the bandwidth use of multiple processes spanning both the host and the NPU. This is achieved by allocating the desired bandwidths to each of the established channels and performing rate limiting based on a weighted max-min fairness method (Marbach, 2003). This component of the framework is however largely for the transmission of network traffic between the two architectures rather than for general communication.

While this message passing approach works well and would be applicable to a large range of NPUs, the overhead of two unidirectional message buffers for each channel may be great for architectures that support a large number of very small processing elements such as the NFP architectures. For this type of architecture the communication resources for each processor will need to be shared lest the communication framework require more processing resources than the intended applications themselves.

2.2.8 Axel

Another heterogeneous implementation that merits a brief mention is Axel (Tsoi and Luk, 2010), a heterogeneous computing cluster, constructed for multiple computing nodes. This cluster adopts the Non-uniform Node Uniform Systems (NNUS) topology where each node contains multiple architectures for computation but with the nodes themselves being uniform. The advantage of NNUS over the alternative UNNS[15] for this system is highlighted when considering inter-node communication. As with most clusters, nodes are connected using gigabit Ethernet or Infiniband connections over which all communication occurs.

By opting to construct the Axel cluster following a NNUS topology, the processing units

[15]Uniform Node Non-uniform System

within a single node can be closely coupled and communicate using a PCIe link. This allows intra-node communication between the disparate architectures to operate and share memory at lower latencies when compared to inter-node communication. By further tailoring the work that is performed on the cluster such that each slice involves multiple architectures, the cluster can take advantage of the spatial locality of work and maximise the portion of communication which occurs within the node.

Though not directly relevant to the research discussed in this thesis, Axel does present a notable approach to communication within a single node. The bulk of data transmission occurs using vendor supplied APIs, however control messages within a node are issued over a medium built on the POSIX interprocess communication (IPC) framework (Tsoi *et al.*, 2010). These messages are formatted to contain the destination ID and the message type being issued with the remainder of the message being largely user defined. This approach to addressing is notable and has influenced the message structure used to identify the source and destination processes involved in each message transaction within our research.

2.3 IXP Derived Research

Although this research aims to develop a framework targeting the NFP architecture family, the NFP architecture is an evolution of the original IXP architecture designed by Intel. The IXP architecture exhibits many similarities to the NFP architecture with the presence of a CPP bus, microengines, and multiple tiers of memory; albeit all on a smaller scale compared to the newer NFP architecture. Due to the similarities between the NFP and IXP architectures, literature relating to the IXP architecture is considered relevant to this research with notable contributions summarised as follows.

2.3.1 Kahn Process Networks

Due to the orientation of the architecture layout of the IXP processor towards network processing, investigations into utilising this architecture for processing stream-based applications have previously been undertaken. One approach used the Kahn process network (KPN) model as an intermediate step and then proceeded to show how stream-based applications could be converted into a format represented by the KPN model (Meijer *et al.*, 2007).

The KPN model, developed by Kahn (1974), describes a simple language which can be used to model streaming processes. Systems represented using this language are modelled as a set of sequential processes that can be executed in parallel. Communication between

these processes is implemented by describing a FIFO queue for each message channel, enabling asynchronous communication. Work by Stefanov *et al.* (2004) has shown that the target stream-based applications could be converted into a KPN model using the Compaan compiler with extensions to this study showing that KPN models could successfully be mapped to the IXP architecture.

2.3.2 Nova

The development of a custom language undertaken by George and Blume (2003) resulted in Nova, a language designed to target the IXP processor. This research was undertaken with the aim of providing an alternative programming solution to IXP targeted application development. Although this investigation focused on the IXP1200, the concepts developed would be applicable to the IXP2800 processor and potentially the NFP architecture after appropriate scaling.

The compiler for the language was designed to account for limitations associated with the IXP architecture when compared with a more conventional language. Areas of focus included register allocation and use, bank assignments, and limited data paths. These were accounted for while other limitations such as the lack of a stack and recursive structures were intentionally ignored.

2.3.3 NP-Click

NP-Click, presented by Shah *et al.* (2003, 2004) borrows the design principles of Click and extends them to produce a software architecture designed to target the IXP1200 NPU (Johnson and Kunze, 2002). The goal of NP-Click is to address a concern relating to the development of efficient and meaningful programs for network processors such as the IXP architectures. To address this, NP-Click describes a programming model intended to bridge the implementation gap between the DSL described by Click and the development of applications for the target architecture. The intent of the programming model is to simplify the development of network applications destined for the IXP architecture while still allowing the developer to leverage architecture specific optimisations related to the IXP processor.

In resolving this implementation gap, existing work was partitioned into four categories: a library of application components, a programming language, refinement from formal models of computation, and runtime systems. Each of these categories was separately evaluated before the research concluded that a runtime system would be the best approach

for NP-click. The runtime solution was selected as it would allow NP-Click to introduce managerial components such as thread schedulers and memory managers which would help hide some of the complexities relating to the development of applications for the IXP processor range.

In the framework proposed in this thesis, the concept of a runtime solution is also very appealing as it would allow for the management of resources required for communication. A runtime could then also be responsible for other meta-operations such as error checking, recovery, and resolving resource contention issues. Another category described by NP-Click which would also be applicable to our research is the refinement from formal models of computation (MoC). This class of solutions uses existing formal models or languages to provide a foundation on which an application can be implemented, usually according to a set of formal semantics for application components. While applications developed in NP-Click are intended to reside exclusively on the IXP processor, our research attempts to produce a design medium that will span both the host and the NPU architecture. To develop such a medium, a communication model that both architectures can conform to must be used. To produce such a communication model, an existing MoC can be shown to be compatible with both architectures and then the communication model can be developed using the formal semantics described by the MoC.

2.3.4 Intel Auto-Partitioning Programming

The auto-partitioning programming model proposed by Intel (2003) describes a construct referred to as a Packet Processing Stage (PPS) in which a user can specify a sequential part of an application. This model allows a system to be defined as a set of interacting PPSs with underlying architectural details such as on which microengine each PPS is executing, abstracted away. Implementing this abstracted model on the IXP architecture is the responsibility of the next-generation C compiler provided by Intel (2003). The compiler is described as being capable of optimising system aspects according to time critical paths and partitioning a PPS across multiple microengines if necessary due to code size and performance requirements.

An aspect of the auto-partitioning programming model that is of interest to the proposed research is the inter-PPS communication, described as a pipe. In the model, pipes are presented as an abstract resource with the responsibility of their implementation being left to the compiler. These pipes allow PPSs to communicate arbitrary data using the appropriate `get()` and `put()` commands and have been described as closely corresponding to the CSP computation model (Li *et al.*, 2005). The presence of additional commands

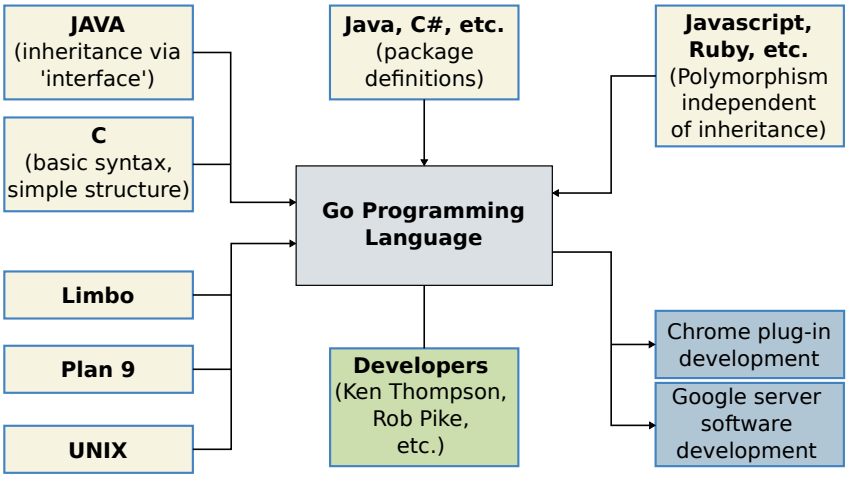


Figure 2.9: Go influence map, adapted from Balbaert (2012).

such as `is_full()` and `is_empty()` associated with a pipe, however, implies otherwise. CSP requires synchronous communication to enforce correct operation between processes, however the pipe construct described in the auto-partitioning programming model seems to allow asynchronous communication as inserting messages into the pipe is a non-blocking operation, provided the pipe is not full.

2.4 Go Programming Language

The Go programming language was developed by Robert Pike, Robert Griesemer, and Ken Thompson in 2007 in an attempt to provide a language targeting multiprocessor development (Chisnall, 2012). The goal of the Go programming language is to facilitate support for the development of low level systems as well as high level applications. The Go programming language is considered to be a member of the C-family of languages such as C++, Java, and C# (Balbaert, 2012). Language design was also influenced by languages such as Pascal and Modula with concurrency concepts drawn from CSP inspired languages such as Erlang, Limbo and Newsqueak. Figure 2.9 maps the major influences which contributed to the design of Go.

2.4.1 Go Language Design

The Go grammar consists of 25 keywords which can be considered minimal when compared to a language like C++11 which, according to the C++ development team (2019), has 84 keywords. In a keynote presented by Pike (2012), Go is however considered to be very “C-like" in terms of language design and structure. There are however, a number of key

variations which should be considered when comparing the two languages, a sample of which are listed below:

- Go does not allow for pointer arithmetic.
- Number conversions must be explicitly specified, Go does not support implicit conversions.
- Array bounds are checked in Go.
- Go does not allow for aliasing of types.
- ++ and −− are statements, and not expressions as in C.

The Go specification (Google, 2015) states that the syntax was designed as a Backus-Naur Form (BNF) grammar, however, due to the structure of the grammer, an LL(k) parser is unable to fully parse the Go syntax and so a 'look-ahead left to right' parser (Balbaert, 2012), first described by Deremer (1969), is required. This limitation is due to ambiguity between function calls and type conversions, which arise from the syntax proposed in Listing 2.1. In this situation, the parser cannot determine whether the operation is a type cast or a function call until the definition of the identifier is parsed. This definition could occur after the first use of the identifier.

```
1 //No way to distinguish type casts from
2 //function calls until name is resolved.
3
4 TypeCast(Value)
5 FunctionCall(Value)
```

Listing 2.1: Function calls and type casts cannot be distinguished without look ahead.

Go differentiates libraries from executable application code by the package declaration names that must be present at the beginning of a valid Go source file. Specifying `package main` informs the compiler that the code associated with this package is executable code, whereas specifying any other package name results in a library being produced for inclusion in a Go application (Aimonetti, 2015).

2.4.2 Concurrency

Concurrency and synchronous communication are core characteristics of the Go programming language (Kozyra, 2014). To support this, Go includes a basic primitive for con-

currency referred to as a goroutine; a play on the more traditional term coroutine that was formally described by Conway (1963). As with a coroutine, a goroutine is intended to allow independent functions or routines, which can be executed asynchronously, to be multiplexed so that they can progress in a concurrent manner (Chisnall, 2012; Google, 2017b).

Kozyra (2014) notes that the actual implementation of the goroutine primitive is abstracted away from the user and is dependent on the compiler and underlying architecture used. The asynchronous behaviour can be achieved by having the compiled application spawn a new thread for each goroutine or implementing a scheduler associated with a single thread, responsible for processing the application and any spawned goroutines (Chisnall, 2012).

2.4.3 Goroutines vs. Threads

As discussed in the language documentation (Google, 2017a), a goroutine is a function that executes concurrently with other goroutines in the same address space. Since this definition is very similar to that of a thread as seen in other common languages such as Java or C++, it is important to highlight the differences between the two.

In effect, a goroutine can be thought of as an independent function that can be resolved asynchronously and is assigned by the Go runtime to a thread for execution. The important factor here is that a single thread can process multiple goroutines concurrently, allowing for many more functions to be executed asynchronously than there are threads available. Should a goroutine executing on a thread block; all other non-blocked goroutines associated with that thread are migrated to different thread so they can continue executing (Google, 2015).

When implementing many small tasks, opting for goroutines over threads also introduces some performance bonuses, particularly in terms of setup, teardown, switching, and memory costs. The creation of a goroutine is designed to be fast and is achieved by having the creation and allocation of the routine performed in user space, thus sidestepping the overhead of requesting a new operating system thread. Given the lack of operating system involvement in goroutine allocation, the destruction of said goroutines is also associated with a performance improvement. Furthermore, the setup is cheap in terms of memory relative to creating a new thread as each goroutine is only allocated a few KB of stack space. Since this stack space is resizeable, it can scale to the requirements of the goroutine at runtime (Google, 2017b).

2.4.4 Communication in Go

As noted by Chisnall (2012), one of the chief advantages of concurrent programming in Go is the ease with which communication between goroutines can be implemented. The primary communication method between goroutines is the channel, which is a synchronous indirect message passing scheme modelled after CSP (Todorova *et al.*, 2013).

In Go, a channel is a first class object that must be typed so as to specify the kind of information that can be passed over it (Balbaert, 2012; Google, 2017b). The provided type must have a fixed size and cannot exceed 2^{16} on standard AMD64 and x86 host systems[16]. Once declared, a channel can either be explicitly passed to a function or simply used if it is within scope even if the caller is being executed concurrently as a goroutine.

Furthermore, channels implemented in Go are bidirectional and asymmetric as they support many-to-many communication. Channels can also be buffered allowing the medium to accept multiple messages asynchronously, only blocking when full. The asynchronous behaviour of the channel can be mitigated by reducing the buffer size to zero or simply omitting the parameter from the constructor (Balbaert, 2012). Goroutines can also perform the equivalent of a selective wait (Burns and Davies, 1993) by using the `select` statement and specifying multiple channels on which to wait. The `select` statement can optionally assign conditions or guards to channels that must be satisfied before they can be considered eligible for selection. Details relating to how the blocking operations are handled are omitted from this summary and instead the authors direct the reader to work by Kozyra (2014) for further information on the topic.

2.4.5 Toolchain Overview

Go was initially described as a language specification and only later implemented with the development of the **gc** and **gccgo** compilers (Taylor, 2012). Changes to the Go specification document are reflected in the implementations; however, updating the compilers to support these changes can be delayed, especially for the **gccgo** compiler which attempts to follow the GCC release schedule.

Go is a compiled language rather than an interpreted one and thus must be translated into a binary format before execution. The first implementation was the **gc** compiler, which is the default compiler used in the toolchain. It currently supports the x86, x64, and ARM instruction sets.

[16]This is an implementation limitation of the compiler and could change in future revisions.

The second implementation of the specification developed was **gccgo**. The **gccgo** compiler has longer compile times than the **gc** compiler but supports code optimisations resulting in theoretically faster execution times of up to 30% (Taylor, 2012). The **gccgo** compiler also supports more processors to bring it in line with the GNU Compiler Collection (GCC) and is now included in GCC versions 4.7.1 and later (Golang, 2015). To ensure convergence in the produced applications, core features of the Go specification are present in both implementations. These features include the memory manager and garbage collector as well as the concurrency components, namely, channels and goroutine scheduler (Taylor, 2012). The location of source files to be compiled by the Go compiler is defined by an environment variable $GOPATH which must be set to point to the current Go project directory to use the compilation tools (The Go Programming Language, 2015). Changing project workspaces requires updating this environment variable to point to the new location.

2.5 Summary

The literature presented in this chapter is intended to provide the reader with contextual information before moving forward with the body of the research. This chapter has intentionally been kept brief as additional literature is presented in subsequent chapters when it becomes relevant to the topics discussed.

To that end, the topics covered in this chapter are mostly intended to familiarise the reader with the research domain. Section 2.1 provides the reader with an overview of the architecture targeted by the proposed framework, while the subsequent summaries provide comparable architectures in the domain of network processing. Sections 2.2 and 2.3 describe the state-of-the-art in terms of communication frameworks for heterogeneous systems. These summaries help to contextualise some design decisions taken during the course of this research.

This chapter closes in Section 2.4 with an overview of the Go programming language. This review is provided to help familiarise the reader with the design and communication philosophy of the Go programming language as it is of particular relevance to subsequent chapters.

Chapter 3 follows on from the work presented here by providing more background information as well as the findings of a preliminary study undertaken to determine the feasibility of designing a communication framework spanning the CPU and NFP.

3

CSP Compliance

The primary objective of this research is to provide a communication framework that will allow processes operating on both the CPU and NFP architectures to communicate at runtime. However, before the design of such an implementation could be contemplated, a preliminary evaluation was required to determine the feasibility of such a framework. At the start of the research, it was unknown whether the NFP architecture was compatible with a runtime communication framework and if shown to be compatible, what type of communication framework would be most effective. This chapter documents this initial investigation of first evaluating supporting literature and then selecting an appropriate model or framework on which to base the evaluation. If the candidate architectures could be shown to support the mechanisms necessary to achieve the described communication model, then a communication framework would be considered feasible provided it conformed to the described model.

This chapter begins in Section 3.1 with a background of the theory of communication. This section focuses on the field of process algebras, concluding with a summary of two prominent contributions to the field, namely Calculus of Computing Systems (CCS) and Communicating Sequential Processes (CSP).

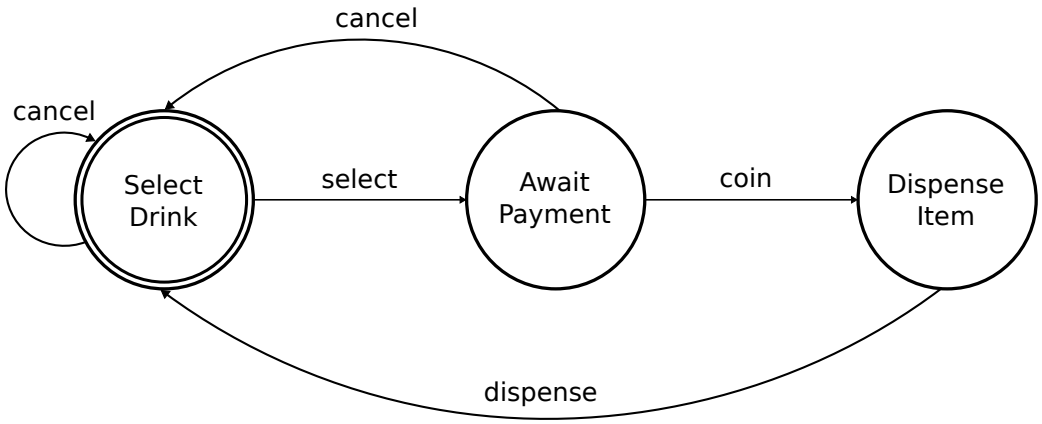


Figure 3.1: Basic LTS representation of a vending machine.

Following the presentation of formal notations used to describe process communication, Section 3.2 discusses two categories of communication commonly seen in parallel and heterogeneous systems: shared memory communication and message passing. A summary of prominent solutions that have made a notable contribution to each of these sections is presented to help support later architectural decisions on the design of the framework.

After the necessary literature has been presented, a discussion on the chosen model or formal notation for the preliminary evaluation is presented in Section 3.3. The evaluation is undertaken with different aspects of the communication model being considered. The chapter concludes with a reflection of the literature presented and an outcome based on the preliminary investigation.

3.1 Communication Theory

When considering communication theory, a good starting point is the concept of process algebras. Baeten (2005) provides a comprehensive summary of the history of process algebras and marks the beginning of the field with the introduction of Petri nets (Petri, 1962). The author notes that the field of concurrency theory began to expand in 1970 with the introduction of three distinct styles of formal reasoning: operational semantics by McCarthy (1961), denotational semantics by Scott and Strachey (1971), and finally axiomatic semantics by Hoare (1969). The work presented in De Nicola (2014) describes each of these categories as follows:

Operational semantics uses a form of label transition diagram (LTS) to model a program. An LTS describes all the possible configurations of the modelled program as a set of states. To represent the possible changes between states, directional transitions are used. Given

a single state in a program, the purpose of the transition edges is to link the current state to all possible states to which the program may transition. Finally labels are associated with each transition stating the action the program must perform to transition to the next state. Figure 3.1 provides a basic example of a simple vending machine which is only capable of dispensing drinks, all with the same price. This program has three states: `Select Drink`, `Await Payment`, and `Dispense Item`. The transitions are represented as directed edges that connect these states and, along with the relevant labels, present all the possible actions that can be taken by the program for every state.

Denotational semantics attempts to formalize the denotation or meaning behind a programming language. This is achieved by representing the expressions of the language as a set of subcomponents called *denotations*. According to Petersson (1997), the philosophy of denotational semantics can be summarised into two statements. Firstly, the semantics of a programming language is ultimately a mathematical function that maps every program produced using the language to a mathematical object. This mathematical object represents the denotation or meaning of the program. Secondly, the semantics associated with any compound syntactical construction can always be represented as a combination of the semantics of its parts. The goal of this field is to represent the capabilities of a program as a domain consisting of the constructed denotations.

Algebraic semantics attempts to present the modelled language as a set of algebraic laws that govern a set of axiomatic semantics, which in turn represents the intended semantics of the modelled language. Algebraic semantics can be used to represent abstract data types where the logical properties are based on the properties of operations that manipulate them. It is important to note that this representation is independent of the data objects themselves or their implementation.

To allow concurrent theory to evolve to support parallel systems, Baeten (2005) notes that changes to two aspects of the existing field must occur. Firstly, the use of global variables in a system model must be avoided as these variables dictate the model state. This approach to modelling would quickly become a limiting factor when considering many independently executing elements where the order in which events are resolved is non-deterministic. A resolution to this limitation is to replace global variables with variables local to each process and instead, describe the exchange of information between these variables. Another core concept that would need to be discarded is that the behaviour within the context of a model could be treated as an I/O function. According to Baeten (2005), this alteration is necessary as the interaction a process has between input and output influences the outcome, disrupting functional behaviour.

3.1.1 Calculus of Computing Systems

One of the key contributions to the field of process algebras is the work done by Milner (1980) with the introduction of CCS. The objective of this work was to model the communication between two parties where the event is indivisible and thus atomic from the context of the communicating processes. To achieve this, the author presents an algebra of processes that are based on three elementary actions: choice (alternative composition), product (sequential composition), and merge (parallel composition). This concept was later extended by Milner (1983) who proves that CCS can also be used to model asynchronous systems, resulting in the presentation of a single framework for modelling systems that interact both synchronously and asynchronously. The introductory notes in (Aceto *et al.*, 2004) provide a good overview of CCS and are recommended as a supporting document when reviewing the primary sources (Milner, 1989). Baeten (2005) also provides a short summary of CCS, chronologically ordering the publications that contributed to the original work and culminating with the current status of CCS.

When modelling a program using the CCS notation, the first aspect to note is that all processes are presented as *process interfaces* which define all the possible interactions that a process can perform on its environment as a set of directional channels. These channels are often labelled with the action that occurs over that channel, similar to how an LTS pictorially represents a state transition. These process interfaces can also be named.

The next CCS feature which is essential to describing programs is the concept of a *nil* process, simply represented by the digit **0**. This is a process that cannot perform an action and thus no further computation can be performed by a process after it enters this state. Following on from the *nil* process is the concept of *action prefixing*, a CCS process constructor that states that if a process $VMachine$ has a label *cancel*, then the resulting $cancel.VMachine$ is also a process. The process $cancel.VMachine$ is simply a process that begins by performing the action *cancel*, and then behaves like process $VMachine$.

To allow a program modelled in CCS to support multiple actions from a given process, the choice operation (+) is used. Considering the vending machine depicted in Figure 3.1, this allows the process to accept two input actions, *cancel* and *coin*, where both events are actions performed on the process by the environment. An example of this machine is presented in Equation (3.1) and shows how the choice operator allows the process to handle branching actions. Currently the only action the $VMachine$ process can partake in after receiving a coin is a *cancel* action as no further actions have been described.

$$VMachine = cancel.VMachine + coin.0 \tag{3.1}$$

Extending the vending machine paradigm, we could allow the machine to provide two drinks, $\overline{juice}$ and $\overline{water}$. These two actions are performed by the $VMachine$ process on its environment and thus the over-line is used to indicate direction. Finally, to allow the $VMachine$ to discern which of these actions to perform, one of two selection actions must initially be performed on it, namely req_{juice} and req_{water}. Now the program presented as the LTS in Figure 3.1 can be defined by a CCS model in Equation (3.2).

$$\begin{aligned} VMachine &= cancel.VMachine + req_{juice}.VMachine_{juice} + req_{water}.VMachine_{water} \\ VMachine_{juice} &= cancel.VMachine + coin.\overline{juice}.VMachine \\ VMachine_{water} &= cancel.VMachine + coin.\overline{water}.VMachine \end{aligned} \tag{3.2}$$

This simplistic implementation of a vending machine shows how the CCS notation can be used to model a single process and its associated operations. To show how processes could interact, the parallel composition operation ($|$) is used. For two given processes P and Q, the operation $P|Q$ dictates that processes P and Q can execute independently and may communicate through complementary channels. To demonstrate this, another process, $Client$, must first be introduced in the program. The actions of the $Client$ process are modelled in Equation (3.3) which depicts an actor wanting to purchase juice but may change its mind after selecting juice and instead purchase water. The resulting system combines both $VMachine$ and $Client$ processes and is presented in Equation (3.4).

$$Client = Work.\overline{req_{juice}}.(\overline{cancel}.\overline{req_{water}}.water.Work + \overline{juice}.Work) \tag{3.3}$$

$$Program = VMachine|Client \tag{3.4}$$

This brief summary depicts a basic scenario of communicating processes modelled using CCS. This summary is very limited and is intended purely to provide enough context for comparative purposes. Readers are encouraged to review the research notes of Aceto *et al.* (2004) as well as the original work presented by Milner (1989) for further information.

3.1.2 Communicating Sequential Processes

Another major contribution to the field of process algebra is the work by Hoare (1978) on CSP. Baeten (2005) notes that a key reason for the significance of CSP is that it obviates the need for global variables and instead adopts the message passing approach to communication.

To assist the reader in understanding the equations represented in this chapter, some of the basic syntax and annotations used are first discussed. This discussion is however heavily reliant on the work by Schneider (1999) and thus conforms to the presented notation. For further reading in the field of CSP, the works by Roscoe (2010) as well as Andrews and Schneider (1983) and Schneider (1999) are recommended.

CSP is a notation for representing concurrent systems involving intercommunicating components. As described by Roscoe (1997), it was presented as a notation for describing and analysing systems involving different components that interact through communication. CSP helps model these systems in a manner that fully describes all communication events, allowing the resulting model to be tested for all possible combinations of states that the system could enter. These tests or traces help identify configurations where the system could enter a deadlock or livelock and thus fail to continue executing.

As the name implies, one of the core attributes of CSP is communication. To that end, according to Roscoe (1997), there are two fundamental assumptions to communication in CSP. Firstly, communication events are instantaneous. CSP abstracts away the realtime latencies of such events such that they are atomic from the context of the communicating parties. Secondly, these events can only occur when both processes involved are capable of entering into the transaction.

Before an example model is presented using the CSP formalisms, it is necessary to introduce the core components of the notation. Firstly, CSP is used to represent or model a system of processes. Schneider (1999) describes a CSP process as an element in a system that can completely be described by how it interacts with the system or environment being modelled. These interactions are performed through *events* which represent an atomic action that the process can partake in. The set of all events which a process can partake in is said to be the alphabet of that process.

In this chapter, the convention of writing process names as uppercase and events as lowercase is adopted. A declaration is defined as a representation of a sequence of events in the form described in Equation (3.5). It is important to note that a process declaration can include a component process in its definition allowing for both abstraction and re-

cursion within a sequence of events. Process definition (3.6) elaborates on this concept by presenting the definition of a process which includes a component process (Q) which is defined elsewhere. Recursion can be implemented in this manner by having the process being defined occurring within the definition, either directly as in definition (3.7), or further buried in a component process as in definition (3.8).

$$PROCESS \;=\; sequence\ of\ events \tag{3.5}$$

$$\begin{aligned} P \;&=\; events \longrightarrow Q \\ Q \;&=\; more\ events \end{aligned} \tag{3.6}$$

$$P \;=\; events \longrightarrow P \tag{3.7}$$

$$\begin{aligned} P \;&=\; events \longrightarrow Q \\ Q \;&=\; more\ events \longrightarrow P \end{aligned} \tag{3.8}$$

The simplest process that can exist in CSP is the `STOP` process. This is equivalent to the *nil* process in CCS; it has no alphabet and thus cannot perform any actions in its environment. CSP also supports the concept of *prefixing*. If the event *cancel* is a communication event which is part of the system alphabet, then $cancel \longrightarrow VMACHINE$ is a process that begins by offering the event *cancel* and then behaving like $VMACHINE$ once it is accepted.

The next notation to consider is the transition operator. This has already been used in Process definitions (3.6) through (3.8) to show the order in which the process evolves as each component process is resolved. In the previous examples the transition operator occurred independently of any condition; however, in many situations, a process may only transition from one state to another when a specific action occurs. This conditional is represented as a label associated with the transition. An example of this is indicated in process definition (3.9) where $VMACHINE$ and $VMACHINE_{PAY}$ are component processes and the label req_{juice} describes the action required for the transition to occur.

$$VMACHINE \overset{req_{juice}}{\longrightarrow} VMACHINE_{PAY} \tag{3.9}$$

To allow a process to accept an action from a range of possible actions that can be applied by an external system, Schneider (1999) presents a menu or prefix choice operator. Process (3.10) uses a prefix choice to describe how the process will behave based on the choice made by the environment. In this situation, A is a set of possible events of which x is an element. Once x has been selected by the environment, the process behaves like $VMACHINE_{PAY}(x)$.

$$VMACHINE = x : req_{juice}, req_{water} \longrightarrow VMACHINE_{PAY}(x) \tag{3.10}$$

Finally, to enable CSP to describe communication between a process and its environment, input and output events are defined (Schneider, 1999). For example, process definition (3.11) describes a process that is allowed to accept some type of input belonging to the set req_{juice}, req_{water} from a channel labelled s before behaving like $VMACHINE_{PAY}(y)$. Output events are described in a similar manner as shown in process definition (3.12). In this situation, $juice$ must be accepted by the environment from the channel label *out* after which the process behaves like $VMACHINE$.

$$VMACHINE = s?y : req_{juice}, req_{water} \longrightarrow VMACHINE_{PAY}(y) \tag{3.11}$$

$$VMACHINE_{PAY}(req_{juice}) = out!juice \longrightarrow VMACHINE \tag{3.12}$$

3.2 Communication Models

LeBlanc and Markatos (1992) note that two of the most popular models for communication between concurrent processes are shared memory and message passing. This observation is still valid today with both models seeing implementations and use in both academia and industry.

When considering SM in the context of heterogeneous or distributed systems, the distributed shared memory (DSM) model is most often adopted. According to Lu *et al.* (1995), a core strength of such a model is the abstraction of globally shared memory. Details relating to the distribution of memory across multiple processing elements or ar-

chitectures is hidden from the user, allowing the resulting system to be treated as having a single coherent memory address space.

The message passing paradigm largely follows the model laid out by formal notations such as CCS and CSP where the only manner in which two or more concurrent processes can communicate is through the exchange of messages. As noted by Lu *et al.* (1995), message passing is more commonly associated with systems that have distributed memory. Though this is not necessarily the case in modern computing as the concept of treating memory regions local to each process as independent is still a popular solution. The resulting programs can be ported to distributed systems with minimal changes to program function.

3.2.1 Shared Memory Models

To achieve communication through DSM, two layers of transactional primitives are required. The first layer is responsible for providing and maintaining the abstracted single address space across multiple memory regions while the second provides a communication framework which resides on top of the DSM model. Implementation of the second layer is largely done through basic synchronisation mechanisms such as semaphores or locks and as such is not explained further in this section.

The transactional primitives required to produce the DSM model is however of interest and as such, three implementations are investigated. The first, IVY, is a shared virtual memory solution proposed by Li (1988) for loosely coupled systems. This is followed by two DSM based solutions: Heterogeneous DSM (HDSM) and Asymmetric DSM (ADSM). Whilst these solutions are dated, the concepts they introduce can still be considered relevant for modern computing environments.

IVY: Though the performance characteristics and networking architecture described by Li (1988) are not necessarily relevant to modern architectures, the associated framework, IVY is a proposed shared virtual memory solution which does introduce concepts relevant to modern distributed computing. The first topic discussed by Li (1988) is the concept of shared virtual memory which is a single address spaces that is shared between multiple processes. The address space is maintained by a set of mapping managers as illustrated in Figure 3.2. Each mapping manager is responsible for creating and mapping the virtual address space to memory local to the relevant CPU. The mapping managers are also responsible for maintaining coherency between the locally mapped address spaces so that all processors have read access to the same version of the shared memory.

IVY was designed as a prototype implementation to explore the feasibility of extending the

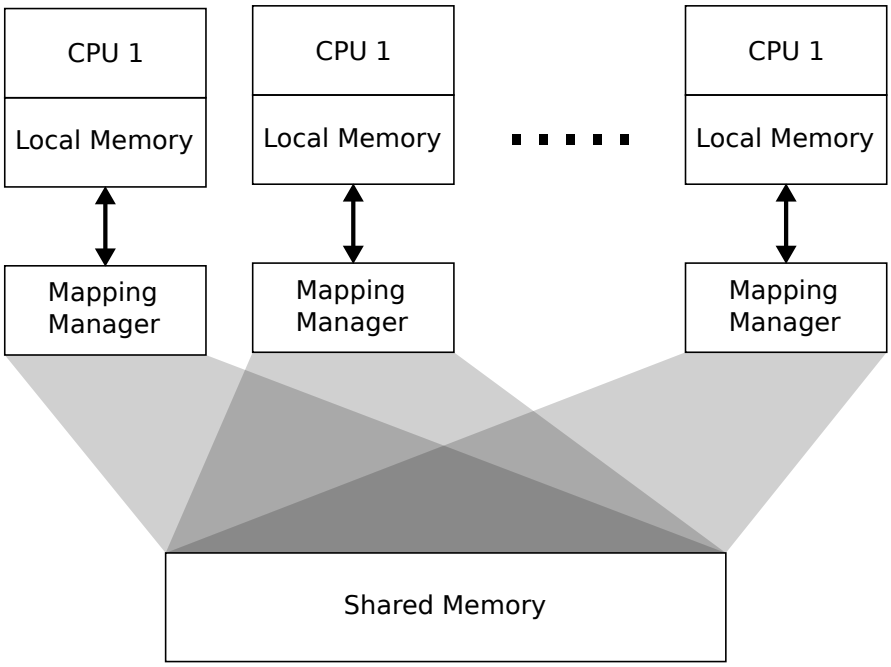


Figure 3.2: Shared virtual memory mapping, adapted from Li and Hudak (1989).

concept of virtual shared memory to span a system including loosely coupled processors. This solution used a set of modules to handle the different aspects of DSM and empirically proves functionality of the resulting framework. Due to IVY being implemented as a user-mode solution, Li (1988) notes that it does suffer from significant overheads which could be mitigated through a reimplementation as a system level solution. As the proposed solution uses the page size as the granularity at which coherency is maintained, the performance cost of maintaining a data structure smaller than the page size is the same as for the page size itself. Furthermore, using small page sizes incurs the same performance cost as larger pages if the communication medium can support a larger bandwidth per message. An interesting point highlighted by Li (1988) is the importance of selecting an appropriate page size, given that small page sizes could impact performance while larger pages could lead to contention concerns.

HDSM: Similar to the work on DSM for loosely coupled CPUs by Li (1988), Zhou *et al.* (1992) investigated the viability of extending the popular DSM model to run on heterogeneous architectures, resulting in the HDSM solution. The authors noted that even though it is expected that communication via DSM would be worse than direct message passing, it was found to be competitive in certain situations. DSM is assumed to underperform when compared to direct message passing because an existing communication medium is required on which DSM can be implemented. Message passing can instead be integrated into the underlying communication medium, reducing overhead. Zhou *et al.* (1992) argue

that two situations where DSM is preferable to message passing are large data transfers and phased execution.

In situations where data is moved between hosts in large blocks, the synchronisation aspects of message passing can become an overhead, resulting in a DSM solution being more efficient. For configurations where processing is done in phases which involve discrete compute and data exchange phases, DSM can again be a more suitable candidate for communication as bulk throughput becomes the limiting factor in the efficiency of the data exchange phase.

Zhou *et al.* (1992) provide a good summary of some of the difficulties arising from attempting to construct a DSM model for heterogeneous systems. Concerns such as endianess are still applicable to modern heterogeneous systems as this requires data structures to be converted between transfer events to conform with specifications of the target architecture. Other issues include thread management and page sizes, the latter being of particular concern for architectures that do not support memory managers. Finally the realisation of a HDSM is dependent on the existence of an inter-architecture communication medium capable of handling the memory transfer operations.

As an implementation, Zhou *et al.* (1992) presented Mermaid, an HDSM model designed to operate across the SunOS workstations and DEC Firefly multiprocessors. The details relating to this implementation are not included in this summary as they are largely architecture specific. Given that both machines provide an operating system, file system, and memory manager, the resulting implementation is not considered relevant enough to warrant further discussion.

ADSM: In software development kits such as NVIDIA CUDA and the Cell runtime management library, as well as the OpenCL specification, memory must be explicitly moved to the accelerator before a kernel operation is invoked. This requirement makes maintaining coherency between the host and the accelerator a responsibility of the developer (Gelado *et al.*, 2010).

A potential memory model that attempts to alleviate this responsibility and abstract away the explicit transfer of data between the host and the accelerator is ADSM by Gelado *et al.* (2010). ADSM provides a shared logical memory space between the CPU and accelerator devices that allows CPU processes to access shared memory objects but protects them from write operations by processes executing on the acceleration units. The rationale for this design is that it allows the CPU to maintain coherence of the shared objects as the CPU is the only element allowed to manipulate the shared data.

ADSM uses a user-level runtime system called Global Memory for Acceleration with the reference implementation designed for interactions between GNU/Linux and CUDA-compatible accelerators. This system consists of an application API, a shared memory manager, a kernel scheduler, and an abstraction layer for both the operating system and the accelerator. The shared memory manager attempts to map both the CPU memory and the accelerator memory to the same virtual address so that a single pointer can be used to refer to the allocated memory with the calling architecture dictating which memory space to access.

To maintain coherence between the accelerator and CPU memory, two protocols are discussed. The first is a batch update protocol and the second is a lazy, or rolling update protocol. The batch update protocol marks shared pages as either invalid or dirty. When invalid, the accelerator contains the latest version of the page and thus, when the CPU performs a read operation, the page must first be transferred from the accelerator memory to the host. When dirty, the host contains the latest version of the page and so it must be transferred to the accelerator memory before a kernel on the accelerator is executed. The batch update protocol operates by simply marking all pages read by the CPU as dirty and all pages transferred to the accelerator as invalid.

The lazy-update protocol improves on the batch update protocol by introducing a read-only state for pages. This state indicates that both the CPU and accelerator contain the latest version of the relevant pages. Pages in the read-only state can be read by the CPU without being marked as dirty. When a kernel on the accelerator is executed, pages in a read-only state do not need to be transferred to accelerator memory. Kernel execution does however cause all pages in read-only state to change to invalid as they may have been modified in accelerator memory.

ADSM is shown to improve the development of applications targeting accelerators by abstracting away the need to transfer memory explicitly between the architectures. This framework is designed to target heterogeneous systems where work is offloaded in batches to an accelerator that does not begin executing until directly indicated to do so. The accelerator architecture of the NPU is more attuned to stream-based processes where the system components executing on the accelerator run for the life of the application. Adopting an approach to abstract away the physical location of data in a CPU-NPU heterogeneous system would be of significant use.

3.2.2 Message Passing Models

As message passing can be implemented as part of the communication medium between system components, such frameworks are usually modelled closely on the formal notation of process algebra. Two implementations which are distinct enough to warrant presenting in this chapter are CCL and MPI.

CCL: The Collective Communication Library (CCL) as described by Bala *et al.* (1995) is a portable and tunable library that can run as a software layer on a wide range of parallel and distributed architectures. This solution is intended to be independent of the underlying communication medium and processing capabilities of component architectures.

The CCL system introduces the concept of *process groups* as a collective term for defining multiple processes belonging to a single set. Each process group is assigned a system wide name which acts as a unique identifier or process group identifier in communication events. Furthermore, each process has a unique process identifier for point-to-point communication. By default all processes associated with the system belong to a superset, the process group `ALL`. By introducing the concept of process groups to communication, CCL allows a system to identify a group and perform a collective transaction on all members of the group in a single call. CCL defines the supported collective transactions such as *bcast*, which supplies every process in the specified group with the message being broadcast.

CCL also supports communication primitives like `send` and `receive` which allow it to support basic point-to-point communication. Due to the portability of CCL, the implementation of these transactions is not formalised and instead a set of properties that the communication must advertise is provided. Firstly, these operations are considered to be *blocking* events but not necessary *synchronous* events. *Process groups* define a *blocking send* as an operation that blocks the issuing process until the message under transmission has left the output buffer. This distinction enables CCL to cater for systems where the underlying communication medium buffers messages under transmission. The *receive* operation must specify the source from which it expects to receive a message and all communication follows a FIFO paradigm so that the ordering of messages is maintained. Finally, *send* operations from multiple sources to a single destination must all be available to the destination for consumption. Provided the target system can support these requirements, a CCL implementation is feasible.

MPI: Arguably the most prevalent message passing framework in modern computing, particularity due to is popularity in high performance computing (HPC), is the message passing interface (MPI) (Rashti *et al.*, 2011; USC center for high-performance computing,

2018; Barney, 2019). The realisation of the MPI standard first began in 1992 at the Workshop on Standards for Message Passing in a Distributed Memory Environment (Walker, 1992) and later released in 1994. The standard borrows from a number of pre-existing message passing solutions (Bala and Kipnis, 1993; Bala *et al.*, 1995; Bosshart *et al.*, 2014).

An important attribute of MPI is that it is a standard and not an implementation. It defines a syntax and the semantics necessary to implement the required routines for supporting the framework. This enables MPI to be language and architecture independent, allowing applications using MPI for communication to be more easily portable across multiple platforms that support the required MPI routines. As of MPI-3, the number of routines defined is over 430 (Gropp *et al.*, 2015), enabling it to support a wide range of use cases. However, having a large collection of supported routines results in many implementations only supporting a a reduced set of them. A consequence of this is that a language or platform supporting MPI is not guaranteed to be fully compatible with all applications designed to use MPI for communication and so care must be taken to confirm compliance when porting (Gropp *et al.*, 2014, 2015).

The strength of MPI is its portability and ability to provide a layer of abstraction for heterogeneous systems. By providing support for a wide range of architectures, applications supporting communication events using MPI can rely on those communication primitives being able to operate on any architecture that boasts a compatible MPI library. This flexibility allows components of an application that communicate using MPI to be deployed for alternative architectures with the MPI library handling the necessary conversions.

Given that MPI has become a popular standard in both industry and academia Snir *et al.* (1998); Open-MPI (2019), building a set of library routines complaint with the MPI standard would result in a communication framework that is both relevant and immediately applicable. Unfortunately, in order to provide an MPI complaint interface for the NFP, an existing communication medium is required. As such a medium is not currently available for handling interactions between host processes and NFP processes, research must first focus on resolving this hurdle. Once a suitable medium is in place, the framework can be extended to support the communication primitives specified by the MPI standard.

3.3 Communication on the NFP Architecture

Considering both the formal notations described in Section 3.1 and the existing communication models discussed in Section 3.2, the initial evaluation of the proposed framework

was undertaken using CSP. Firstly, the decision to select CSP over CCS moving forward was largely driven by the host component of the framework, Go, being heavily influenced by the work presented by Hoare (1978). Go has been shown by Kappler (2016) to be fully capable of describing systems with message passing primitives that are easily mapped to the communicating events of CSP. Provided communication and concurrent process execution in the NFP architecture can also be represented using the CSP notations, a framework compatible with both architectures should be feasible.

As CSP is a formal notation, it does not describe the implementation of the system being modelled. Provided concurrent processes are able to progress and interactions between these processes can be modelled using blocking atomic events, the system can be modelled in CSP. An important consideration during the initial stages of this research was whether communication occurs through shared memory or message passing.

One of the major strengths of communicating using shared memory is the reduced overheads when attempting to transfer large data structures. This, however, is only applicable in system configurations where all components have access to a shared memory region where the components reside. For systems with disparate memory regions such as the proposed CPU-NFP configuration, data shared between the architectures would need to be physically moved to a memory region local to its destination.

In terms of existing communication models, the solutions summarised in Section 3.2.1 focus on the presentation of an abstract shared memory space between disparate processing elements. Although such an approach could be used to design a communication framework, the absence of a memory manager on the NFP to handle coherency as well as any form of dynamic memory allocation for address mapping result in such an implementation being potentially infeasible. This is further compounded by the memory architecture of the NFP being partitioned into multiple tiers with each tier potentially consisting of multiple instances as discussed in Section 2.1.1. The concept of providing a shared memory space without abstracted coherency or synchronisation is however valid and worth consideration.

The existing communication models discussed in Section 3.2.2 provide good insight into some of the common approaches taken to message passing on parallel systems. The concept of process groups presented by CCL is of particular interest given the number of contexts that can be executed on a single NFP device. The state-of-the-art in terms of message passing in parallel systems however, leans heavily towards MPI due to its adoption in many areas of both industry and academia. Considering this, MPI is a clear choice for further investigation, however, before an MPI framework can be implemented,

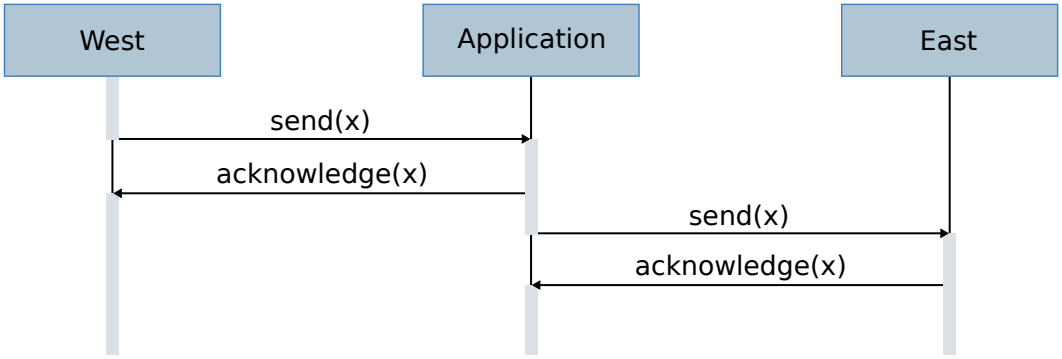


Figure 3.3: Process configuration used for the CSP test programs.

a reliable existing communication medium is needed. The intent is to instead focus on providing a stable communication framework on which an MPI implementation could be designed as future work.

3.3.1 Identification of Required Semantics

To ensure that the NFP architecture would be capable of implementing systems whose communication mechanisms have been represented fully using the CSP formalisms, the architecture must advertise support for all rules and functionality that underlie these formalisms. As a technical discussion, Kappler (2016) presents the solutions to the sample problems described by Hoare (1978) written exclusively in Go. To mimic this, a subset of the described exercises have been implemented on the NFP architecture. For the sake of brevity, only the first two examples are presented in this chapter, COPY and SQUASH, with the remainder of the exercises provided as part of the supporting digital material.

To implement these sample problems a suitable environment in which to test the solutions needed to be constructed. As both examples require the processes *east* and *west*, these two processes were implemented and are recorded in Appendix A.1. Process *west* acts as a basic producer, continuously submitting a value to process *east* and then waiting for an acknowledgement. Process *east* is a consumer process, continuously waiting for a message from process *west* and then acknowledging it. The processes *east* and *west* were implemented on separate microengines within the NFP so that they could execute in parallel.

To allow the sample problems to be implemented between *east* and *west*, a third process was added to the environment. This process acts as an intermediary, receiving, modifying, and relaying both messages and signals as required by the relevant sample problem. The final configuration of the default set-up for these tests is presented in Figure 3.3.

COPY: As explained by Hoare (1978), "Write a process X to copy characters output by process west to process east". This is effectively an implementation of a message pass event between the two relevant processes with the required NFP code presented in Appendix A.1. As discussed in Section 2.1.1, the NFP architecture performs the majority of communication between microenegines using a reflect operation. The helper function presented in Listing 3.1 provides this functionality by sending one word of the data stored in register `r_west` to register number `R_APP` on the microengine located at `APP_LOC`. Once submitted, it asserts the signal number `S_APP` on the target microengine. The process *west* can then sleep until the signal `s_west` is asserted by an external actor. The communication principle is that the signal will be asserted as an acknowledgement for the message submitted. This allows the communication event to be synchronous provided the sending process chooses to sleep immediately after submitting a message and, upon receiving the message, the receiving process immediately asserts the relevant signal on the sending process.

```
1 #define APP_LOC 5
2 #define S_APP 15
3 #define R_APP 1
4
5 __declspec(write_reg)          unsigned int r_west;
6 r_west = '*';
7 cls_reflect_write_sig_remote(&r_west, APP_LOC, R_APP, S_APP, 1);
```

Listing 3.1: Helper function for performing a reflect write operation within the NFP architecture.

SQUASH: As stated by Hoare (1978), "Adapt the previous program [COPY] to replace every pair of consecutive asterisks '**' by an upward arrow '^'. Assume that the final character input is not an asterisk.". Firstly, if the character received from *west* is not a '*', then it can simply be forwarded to *east*. Otherwise, if the subsequent character from *west* is also a '*', then submit a '^' to *east*. Finally, if the last received character is not a '*', then submit a '*' followed by the most recently received character to *east*. The NFP implementation of this example is provided in Listing A.1.

3.3.2 CSP Concurrency

The next topic to address is concurrent process execution on the NFP. Given the layout of the NFP architecture presented in Section 2.1.1, it is clear that the execution of concurrent processes in parallel is indeed possible on the NFP architecture. At the microengine level, processes can execute in an independent manner as each microengine boasts it own

processing hardware and local registers. At the level of an individual context within a micorengine however, this is no longer the case as all contexts must share the same ALU.

In the case of a more generic multithreaded processor, this would not pose an issue as a pre-emptive scheduler would ensure fair usage of the ALU amongst active contexts. In the case of the NFP microengines, such pre-emptive scheduling is not natively supported and instead, each context is expected to relinquish control of the ALU manually to allow another context to operate. The resulting issue is that we cannot guarantee fair processing time among active contexts.

Though potentially unfair time-slicing among contexts may be a legitimate concern in realtime applications, it does not technically violate the CSP formalisms, provided all contexts can be guaranteed to complete eventually. Though not pre-emptive, each microengine does include a scheduler to determine which context should receive control of the ALU after control is relinquished. The only requirement we need to enforce is that each process does relinquish control of the ALU at some point during operation, particularly at a synchronous event such as communication.

3.3.3 Recursion

As discussed in Section 2.1.1, common features present on general-purpose processors which are not included in the NFP architecture include a data or subroutine call stack. A consequence of this limitation is that applications developed for the NFP architecture must avoid the use of recursive functions. The Network Flow C Compilers User's Guide by Netronome (2008) describes the possibility of implementing a software stack using IMEM, however this is not recommended due to the comparatively large memory latencies.

Recursion however, is a common mechanism in CSP for describing processes and so careful study of the nature of such recursion must be undertaken to determine if the implementations of recursive CSP formalisms must themselves be recursive. Recursion in CSP is described by Hoare (1985) as a method of representing the repetitive behaviour exhibited by processes in a more concise notation as opposed to explicitly specifying every iteration.

Section 3.1.2 briefly touches on this concept in process definition (3.7) which describes such a recursive process. Due to the nature of CSP formalisms however, a process cannot be used as a prefix as described in process definition (3.13). A consequence of this is that process P can only appear as the last event in a process description. This concept of recursion can be extended by allowing multiple functions to be defined in terms of each other as described in Equation 3.14, resulting in the functions becoming mutually

recursive.

$$\begin{aligned} &P = event \longrightarrow event \longrightarrow P && \text{— allowed} \\ &P = event \longrightarrow P \longrightarrow event \longrightarrow SKIP && \text{— cannot include a process as a prefix} \end{aligned} \tag{3.13}$$

$$\begin{aligned} P &= F(G) \\ G &= F(P) \end{aligned} \tag{3.14}$$

As the recursive call is limited to being the last event in a process definition, the type of recursion described in CSP is thus limited to a subset referred to as tail recursion (Clinger, 1998). Tail recursion itself can be considered a restricted case of tail call where the call to the next procedure is guaranteed to be the last action taken by the current process. More formally, a process or function F can be considered tail recursive iff its definition is tail recursive. The definition is considered tail recursive iff it includes a tail call to the function F where a tail call is a function call that is guaranteed to be the last operation executed by the current function or process (Cowles and Gamboa, 2004; Burch, 2012).

An important feature of tail recursion however, is that it can always be converted to an iterative operation. Many compilers perform this optimisation during compilation with some languages such as Scheme explicitly specifying tail call optimisation as a compiler implementation requirement (Abelson *et al.*, 1998; Liu and Stoller, 1999). The result of this limitation is that processes described in CSP that exhibit a self-referentially recursive or mutually recursive definition can be implemented on the NFP architecture in an iterative manner and thus do not require a stack to be implemented.

3.3.4 Barrier

The final element considered in this chapter is the synchronisation of events. Event synchronisation plays an important role in describing the functionality of a concurrent system in CSP by enforcing that all processes that include a shared event in their alphabet must be ready to execute the event before any of the processes continue. The common implementation of this event synchronisation is to define a barrier (Welch and Austin, 2009; University of Kent, 2007) which allows an arbitrary number of processes to synchronise at a particular point before continuing.

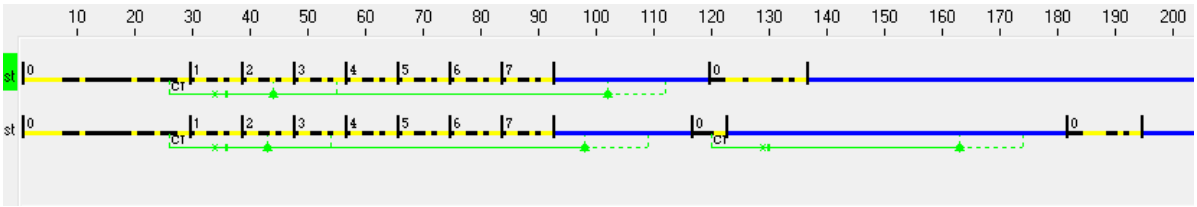


Figure 3.4: Cycle history of two threads during barrier simulation.

After due consideration, it was concluded that reflect operations and inter-thread signalling would not result in a suitable implementation of a barrier equivalent construct on the NFP architecture. The reasoning for this is due to the arbitrary number of concurrent events that may need to synchronise on a single barrier. If inter-thread signalling is used, each thread would require explicit knowledge of every other thread synchronising so as to signal each one, indicating that it is ready to synchronise. This approach would scale poorly as each process would need to reserve a signal for every other process synchronising on a specific barrier.

A better solution would be to implement the barrier as a counting semaphore in shared memory which is initialised to the number of processes that are required to synchronise on it. Each process, when reaching the point of synchronisation, could then decrement this value using an atomic decrement command and spin until the barrier has been depleted completely. Once depleted, all processes blocked by the barrier should then be allowed to continue executing.

To implement this on the NFP architecture, a simple barrier object is defined that includes a reference to a semaphore. This integer can be considered the barrier which must be depleted before all processes using the barrier as a synchronisation point can continue execution. The source code associated with this minimal implementation of a barrier is shown in Appendix A.2 with the source code executed on two microengines in Listing A.5. Simulating the application when operating on the same island produced the thread history graph described in Figure 3.4. This minimal application operated on one thread per microengine and terminated in under 200 cycles.

Implementing synchronisation in a manner that causes contexts to ‘spin’ is not an optimal solution. However, in the context of a barrier, this results in significantly less synchronisation overhead when compared to synchronous channel communication. Furthermore, the NFP architecture is well equipped to take advantage of situations where a thread could ’spin’ on a shared variable. To quantify this, consider an extension to the example code shown in Listing A.5 such that the `__ctx() == 0` limitation is removed and the

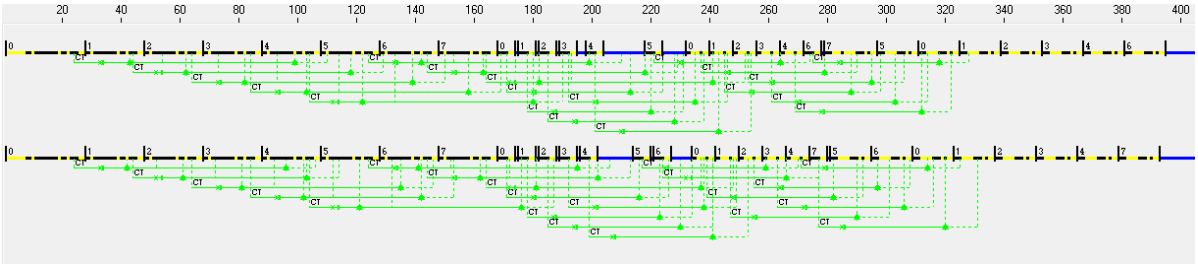


Figure 3.5: Cycle history of 16 threads during barrier simulation.

initial value of the barrier changed from 2 to 16. Running the simulation again on two microengines generated the context history trace shown in Figure 3.5. By removing the `__ctx() == 0` limitation, the application is executed by all eight contexts on each microengine, resulting in 16 instances of the application being executed. The point of this example is to show that even though half the processes required to synchronise on the barrier are on the same microengine, by enforcing that a context sleeps whenever it waits to access the lock value, the system can still complete on the NFP architecture.

3.4 Summary

The purpose of this chapter is to discuss the preliminary evaluation of the architectures involved in the proposed communication module to determine if such a model would be feasible. Section 3.1 precedes this evaluation with a discussion on process algebras and a summary of two prominent formal notations. This is followed by a presentation of different communication models in Section 3.2 with the models categorised by how communication is achieved.

After a review of both the formal notations of CCS and CSP as well as existing communication models, it was concluded that the formal notation CSP could be used to undertake this initial evaluation. As noted in Section 3.3, the host language, Go, was considered capable of implementing models defined in CSP with examples described by Hoare (1978) being implemented in Go (Kappler, 2016). A subset of equivalent exercises implemented on the NFP architecture are presented in Section 3.3.1, indicating the capability of the NFP to represent communication as synchronous and atomic events. Furthermore, the parallel nature of the NFP architecture leads naturally to the ability to support multiple concurrent communicating process. The concept of concurrency is discussed further in Section 3.3.2 where it is noted that concurrent operation in the context of a single microengine is feasible, provided close attention is paid to the cooperative scheduling mechanism of the architecture.

Finally, the chapter considers two additional mechanisms of the CSP notation in the context of the NFP architecture. Section 3.3.3 tackles the concept of recursion on the NFP while Section 3.3.4 discusses a synchronisation primitive, the barrier. It is concluded that a communication framework between the NFP architecture and Go is feasible, giving the go-ahead to the design of the framework.

As the feasibility study presented in this chapter showed that a communication framework between the CPU and NFP was viable, design and implementation of such a framework could be undertaken. Chapter 4 presents the development of the NFComms framework, a communication framework built on the findings presented in this chapter for enabling synchronous communication between the CPU and NFP.

4

Framework Design

In the previous chapter, the feasibility of producing a communication framework spanning both the NFP architecture and the host CPU was explored. As defined in Section 1.2, one of the objectives of this research is to provide such a framework for targeting the programming language Go. The previous chapter evaluated both Go and the NFP architecture and it was concluded that a communication framework based on the communication principles of CSP was feasible.

This chapter aims to realise this objective with the presentation of the NFComms framework, a communication framework for synchronous message passing between applications written in Go and deployed on the host and those executing on the NFP architecture. Due to the reactive manner in which the design and implementation was performed, the design of the framework is presented through an evaluation of its implementation. A summary of work presented in this chapter was initially published in Pennefather *et al.* (2017) with an extended version published in Pennefather *et al.* (2018a).

This chapter begins by discussing an important aspect of disparate communication systems: naming and symmetry. This section first presents a brief description of each aspect and then highlights how these communication semantics differ between various candidate

frameworks. Section 4.2 follows with an overview of how basic communication between the NFP architecture and the host kernel is facilitated. The section begins by providing a brief summary of the capabilities of the existing driver and identifying how key communication components can be introduced to allow for synchronous message passing. These communication components are discussed throughout the remainder of the section.

An equivalent discussion on achieving basic communication between the kernel and user-space applications is presented in Section 4.3. The section includes an investigation into the different approaches to providing an interface between user-space applications and the kernel as well as how to facilitate the sharing of a single interrupt across multiple driver routines. The driver API and driver extensions are presented as well as an introduction to the NFComms runtime.

To explain both the design and implementation of the message passing engines, Sections 4.4 and 4.5 discuss the operations involved in the synchronous transaction of a single message through the framework. While portions of this work has previously been published (Pennefather *et al.*, 2018a), due to additional revisions of the framework, the provided results are no longer representative of the performance capabilities of the final framework developed in this thesis. This is largely due to the inclusion of additional components for error detection and mitigation impacting overall performance. Section 4.4 focuses on messages originating from the NFP that target the host, while Section 4.5 discusses messages being transmitted in the opposite direction. For both events, all the operations required to realise a successful transaction as well as the handling of failures are detailed. This section concludes with a brief review of interrupt coalescing.

Preliminary testing of the NFComms framework is described in Section 4.6 which is split into two categories: application testing and performance testing. The focus of application tests is to present two simple heterogeneous applications that utilise the NFComms framework to operate. The goal of this testing is to show empirically that the framework is functional. This is followed by an evaluation of the performance capabilities of the implemented prototype. A summary of this chapter as well as a discussion on the findings from the testing section are presented in Section 4.7.

4.1 Communication between Concurrent Processes

Concurrent programs are generally designed to incorporate two or more cooperating processes, which are expected to be executed in a concurrent manner. Andrews (1991) notes that during execution, these processes may be required to interact at specific stages, imply-

(a) Example event, adapted from (Hyde, 1995). (b) Example synchronous event.

Figure 4.1: Abstract representation of message passing events between threads A and B.

Source	**send** message **to** *Destination*
Destination	**receive** message **from** *Source*

Figure 4.2: Abstracted syntax for message passing.

ing the need for both a communication and synchronisation primitive. For this research, synchronous message passing is introduced as such a primitive. At the application level, message passing is an event involving two or more threads within the current program that wish to exchange information. The information is exchanged in a unidirectional manner as depicted in Figure 4.1(a). Any information transmitted in the opposite direction must be presented as a separate event.

As discussed in Section 3.1.2, CSP, the formalism on which this communication model is based, requires that all threads that have initiated a message passing event cannot continue executing until the event has been resolved. As a message passing event requires the interaction of two participating processes, both processes must have reached the appropriate state in their respective execution paths before this resolution can occur. This constraint effectively enforces that message passing becomes synchronous by causing whichever thread first initiates the event to wait until the corresponding thread reaches the equivalent state.

To implement this synchronous behaviour, a message passing event usually requires at least two interactions between the participating parties. As depicted in Figure 4.1(b), the first interaction is the transmission of the message from the initiating process while the second is the acknowledgement message from the receiving process.

4.1.1 Naming

To implement synchronous communication through message passing such as depicted in Figure 4.1(b), an important attribute that must be considered is how the relevant parties are identified. Considering an abstracted implementation of message passing, Andrews and Schneider (1983) describe a syntax similar to Figure 4.2.

In this implementation, the variable *message* represents the information being passed

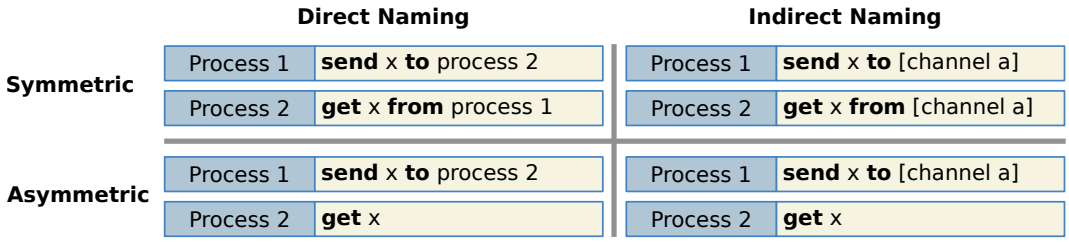
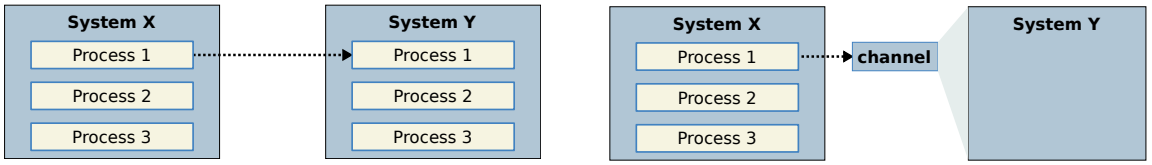


Figure 4.3: Four types of communication considering naming and symmetry aspects.



(a) Direct naming (b) Indirect naming

Figure 4.4: Abstract example of communication using different naming semantics.

from the source process to the destination process. The variables *source* and *destination* are used to allow the processes to identify other members involved in the transaction. This identification depends on the communication convention to which the event implementation conforms.

Determining the appropriate naming convention depends on the underlying hardware of the target system as well as the target application domain. According to Burns and Davies (1993), there are two elements that should be considered when defining the communication convention. The first is whether direct or indirect naming should be used to define endpoints in a communication event and the second refers to whether this communication should be symmetrical or asymmetrical. Considering the different naming aspects, synchronous message communication can be divided into one of four quadrants as shown in Figure 4.3 and described further in this section.

For a message transmission event to be resolved correctly, the transmitting process must provide a description of the destination so that the enviroment can determine which processes are expected to be involved in the event. The simplest method of achieving this is to use direct naming where some unique identifier of the target process is provided for the message transmission event (Dinning, 1989; Silberschatz *et al.*, 2005). An example of this is CCL (discussed in Section 3.2.2) which achieves point-to-point communication by assigning all processes with a unique process ID. This is equivalent to the abstracted syntax shown in Figure 4.2 which presents an example of direct naming where both communicating parties identify their counterpart processes by name.

Bertolotti and Hu (2015) note that an alternative to direct naming is to provide a communication medium such as a mailbox, message queue, or channel to act as an intermediary between the communicating parties. The specified medium is then associated with the communicating parties so that the *send* and *receive* primitives only need to specify the intermediary as the channel endpoints. Existing heterogeneous solutions such as ClickNP (Li *et al.*, 2016) and RIFFA (Jacobsen *et al.*, 2015) (discussed in Section 2.2) are both examples of this as they declare a finite set of IO channels that processes can use for communication.

One notable advantage of indirect naming is that it greatly improves flexibility of code. As the authors note, consider the environment described in Figure 4.4(a) where process X_1 from system X wishes to communicate with process Y_1 in system Y. Should direct naming be used, process X_1 would need to explicitly specify process Y_1 in the communication event and thus have to know the internal configuration of system Y. Should system Y ever change its internal configuration, the communication with process X_1 would need to be evaluated to ensure it is still targeting the correct process.

Were indirect naming to be used, system Y could effectively be replaced by a black box from the context of process X_1 as depicted in Figure 4.4(b). System X would only require that the communication medium be visible. Now, knowledge of the internal configuration of system Y is not required for process X_1 to communicate with the intermediary. System Y can be independently responsible for the internal configuration of its processes without having to check if communication with other systems has been broken.

4.1.2 Symmetry

The symmetry of a communication event describes the relationship between the participating processes. For an event to be considered symmetric, the relationship between the sending and receiving processes should be one-to-one. This relationship implies that for a particular communication event, the transmitting process can only send a message to a specific target process. The target process in turn, accepts a message only from the specified transmitter (Koymans, 1992). Symmetric communication works well in situations such as the pipeline paradigm where a process receives information from a specific upstream entity and submits processed information to a known downstream entity.

For some problems however, the one-to-one relationship enforced by symmetrical message passing on communicating parties can be too restrictive. Such problems would instead benefit from an asymmetric communication model which allows for a single process to broadcast a message to multiple receivers (Baiardi and Vanneschi, 1987).

A notable advantage of an asymmetric communication model is that it can effectively act as a less restrictive variant of a symmetric model. The notation for asymmetric communication is usually the same as its symmetric counterpart, except that the receiving process does not specify a *from* clause. An alternative presented by Burns and Davies (1993) is the selective wait where the receiving process specifies a range of sources from which a message can be received. The asymmetric model could effectively reduce to the symmetric model by simply restricting the multiplicity of partners participating in a message passing event to one. A consequence of this model is the loss of identity for processes waiting on a message from multiple sources. Knowledge of the transmitting identity must also be relayed to the receiving process along with the message so that a receipt for the message can be sent to the correct process.

4.1.3 Comparison of Message Passing in Go and the NFP

Section 2.4 provides an overview of Go including a brief discussion on how concurrency and communication are handled by the language. As noted in this discussion, communication in Go can be considered asymmetric as the channel construct is capable of supporting communication events with a many-to-many configuration. Considering Figure 4.3, it is clear that message passing in Go is positioned in the bottom right quadrant as the use of channels is an attribute of indirect naming and their implementation in Go is clearly asymmetric.

Communication on the NFP is achieved through the use of reflect operations and interthread signalling as discussed in Section 3.3. This approach to communication results in the NFP architecture residing in the top left quadrant of Figure 4.3 as the ID of each process involved in the message passing event must be specified. Furthermore, the NFP architecture does not natively support intermediaries that would allow for indirect naming. An intermediary could be created in shared memory but this risks introducing a negative performance impact on any system implementing such an approach. The architecture does however support hardware rings, which could act as the intermediary for indirect naming; however, these are limited in number thereby impacting scalability. Though both architectures support message passing that is synchronous and conforms to the message passing events described in CSP, it is clear that the two protocols are not compatible. To resolve this communication mismatch some form of translation must be implemented.

Another limitation to consider is how data is transmitted between the CPU and NFP. Section 2.1 describes the PCIe interface through which the NFP can communicate with the

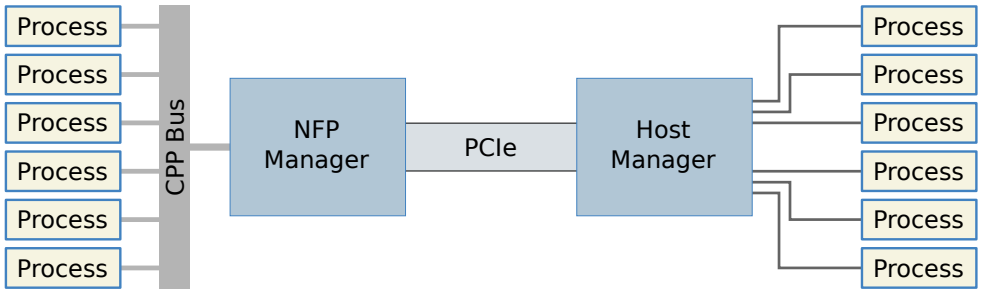


Figure 4.5: Channel engine overview.

host; however, the PCIe interface is handled by the PCIe block on the NFP (Netronome, 2016). For message passing, this means that all data in transmission will converge at the PCIe block before being transmitted over the PCIe bus or being distributed to the different NFP contexts.

4.1.4 Bridging the Semantic Divide

Given the limited number of hardware rings on the NFP and the fact that all communication takes place over the PCIe interconnect, the proposed solution is to implement a pair of dedicated channel management engines to span both the NFP and the CPU. Together, these engines will be responsible for relaying messages between the two architectures as well as bridging the semantic divide in both naming schemes and symmetry as depicted in Figure 4.5.

From the perspective of the NFP architecture, the engines act as the required intermediary which all NFP processes can address for message transmission. This allows processes on the NFP to address a single target for the reflect operation and to encapsulate the actual message in a header indicating the target process.

For a Go application executing on the CPU, interactions with the engine can simply be done through the use of a channel. From the perspective of the Go application, the channel is actually connected to the target process regardless of whether that process is being executed on the CPU or the NFP. Should the target process be executing on the NFP, the host engine is responsible for receiving, encapsulating and transmitting the message to its actual destination.

The design of each engine is distinct so as to cater for the type of message passing it is expected to process. The goal of these engines is to allow synchronous transmission without introducing excess latency on the communication events. Each message transmission is performed using a combination of NFP specific signals, shared IO memory for

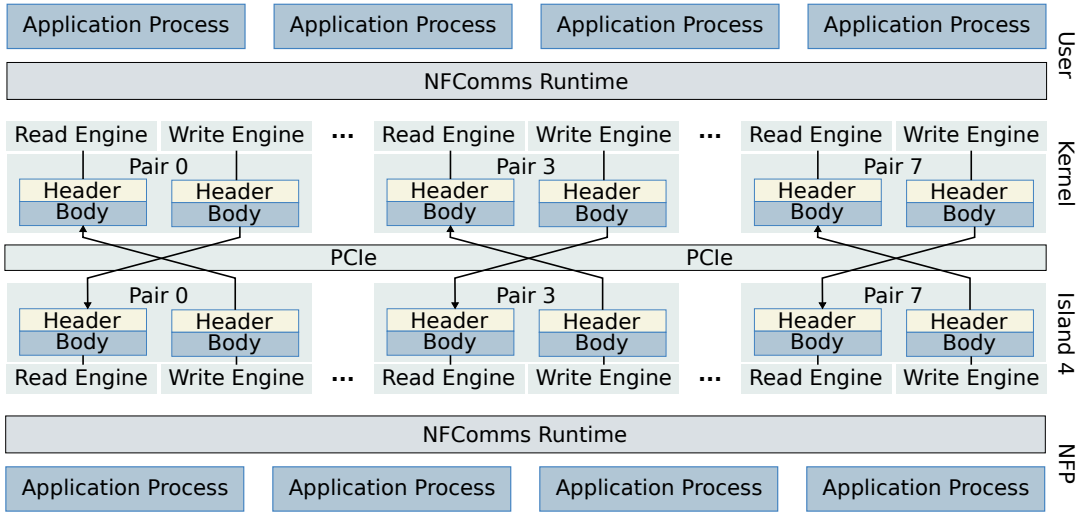


Figure 4.6: Overview of the common actors involved in a message passing event.

the message body, and in the case of the NFP Write Engine, MSIX signals discussed in Section 4.2.2.

To help describe the different actors involved in a message passing event, a basic interaction diagram is provided in Figure 4.6 which depicts the major actors involved in a message transmission. This figure is intended to provide an abstracted overview of the framework and thus only includes the major actors involved in the operation of the framework.

4.1.5 Message Passing Overview

During the initial design of the message passing fabric, it became apparent that a format for how messages were transmitted as well as what metadata in addition to the message body was required, would need to be formally defined. Given that the recipients of the messages were not only developed using different languages, but also reside on different architectures highlighted this requirement in order to simplify further development and help ensure inter-architecture compatibility.

Initial Design

During the initial stages, a single message was designed to cater for a maximum size of 4096 bytes as this coincided with the page size supported by the host. This size matched the maximum payload size that could be represented by a single DMA descriptor and thus could be processed as a single DMA transfer event by the NFP PCIe module (Netronome, 2016).

During implementation however, it was found that the proposed message size of 4096 bytes was unnecessarily large and would impact negatively on the transmission performance of notably smaller transactions. A significant contribution to the limited performance incurred by large message payloads was the inclusion of multiple memory transactions. These transactions would be required to migrate a message between shared memory locations associated with the client processes and the message passing engines situated on the NFP device.

Multiple memory transactions are necessary due to the limited number of transfer registers that could be reserved for a single transaction. As introduced in Section 2.1.1, each context executing on a microengine is only allocated 32 transfer registers, thus both the communicating NFP process and the NFP-based engine facilitating the communication would require that larger messages be transferred as multiple separate events. Not only does this complexity increase the latencies associated with a message transfer event, it also breaks atomicity.

Furthermore, storage of the collected memory would also become a concern when introducing multiple engines as each would need its own message buffer in which to collect the data before servicing it. As these engines are expected to process multiple messages in a concurrent manner, these temporary message regions may not be shared.

Message Format Refinements

During implementation and further testing, some system instability was noted, particularly when transmitting messages exceeding 56 bytes. Investigating this issue further revealed that utilising reflect operations to act as the primary mechanism for transferring data between engines without the use of shared memory imposed an additional size limitation. As discussed in Section 2.1.1, reflect operations limit the amount of data that can be transferred in a single operation to either 14 or 16 words, depending on the architecture revision. As this research utilises the NFP 400 for its implementation, the maximum amount that can be transferred in a single event is 14 words or 56 bytes.

Though the reservation of some memory is unavoidable as the engines will need to buffer a message before it can be read by the PCIe module for transmission, reserving 4096 bytes to support the initially proposed message size per engine was considered excessive. Given the limitation imposed by the reflect operations and the limited number of transfer registers that could reasonably be made available to a single context for message transmission, it was decided to reduce the message size from 4096 bytes (including 20 bytes of metadata), to 56 bytes (excluding metadata). The metadata associated with a message is required to

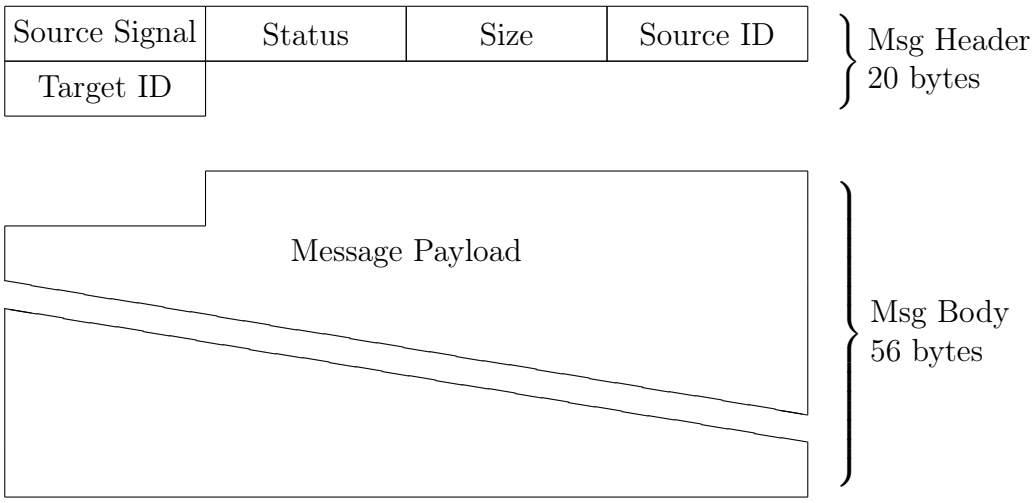


Figure 4.7: Layout of the DMA message structure.

guarantee correct transmission between the two architectures and is transferred separately.

This reduced message size would make it feasible to utilise direct message passing between the framework engines and the initiating or receiving processes on the NFP device. This limited memory size would also reduce the amount of CLS memory required by each of the 16 engines to 76 bytes for PCIe communication. The message structure is presented in Figure 4.7 with the different fields detailed as follows:

- **Source Signal:** A signal associated with the source NFP engine, which can be used by the host to relay either an acknowledgement signal or an initialisation signal.
- **Status:** A reserved field for indicating the status of the message under transmission. A list of the possible statuses is presented in Table 4.1.
- **Size:** The size of the message payload in words which must be between 1 and 14. This field is used by the host NFP driver to determine the size of the message buffer.
- **Source ID:** A field containing the ID of the initiating process which is used for assertions and error checking when allocating messages to channels.
- **Target ID:** Identity of the destination process. For messages emanating from the Host, this field contains both the signal to assert on the target context and the register into which the engine return address will be written. For messages emanating from the NFP, this field contains an ID associated with the channel over which the message payload must be relayed.
- **Message:** The actual message payload to be transmitted to the destination process.

Table 4.1: Possible states for the DMA message struct.

Code	State Name	State Description
0x88	DMA_STATE_IDLE	The engine is in an idle state and is ready to handle a new transaction.
0xFF	DMA_STATE_ERR	An error has occurred during transmission.
0x01	DMA_STATE_READ	The engine has received a read operation.
0x02	DMA_STATE_READ_ACK	The engine has acknowledged the read operation.
0x03	DMA_STATE_READ_FAIL	The engine has acknowledged the read operation but has failed to transmit.
0x04	DMA_STATE_WRITE	The engine has received a write operation.
0x05	DMA_STATE_WRITE_ACK	The engine has acknowledged the write operation.

4.2 Communication Between Host and NFP

Before attempting to develop the proposed framework, one additional aspect that had to be considered, namely how communication is physically handled between the host and the NFP. Currently the NFP physical function (PF) driver is capable of providing communication support for uploading firmware to the architecture as well as an interface for a host of tools to access memory regions and registers for debugging purposes. The driver is also capable of presenting the NFP device as a set of virtual network interfaces for the transfer of network traffic both to and from the device.

4.2.1 Existing Driver Review

An evaluation of the existing NFP driver indicated that two key features required to realise message passing between the two candidate architectures were missing. The first was a feature to allow applications operating on the NFP device to trigger events on the host, effectively allowing an NFP-based process to indicate the availability of messages that require host interaction to resolve. With the current driver this could be implemented by having the host poll reserved regions of RAM that are associated with the device. This however, was not considered an acceptable solution for the proposed framework as message latency is a significant concern.

The second feature considered necessary for the implementation of a message passing fabric is a common memory region accessible by both the host and the NFP device. During the initial development stages of the message passing system, this was implemented by reserving a region on Base Address Register 0 (BAR0) of the NFP device which was addressable by both the host kernel and applications running on the NFP. Though

functional, this approach resulted in the shared memory for the framework being situated on the NFP device itself and so the CPU must perform a non posted read operation to access the message data. The consequence of a non posted read is that the CPU stalls until a completion Transaction Layer Packet (TLP) is received from the endpoint device (Lawley, 2014), consuming resources and introducing increased read latency. Write operations in this situation are posted transactions so no completion TLP is required from the NFP device to indicate a successful transmission and as a result, the operation could exhibit a much higher throughput.

Aside from these limitations, further reviews of the NFP driver itself and the support it provided in many other aspects of device operation made it clear that developing a new driver would not be a feasible solution. Instead it was proposed that the existing driver be extended as it would provide an excellent platform on which extensions could be implemented.

4.2.2 MSIX

The proposed approach to supporting the first missing feature was to extend the driver to detect MSIX (Coleman, 2009; Xilinx, 2014) which the PCIe module, contained in the NFP device, is capable of generating at the request of an internal process. These interrupts would allow applications operating on the NFP to signal the host via the driver, indicating that an event had occurred. For this implementation such signals would be used for synchronisation as they would allow applications within the NFP to send an event signal to the host driver which could be interpreted to indicate that a message related event had occurred. As the current driver already allocates an MSIX interrupt vector for general operation, this feature was implemented by having the driver register an additional MSIX interrupt. For the message passing system, only one interrupt signal would be necessary as additional context would be provided by the state in which the message passing engines reside when the interrupt is detected.

The alternative to interrupt generation is polling as used by iPipe (discussed in Section 2.2.7). The premise of polling is to disable interrupts altogether, effectively removing the overhead relating to interrupt handling from the system. Instead, the host CPU periodically polls the target memory for new elements. There are two main drawbacks to polling. The first is that a polling event is never guaranteed to find an element to process. In the context of the research presented in this thesis, this often occurs when components of the heterogeneous system are involved in a compute event and not communicating. During such times, the polling for events is a waste of CPU resources. The second issue

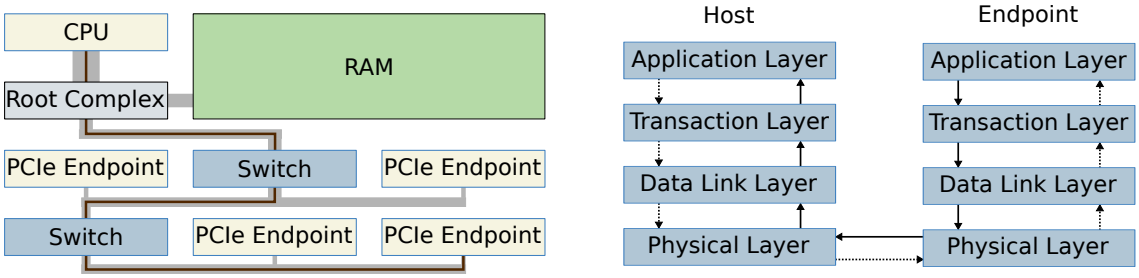
is latency related. When elements do arrive at the host for processing, they can only be handled when detected during a polling event. If such elements arrive just after a polling event, processing is delayed by the wait time between polling events, introducing additional latency.

4.2.3 DMA Mapping

The second modification needed is the reservation of a continuous region of physical memory which can be performed by using the *dma_zalloc_coherent()*, provided as part of the Linux kernel DMA API (Pinchart, 2014; Bottomley, 2017). When called, this function allocates a consistent memory region of a specified size that can be used until freed. As part of the function input parameters, a reference to a DMA address variable is required. The physical address for the beginning of the allocated memory block in RAM is inserted into this address. Once resolved, this function returns a virtual address for where the physical memory is mapped to the CPU virtual address space for access. This function also handles interactions with the IO Memory Managment Unit (IOMMU) for white listing of the NFP device when attempting to access the reserved memory region. The advantage of using consistent memory is that both the NFP device and the CPU are able to access the data as it is currently stored in the reserved region without being concerned with caching. As the NFP device is only capable of 40-bit addressing, a 40-bit DMA address mask is first applied to the NFP device handle before the function call so that the returned physical address is limited to reside within this address space (Miller *et al.*, 2017).

As the current implementation of the message transfer fabric is expected to support up to 16 concurrent message passing events, the memory to be reserved by *dma_zalloc_coherent()* must reflect this. Each message passing event requires one DMA structure as described in Figure 4.7 and so a memory region large enough to contain 16 instances of this structure must be reserved. For simplicity, this memory can logically be represented as an array of 16 elements with the first eight being allocated to the transmission of messages from the NFP device to the host. The second eight are allocated to messages passed in the opposite direction.

To allow the NFP device access to the reserved memory region, the 40-bit bus address returned by the *dma_zalloc_coherent()* must be stored in a location that is accessible by the NFP message passing engines at startup. As the previous message passing implementations used BAR0 for communication (see Section 4.2.1), support was already in place for allowing processes on the NFP device to access the physical address by having it written



(a) TLP transmission (b) Protocol layers

Figure 4.8: Interactions between endpoint and host over PCIe.

to BAR0 by the host on driver initialisation.

4.2.4 PCIe

As an aside, it is important to briefly discuss the PCIe architecture due to its relevance to the message passing system. As noted by Budruk *et al.* (2003), PCIe is considered to be the third generation of bus interconnects used on computing platforms and was designed as an evolution of PCI.

Though considered the current standard, PCIe maintains many similarities with its PCI predecessor such as maintaining a similar transaction layer for read and write operations and the use of device configuration through BARs instead of physical jumpers on the device itself (Solomon, 2014). An advantage of maintaining this approach to I/O operations is to allow for backward compatibility of host drivers written for PCI on modern architectures. From an abstracted overview, both PCI and PCIe operate in the same manner however, these differ greatly in their implementation. While both PCI and PCIe perform serial communication, PCI operates as a bus topology with a single bus servicing multiple endpoint devices while PCIe operates more as a star topology which implements switch based technology to create and manage multiple interconnects (Shanley and Anderson, 1999; Budruk *et al.*, 2003).

Communication between the host CPU and a PCIe peripheral occurs by having the CPU initiate a read/write transaction targeting the intended peripheral. This operation is also referred to as a programmed IO instruction (Budruk *et al.*, 2003). The front side bus of the host CPU however is not directly connected to the bus associated with the target peripheral and the instruction is instead received by the route complex which directs the transaction down the correct route towards the specified peripheral. Completing this operation may require routing the transaction through multiple switches depending on the

architecture layout before it reaches the bus physically associated with the destination peripheral as illustrated in Figure 4.8(a). To achieve reliable communication between endpoints on the PCIe fabric, each application level transaction must pass through three layers associated with the PCIe protocol as described in Figure 4.8(b).

The first layer of the PCIe protocol is the transaction layer which is responsible for the encapsulation of the application data into a TLP (Ravoori *et al.*, 2017). All transactions generated by this layer occur as two separate operations. In the first operation, the TLP is handed to the data link layer for transmission while the second operation involves the reception of an acknowledgement TLP from the data link layer some time after the initial operation. This two stage operation is referred to as a split transaction and allows multiple transactions to occur at once (National Instruments, 2014).

The second layer is the data link layer which is largely responsible for the reliable delivery of each TLP passed down from the transaction layer (National Instruments, 2014). The data link layer encapsulates the TLP into a Data Link Layer Packet (DDLP) which applies a packet sequence number and a cyclic redundancy check (CRC) before handing the packet to the physical layer for transmission (Ravoori *et al.*, 2017).

The final layer is the physical layer which performs the actual transmission of a DLLP onto the PCIe fabric. This layer is responsible for serialising the received DDLP before transmitting it across the bus to the downstream node where it will be de-serialised back into a DDLP. The number of bits in the stream which are transmitted in a single cycle depends on the width or number of lanes of the bus. Each lane represents two pairs of differential signals allowing for full duplex communication per cycle (National Instruments, 2014).

4.2.5 DMA Initialisation

Before the different actors depicted in Figure 4.6 which are involved in a message transfer event can be discussed, the startup sequences required by some message passing components within the NFP architecture must first be described to provide better context as to how some events are resolved during operation. Ensuring the system includes an initialisation stage is essential for allowing the NFP message passing engines to properly set up memory regions and acquire bus addresses for performing DMA read and write operations. As the NFP component of the system includes two types of message passing engines, the NFP Write Engine and the NFP Read Engine, two different DMA initialisation sequences exist. The sequences however are very similar and both can be described by Algorithm 1.

if *DMA initialisation requested* **then**
1. Create a message offset;
2. Setup **dma_descriptor_configuration** and write it to descriptor config 0;
3. Create and populate a DMA descriptor with default values;
4. Read in initial page address from BAR0;
5. Offset page address and update DMA descriptor;
6. Setup PCIe module address to point to DMA descriptor high priority queue for future operations.;

end

Algorithm 1: Initialisation procedure for DMA engines.

Initialisation begins at step **(1)** where each engine determines its relative offset based on the context on which it operates. As detailed in Section 4.2.3, the reserved memory region in host RAM is divided into 16 elements; one for each of the message passing engines. Rather than write the associated page address for each engine to the NFP BAR0, the memory region is mapped as a continuous region and only the initial offset is written to BAR0, minimising the amount of memory on BAR0 required for this framework. The message offset that each engine must calculate is the starting location of its specific region with respect to the provided physical address.

Interactions between the NFP device and the PCIe bus occur through a dedicated module referred to as the PCIe module. This module accepts read and write requests from processes executing on the NFP architecture in the form of DMA descriptors which must be written into one of six descriptor priority queues. These descriptor priority queues are split into two sets, read and write, with three queues in each. The three queues are used to represent high, medium, and low priority with the priority indicating the order in which they are evaluated for work. Work allocated to a lower priority queue is not evaluated unless all higher priority queues are empty.

For a DMA operation to be serviced by the PCIe module, a minimum of two components are needed. The first is the DMA descriptor which must be written by the initiating process[1] and provides the PCIe module with configuration details and basic information required for the operation itself. The second component is the population of a DMA descriptor configuration register, depicted in Figure 4.9, which provides further details and context to be associated with a DMA descriptor.

For this implementation, most of the configuration details associated with a DMA transfer do not need to be modified as the system is only using the PCIe module to perform basic

[1]In the context of this system, this is one of the NFP message passing engines.

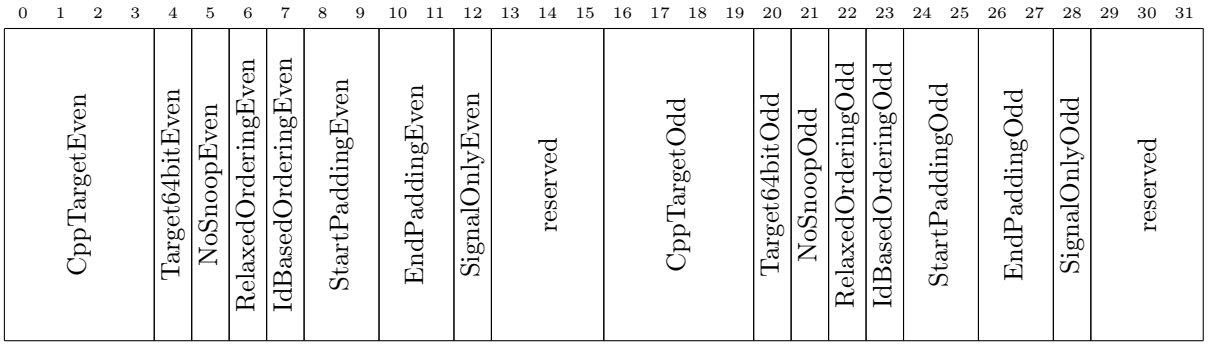


Figure 4.9: Layout of the DMA configuration register.

read and write operations. With this in mind, the DMA descriptor configuration register can be set to zero in step **(2)** of Algorithm 1 with the exception of the bits associated with defining the memory type of the CPP bus target (Netronome, 2016).

The DMA descriptor provides the PCIe module with a reference to the memory region where it must read or write data for the DMA operation; however, as this reference does not tell the PCIe module with which memory type it is associated, this information must be provided as the CPP bus target field in a DMA descriptor configuration register. For this implementation, all memory references are with respect to the CLS on Island 4 and so the CPP bus target field on the DMA descriptor configuration register can be populated appropriately. Considering the components of the DMA configuration descriptor which need to be specified for a message passing operation to be completed, the following information must be provided:

- The memory address of where to find the data to be transmitted or where to put the received data.
- The DMA descriptor configuration register which contains additional information.
- The type of mode to use for the DMA operation (we are using normal DMA with auto push).
- The page address within the host memory where the message must be delivered or from which data must be requested.
- The size of the message (in bytes) which must be read or written.

For write operations, once a DMA descriptor is written by the initiating process into one of the appropriate queues, the PCIe module then reads from the specified memory reference the number of bytes indicated in the descriptor. This data is stored in an allocated internal memory buffer of the PCIe module before being inserted into a TLP

for transmission over the PCIe bus. Once the TLP has successfully been submitted, the initiating process is signalled, informing it of the completed transmission. Read operations happen in a similar manner from the context of the initiating process; the only difference is that the data is instead read from the host page address and stored in the specified CPP bus address.

Returning to the initialisation procedure in Algorithm 1, once the DMA descriptor has been populated with default values, the initial page address is read from BAR0 by performing an internal PCIe read request in step **(4)**. The returned result is a 40-bit address spread across two words. The initially calculated offset is then added to this address giving the page offset for the host DMA message structure reserved for the current engine. This page address is inserted into the DMA descriptor and the PCIe address is set to point to the high priority descriptor queue to be used in future operations.

At this point the DMA initialisation sequence for an NFP Write Engine is complete, however the NFP Read Engine requires one additional step. In order for a Host Write Engine to interface with an available NFP Read Engine, it needs to be able to uniquely identify it. The current implementation facilitates this by requiring that each NFP Read Engine initially populates the *source_signal* field in the DMA message struct (Figure 4.7) allocated in the host RAM. This *source_signal* can then be used by the modified NFP driver and an existing library to generate an interthread signal within the NFP device, informing the corresponding NFP Read Engine that there is data available for it to handle in the host RAM. To populate this field, a DMA write operation is requested by each NFP Read Engine to write a signal address to the first field in its allotted DMA struct. Once complete, the NFP Read Engine can then sleep on the relevant signal until a message is available for it to service.

4.2.6 Interrupt Coalescing

An import attribute of the proposed framework which must still be addressed, particularly in terms of the generation of MSIX interrupts, is interrupt coalescing. Interrupt coalescing[2] is a term more commonly used when referring to network traffic related communication or high speed secondary storage and is used to describe the delaying of interrupt generation for additional traffic (Prasad *et al.*, 2004). The result is a reduction in the number of interrupts generated as multiple traffic events can be coalesced into one interrupt. Traditionally, as explained by Ahmad *et al.* (2011), interrupt coalescing is intended to optimise two parameters; the maximum interrupt delivery latency (MIDL) and

[2]also referred to as interrupt moderation

maximum coalesce count (MCC).

Salah (2007) observes that interrupt coalescing can be divided into types: count-based and time-based. Count-based coalescence describes approaches where interrupts are only generated after a set number of elements are available that the host CPU must be made aware of. In the case of this research, these would be the message events involved in message transmission between the architecture engines. The goal of this interrupt coalescing type is to allow a single interrupt to represent a batch of elements, effectively reducing the number of interrupts by the size of the batch. The second interrupt coalescing type is time-based coalescing. The goal of this type of coalescing is to delay the generation of an interrupt by a fixed interval. During this time, additional elements may become available to the host CPU and thus, once the interval expires, a single interrupt can be used to represent multiple elements. Although very similar, the differentiating factor between time- and count-based interrupt coalescing is that the former issues an interrupt after a fixed duration while the latter only issues an interrupt after a fixed batch size is available.

For this research, the goal is to minimise the latency associated with a message passing event. To that end, a pure polling implementation was not considered due to the concerns raised in Section 4.2.2. Considering the framework, all message passing is synchronous and the current implementation supports a total of 16 engine pairs, thus the total number of queued interrupts at a given time can be at most 16. Furthermore, due to the framework not explicitly being designed for applications performing phased execution as discussed when reviewing communication models in Section 3.2, no assumptions can be made on when communication events will occur.

Such assumptions however are only a concern for situations where the message originates from the NFP architecture. For communication events originating from the host, no polling would be required as the transmission of the associated message to the NFP begins in the driver. For the subsequent transactions within these events however, polling is optimal due to the synchronous nature of the messages. Once a message transaction has been initiated, the driver can then poll a relevant memory region for the acknowledgement rather than waiting for an interrupt.

Considering communication events originating from the NFP, it is feasible for an entire application to only entertain a single message passing event. This would result in the driver wasting CPU resources in continuously polling for new messages in a situation where very few actually occur. Regarding the two types of interrupt coalescing, the same reasoning can be applied to the count-based approach, as generating fewer than the minimum number of events over the entire application could result in an interrupt

never being generated. Time-based interrupt coalescing on MSIX interrupts is thus the approach taken for detecting messages originating from the NFP, although the number of outstanding elements is also taken into account.

```
1. Set credits to 6;
2. Increment outstanding_interrupts;
while outstanding_interrupts is not 0 do
    if (credits − outstanding_interrupts) <= 0 then
        3. Attempt to acquire a lock for interrupt generation;
        if lock acquired then
            4. Set outstanding_interrupts to 0;
            5. Issue an interrupt;
            6. Release lock;
            7. exit;
        end
        else
            8. Decrement credits by 1;
            9. Sleep;
        end
    end
end
```

Algorithm 2: Pseudocode for performing basic interrupt coalescing of MSIX interrupts.

Table 4.2: Recorded latency/interrupt impact for different coalescent credits.

Credits	**Interrupt Drop (%)**	**Latency Increase (%)**
0 credits	0.00	0.00
1 credits	4.18	1.17
2 credits	7.07	1.32
4 credits	6.37	4.01
6 credits	11.63	5.21
8 credits	15.73	8.71

When an interrupt is requested by one of the eight NFP Write Engines, the issuing thread enters the interrupt generation routine described by Algorithm 2. This routine begins by allocating credits to the process in step **(1)**. Although there can be a total of 8 events at a given time, the default credit number is set to 6. This credit value was empirically selected to give the optimal decrease in generated interrupts for increase in latency as presented in Table 4.2. Following the credit allocation, step **(2)** increments the shared *outstanding_interrupts* variable using an atomic increment operation. This indicates to all instances of the interrupt routine that there is an additional communication element outstanding.

The routine then enters a loop until the number of outstanding elements recorded by *outstanding_interrupts* is 0. The loop begins by checking if the number of available credits has either degraded to 0 or the number of outstanding messages has equalled or exceeded the number of available credits. In either case, this indicates that the current thread should attempt to issue an MSIX interrupt. The current thread then attempts to acquire a shared lock using a set of atomic operations in step **(3)**. If this fails, it implies that another thread is in the process of sending an interrupt at which point the current thread simply degrades its credit count in step **(9)** before sleeping. As the NFP uses cooperative scheduling, the sleep operation is very important as it allows other engine threads to operate and provides the interrupt routine with a short delay before attempting another iteration through the loop.

Should step **(3)** succeed, the current thread can safely issue an MSIX interrupt. Before doing so, the current thread first sets the *outstanding_interrupts* to 0 so that other instances of this routine can be made aware that the current thread is about to generate the interrupt. The current thread then issues the actual interrupt in step **(5)** before releasing the acquired lock. As the current thread has now issued an interrupt, it can immediately exit the routine.

4.3 Communication between Kernel and Host Application

By mapping a continuous DMA memory region on the host and acquiring the physical address associated with the region, interactions between the driver and the NFP device can be facilitated. Catering for interactions with the message passing component operating as part of the host application however still requires further modifications. Firstly, an interface must be provided through which the host component can establish communication with the driver and secondly, a mechanism must be implemented to somehow block or interrupt the host to indicate that the NFP driver has received a message from the NFP device which requires further processing. Section 4.2.2 concluded that MSIX interrupts could be used to achieve this, but the handling of such events must still be determined.

4.3.1 Host Communication

Though both the driver and host application operate on the same architecture, they cannot directly interact unless a communication interface between the two actors is established. The proposed approach is to implement a runtime similar to what was described by NP-Click in Section 2.3.3. To establish this, two approaches were considered.

The first approach was to use the `\proc` file system which is commonly used by the kernel to publish information for user-space applications to read. The content of these files is generated by an associated function when a read operation against them is initiated. Write operations to these files operate by calling the associated write function in the kernel to process the character stream received. As noted in Corbet *et al.* (2009) however, use of the `\proc` file system was originally intended for running information and use of this directory for more generic I/O operations is discouraged.

Though use of this file system for the purposes of this research is discouraged, it was decided that a test implementation should nevertheless be developed for evaluation. To test this approach, the NFP driver was modified to create an entry in the `\proc` file system during initialisation. To allow interactions between the DMA message queues and user-space applications, function handlers for the read and write operations were also designed. These handlers were allocated to a *file_operations* struct which was then used to create an entry in the Linux `\proc` file system using *proc_create()*. Using this approach allowed user-space applications to perform read and write operations on the generated file which were then relayed as messages to the NFP device.

An alternative approach that was preliminary investigated was to use the `\sys` file system which, as noted in the Linux Kernel documentation (Mochel and Murphy, 2011), was initially based off `ramfs` and was designed as a directory to house interfaces to kernel objects such as drivers. The `\sys` file system was introduced as a more structured alternative to the `\proc` file system and is intended to be used by kernel modules to export information to user-space rather than having such information exposed through the `\proc` file system. Due to similarities in function between `\proc` and `\sys`, the `\proc` implementation was adapted to test use of the `\sys` file system and it was found to function in the same manner. Given that the reason for introducing the `\sys` file system was to migrate driver interactions away from the `\proc` file system, the latter approach was considered a more viable solution.

The third approach investigated was the use of system calls to facilitate this stage of the communication. A system call can be succinctly described as a request by the application to the kernel to perform an operation or provide a service on behalf of the caller (Love, 2010). From the context of the user-space process, interactions with a service via a system call operate in the same manner as reading or writing to a file, simplifying application development. By utilising system calls to interact with systems operating in kernel space such as device drivers, the kernel can also implement basic access control by setting the user access permissions on the system file descriptors (Love, 2010).

Rather than attempt to create and register a set of new system calls within the Linux kernel, the IOCTL system call is used with handler functions for the additional IOCTL numbers. Long (1989) described IOCTL as a function through which a variety of control functions for a specific device can be performed. For this implementation, the IOCTL interface was required to represent six control functions: *Read()*, *Write()*, *ReadAcknowledge()*, *ReadFail()*, *Shutdown()*, and *Startup()*. Each of these entries corresponds to a separate control function implemented in the driver as further explained in Section 4.3.3.

After testing the above approaches, it was decided that the current implementation would use IOCTL to provide communication between the NFP driver and applications executing in user-space. This approach was selected predominantly for simplicity and ease of relaying and copying references between user-space memory and the kernel.

4.3.2 Kernel Wait Queue

Having determined that MSIX interrupts would be used as the primary mechanism for allowing the NFP device to signal the kernel, we now consider handling of such events. To incorporate an event mechanism into message passing interactions between the host

application and the NFP driver, an event-based synchronisation, implemented using a kernel wait queue[3], was proposed to store waiting processes. By implementing this wait queue within the NFP driver, functions requesting event related services do not need to account for situations where the event related resource is not currently available, instead allowing this to be the responsibility of the NFP driver.

When a user-space application requests a resource, the handling function within the driver first checks the availability of the resource and if not present, it can mark itself as sleeping and remove itself from the set of runnable threads for selection by the scheduler. When the handling function marks itself as sleeping, it must also select one of two states: `TASK_INTERRUPTABLE` and `TASK_UNINTERRUPTABLE` (Love, 2010). In the context of this research, the sleeping mechanism is used to allow a process to wait for an MSIX interrupt from the NFP and so the first of the two states is used exclusively (Corbet *et al.*, 2009).

Once a process has been marked as sleeping, it is assigned to the driver wait queue so that it can be woken up by the queue when a new event occurs. As all processes assigned to the queue in this manner will be woken up on any event, a wakeup condition must also be associated with a process when it is allocated to the queue. Upon wakeup, the process can first check if this condition has been met and if not, return to the wait queue until the next event occurs.

4.3.3 Host Driver Interactions

```
1 void InitDriver(int fd);
2 void WriteToDMA(int fd, int buffer, int source_signal, unsigned int size,
3                 unsigned int source_id, unsigned int target_id,
4                 uint32_t* message);
5 void ReadFromDMA(int fd, int buffer, uint32_t* meta , uint32_t* msg);
6 void AcknowledgeRead(int fd, int buffer);
7 int  FailRead(int fd, int buffer);
8 void RequestShutdown(int fd);
```

Listing 4.1: Function prototypes for NFP API in C.

As noted earlier in this section, to implement communication between the host application and the NFP driver, six control functions are required. These functions are implemented in the driver and an associated API library has been designed in C for interacting with them from applications operating in user-space. The prototypes of the API functions are provided in Listing 4.1. To allow the API to be interfaced with from the NFComms

[3]Functions associated with this structure are described in `linux/wait.h`

runtime, a library of wrapper functions was also implemented. The wrapper library was necessary to handle translations of parameters from those defined in Go to their equivalent types in C. The functionality of the six API functions is described as follows.

InitDriver: This initialisation function must be called during the initialisation of an application and is responsible for ensuring that the NFP driver is in a state where it is ready to accept DMA messages from both the host and the NFP device. Currently this function is limited to clearing the shutdown state and zeroing the DMA buffers so that no false message events are generated during the initialisation period.

WriteToDMA: The DMA Write function, as its name implies, is responsible for performing a DMA write operation on behalf of the host application. To achieve this, the host handler function requires a buffer number and a list containing four message specific parameters which are packed into a single *HostDmaMessage* structure. The structure content is as follows:

- **Size:** This field contains the size of the message payload. This must be a value between 1 and 14 due to limitations discussed in Section 4.1.5. This field is used by the NFP Read Engine to determine the number of registers to allocate to this message. The framework also requires that the size field supplied to the DMA write function corresponds with the number of words expected by the NFP destination process.

- **Source ID:** This field contains the ID of the initiating process which is used for assertions.

- **Target ID:** This field contains the ID of the destination process which is a single 32-bit value that contains both the signal to assert on the target context and the register in which to write the engine return address as part of the communication event.

- **Message:** The actual message payload in bytes. Currently payloads have a maximum size of 56 bytes due to the limited buffer size discussed in Section 4.1.5.

ReadFromDMA: This function is designed to mirror the *WriteToDMA* from the perspective of the host application and so accepts the same parameters. Unlike the DMA Write function however, the initial contents of this *HostDmaMessage* structure is not relevant and are overwritten with the particulars of the message received from the NFP device. The structure is however still required so that the appropriate memory is reserved by the calling application for storing the message.

AcknowledgeRead: Once a read operation has been completed, the host must acknowledge the operation. To achieve this, the Read Acknowledge API call should be used. This function is intended to be called after a DMA read event has occurred and is used to unblock the NFP Write Engine which is associated with the current message transfer event. Unlike the data oriented IOCTL commands, only the buffer number of the *HostDmaMessage* structure must be supplied so that the relevant NFP Write Engine can be identified.

Shutdown: The final control function implemented was the Shutdown function. During implementation and testing it became apparent that a stable method of unblocking all threads associated with the NFP driver and clearing the DMA buffers was required. Initially this presented an issue as multiple threads within the host application operated in a loop where they would either be processing a message or be blocked in the NFP driver, waiting for a message to process. To resolve this, a shutdown flag was added to the NFP driver which would be set as part of a shutdown event and cleared as part of a startup event. Any interactions with the flag would have to be performed via atomic operations[4] to protect against read errors. When asserted, the intent of the shutdown event is to unblock all threads and stop all communication with the NFP device which relates to message passing events.

4.3.4 Driver Functions

To better elaborate on the operations associated with the different API calls which form part of the NFComms framework, the associated handler functions must be detailed. Associated with each API call is a unique IOCTL number which is supplied along with the necessary parameters when a call to the driver is made. This IOCTL number is registered with the NFP driver on startup and used by the interrupt register queue (IRQ) manager to determine which handler function should be used to process the supplied parameters.

When the API function *WriteToDMA* is called, a reference to the supplied *HostDmaMessage* structure along with the buffer number is passed as a parameter to the IOCTL command with the appropriate number for the DMA Write Handler. The driver IRQ handler then relays the parameter to the DMA Write Handler whose functionality is described in Algorithm 3. The DMA Write Handler begins by first copying the message struct reference and message payload from user-space into an allocated kernel space region where it can be interacted with directly in step **(3)**. The message parameters are

[4]For this implementation, the operations described in `asm/atomic.h` were used.

```
if message write event requested then
    1. Allocate space for a HostDmaMessage structure in kernel memory;
    2. Allocate space for a nfp_dma_message structure in kernel memory;
    3. Move user-space HostDmaMessage structure to allocated kernel memory;
    4. Populate nfp_dma_message structure;
    5. Assert dma_message.source_signal;
    6. Periodically check shutdown and dma_message.status ;
    if shutdown set then
        7. return SHUTDOWN;
    end
    if dma_message.status == DMA_STATE_WRITE_ACK then
        8. return SUCCESS;
    else
        9. return ERROR;
    end
end
```

Algorithm 3: Pseudocode for the DMA Write Handler.

then used to populate a *nfp_dma_message* structure indexed by the buffer number in step **(4)**. Once populated, the function asserts the current *nfp_dma_message* structure source signal in step **(5)**, informing the associated NFP Read Engine that there is a message available for it to process.

At this point, the DMA Write Handler must wait until the current message has been copied to the NFP device so as to ensure that no premature writes occur which could cause data loss or corruption. To implement this, the DMA Write Handler periodically checks to see if *dma_message.status* has changed or if a shutdown event has occurred. The expected value of *dma_message.status* if changed is DMA_STATE_WRITE_ACK, indicating a successful transmission. If any other state is received, a panic event is generated. Further information relating to the different status values is provided in Section 4.1.5. This approach is noted to be a suboptimal solution as it introduces a superfluous DMA write operation by the NFP, however this does provide an added layer of error checking to help maintain a stable framework which is favoured during the prototyping stages.

If the API function *ReadFromDMA* was called, the DMA Read Handler, described in Algorithm 4, is used to handle the request. This handler begins by first checking if the system is in an active state in step **(1)**. This is done in case the host application attempts to allocate read threads to the NFP driver when there is no active application, resulting in those threads blocking indefinitely. If the system is in a shutdown state, this function immediately returns.

```
if message write event requested then
    if shutdown flag set then
        1. return SHUTDOWN;
    end
    2. Allocate space for a HostDmaMessage structure in kernel memory;
    3. Move user-space HostDmaMessage structure to allocated kernel memory;
    4. Enter the wait queue and sleep until woken;
    if shutdown flag set then
        5. return SHUTDOWN;
    end
    if dma_message.status == DMA_STATE_READ then
        6. Copy message from nfp_dma_message to HostDmaMessage structure;
        7. Move HostDmaMessage structure back to user-space;
        8. dma_message.status = DMA_STATE_IDLE;
        9. return dma_message.size;
    end
    if dma_message.status == DMA_STATE_ERR then
        10. return ERROR;
    end
end
```

Algorithm 4: Pseudocode for the DMA Read Handler.

Following the shutdown test, the DMA Read Handler allocates memory in the kernel for a *HostDmaMessage* structure. The contents of the user-space *HostDmaMessage* structure is then copied into this allocated space in step **(3)**. Once copied, the handler allocates itself to the kernel wait queue in step **(4)** until woken. The wakeup condition for this blocking event is whether the status field of the DMA message struct indexed by the supplied buffer number has its status field change, or if a shutdown event has occurred.

When awoken, the status field is checked to confirm a DMA_STATE_READ has been received. The DMA read handler then populates the *HostDmaMessage* structure with the received information in step **(6)** before the structure is returned to user-space. The *dma_message.status* field is then set to DMA_STATE_IDLE in step **(8)** to stop it from generating duplicate events. Finally, the handler exits, returning the number of words transferred to the caller of *ReadFromDMA*.

The final handler function to be discussed is the DMA Read Acknowledge Handler, which is used when *AcknowledgeRead* is called. The supplied buffer number is used by the handler to index the correct *nfp_dma_message* structure and read its associated source signal field. Before any signal is actually generated, the handler function first checks if the system is in an active state and returns immediately if not. As with the DMA Read Handler, this check is done to protect against the situation where the NFP Write Engines may not

be in a waiting state, or the source signal field has not been initialised. In either situation, generating a signal risks putting both architectures in an unstable state. If the system is in an active state, the *dma_message.status* field is first set to DMA_STATE_READ_ACK before the source signal is used to generate a signal event which is relayed to the NFP device. The handler function then polls the *dma_message.status* field for a status change to DMA_STATE_IDLE. In this situation, the driver is expecting a response for the NFP driver shortly after generating the signal event. As the overhead of allocating the handler function to the kernel wait queue as well as the generation and handling of an interrupt is considered comparatively expensive, polling over a relatively short duration is favoured. Once *dma_message.status* has been changed to DMA_STATE_IDLE, the operation has completed successfully and the handler can safely return. If any other status value is received, an error has occurred and this is propagated to the caller of *AcknowledgeRead()*.

4.3.5 NFComms Runtime

Before continuing with the presentation of the communication framework, it is important to first introduce the host-based runtime (referred to as the NFComms framework) which facilitates communication between the host applications and the driver. In particular, the goal of the runtime is to provide the communication layer between the API functions discussed in Section 4.3.3 and the host application in a manner that abstracts away the setup, teardown, and runtime procedures.

The NFComms runtime is constructed around the *NfpComms* data structure that contains persistent elements required throughout the execution of a heterogeneous application. Associated with this data structure are a number of functions, both internal to the runtime, and available to the host process interfacing with it. The persistent elements are listed as follows:

writeChan: As discussed in Section 4.1.5, both the host and NFP architectures communicate using a set of dedicated managers or engines. On the host, there are a total of 16 engines, eight for receiving messages from the NFP and eight for sending messages. All engines execute as separate routines although the use of the Host Write Engines needs to be balanced to maximise performance. The proposed solution to this is to use the channel *writeChan* to serialise traffic before presenting it to the Host Write Engines. All messages destined for the NFP architecture are inserted into *writeChan* and all Host Write Engines can act as consumers on the channel. Traffic is then allocated to the Host Write Engines on a first-come-first-serve basis.

shutdown: To manage the safe exiting of the runtime, a dedicated channel available

to all internal routines is needed to broadcast a shutdown message when the runtime is required to exit.

wg: In conjunction with the *shutdown* channel, the WaitGroup *wg* is required to assist the exiting routine in ensuring a safe shutdown. Whenever an internal routine is initiated, the *wg* counter is incremented, indicating the existence of an additional thread. Before an internal routine exits, it decrements this counter. The exiting routine can simply wait until the *wg* counter is zero, implying that all internal routines have exited before completing the shutdown procedure.

closed: The *closed* flag is used to indicate that the runtime has entered the shutdown procedure. Once set, any attempt to read or write from the runtime immediately returns with an error indicating this.

connections: The final element of the *NfpComms* structure is a map containing connection elements. Each connection element consists of an address and a channel which can pass a structure containing a *HostDmaMessage* and an acknowledgement channel. This map effectively lists all valid host channels with which the NFP architecture can communicate. The address element contains the value which must be supplied by an NFP process in order to communicate with that channel. Details pertaining to the other elements in the channel are presented as part of the implementation discussion in Sections 4.4 and 4.5.

```
1 func New(number uint32) NfpComms
2 func (n NfpComms) AddConnection(id uint32, island int, microengine int,
3                                 context int, signal int, reg int) (err error)
4 func (n NfpComms) Start() (err error)
5 func (n NfpComms) Stop() (err error)
6 func (n NfpComms) Send(id uint32, data interface{}) (err error)
7 func (n NfpComms) Receive(id uint32,  data interface{}) (err error)
```

Listing 4.2: NFComms runtime user functions.

To interact with the NFComms runtime, the functions presented in Listing 4.2 are made available to user-space applications. The purpose of each of these functions is described as follows:

New: When interfacing with the NFComms framework, *New()* must be called to create an *NfpComms* object. This object is returned by the function and is required in all future interactions with the framework. The number supplied to this function is currently not used but required for future revisions. Future work on the NFComms framework includes

introducing support for hosts containing multiple NFP devices. This number will be used to identify the relevant NFP device.

AddConnection: As discussed earlier in this section, the *connections* map contained in the *NfpComms* object lists all connections registered with the framework. To add a connection to this map *AddConnection()* is provided. This function accepts a unique ID for the connection as well as the physical location of the target NFP-based process with which the connection should be associated. This location is described by providing the island, microengine, and context of the target process as well as a signal number and register which is reserved on the target process for communication. To support intra-architecture communication, the address parameters can all be set to zero to indicate that the destination process resides on the host. This is done for completeness of the framework so that a single approach to communication can be used throughout an application. For performance purposes, it is recommended that more traditional channels discussed in Section 2.4 be used instead.

Start: Once the framework is ready to be initialised, *Start* must be called. This function spawns the 16 engine routines and updates the *NfpComms.wg* field to reflect this. At this point the NFComms runtime is ready to interface with the NFP Read and NFP Write engines.

```
1. Create a 14 word array message for the message;
2. Use ID to select the registered channel con;
if selection failed then
    3. return error;
end
4. Try pack data object into message;
if packing failed then
    5. return error;
end
6. Create an acknowledgement channel ack;
7. Construct a DmaMessage;
if NfpComms.closed is set then
    8. return error;
end
if con.address == 0 then
    9. con.channel <- DmaMessage;
else
    10. NfpComms.writeChan <- DmaMessage;
end
11. <- ack;
```

Algorithm 5: Pseudocode for a Send operation via the NFComms runtime.

Owing to how channels function in Go, it is not possible to read a value from a channel without consuming it (Google, 2015). Therefore, any intermediary within the communication chain effectively acts as a buffering element, increasing the number of elements that can be written to a channel before it blocks. This poses a major issue as it is no longer possible to have a synchronous transaction when relying exclusively on the channels provided natively by Go. To remedy this, message passing takes place through the *Send()* and *Receive()* functions that handle transmission and synchronisation internally.

Send: *Send()* is a blocking operation that can be used to send a message to a process executing on the NFP architecture. This function accepts an *ID* and *data* element as parameters. The function begins by allocating a static 14 word wide array in step **(1)** of Algorithm 5. This array is used to submit the message body through the host component of the framework. The supplied *ID* must match a previously subscribed connection in step **(2)**, which can be achieved using *AddConnection()*. If no matching connection is found, an error is generated.

The *data* element is a generic interface which must be static and cannot exceed 56 bytes in size. If the object does not meet these requirements, the packing procedure in step **(4)** fails and an error is generated.

An acknowledgement channel as well as the supplied data and connection details collected from the *connections* map are used to construct a *HostDmaMessage* object in step **(7)**. Before transmission can occur however, *NfpComms.closed* is checked to confirm that the framework is not in a shutdown state. Next the connection address is checked to see if the message is intended for a process local to the host and if so, the *HostDmaMessage* is sent directly to the target channel in step **(9)**. If not, the *HostDmaMessage* is submitted over *NfpComms.writeChan* where the first available Host Write Engine services it. To maintain a blocking event, *Send()* must then wait until a message is received over the acknowledgement channel, indicating successful transmission of the message.

Receive: To receive a message from the NFComms framework, *Receive()* must be used. This function accepts an *ID* and *data* element as parameters. The *ID* indicates which registered connection the calling process expects to receive data from and the *data* element presents the location where the received message will be written. As with *Send()*, the supplied *ID* must match a previously subscribed connection or an error will be generated. This function operates by first acquiring the connection channel associated with the supplied *ID* and then waiting until a message becomes available for consumption. The consumed *HostDmaMessage* is then decoded and written to *data*. Should the decode routine fail due to insufficient size or an invalid location, an error is generated. A mes-

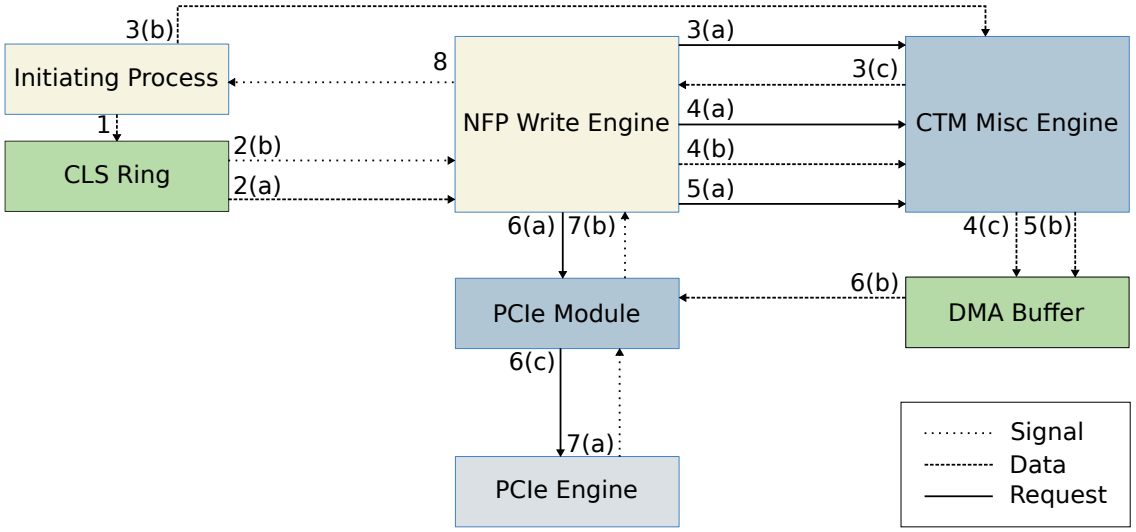


Figure 4.10: Overview of message transmission from the NFP to the PCIe bus.

sage is then sent over the acquired acknowledgement channel regardless of whether the decode routine completed successfully to protect the framework from entering an unpredictable state. This acknowledgement message informs the framework that the message was received and is propagated back to the initiating process.

4.4 NFP to Host Transmission

The transmission of a single message from an NFP-based process to a host-based process can be broken down into four stages divided between the two perspectives. Figure 4.10 provides an overview of the different actors involved in the event within the context of the NFP device, though for simplicity, some of the interactions between message related actors and memory actors have been omitted. Each event within the diagram is numbered to show the order in which each action must be completed to move the message from the initiating process to the PCIe bus as well as relay the acknowledgement signal from the PCIe bus back to the initiating process. To better explain this sequence of operations, the message passing event is first explained from the perspective of the initiating process, and then from the perspective of the NFP Write Engine responsible for handling it.

4.4.1 Initiating the Message Transfer Event

For a message transfer event to occur, the initiating process must reach a point within its execution where it performs a message passing operation. At this point, the process can prepare and initiate a message transfer event by following the sequence of events

detailed in Algorithm 6. The sequence begins with step **(1)** where the message data is first stored in a *xfer_message_buffer* which can only be read by actors external to the initiating process. Once the message data has been moved, the source ID of the initiating process must be generated in step **(2)**. In situations where the destination process resides on the host, the source ID is used by the NFP Write Engine servicing a request to identify the associated requesting process as well as where to find the message data and how much to read. In situations where the destination process resides on the NFP, this data is used by the destination process to determine where to find the message data as well as which signal to assert upon successful transmission. In the current revision of the framework, the source ID contains the following information:

- The island on which the initiating process resides.
- The microengine containing the initiating process.
- The specific context within the microengine responsible for executing the initiating process.
- The register offset of the write transfer array relative to the first register associated with the executing context.
- The number of consecutive words in the write transfer array containing valid message data.
- The signal number associated with the signal that the initiating process sleeps on until the message transfer event has been resolved.

```
if message write event requested then
    1. Move message data to the xfer_message_buffer;
    2. Construct a transfer request message and insert it into the xfer_request_buffer;
    if destination_id > 500 then
        3. Construct a reflect_write_address;
        4. Request CT send the xfer_request_buffer to the destination context;
    end
    else
        5. Request the CT move the xfer_request_buffer to hardware ring 0 on island 4;
    end
    6. Sleep on the signal encoded in the xfer_request_buffer;
end
```

Algorithm 6: Pseudocode for initiating a write operation from the NFP.

Along with the source ID, the destination ID of the target process is also required. To make the communication framework handle both inter and intra-architecture message

passing events, messages originating from the NFP and destined for the host must have a message ID under 500. Message IDs exceeding this value indicate that the destination process resides on the NFP architecture and a NFP local communication routine should be used. The upper limit on message IDs was not selected due to its significance to the framework but rather as a baseline for this implementation. Should more message channels be required between the host and NFP, this value can be changed.

In situations where the ID is less than 500, it is used by the Host Read Engine servicing this event to determine which channel the received message should be relayed over. In the situation where the ID is greater than 500 it is actually the destination address of the target process, containing the same components as the source address although the actual values are specific to the destination context. Both the source and destination IDs are coupled to form a transfer request token which is written into the *xfer_request_buffer*.

For intra-architecture communication, a *reflect_write_address* is first constructed in step **(5)** by unpacking the destination ID into its components. This address is then used in step **(6)** to request that the local CT engine perform *reflect_write* event to move the *xfer_request_buffer* to the destination context and signal it on the signal number provided as part of the destination ID.

For inter-architecture communication, the initiating process submits a request to have the *xfer_request_buffer* copied to a hardware ring which is situated in the CLS of Island 4 in step **(5)**. This allows the request to be treated as a work unit to be processed by subscribed NFP Write Engines. Referring back to Figure 4.10, this process represents event ① of the transfer operation.

At this point, the initiating process has completed its involvement in the message transmission stage of the event as the remainder of the message pass request is handled externally. For intra-architecture communication, the remainder of the transfer is the responsibility of the destination process which must copy the message body into its local register and then signal the initiating process upon completion. For inter-architecture communication, this is handled by the next available NFP Write Engine.

The initiating process can now safely sleep in step **(6)** of Algorithm 6 until the signal submitted as part of the source ID is asserted. This allows other contexts to operate on the host microengine. Once the message stored in the write transfer array has been received and acknowledged by the intended recipient, this signal is asserted, either by an NFP Write Engine as represented by event ⑧ in Figure 4.10, or by the NFP local destination process, allowing the initiating process to continue execution.

4.4.2 Servicing the Message Transfer Event

Continuing with the sequence of events depicted in Figure 4.10, a message transfer request token has been submitted by the initiating process to the CLS hardware ring on Island 4. The application executed by each of the NFP Write Engines is described in Algorithm 7.

During the initialisation phase of the system, a set of NFP Write Engines are initialised and subsequently register themselves as work units for the CLS hardware ring. This process is presented from step **(1)** to step **(8)** in Algorithm 7 where context 0 is responsible for setting up the hardware ring in CLS while the remaining seven contexts block until this has been resolved. Step **(5)** is important as it forces all contexts that reach this point in the application to release control of the ALU. As discussed in Section 2.1.1, thread management on a single microengine is achieved through cooperative scheduling and thus, failing to allow other threads to execute at this stage of the application could cause context 0 to never complete initialising the hardware ring, causing the NFP Write Engine to deadlock. Once initialisation of the hardware ring is complete, DMA initialisation as detailed in Section 4.2.5 is performed in step **(7)**. When the transfer request token is submitted to the ring, a registered NFP Write Engine is assigned to the request token and woken up in step **(8)**.

Now that an NFP Write Engine is actively servicing the write request, it first updates the *source_id* and *destination_id* fields of the *nfp_dma_message* with the values supplied in the request token. In step **(9)**, the supplied source ID is used to construct a *reflect_read_address*. This address is then supplied to the CT engine in step **(10)** to request that it handles the transmission of the message body from the write transfer registers on the initiating process to the *message_xfer_input* on the active NFP Write Engine. This procedure is represented by event ③ in Figure 4.10. As the message data is not relevant to the transmission process, this data can immediately be transferred to the *nfp_dma_message* structure as represented by event ④ in Figure 4.10. To achieve this, the data must first be moved from *message_xfer_input* to *message_xfer_output* in step **(12)**. As the *nfp_dma_message* structure resides outside of the microengine, transferring data to it can only be performed by first moving the relevant data into a set of write registers, in this case the *message_xfer_output* array. The message body can then be transferred to the *nfp_dma_message* structure in step **(13)**.

After the message body has successfully been stored, the remaining attributes of the *nfp_dma_message* struct can be populated in step **(14)**. The layout of this struct is discussed in Section 4.1.5 and the fields are populated accordingly. This additional data is then also moved to the DMA message struct in event ⑤ of Figure 4.10.

```
1. Create a nfp_dma_message struct in CLS on island 4;
2. Set up the message_xfer_input and message_xfer_output arrays;
if context 0 then
    3. Initialise the CLS hardware ring;
    4. Increment ring_lock using an atomic instruction;
end
else
    while ring_lock == 0 do
        5. Perform context switch;
    end
end
6. Initialize DMA;
7. Populate nfp_dma_message struct with DMA address and source_signal;
while the NFComms runtime is active do
    8. Subscribe to the CLS hardware ring for work;
    9. Update nfp_dma_message with source and destination IDs;
    10. Use source ID to construct a reflect_read_address;
    11. Request that CT read message data into message_xfer_input array;
    12. Copy message_xfer_input to message_xfer_output;
    13. Move contents of message_xfer_output to nfp_dma_message;
    14. Populate outstanding fields of nfp_dma_message;
    15. Call DMA write subroutine to send nfp_dma_message to host;
    16. Sleep until signalled by host;
    17. Read nfp_dma_message from host RAM to CLS;
    if dma_message.status == ACK then
        18. Signal initiating process;
    end
    else if dma_message.status == FAIL then
        19. Return work unit to back of CLS hardware ring;
        20. Set dma_message.status to IDLE;
        15. Call DMA write subroutine to send nfp_dma_message to host;
    end
    else
        20. Set dma_message.status to ERROR;
        21. Call DMA write subroutine to send nfp_dma_message to host;
    end
end
```

Algorithm 7: Pseudocode for the NFP Write Engine.

With the relevant components of the DMA struct populated, the message is finally ready to be submitted to the PCIe module for transmission over the PCIe bus in step **(15)**. This event is equivalent to event ⑥ in Figure 4.10. During the initialisation stage of the system, each of the NFP Write Engines initialises a DMA descriptor as discussed in Section 4.2.5 as well as acquiring the physical addresses for where the populated DMA message structures should be written into host RAM. Using this information the active NFP Write Engine can perform a DMA write operation to relay the DMA message from CLS memory to host RAM. As part of this submission, the handling function must also generate a MSIX interrupt immediately after the DMA transfer is complete to indicate the occurrence of a message related event to the host architecture.

Once the message has successfully been moved to the host architecture, the NFP Write Engine can simply sleep on an acknowledgement signal which the host asserts when the message transfer event is complete as indicated by step **(16)**. This assertion is propagated from the PCIe Controller Core to the NFP Write Engine by the PCIe Module in event ⑦ of Figure 4.10.

When asserted, the active NFP Write Engine must confirm correct transfer before completing the transaction. This is achieved by first copying the *nfp_dma_message* from the host to the NFP and then checking the status field of the message. If the status field contains state `DMA_STATE_READ_ACK`, the message has been acknowledged and the active NFP Write Engine can submit an acknowledgement signal to the initiating process by asserting the signal number that was passed to the active engine as part of the work unit. If the status is `DMA_STATE_READ_FAIL`, the message was correctly transmitted but was not accepted by the destination and thus must be queued for retransmission.

In step **(19)** of Algorithm 7, the retransmission is handled by placing the message request token back in the hardware ring. The active NFP Write Engine then sets the *nfp_dma_message* status field to `DMA_STATE_IDLE` before returning the *nfp_dma_message* to the host in step **(20)**. This additional step is needed to inform the host that the failed message has been acknowledged and handled by the active NFP Write Engine. If any other state described in Table 4.1 is received, the *nfp_dma_message* status field is set to `DMA_STATE_ERROR` before being returned to the host. If the host receives a message status `DMA_STATE_ERROR`, a panic event is generated as this indicates a communication fault. Future implementations could include an extension for handling and recovering from such situations.

Once the active NFP Write Engine has responded appropriately to the *nfp_dma_message* status, the message transfer event can be considered complete, whether successful or

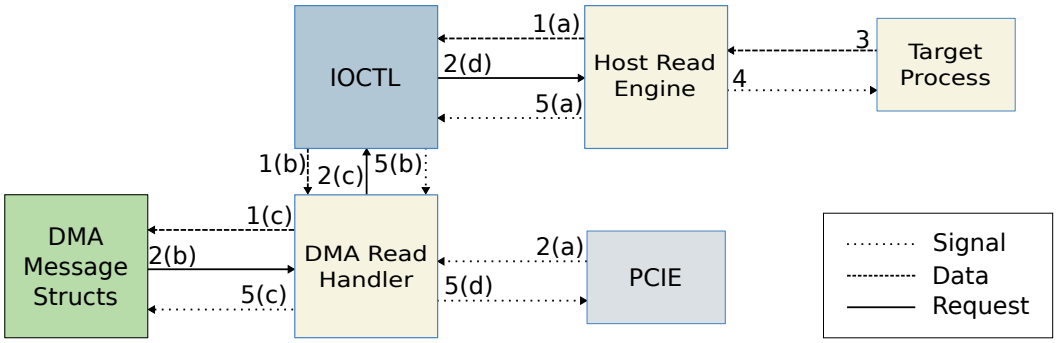


Figure 4.11: Overview of the different events involved in the reception of a message from the PCIe bus.

otherwise. The active engine can resubscribe itself to the CLS hardware ring, allowing it to service future message transfer events.

4.4.3 Reception via the NFComms Runtime

When considering the remaining operations that must occur to resolve the current message transfer event, it is easier to switch perspective from that of the NFP device to that of the host architecture. Figure 4.11 provides an overview of the different actors involved in the message transfer event from the perspective of the host as well as their interactions.

Before the current message transfer event can be resolved by the host, the Host Read Engines must have been initialised as part of the application startup sequence. For this implementation, eight Host Read Engines are established, one for each of the eight host DMA read buffers.

Currently the Host Read Engine is implemented as a routine that is spawned when *Start()*, discussed in Section 4.3.5, is called by the host component of the application. This routine is associated with the current *NfpComms* object and has access to the associated elements. As a result, the Host Read Engine only requires a buffer number as an input parameter when called. As eight instances of this function are executed, the buffer is necessary to act as the distinguishing element, allowing the routine to index one of the DMA buffers discussed in Section 4.2.

Operation of a Host Read Engine is described in Algorithm 8 and is designed to run for the life of the current application. The function begins in step **(2)** by calling *ReadFromDMA*. This operation requires the supplied buffer number and a reference to a message struct in which to store the returned data. This is a blocking operation that is detailed in Section 4.3.4. When this operation resolves, it implies that one of three situations has occurred:

```
1. request = DmaMessage;
while system is running do
    2. state = ReadFromDMA(request, bufferID);
    if state == 0 then
        3. return;
    end
    if state < 0 then
        4. panic;
    end
    5. use request to select the destination channel dst.chan;
    6. create a MessagePair structure including an ack channel;
    begin
        switch on selected channels do
            case dst.chan <- MessagePair do
                7. Wait for message over ack;
                8. state = AcknowledgeRead();
                if state < 0 then
                    9. panic;
                end
            end
            case <- timeout.chan do
                10. state = FailRead();
                if state < 0 then
                    11. panic;
                end
            end
        end
    end
end
```

Algorithm 8: Pseudocode for the NFComms Host Read Engine.

either a message has been received, the framework is shutting down, or an error has occurred. If the returned *state* is 0, then a shutdown is in effect and this function simply returns in step **(3)**, decrementing *NfpComms.wg* to indicate its removal. If the returned *state* is negative, then an error has occurred and a panic event is triggered.

Situations where the returned *state* is greater than 0 indicate that a message has been received which must be handled. When this event occurs, the read request populates *request* with the current message which must be transferred. This operation is represented by event ② in Figure 4.11.

The Host Read Engine uses *request.target_id* as an index into the *NfpComms connections* map in step **(5)**. If successful, the associated channel, *dst.chan*, acts as the destination port for the message. Should this lookup fail, the Host Read Engine generates a contextualised error which it logs as a fatal event, causing the system to exit. Though extreme, if the Host Read Engine receives a target ID that does not correlate to a channel in the *connections* map, the system is considered to be in an unstable state.

Before forwarding the message data to the indexed channel, the Host Read Engine first creates an additional channel, *ack*, over which an acknowledgement signal can be sent. A reference to *ack* is then packed with the message from *NfpComms* into a *MessagePair* structure in step **(6)**. The *MessagePair* is pushed to *dst.chan* for the destination process to consume. This event is represented as event ③ in Figure 4.11.

To account for situations where the destination process is not currently available, a timeout event is also initiated. The timeout event is intended to protect the framework from deadlock that could occur by having all Host Read Engines block, waiting for a message to be consumed. A `select` statement, detailed in Section 2.4 is used to allow the Host Read Engine to wait on both the consumption from *dst.chan* and the timeout. If the timeout resolves first, a *FailRead* is issued in step **(10)** to inform the framework that the destination process is not ready to consume a message. Section 4.3.4 details the handling of a *FailRead* event.

Should the message consumption event resolve first, the Host Read Engine waits for a message to be received over *ack* in step **(7)**. This event acts as an acknowledgement of the message by the destination process represented by event ④ in Figure 4.11. Details of how this transaction is handled are given in Section 4.3.5. When received, the Host Read Engine calls *AcknowledgeRead()* to relay the successful message transmission to the NFP driver.

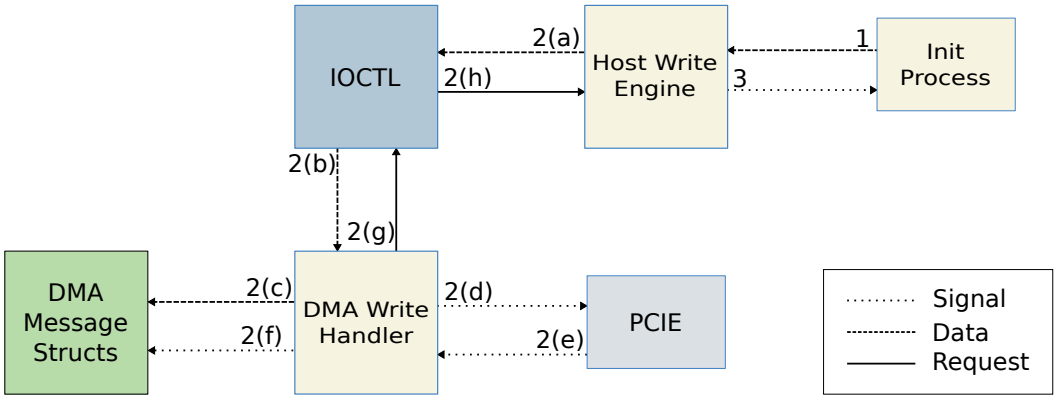


Figure 4.12: Events involved in message transmission from the host to RAM accessible by the NFP.

4.5 Host to NFP Transmission

The transmission of a message from an application executing on the host to a process executing on the NFP device begins when a host process initiates a communication event. To better describe the interactions involved in transferring a single message from the host process to a memory region accessible by a NFP Read Engine, an overview of the relevant actors is provided in Figure 4.12.

4.5.1 Initiating and Servicing the Message Transfer Event

From the context of the host application, the process of submitting a message to the NFP is succinctly achieved in event (1) of Figure 4.12 by calling the NFComms runtime function *Send()* to submit the message to the Host Write Engines. The details relating to this function are presented in Section 4.3.5 which describes the function as a blocking operation. When this call resolves, it indicates that the message has successfully been transmitted to the destination process on the NFP device and the initiating process can continue normal execution.

Transmission then becomes the responsibility of the Host Write Engine that consumed the transmission request. The Host Write Engine executes for the life of the runtime and begins by waiting for an event from one of two channels as depicted in Algorithm 9. As with the Host Read Engines, the Host Write Engines are implemented as a set of routines, spawned when *Start()* is called by the host application. The Host Write Engine is also associated with the current *NfpComms* object and has access to the associated elements thus it too only requires a buffer number on startup to index the appropriate DMA buffer discussed in Section 4.2.

```
while system is running do
    begin
        switch on selected channels do
            case <- NfpComms.shutdown do
                1. return;
            end
            case message <- NfpComms.writeChan do
                2. state = WriteToDMA(request, bufferID);
                if state == −1 then
                    3. Send a value over message.ack;
                    4. return;
                end
                if state ! = 0 then
                    5. panic;
                end
            end
        end
        6. Send a value over message.ack;
    end
end
```

Algorithm 9: Pseudocode for the NFComms Host Write Engine.

The first channel is *NfpComms.shutdown* while the second is *NfpComms.writeChan*. If a message is consumed from *NfpComms.shutdown*, this indicates that the NFComms framework is shutting down and this function simply returns in step **(1)** of Algorithm 9, decrementing *NfpComms.wg* to indicate its removal.

A message transfer event in the context of this function begins when the engine detects and consumes a *message* off *NfpComms.writeChan*, represented by event ① in Figure 4.12. The message received in this manner contains all the necessary information to identify the destination process as well as a channel over which to submit an acknowledgement once the message transfer event is complete. *WriteToDMA* is then called, supplying the data component of *message* and the buffer number supplied to the Host Write Engine during startup. *WriteToDMA* acts as a blocking operation and is responsible for event ③ in Figure 4.12 with the operational details discussed in Section 4.3.3.

When this function resolves in step **(2)**, the returned *state* value is checked to determine if a successful transmission occurred. If -1 then a shutdown event occurred between the reception of a message over *NfpComms.writeChan* and the calling of *WriteToDMA*. Before the shutdown is handled, a value is first sent over *message.ack* so that the initiating process does not remain in a blocked state. The shutdown is then handled in the same

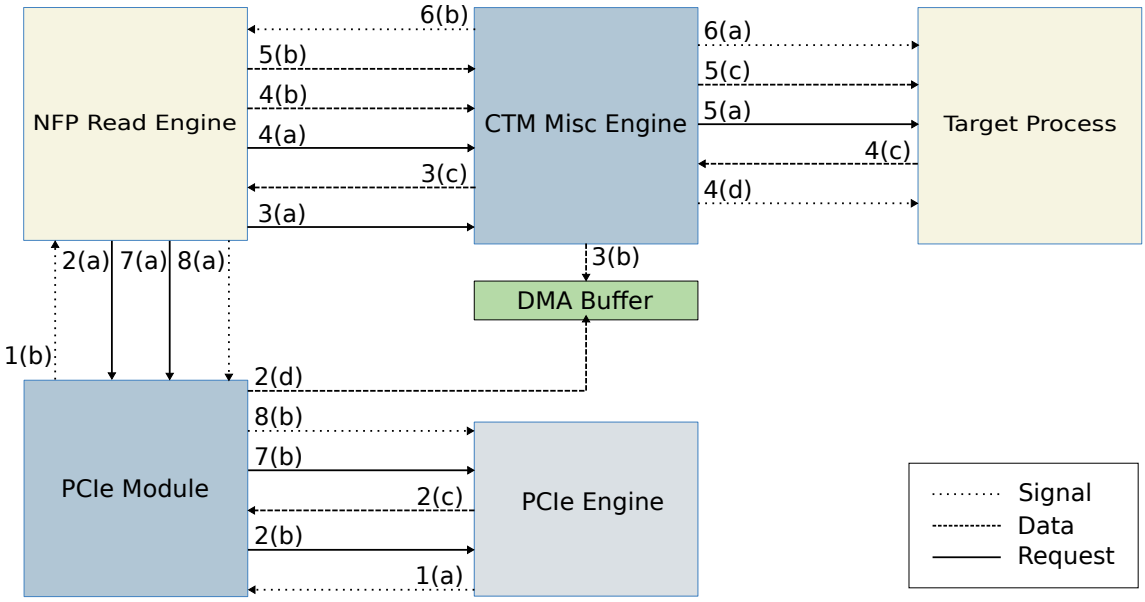


Figure 4.13: Overview of events involved in message reception from the PCIe bus.

manner as discussed for step **(1)**.

If *state* contains any value other than -1 or 0, then an error has occurred and the Host Write Engine generates a contextualised error which it logs as a fatal event, causing the system to exit. If 0 then the message transmission event has been completed successfully and a value is sent over *message.ack* to indicate that the Host Write Engine's involvement in the transaction is complete.

4.5.2 Reception and Forwarding the Message Transfer Event

Once the NFP driver has submitted the signal assertion request to the NFP device, the remainder of the message transfer event is handled within the NFP architecture. An overview of the event within the scope of the NFP device is depicted in Figure 4.13 with the sequential states being numbered to indicate the order in which they occur.

From the perspective of the NFP device, a write transaction to the card begins with the assertion of a signal by the host represented by event (1) of Figure 4.13. This assertion wakes up the corresponding NFP Read Engine, informing it that there is a message under transmission which it must service. The active NFP Read Engine begins processing the message according to the sequence of events shown in Algorithm 10.

The NFP Read Engine begins execution by first allocating a *nfp_dma_message* structure in CLS for the storage of messages requested through the DMA operations. This is followed

1. Create a **nfp_dma_message** struct in CLS on island 4;
2. Set up the **message_xfer_input** and **message_xfer_output** arrays;
3. Create a **message_sig_host** and **message_sig_client** ID;
4. Initialize DMA;

while *the NFComms runtime is active* **do**

   5. Sleep until **message_sig_host** is asserted;
   6. Call DMA read subroutine to read from the host to **nfp_dma_message**;
   7. Request that CLS read **dma_message.message** into **message_xfer_input** array;
   8. Copy **message_xfer_input** to **message_xfer_output**;
   9. Construct **message_reference** using **nfp_dma_message** and **message_sig_client**;
   10. Request that CT write **message_reference** to **dma_message.target_id**;
   11. Sleep until **message_sig_client** is asserted;
   12. Set **dma_message.status** to DMA_STATE_WRITE_ACK;
   13. Call DMA write subroutine to send **nfp_dma_message** to host;

end

Algorithm 10: Pseudocode for the NFP Read Engine.

by the reservation of *message_xfer_input* and *message_xfer_output* arrays in step **(2)** of Algorithm 10, and the signals *message_sig_host* and *message_sig_client* in step **(3)**. The signal *message_sig_host* is used to construct a *source_signal* which is required for the DMA initialisation routine in step **(4)**. Details relating to the DMA initialisation for the NFP Read Engine are presented in Section 4.2.5. Once complete, the NFP Read Engine sleeps on *message_sig_host* which is asserted by the host during a message transaction.

Once asserted, the active engine can safely assume that the message to be transmitted has been stored in its reserved message struct within RAM. The active engine can begin by issuing a DMA read request to the PCIe module which is represented by step **(6)** in Algorithm 10. As the size of the associated message is unknown, the read operation moves the entire *nfp_dma_message* from the host and stores it in the allotted memory address in CLS. Once stored, the active engine reads the message size from the struct and issues a read request to the CLS engine to move the message payload from CLS into the *message_xfer_input*, represented by step **(7)**. In order to make the message payload available for collection by the intended target process, the data must be moved from *message_xfer_input* to *message_xfer_output*. Once the transfer is complete, the active engine begins preparations to inform the target process of the event.

The signal *message_sig_client* as well as *dma_message.target_id* and *message_xfer_output* are used to construct a *message_reference* in step **(9)**, which contains all the information required to allow the destination process to identify the current NFP Read Engine as well as where the message payload is located within its write registers. This message also

includes which signal to assert once the transfer has been completed.

Once complete, the active engine can request a reflect write request in step **(4)** to move *message_reference* to the destination process. The target address and signal used to initiate the reflect operation are extracted from *dma_message.target_id*. It is the responsibility of the host to correctly populate this field before signalling the NFP Read Engine to continue processing the event.

At this point the active NFP Read Engine must again sleep but this time on the signal which it has relayed to the target process as part of the reflect operation. When this signal gets asserted during step **(11)**, the current Read Engine can assume that the message has been accepted by the target process.

An acknowledgement of this can be relayed back to the initiating process using step **(12)** and step **(13)** which represent event ⑦ in Figure 10. This is achieved by setting *dma_message.state* to `DMA_STATE_READ_ACK` and submitting the *nfp_dma_message* back to the host. As the Host Write Engine involved in the current message transfer event is expecting a response, it will be monitoring the relevant DMA message struct for this update.

4.5.3 Receipt and Acknowledgement of Message

if *Message read event requested* **then**
1. Set up the **message_xfer_input** array;
2. Sleep on **msg_sig** until woken;
3. Read message from *msg_reg*;
4. Request that CT read message into **message_xfer_input**;
5. Request that CT assert signal contained in *msg_reg*;

end

Algorithm 11: Pseudocode for the NFP Read Engine.

After the NFP Read Engine servicing the current event performs a reflect write operation indicated as step **(10)** in Algorithm 10, the next stage of the message transfer event is best viewed from the perspective of the target process itself.

Initially, the target process could be in an execution state where it is not ready to receive a message from the initiating process and as a result, all other actors involved in this event must wait until the target process reaches the appropriate execution stage. In this situation, the signal specified by the host on the target process is asserted so that the message transfer event can continue to resolve when the target process is ready.

Once the appropriate execution stage has been reached, the target process begins by allocating *message_xfer_input* into which the message can be stored. The target process then attempts to sleep on the reserved signal, *msg_sig*, as indicated by step **(2)** in Algorithm 11. If the signal has already been asserted, the target process can request a reflect read operation using the signal address provided by the active NFP Read Engine to fetch the message. This step is represented by event ④ in Figure 4.13. Once the reflect operation completes, the message has successfully been delivered to its intended destination and an acknowledgement signal can be relayed to the initiating process in step **(5)** of Algorithm 11. To begin the message relay procedure, the target process again uses the signal address provided by the Read Engine but this time requests an interthread signal operation to inform the engine of the completed transfer. From this point, the target process can continue normal operation since, from its perspective, the message transfer event has been resolved. This acknowledgement signal is propagated back to the NFComms runtime, finally being received by the blocked *Send()* call. This call can then resolve, unblocking the initiating process and completing the transfer operation.

4.6 Preliminary Testing

To determine both the correctness and functionality of the NFComms implementation, a preliminary set of tests were undertaken. These tests were divided into two categories: correctness testing and performance testing. The goal of the correctness testing was to evaluate whether the NFComms framework presented in this chapter operated as expected. This testing was performed by implementing two example applications utilising both processing elements deployed to both the CPU and the NFP. These elements would then interact using the prototype NFComms framework. Following the correctness tests, the general performance capabilities of the NFComms framework were evaluated. These tests were intended to determine the message passing latencies and throughput achievable by the implemented prototype.

4.6.1 Application Testing

Given that the intended target domain of the NFP device is network processing, the range of features supported by the architecture can be viewed as restrictive when compared to a modern CPU. Further details relating to these restrictions are given in Section 2.1.1. As noted in the initial design goals presented in Section 1.2 however, the proposed framework was intended to be used for generic processing. With this in mind, the applications selected were not related to network processing but could still be considered embarrassingly parallel

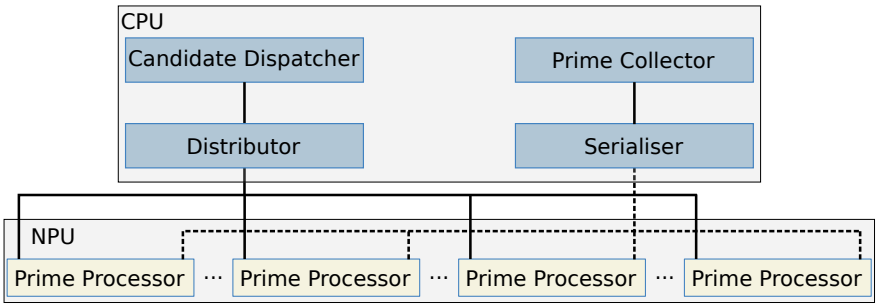


Figure 4.14: Process layout for prime number application.

and communication intensive to enable testing of the NFComms framework.

Prime Number Generation

As a first application, a naive prime number generator was implemented. The goal of this application was to have processes operating on the NFP device determine whether a given candidate number was a prime number using the simple algorithm given in Listing 4.3. The application component running on the host would be responsible for handling the distribution of candidate numbers to the NFP processors and collection of the results. The distribution and collection was achieved through synchronous message passing using the NFComms framework. The resulting application layout is illustrated in Figure 4.14 which shows the application components running on each architecture. Additional serialiser and distributor processes were used to manage the serialisation of messages between the NFComms runtime and the evaluation process. By including these intermediary processes, the application could easily scale the number of processes running on the NFP device with minimal changes. This application example was presented as part of a previous publication (Pennefather *et al.*, 2018a).

Table 4.3: Execution times for naive prime search.

Candidate Count	Processes	Time [s]
40000	1	103.065
40000	8	79.953
40000	16	39.973
40000	32	20.028
40000	64	10.049
40000	128	5.074
40000	256	2.616
40000	480	1.467

```
1  int main() {
2    __xread  volatile unsigned int data_input[2];
3    __xwrite  volatile unsigned int data_output[1];
4    __declspec(cls4)  volatile unsigned int candidate;
5    volatile unsigned int i, response;
6    while(1) {
7        channel_read((void*)data_input, 2, &reg, &sig);
8        candidate = data_input[0];
9        response  = data_input[1];
10       for(i = 2; i < candidate; i++) {
11           if ((candidate % i) == 0) {
12             candidate = 0;
13             break;
14           }
15       }
16       data_output[0] = candidate;
17       channel_write((void*)data_output, 1, response);
18   }
19 }
```

Listing 4.3: Prime check application.

To test the implemented application, a list of 40000 sequential numbers from 2 to 40002 were used as candidates. The testing involved 20 runs of the application initially using a fixed number of NFP processes with the resulting completion times being the average across all runs. This test was repeated for varying numbers of NFP processes and the results are recorded in Table 4.3. As a standard across all tests, compiler optimisations were disabled for both the CPU and NFP components of the application.

The results presented in Table 4.3 present two noteworthy observations. The first is that doubling the number of processes being allocated to the application approximately

Table 4.4: Execution times for naive prime search on a single microengine.

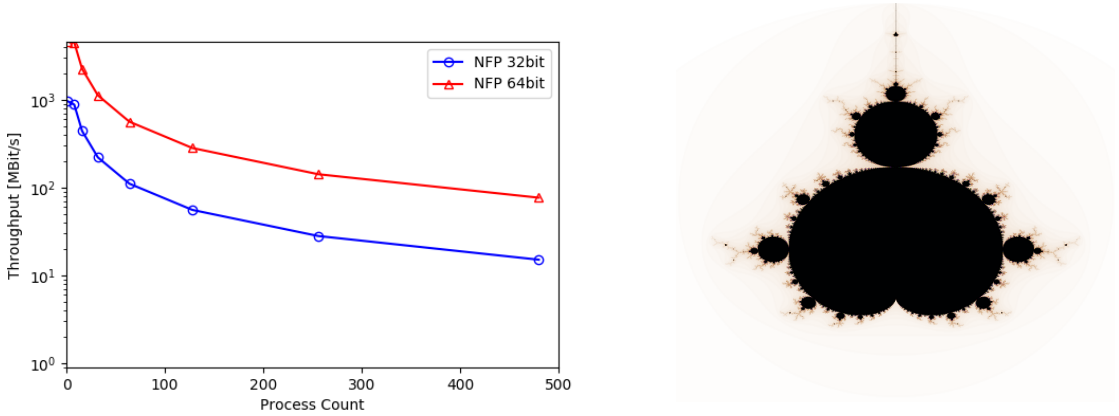
Candidate Count	Processes	Time [s]
40000	1	103.065
40000	2	85.481
40000	3	81.915
40000	4	80.810
40000	5	80.397
40000	6	80.620
40000	7	80.101
40000	8	79.945

halves the processing time required to complete the test. This result was expected as each microengine operates in parallel and so tasks running on separate micorengines can progress simultaneously. Considering a single micorengine however, the results are significantly different.

The above tests were repeated but targeting only the contexts on a single microengine with the results recorded in Table 4.4. As noted, aside from a reduction in processing time when moving from one process to two processes, increasing the process count provides a negligible improvement in time to complete. This plateauing of execution times shows that running multiple instances of the prime searching algorithm on the same micorengine does not improve application performance greatly. The noted improvement when transitioning from one process to two can mostly be attributed to the hiding of communication latencies associated with sending and receiving messages.

The second observation is that the overall performance of the naive prime application example is significantly worse than expected, with the equivalent multi-threaded solution completing in 78.62 ms on a Intel Haswell Core i5 4670k processor. Further investigations revealed that the modulo operator, which is heavily used by this application, is a relatively expensive operation on the NFP due to the absence of the division operation (Netronome, 2008). It is suspected that this expensive operation contributes significantly to the relatively poor performance.

To better evaluate how expensive a modulo operation is on the NFP, the original testing was repeated with the candidate list replaced by 10000 instances of the prime number 139397 so that the loop present in Listing 4.3 would always need to run to completion. This test was repeated a third time with the modulo operator in Listing 4.3 replaced with a subtract operator so that each process would still need to run through the entire loop in each iteration but not perform the modulus operation. This testing would of course



(a) Execution times

(b) Sample image

Figure 4.15: Results for 32- and 64-bit implementations of fractal generation on the NFP.

not yield any prime numbers but would still require the application to run to completion as if the candidate numbers were being evaluated correctly. The results of both tests indicated that application runs without the modulo operation were 28 to 35× faster than those utilising the modulo operator.

Fractal Generation

As a second application, the concept of using the NFP to assist in the generation of fractal shapes pertaining to the Mandelbrot set (Mandelbrot, 2004) was explored. The goal of this application was to have processes operating on the NFP device calculate the depth for a given complex number which would correlate to the colour of a pixel in the fractal image. In this application, the host was responsible for the generation of complex numbers and rendering an image from the depth values returned by the NFP processes. This application example was also presented as part of a previous publication (Pennefather *et al.*, 2018a).

As the NFP architecture does not support floating point or division operations, the approach taken was to implement this application using fixed point arithmetic (Barina, 2014) and shift operations. The layout of the implemented application follows the same design as depicted in Figure 4.14 with the Candidate Distributor and Prime Collector replaced with Complex Distributor and Depth Collector respectively. Three variants of the application were designed: an NFP 32-bit implementation, NFP 64-bit implementation, and a host-only 32-bit implementation. In both NFP implementations, the applications were technically heterogeneous as the submission and collection of pixel data for rendering was handled by the host, however no pixel depth processing

was performed by the host in these applications.

To gauge the performance of the NFP applications, both applications were tested to evaluate each pixel in a 300x400 image to a maximum depth of 0xFFFFFF. These tests were then repeated for varying process counts with the results recorded in Figure 4.15(a). A sample image rendered from the 32-bit implementation is displayed in Figure 4.15(b).

From the recorded results it is clear that the 64-bit implementation of the application takes significantly longer to complete than the 32-bit implementation, taking on average 5× longer to resolve. Evaluating the generated assembly code, this is due to each 64-bit operation being broken down into multiple 32-bit operations as general registers within the microengine are only 32 bits wide.

To compare the performance capabilities of the NFP and the CPU, the host-only 32-bit implementation of the application was tested. As with the other host-based tests, this application was run on a Intel Haswell Core i5 4670k processor with the application designed as a multi-processor implementation to take advantage of the CPU logical processors. Compiler optimisations were similarly disabled to make the comparison fairer. Over 20 iterations the host-only implementation completed in 17.60 s, 16.7% slower than the 32-bit NFP implementation when running 480 processes.

As a final test, a heterogeneous variant of the 32-bit application was implemented. In this variant both the CPU and the NPU were used to perform pixel depth processing. Setting the maximum number of concurrent processes allowed on both candidate architectures to 480 and again setting the maximum iteration depth to 0xFFFFFF, the application was tested and the average of 20 iterations was recorded. The resulting average was 10.66 s, a 65% increase on the host-only implementation and a 41% increase on the NFP-only implementation. This result does not infer that utilising an NFP-CPU heterogeneous system for fractal generation results in a significant speedup on a host-only implementation. In fact, given the inclusion of an entire additional processing unit, these results could be considered very underwhelming. However, given that the current framework is largely still in the exploration stage with poor optimisations, this result does imply that the NFP could act as a viable coprocessor to form a CPU-NFP heterogeneous system capable of accelerating existing applications.

4.6.2 Performance Testing

Though the application testing was able to show that the NFComms implementation functioned correctly, it also highlighted concerns relating to performance. To further

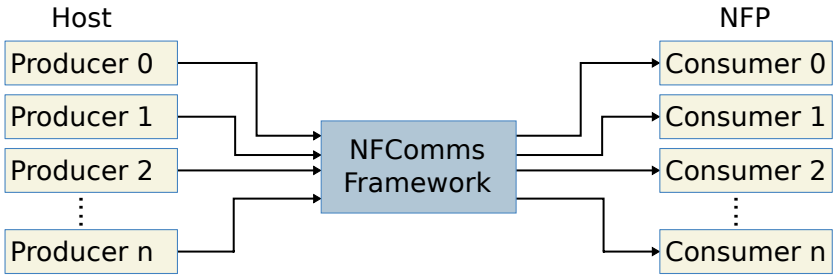


Figure 4.16: Overview of latency test setup.

investigate these issues, two aspects of the framework were tested, namely, latency and throughput.

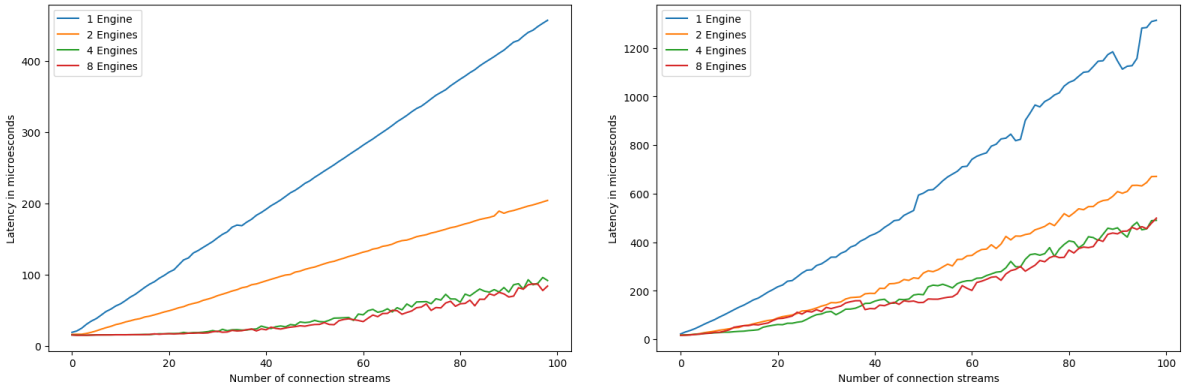
Latency Testing

To test latency, a simple producer-consumer test environment was designed as presented in Figure 4.16. The environment consists of a host component written in Go and an NFP component. For the first iteration of tests, the host component consists of only producer elements while the consumers reside on the NFP. This configuration allows for the latencies associated with message transmission from the host to the NFP to be recorded in a situation where multiple concurrent communication streams are established.

The goal of this initial test was to determine the impact increasing the number of concurrent streams can have on the communication latency between host and NFP. One important attribute of the environment is however noted: the implemented framework consists of eight read engines and eight write engines allowing for the handling of up to eight unidirectional streams in either direction concurrently. Additional streams can be handled but are delayed until a relevant engine is available to handle the transmission.

Using the testing environment, 1000 synchronous messages were sent from a single host producer to a single NFP-based consumer and the average latency was recorded. This test was then repeated for an increasing number of producer-consumer pairs with the results illustrated in Figure 4.17(a). To determine the impact that multiple engines could have on message latency, the number of NFP Read Engines was reduced to one. The above test suite was rerun with the results also recorded in Figure 4.17(a). These tests were again repeated for configurations consisting of two and four engines, the results of which are also shown in Figure 4.17(a).

To investigate the latencies associated with message transmission originating from the NFP, the roles of the architectures depicted in Figure 4.16 were reversed. Now the pro-



(a) Host to NFP

(b) NFP to host

Figure 4.17: Framework latency tests.

ducer elements were implemented on the NFP while the consumer processes resided on the host. To initiate a transfer test, the host was required to send the producer an initiating message, detailing the target process ID and the number of messages to send. Timing of the transfer only occurred after this message was acknowledged by the NFP. To make the results of this test comparable with those of the transfer event from the host to the NFP, the same testing procedure was followed. Each producer would send 1000 synchronous messages to its respective consumer and the average latency was recorded. This test was then repeated with an increasing number of producer-consumer pairs. The results of these tests are recorded in Figure 4.17(b). To see the impact the number of engines could have on these unidirectional communication streams, the above tests were repeated for additional configurations supporting one, two, and four engines, the results of which are also presented on Figure 4.17(b).

Firstly, considering the results presented in Figure 4.17(a), it is immediately apparent that increasing the number of concurrently communicating processes increases the average observed latency per message. In the situation where only one NFP Read Engine was available for processing, the latency rapidly increases at a rate of approximately 4.5 μs per additional communicating pair, from 19 μs to 456.72 μs. Adding an additional NFP Read Engine resulted in a notable improvement in performance with the message latencies still growing at a linear rate, but now at approximately 2 μs per additional communicating pair, a marked improvement on the single engine configuration.

An improvement is again seen when two additional NFP Read Engines are added as this allows the framework to handle four messages concurrently. This configuration however no longer shows a linear increase as the number of communicating elements increases. Initially, the rate of change is comparatively small, with message latencies increasing from

15.5 μs with a single communicating pair to 20.3 μs when 30 message passing units are present. From this point however, the rate at which message latencies increase begins to trend upwards while also showing an increase in variance for the average latency between successive connection streams. Although distinctly more sporadic when compared to configurations using one or two NFP Read Engines, the overall latency increase is less, reaching a maximum of 96.2 μs.

Next, the configuration using all eight NFP Read Engines was considered. It is immediately apparent that the latency improvement seen when increasing the number of NFP Read Engines is greatly diminished after four NFP Read Engines have been introduced. Furthermore, the eight NFP Read Engine configuration exhibits the same characteristics as the four NFP Read Engine one, with the recorded average latency increasing from 15.6 μs with a single communicating pair to 18.6 μs when 30 message passing units are present. The maximum recorded latency in this configuration is 87.1 μs, a relatively small decrease compared to the four NFP Read Engine configuration. It is suspected that both the increasingly sporadic rate at which the latencies of the four and eight NFP Read Engine configurations increase as well as the relatively small improvement in overall latency implies that the number of NFP Read Engines used beyond four is being bottlenecked by some artefact of the framework.

All NFP Read Engines are deployed on a single microengine to minimise the resource footprint of the framework. Delays while waiting for memory and communication events to resolve externally provide an opportunity for the sharing of the microengine computing resources between contexts not currently blocked on such events. However, it is suspected that operating more than four NFP Read Engines results in contexts waiting for access to the ALU outside of such events, resulting in a performance reduction per context.

For communication events recorded in Figure 4.17(b), a similar pattern is seen when compared to the message passing tests originating from the host. Messages originating from the card however do exhibit a much greater latency, particularly once a large number of communication elements are present in the environment. In the situation where only one Host Read Engine is available for processing, the maximum observed latency is 1314.3 μs, 2.88$\times$ greater than the average host-based transaction in a similar environment. The rate at which latencies increase for NFP-based communication events is superlinear for all counts of Host Read Engine, with increased variance seen in the recorded latencies as the number of introduced communicating elements increases. As with host-based transaction testing, increasing the number of concurrent communication streams through the framework improves latency, however this benefit degrades more rapidly. Considering

the test environment where 100 communication pairs are present, each transmitting 1000 synchronous messages, the average latency recorded when two Host Read Engines are used is 671 μs, 1.96× smaller than the equivalent test using a single Host Read Engine.

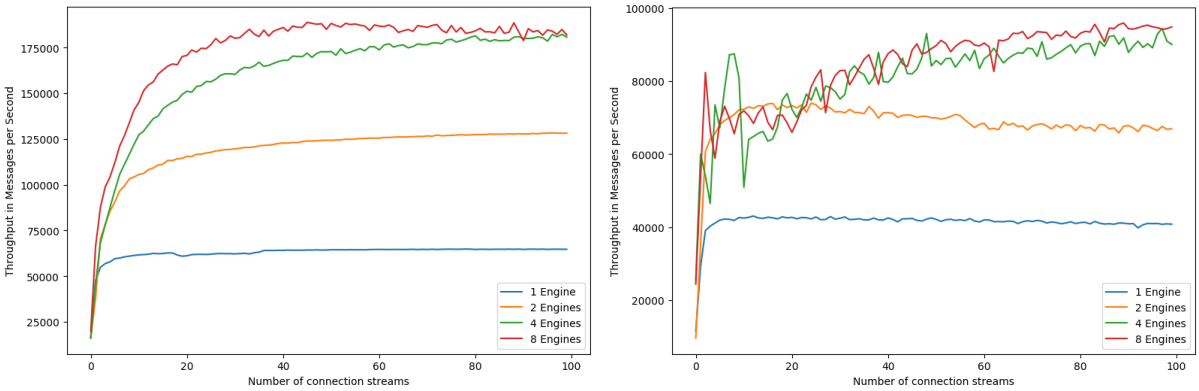
Considering the same test but with four engines available, the average recorded latency is 490.5 μs resulting in a latency improvement of only 1.37× while the equivalent eight engine configuration actually yielded a higher latency at 499.2 μs. Due to the high level of variance noted in the average latency of successive tests for both the four and eight engine configuration, it is not possible to draw a meaningful conclusion as to which configuration yields the best latency overall.

Throughput Testing

To test the throughput achievable by the NFComms framework, a similar approach to the latency testing was undertaken. This implementation again used the testing environment described in Figure 4.16 with an increasing number of independent producer-consumer pairs being presented to the test on each iteration. For these tests however, instead of recording the time taken for a single message transaction to resolve, the time taken for a set number of transactions was recorded. This was resolved to the number of messages transmitted per second through the framework for different configurations of producer-consumer pairs.

Initially, message throughput from the host to the NFP was tested with the results presented in Figure 4.18(a). For comparative purposes, this test was repeated while varying the number of message passing engines active in the NFComms framework. This entire process was then repeated with the producer elements residing on the NFP and consumers on the host. The results of this final batch of testing are presented in Figure 4.18(b).

The goal of the throughput tests was to determine the maximum number of messages that the NFComms framework can handle per second. By repeating the tests for an varying number of NFP Read Engines or Host Read Engines, the impact of multiple engines can be observed. The results in Figure 4.18(a) show a rapid increase in message throughput as the number of message passing elements increases. Once the supplied engines become saturated, the number of messages that the framework can handle plateaus, depicting the peak message throughput achievable. As expected, increasing the number of NFP Read Engines increases the maximum throughput; however, as observed with the latency testing, this has a deteriorating effect with the four and eight engine configurations converging to the same peak throughput at just over 18000 messages per second. The primary concern however, is the maximun data throughput the framework can achieve. Given that



(a) Host to NFP

(b) NFP to host

Figure 4.18: Framework throughput tests.

each message transaction is limited to a maximum payload of 56 bytes, the average data throughput achieved during testing of eight NFP Read Engines in an environment consisting of 100 producer-consumer pairs was 77.8 MBit/s. Furthermore, the peak throughput for testing described in this section was 102.4 Mbit/s.

The concerns relating to throughput are further highlighted when reviewing the results presented in Figure 4.18(b). As with the NFP-based latency tests, these results exhibit a high level of variance between successive tests, especially for the four and eight engine configurations. Comparing the results depicted in Figures 4.18(a) and 4.18(b) the most notable difference is the throughput achieved. For NFP-based communication, the peak average message throughput recorded was 95,891 messages per second with the peak average data throughput being 41 MBit/s, almost half that of the host-based message transactions.

4.7 Summary

This chapter presented the design and implementation for the first phase of the NFComms framework. At the start of the design process, concerns relating to naming and symmetry in the context of message passing were investigated in Section 4.1. This section highlighted how these communication semantics differ between the candidate frameworks and proposed a solution for resolving this disparity through the use of engines. This solution was elaborated on to produce an overview of how synchronous message passing would be realised within the message passing framework in Section 4.1.5 as well as the formalisation of a message structure understood by all engines.

A detailed discussion on communication between the kernel and the NFP architecture has

been provided in Section 4.2 where it was concluded that the proposed communication framework would be implemented by extending the existing driver. To achieve this, absent mechanisms essential for synchronous communication were identified.

Section 4.3 provided a similar discussion on facilitating communication between the kernel and user-space applications. This discussion considered different approaches for creating an interface between the kernel and user-space, concluding that IOCTL commands would provide the best solution. This section concluded by introducing the concept of the NFComms runtime, presenting the runtime API functions that have been exposed for user-space applications to interact with.

Sections 4.4 and 4.5, described the bulk of the chapter, detailing the interactions required to facilitate a message transaction originating on each of the architectures. Section 4.4 discussed the functionality of the NFP Write Engine as well as the Host Read Engine, including the actions necessary for an NFP-based process to submit a message and the host application to consume one. Section 4.5 described how a host application could submit a message to the NFComms runtime and the steps taken by the Host Write Engine to facilitate it. A description of the involvement of the NFP Read Engine followed as well as how the submission of a received message to the destination process is handled.

Finally, Section 4.6 presented the initial testing of the prototype framework. Testing was split into two focuses: application testing and performance testing. For the application testing component, two example applications were presented. The first was a prime number generator designed solely to test the functionality of the framework. This application operated by having multiple processes executing on the NFP to check candidate numbers and determine if they were prime. The host component of the application was responsible for distributing these candidate numbers to the NFP and collecting the results. The second application was intended to be more functional while still testing the capabilities of the NFComms framework. This application used the processing capabilities of the NFP to generate a fractal image by allocating each pixel of the fractal image to a NFP context for processing.

For performance testing, both the latency and the maximum average throughput of the NFComms framework were investigated. In both configurations the same testing environment was used which would progressively increase the number of communicating producer-consumer pairs that would need to be facilitated. The latency tests recorded a minimum latency of 15.5 μs when submitting messages over the framework originating from the host and 18 μs for message originating from the NFP. These latencies increase for successive tests as the increased number of communicating pairs saturate the framework,

causing transaction components to be buffered.

During application testing it was observed that increasing the number of communication engines beyond four did not introduce a significant improvement in message throughput or latency. It was suspected that this is due to all eight unidirectional NFP-based engines being executed on a single microengine. Another observation was the poor data throughput achieved by the framework, achieving a peak throughput of 102.4 Mbit/s. Although it is acknowledged that the NFComms framework is a prototype, the observed performance is not considered suitable for practical applications.

These performance limitations are addressed in Chapter 5 where the source of the throughput limitations is identified. These bottlenecks are resolved through the introduction of a set of extensions that enable transfer operations supporting an improved throughput performance. Chapter 5 details the design of these extensions.

5

Framework Extensions

Chapter 4 presented the design and implementation of the NFComms framework, a functional communication framework for the synchronous transmission of messages between the host and NPU architecture at runtime. The identification and resolution of concerns relating to symmetry and naming were initially discussed after which the NFComms framework was presented by detailing its implementation. Once implemented, the framework was used in two basic applications to test functionality followed by latency and throughput testing.

Though the initial implementation of the NFComms framework was shown to be functional, an analysis of the maximum achievable throughput recorded a peak transmission rate of 102.4 Mb/s which was considered inadequate for any bulk transfer operations. This poor performance was chiefly due to the very restrictive 56-byte message size enforced by the 14-register transfer limit discussed in Section 2.1.1. Considering this, an investigation into implementing an alternative interface that could either operate in conjunction with, or utilise the existing framework to advertise better throughput capabilities was carried out. This chapter explores the feasibility of introducing a set of extensions to the NFComms framework to support bulk data transfers. The goal of these extensions is to provide a communication interface that conforms to the same requirements as the existing

model but provides a greater data throughput and supports larger message sizes. The work presented in this chapter has largely been published in (Pennefather *et al.*, 2018b) with an invited extension being submitted as (Pennefather *et al.*, 2019a).

This chapter begins in Section 5.1 with an identification of artefacts in the existing framework that contribute to the poor performance observed. Architectural characteristics such as limited memory sizes and register limits are discussed, highlighting their impact on the overall throughput. This identification is followed by the proposed solution to the NFComms framework to mitigate the poor throughput. The section concludes with a brief discussion on zero-copy operations, an important mechanism for minimising the number of data copy events required to realise the proposed solution.

Following the problem identification, the remainder of the chapter focuses on presenting the changes made to the existing NFComms framework to accommodate the proposed solution. This presentation is done by dividing the framework into different regions and discussing the modifications to each region separately. Section 5.2 begins by discussing the modifications to the NFComms driver handler functions to facilitate the bulk transfer operations. This section also includes a discussion on extending both the existing DMA message structure and the associated list of status codes. Section 5.3 focuses on the changes made to the NFP Read and NFP Write engines while Section 5.4 describes the required extensions to both the NFComms runtime and the NFP communication library.

Testing of the revised framework is presented in Section 5.5 with an initial focus on latency and throughput testing. Following the initial tests, the application example discussed in Section 4.6.1 is revisited and altered to operate using the bulk transfer operations providing a qualitative comparison between the architecture revisions. A discussion on the findings from these initial tests is presented in Section 5.6 along with a summary of the chapter.

5.1 Problem Identification

In the initial prototype, all communication between the two architectures occurs through synchronous message transmission where each message is passed using a single reflect operation. Section 4.1.5 describes the message format used by the prototype as well as the reasoning for the selected sizes. In its current configuration, all messages are restricted to 56 bytes (excluding metadata) to allow them to be passed in a single reflect operation.

As discussed in Section 4.2.4 however, the PCIe operates by passing TLPs that can support

a payload of up to 4096 bytes (Lawley, 2014). As the individual messages are not gathered for a single transaction so as to minimise latency, the current implementation is only able to use approximately 1.4% of the available capacity for a single message. This bandwidth reduction is further compounded by the synchronous nature of these messages, requiring the backward propagation of an acknowledgement signal to indicate completion for each transaction.

A potential solution to this issue is to simply remove the message size limitation so that NFP intra-communication can occur by breaking down the larger message and transferring it as a sequence of operations. This approach was briefly discussed in Section 4.1.5 where it was concluded that the benefits of such an approach do not outweigh the incurred costs.

Considering the NFP architecture as a whole, all transactions to and from contexts within microengines utilise these transfer registers to resolve. As the NFComms framework is expected to work in conjugation with other applications deployed on the NFP, the number of registers reserved by the framework should be kept minimal. Given that the maximum number of bytes that can be moved in a single reflect operation requires up to 14[1] registers, it can be argued that supporting multiple reflect transactions in a single message pass event would simply not justify the overhead it could incur. The framework would either need to reserve the majority of the transfer registers, or need the relevant context to be involved actively in the transfer operation, thereby breaking atomicity.

Another aspect to consider is the limited memory within a single microengine. As discussed in Section 2.1.1 each microengine only supports 4 KB of local memory which is shared between all eight contexts. If equal partitioning of local memory between the eight contexts is assumed, each context has 512 bytes of local memory which would require only 10 reflect operations to transfer. This leads to two concerns. Firstly, the intent of introducing communication support for bulk transactions is to allow for a higher throughput when moving relatively large volumes of data. With the local capacity of a single context being 512 bytes (assuming equal partitioning), there is simply not enough local capacity to support a large data transactions. The second concern is that the NFP Read and NFP Write engines would need to buffer these requests while still reserving enough capacity for correct operation. These limitations could be mitigated by storing the memory outside of the NFP Read and NFP Write engines and having it moved into the engine as part of the transfer sequence, however this is expected to result in a limited improvement to the transfer throughput.

[1] 16 registers on the newer NFP-6 architecture

It is acknowledged that an application utilising the NFComms framework could include events that may require a large body of data to be transferred between the NFP and the host in a timely fashion. Such situations could include the uploading of rule tables for an intrusion detection or prevention system, uploading of unprocessed work to be computed on the NFP, or the transfer of collected records for processing on the host. Thus, being able to transmit a large body of data to or from the NFP would be beneficial although such data cannot be stored in the local memory of a single microengine due to limited capacity.

5.1.1 Proposed Solution

Section 3.4 concluded the initial investigations by confirming that a synchronous message passing system between the host and NFP architectures would be feasible. This became the basis for the implementation presented in Chapter 4. Although the goal of the research presented in this chapter is to resolve the concerns relating to throughput and message capacity, it is important to ensure that such extensions follow the methodology on which the initial implementation was based. As an initial goal of the message passing framework required that all communication would occur through synchronous events, these bulk memory transactions should adhere to the same restriction.

Another characteristic of the bulk transfer operations to consider is capacity. As stated in Section 5.1, the capacities of individual microengines make them poor candidates for bulk data transactions and as such, the viability of selecting a memory region outside of a microengine should be explored. As presented in Table 2.1, the NFP architecture supports four separate types of memory outside the scope of a single microengine. Each memory type supports a different capacity that is inversely proportional to the latency that a single microengine experiences when attempting to interface with it. For the proposed extensions, the three memory locations of particular interest are CTM, IMEM, and EMEM (see Table 2.1) as they have capacities large enough to justify bulk data transactions.

For the sake of clarity, the read and write components of the initial communication, as presented in Chapter 4, henceforth referred to as ‘direct read’ and ‘direct write’ operations. These operations are labelled as such because the NFP-based memory that the message is either sourced from or destined for, resides within the microengine involved in the transaction. Thus, the message and associated data is being transmitted directly to or from the relevant microengine. Message transactions relating to the extensions proposed in this chapter are referred to as ‘indirect read’ and ‘indirect write’ operations owing to the message data being indirectly transferred to the relevant microengine by having it

Table 5.1: CPP bus targets addressable by the NFP PCIe module.

CPP bus Target ID	CPP bus Target Name
6	ILA
7	CTM, IMEM, EMEM
9	PCIe
10	ARM
12	Crypto
15	CLS

sourced from or stored in an accessible memory region on the NFP but not within the relevant microengine.

Section 4.2.5 discussed the basic layout and configuration of the PCIe module within the NFP. For direct write operations originating from the NFP, the message data is first collected from the initiating context by the relevant NFP Write engine via a reflect read operation. This data is then written to the DMA message structure reserved in CLS and the PCIe module is requested to handle the DMA operation. The component of the write process that is particularly relevant to this discussion is how the PCIe module is informed of the location where the DMA message structure resides. For direct message passing, this location is always the CLS memory region of island 4; however, by changing the CPP bus target field of the DMA configuration register, the type of memory from which the PCIe engine pulls data in or pushes data out can be changed. Table 5.1 lists the addressable targets for the NFP architecture.

As noted in Table 5.1, all three memory regions with which the indirect communication extensions intend to interface (namely, CTM, IMEM, and EMEM), happen to share the same CPP bus target ID. This is significant for two reasons. Firstly, this implies that the PCIe module can address the NFP source and destination memory regions directly and be responsible for the actual data transfer. Secondly, as all memory regions share the same CPP bus target ID, a single DMA configuration register can be used to represent all indirect operations. The address of the relevant memory location can be supplied as part of the DMA descriptor, informing the DMA engine where either the source or destination memory region is located.

To enforce synchronisation as well as to acquire the correct memory address for a transaction, the underlying aim of the extension is to have the transaction involve an NFP process rather than statically assigning address locations on the NFP device. Although this adds an additional step to the transaction, it makes the NFP process responsible for supplying the address associated with the intended source or destination of a trans-

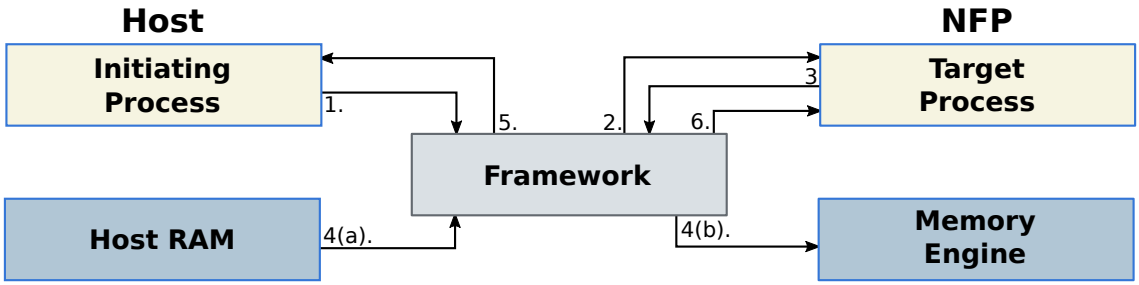


Figure 5.1: Overview of the events involved in an indirect message transmission event.

fer event. Synchronisation can then be enforced between the host and NFP process by utilising the existing direct message passing model to transmit the metadata.

Conceptually, an indirect write event from the host to the NFP follows the abstracted sequence of events presented in Figure 5.1. This transaction begins with a message being submitted to the framework in event ①. The framework synchronises with the target process in event ② and requests an address where the data associated with the message should be stored. At the point where the target process submits the destination address to the framework in event ③, both the initiating and target processes are blocked and must wait for the memory transfer presented by event ④ to complete. The framework then signals the involved processes in events ⑤ and ⑥, indicating that the transmission has been completed.

For a transaction operating in the opposite direction, the events occur in a similar fashion to those described in Figure 5.1 except that the originating process resides on the NFP device. For these transactions however, there is an additional message passing event that moves the destination address for the message from the host to the NFP. This is required because in both instances, the actor responsible for moving the actual data is the DMA engine on the NFP device. For transactions originating from the host, this address is submitted as part of the original message and thus no additional transaction is necessary. The destination address requested in event ③ does not require a transaction involving a DMA operation as the requesting component of the framework also resides on the NFP device. It is expected that the additional operation needed to initiate an indirect message passing event could impact the time taken to resolve the transaction, but should become relatively insignificant when transmitting suitably large messages.

5.1.2 Zero-Copy Operations

Before continuing with a presentation of the bulk transfer extensions, it is important to first introduce the concept of a zero-copy operation. As the goal of these indirect

transactions is to improve throughput, it is obviously necessary to minimise the time taken to physically move data between the source and destination actors.

For direct communication, the message passing framework comprises three distinct regions: user-space, kernel-space, and the NFP device. Moving data from user-space to the NFP architecture via kernel-space currently involves copying the message data to an intermediate kernel buffer. This buffer is allocated as a continuous region in physical memory (Corbet *et al.*, 2009) with the associated physical pages registered with the IOMMU (if present) for the NFP device (Intel, 2017). The DMA engine of the NFP device can then read the data associated with the host memory region via the IOMMU, thus moving the message payload to a memory region onboard the NFP architecture for further processing. Submitting a message originating on the NFP architecture to the user-space component of the framework uses an equivalent copy operation from a kernel-space buffer to a user-space buffer.

For small messages, the overhead of this intermediate copy operation is relatively small when compared to the actual DMA transaction. When scaling to much larger messages such as that which indirect message passing is expected to support, this intermediate copy operation may quickly become a limiting factor in throughput. To quantify this limitation, a preliminary implementation of the indirect message passing extension was designed to dynamically allocate a block of kernel-space memory to act as an intermediary into which the current message under transmission could be stored. Attempting to transfer 10 MB of data from the host to the card resulted in an average sustained throughput of 5 Gbit/s. Although this observed throughput is significantly better than the recorded throughput of 102.4 Mbit/s associated with direct communication, it is anticipated that the indirect transfer events could achieve even greater transfer speeds.

To improve on this observed performance, it was proposed that the bulk memory transfer operations be reimplemented using zero-copy operations. The term zero-copy is often used to describe a collection of common techniques for minimising the number of memory accesses involved in a potentially memory intensive operation. As the name implies, these techniques focus on avoiding the copying of memory until it is actually necessary (Brose, 2008).

In the context of this research, the term ‘zero-copy’ is used to describe techniques used to remove the kernel from the data-path between the user-space application and the NFP device. Besides facilitating the transmission of the data, the kernel is not involved in any of the processing and thus does not actually need to be able to perform any read or write operations on the data. To achieve this type of zero-copy operation, references

to host memory regions into which data must either be sourced or inserted are supplied to the kernel. The virtual pages associated with the user-space memory region can then be collected by the kernel, allowing it to reserve underlying physical pages for DMA operations. This enables the DMA operation to directly interact with the user-space memory, bypassing the need for kernel buffers and reducing the number of intermediate copy operations.

5.2 Driver Functions

To account for the extended functionality of the NFComms framework, two additional driver function handlers are required. These handlers support the allocation and reservation of physical pages for transmission between the two architectures. As with the existing handler functions, each is associated with a unique IOCTL number which is supplied along with the necessary parameters when a call to the driver is made. This IOCTL number is registered with the NFP driver on startup and used by the IRQ manager to determine which handler function should be used to process the supplied parameters. By separating the indirect communication events from their direct communication counterparts, the indirect handler functions can act independently from existing driver functions. Thus, when a call to the functions introduced in this section is made, it can safely be assumed to be an indirect operation. In addition to the inclusion of additional handler functions, the current DMA message structure introduced in Section 4.1.5 must be extended.

5.2.1 DMA Message Structure Extension

To help distinguish an indirect message from a direct one, the DMA message structure described in Section 4.1.5 is extended to include an additional *type* field. This field can be used to help identify the presence of an indirect request from a direct one. The choice of adding a separate field to help in this identification is to provide a reliable method of distinguishing message types that could be extended to include additional types as part of future work. Such types could include asynchronous messages, control messages or debug messages.

To allow for zero-copy operations when transmitting the actual message reference, one additional field was added to the DMA message structure, the *address* field. This field is set to be an `unsafe.Pointer` allowing the message to carry a reference to the payload location without having to convert that reference into a format that could be stored in the *message* field. The revised DMA message struct is presented in Figure 5.2.

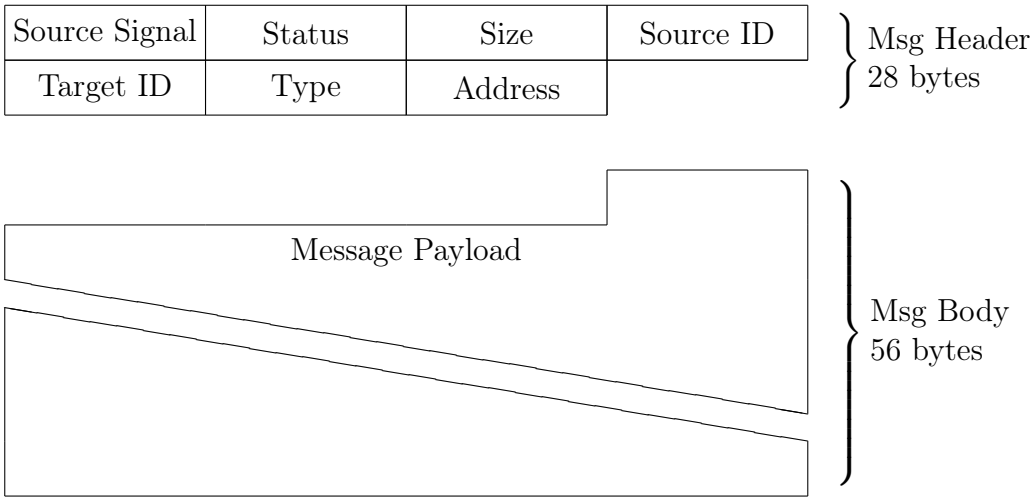


Figure 5.2: Revised layout of the DMA message structure.

5.2.2 Status Code Extensions

To account for the additional transactions that the handler functions support, the set of status codes described in Table 4.1 was extended to include additional indirect specific codes. Firstly, to allow the NFP engines to identify an indirect message received from the host, the status code `DMA_STATE_IND_XFER` was added to the supported list. This code is used for all indirect messages originating from the host that are neither error nor completion messages. To account for these additional cases, the respective status codes `DMA_STATE_IND_ERR` and `DMA_STATE_IND_DONE_ACK` were also included. For indirect messages originating from the NFP, the status code `DMA_STATE_IND_ACK` was added to allow the NFP to acknowledge the intermediate transfer events and `DMA_STATE_IND_DONE_ACK` to acknowledge a complete indirect message transfer. For initiating a transfer event and propagating the occurrence of an error to the host, the described `XFER` and `ERR` codes can be used. The final set of error codes supported by the NFComms framework is presented in Table 5.2.

5.2.3 DMA Indirect Handler

To support indirect communication, two handler functions have been added to the driver: DMA Indirect Write and DMA Indirect Read. The approach taken in resolving an indirect communication event in both cases however, is almost identical. Because of this, these handler functions act as intermediaries with the body of the logic residing in a common function, the DMA Indirect Handler.

The goal of the DMA Indirect Handler is to manage the acquisition of user-space memory

Table 5.2: Possible states for the DMA message struct.

Code	State Name	State Description
0x01	DMA_STATE_READ	The initiator has received a read operation.
0x02	DMA_STATE_READ_ACK	The initiator has acknowledged the read operation.
0x03	DMA_STATE_READ_FAIL	The initiator has acknowledged the read operation but has failed to transmit.
0x04	DMA_STATE_WRITE	The initiator has received a write operation.
0x05	DMA_STATE_WRITE_ACK	The initiator has acknowledged the write operation.
0x06	DMA_STATE_IND_XFER	The initiator requested an indirect transfer.
0x09	DMA_STATE_IND_DONE_ACK	The initiator acknowledged the completion of an indirect transfer.
0x21	DMA_STATE_IND_ACK	The initiator acknowledged an indirect transfer.
0x80	DMA_STATE_IND_DONE	The initiator announced the completion of an indirect transfer.
0x88	DMA_STATE_IDLE	The initiator is ready to handle a new transaction.
0xFF	DMA_STATE_ERR	The initiator has encountered an error.
0xFFA	DMA_STATE_IND_ERR	The initiator has encountered an error.

from a requesting application and make it available for DMA operations performed by the NFP. To operate, this function requires a buffer number and a DMA extended message structure. During the initial stages, the operation of this handler is similar to both the DMA Read and DMA Write Handlers presented in Listings 3 and 4, respectively, as all three begin by first moving the relevant message data from user-space to kernel-space and using it to populate an *nfp_dma_message* structure. The indirect handler function then begins to deviate as it must first prepare the pages associated with the user-space address for DMA operations before initiating a transfer event. This process begins in step **(5)** of Algorithm 12 where the offset into the first page associated with the memory to be transferred is calculated. As described by Corbet *et al.* (2009), this can be done by separating the supplied user-space virtual address into the offset and page frame number. This offset is used to tell the NFP where within the first page it should begin the read operation. Following this, the number of pages associated with the message contents is determined in step **(6)** and is used to reserve a list of page structures in step **(7)**. This list is used in conjunction with `get_user_pages` to pin the user-space memory in step **(8)** so that it cannot be moved until the DMA operation has been resolved. The final component required before initiation of the transfer event can begin is a gather list into which references to all pages along with their offsets can be stored.

```
if message write event requested then
    1. Allocate space for a DmaMessage structure in kernel memory;
    2. Allocate space for a dma_message structure in kernel memory;
    3. Move user-space DmaMessage structure to allocated kernel memory;
    4. Populate dma_message structure;
    5. Construct page_offset from user address;
    6. Determine page_count of message body;
    7. Use page_count to reserve a page_list;
    8. Populate page_list with physical pages associated with user address;
    9. Allocate gather_list of 6 page elements;
    while there is transferred data do
        10. Populate the gather_list ;
        11. Write populated gather_list to dma_message;
        12. Set dma_message.status to DMA_STATE_IND_XFER;
        13. Assert dma_message.source_signal;
        14. Periodically check shutdown and dma_message.status ;
        if shutdown set then
            15. Free reserved memory and pages;
            16. return SHUTDOWN;
        end
        if dma_message.status == DMA_STATE_IND_ERR then
            17. return ERROR;
        end
        if dma_message.status == DMA_STATE_IND_ACK then
            18. continue;
        end
    end
    19. Set dma_message.status to DMA_STATE_IND_DONE ;
    20. Assert dma_message.source_signal;
    21. Sleep on shutdown and dma_message.status ;
    if shutdown set then
        22. return SHUTDOWN;
    end
    if dma_message.status == DMA_STATE_IND_DONE_ACK then
        23. return SUCCESS;
    else
        24. return ERROR;
    end
end
```

Algorithm 12: Pseudocode for the DMA Indirect Write Handler.

As the NFP device does not support scatter-gather operations when interfacing with host memory[2], the proposed approach is to implement a software level variant of this operation which is first introduced in step **(9)** of Algorithm 12.

The first notable characteristic of this list is that it can only contain six elements. As the intended solution to introducing bulk transfer operations is to use the existing direct messaging implementation for the transmission of metadata, the size of the gather list has been dictated accordingly. For this implementation, the gather list supports up to six entries as this is the maximum number that can placed in a 56-byte buffer along with basic metadata. Each list entry contains the necessary information needed by the NFP DMA controller to move a single page of memory from the host to the NFP device. This basic information is as follows:

- number of bytes within the page containing the message data
- page offset to where the message data begins
- physical address of the page compatible with the IOMMU (if present).

Once the gather list has been instantiated, the handler function begins the process of submitting pages to the NFP for it to use in DMA operations. Referring back to Figure 5.1, this process effectively represents events (1c) and (1d) where the actual transfer operation takes place. The goal for the remainder of this algorithm is to iteratively mark each page for a DMA operation and submit it to the relevant NFP Read Manager. As each page is marked, it is inserted into the gather list until either the whole message has been pinned or the gather list is full. Once step **(10)** is complete, the gather list is packed as a message to be submitted to the NFP device, representing event (1d) of Figure 5.1. The number of list elements as well as the number of bytes outstanding are added to the message before it is submitted to the NFP device using the direct message passing routine.

Once submitted, the handler allocates itself to the kernel wait queue in step **(21)** of Algorithm 12 until woken. The wakeup condition for this blocking event is if the DMA message struct indexed by the supplied buffer number has had its *status* field changed, or if a shutdown event has occurred. The decision to use an interrupt based approach for receiving an acknowledgement for this transaction rather than polling is largely because of the expected volume of data to be transmitted. For direct communication, each message can at most span two TLP transfers and as a result, the anticipated latency between submission and acknowledgement is short. It is anticipated that indirect transfers will be

[2]It does support a variant of scatter-gather for memory regions within itself.

used to move larger volumes of data. This will result in many intermediate transactions operating on a fully populated gather list, requiring eight TLP transfers. Due to the increased duration between acknowledgements, it was determined that the resources required to register the handler function for an interrupt and subscribe it to the wait queue would be less than those required for polling.

When woken, the shutdown flag is checked to determine if a shutdown event is underway. If the shutdown flag has been set, the handler function cleans up all allocated memory and unpins the user-space pages before returning. Otherwise, if the *status* field contains `DMA_STATE_IND_ERR`, an error has occurred and this is propagated back to the calling handler function. If the *status* field contains `DMA_STATE_DONE_ACK`, the data associated with the currently populated gather list has successfully been handled by the NFP and the transaction can continue.

Along with the updated *status* field, the NFP also sets the size field to its expected number of outstanding bytes. If this value does not match what the handler function expects, then one of two events occur. If the NFP thinks the number of bytes outstanding is more than the handler function and the difference correlates with the number of bytes the previous gather list represented, the list is resent. If the NFP reports an outstanding byte count that is less than the handler of the function or if the outstanding count does not match the size of the last gather list, it is considered an erroneous transaction and this is reported to the calling handler function.

If the transfer was a success and there is still outstanding data to transfer, the algorithm returns to step **(10)**, continuing the process of preparing pages for the NFP to interface with. If there is no outstanding data, the process is complete and the handler function submits an empty message to the NFP in step **(20)** with the *status* field set to `DMA_STATE_IND_DONE`. This informs the NFP that the transfer is complete from the context of the DMA Indirect Handler to which it must either reply with `DMA_STATE_IND_DONE_ACK`, acknowledging the transfer, or `DMA_STATE_IND_ERR`, indicating a failure occurred.

5.3 NFP Engine Extensions

To facilitate an indirect transfer from the context of the NFP, both the NFP Read and NFP Write engines would need to be extended to support the additional functionality. Initially separate indirect specific engines were considered, however as outlined in Section 1.2, one of the objectives of this work is to provide a framework with a minimal resource

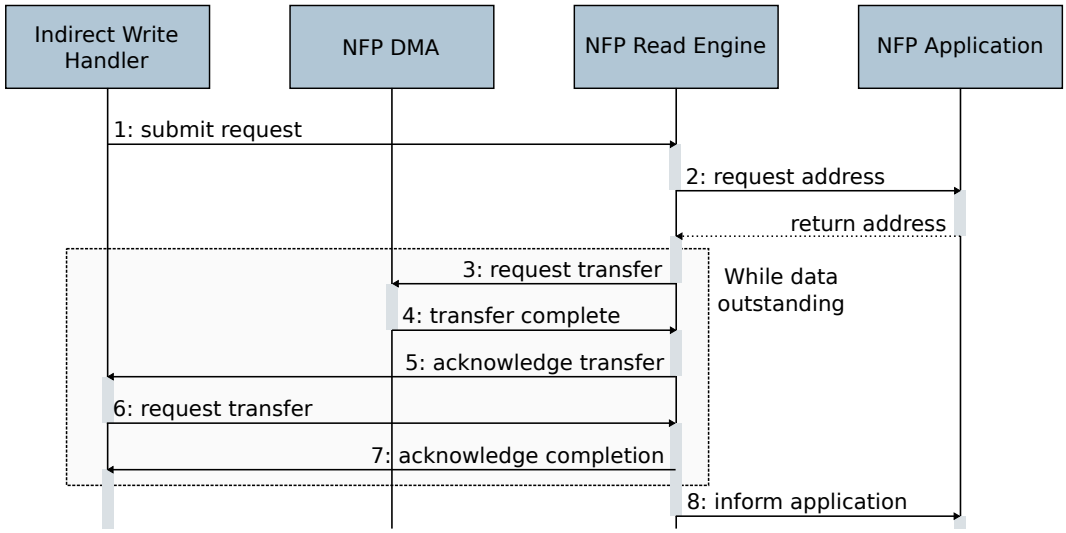


Figure 5.3: Indirect read transaction facilitated by the NFP Read Engine.

footprint on the NFP. Considering this, it was decided that the existing engines would be responsible for both types of transactions.

5.3.1 Read Engine Extensions

Considering the original framework configuration described in Section 4.4, when the NFP Read Engine received a signal from the host, it indicated the presence of a transfer event that the engine was expected to facilitate. Although this component of the transaction is unchanged, the message received could now be either a direct or an indirect message. To distinguish the start of an indirect transfer from a direct message event, the extensions made to the DMA message structure in Section 5.2.1 have been duplicated for the NFP specific implementation of the equivalent structure. This allows the host to include the *type* field in the metadata of the message being transmitted which can be used to indicate whether the message is part of a direct or indirect transfer. On receipt of a message, the managing thread can either forward the message to the destination process as a direct transaction or initiate the indirect transaction routine based on the *type* field in the extended message header. Listing 13 loosely describes how the indirect read routine handles a message transaction with the interactions between the NFP Read Engine and external parties being presented in Figure 5.3.

To initiate indirect read operation from the perspective of the NFP, the Indirect Write Handler first submits a write request to the NFP Read Engine, represented by event (1) in Figure 5.3. After the request is confirmed to be an indirect transfer, the NFP Read Engine begins by requesting the destination address for the message body from the destination

process. If the destination process is not ready to partake in the transaction, the routine blocks until either the destination is ready or a time-out occurs. Assuming the former, the destination responds to the request, supplying the destination address as well as the expected size of the transaction. The message size is compared with the size received from the host and if they are not equal, an error message is generated and relayed to the host in steps **(2 - 5)** of Algorithm 13. The routine then exits, returning the manager to the idle state. Should the message sizes match, the read routine prepares a DMA configuration for targeting memory regions within the NFP.

Considering Algorithm 12, for each iteration of the internal loop from step **(10)** through to step **(18)**, the events ③ through ⑥ described in Figure 5.3 must occur. This sequence of events describes how the body of the message is transmitted. The initial message submitted to the NFP Read Engine not only requests the initiation of the transfer event, but also contains the first gather list for the transfer operation. The engine iterates through each entry in the gather list and constructs a DMA descriptor in step **(8)** to represent the transaction. This transaction is then submitted to the DMA engine requesting that the required number of bytes be read from the supplied DMA address to the location specified by the destination process. Once all entries in the gather list are submitted and acknowledged, the engine updates the outstanding bytes and page counters before sending an acknowledgement back to the Indirect Write Handler, informing it that the request has been processed. If the number of outstanding bytes is not 0, the NFP Read Engine expects another transfer request to be sent from the host which it must process. If the engine receives a request with the *status* field set to `DMA_STATE_IND_DONE`, it immediately responds with `DMA_STATE_IND_ERR` as an error has occurred and not all the data has been transferred.

Once all DMA requests have been fulfilled, the read manager can signal the host process of a successful transaction and wait for an acknowledgement message to be returned. Following this, the target process is signalled indicating that the message body has been transferred successfully. At this point, the transaction is considered complete and the indirect read routine returns, allowing the NFP Read Engine to facilitate a new transaction.

5.3.2 Write Engine Extensions

The extensions to the DMA Write Engine operate in a similar manner to the extensions described for the DMA Read Engine which is to be expected given that both ultimately interface with the same handler function, the DMA Indirect Handler. Thus, rather than reiterate the transfer specific points described in the previous section, only the initiation

```
if the request is for a bulk read transaction then
    1. Request the destination address from target process;
    if target process size <> source process size then
        2. Add message size mismatch error code to DMA message;
        3. Add target process expected size to DMA message;
        4. Add source process expected size to DMA message;
        5. Send message to host;
        6. Exit the indirect read routine;
    end
    7. Set up a DMA configuration register for memory unit addressing;
    while there are still outstanding pages to process do
        for each page address supplied do
            8. Build a DMA descriptor for the transaction;
            9. Submit the descriptor requesting a DMA read operation for the supplied page
               address;
        end
        10. Wait until all pages have been handled by DMA engine;
        11. Update outstanding bytes and page counters;
        12. Create an acknowledgement message including the updated counters;
        if there are no more outstanding pages then
            13. Set the message status field to indicate that the transfer is complete;
        end
        14. Send the acknowledgement message to NFP using direct messaging framework;
    end
    15. Wait for a transfer complete message from the host;
    16. Handle any error state;
    17. Signal the destination process that the transfer is complete;
end
```

Algorithm 13: Pseudocode executed to perform a bulk read transaction on the NFP.

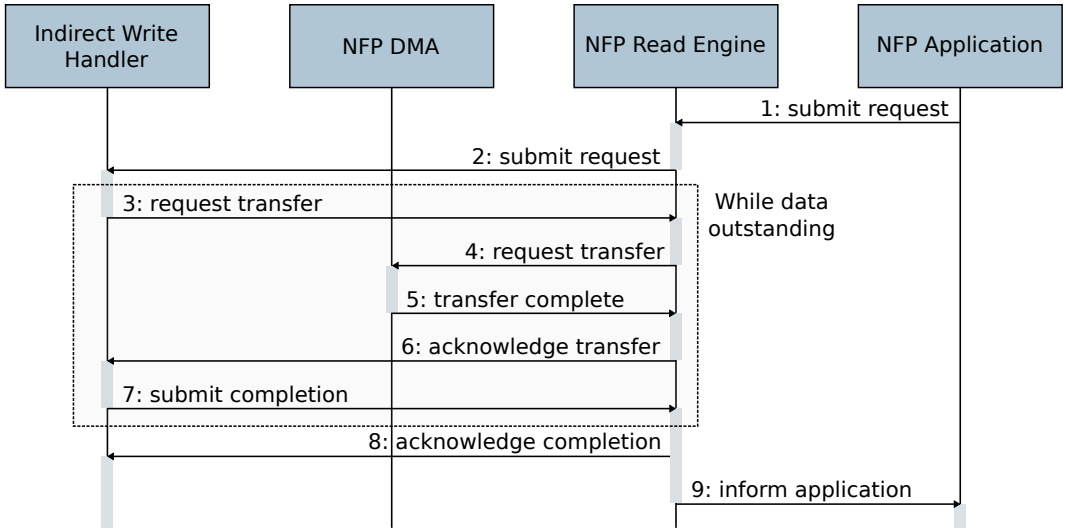


Figure 5.4: Indirect write transaction facilitated by the NFP Write Engine.

and completion components of this transaction are detailed.

For an indirect write transaction originating from the NFP, the operation begins with the relevant NFP process submitting a request to the NFP Write Engine using the procedures described in Section 4.4. As there is no type field to check and the number of available bytes to reserve for metadata is very limited at this stage of the transaction, the *size* field is instead used to differentiate direct messages from indirect. If the size exceeds the maximum allowable size of a direct message, the message is assumed to be using indirect message passing and treated accordingly.

Once the active NFP Write Engine confirms that the submitted request is an indirect request, it constructs and submits a message to the host using the existing direct message routine. At this point the active engine waits until it receives a signal from the host indicating that the response to the initial message is ready to be processed. The active engine reads the message response from RAM and checks the *status* field to determine if the initial message transmission was successful and the destination process is ready to engage in the transaction or not. If the destination process is not ready, the *status* field is set to `DMA_STATE_READ_FAIL`, indicating that the handling of this message should be re-queued so that the current resources can be used to handle other messages. The details relating to how this is handled are the same as described in Section 4.4.2.

If the initial message was successfully transmitted and the destination process is ready to engage in the transfer, the *status* field is set to `DMA_STATE_IND_READ`, indicating that the host is ready for the transfer event to occur. Along with this message, the host includes

the first gather list of page addresses where the NFP can write the message data. This message acts as event ③ in Figure 5.4 which begins the transfer loop as described in the previous section. As these transactions were initiated by the NFP source process, the memory address for the message body was submitted as part of the initial request so the transfer can begin immediately. Considering the transfer operations described for Algorithm 13, the read operations occur in the exact same manner except the descriptors supplied to the DMA controller are set to push data to the supplied RAM address rather than pull data.

Once the transfer operation is complete, the host acknowledges the request by setting the *status* field of the last message to `DMA_STATE_IND_DONE`. The active engine responds to this with an empty message containing the status `DMA_STATE_IND_DONE_ACK` and relaying the completion of the transaction to the initiating NFP process.

5.4 Library Extensions

An important consideration when implementing the indirect extensions is to minimise the impact that such extensions could have on the operation of the existing NFComms framework. By preserving the current functionality, backwards compatibility of the extended framework with existing applications can be maintained. To allow applications to utilise the extensions however, two additional library functions for both the NFP and the host must be included. For the NFComms library these are `SendInd()` and `ReceiveInd()` while for the NFP, `channel_write_ind()` and `channel_read_ind()` are used.

5.4.1 Host Library Extensions

For the NFComms library, the two new functions introduced follow the same format as the functions presented in Listing 4.2. As with `Send()` and `Receive()`, the corresponding extended functions accept the same set of parameters, but also require an additional size parameter. This size parameter allows the function to indicate the volume of data (in bytes) which should be transmitted to the destination process. This additional parameter has been included to allow for a slice of a very large data object to be transmitted without requiring subdivision prior to submission. The extended set of runtime functions is presented in Listing 5.1.

```
1  func New(number uint32) NfpComms
2  func (n NfpComms) AddConnection(id uint32, island int, microengine int,
3                  context int, signal int, reg int) (err error)
4  func (n NfpComms) Start() (err error)
5  func (n NfpComms) Stop() (err error)
6  func (n NfpComms) Send(id uint32, data interface{}) (err error)
7  func (n NfpComms) Receive(id uint32,  data interface{}) (err error)
8  func (n NfpComms) SendInd(id uint32, data interface{}, size int)
9                  (err error)
10 func (n NfpComms) ReceiveInd(id uint32, data interface{}, size int)
11                 (err error)
```

Listing 5.1: Extended NFComms runtime user functions.

Within the host, the initiation of an indirect message passing event to the NFP begins with a `SendInd()` call to the NFComms library. As with `send()`, the application must supply a valid target ID that has previously been subscribed to the active `NfpComms` object. The second parameter is a reference to a declared object with a static size. This object contains the data that the message passing event will attempt to transmit to the NFP. The final parameter is the number of bytes to be transmitted starting from the supplied memory reference.

When called, `SendInd()` begins by confirming that the supplied ID is in fact valid and if not, returns an error. Following this, the supplied data reference is evaluated. This data object is set to a generic interface to allow for a wide range of possible object types to be passed through this framework. In order to pass a reference of this object to the NFP API however, a C-compatible pointer is required that cannot be created unless the type of the object is known.

Unfortunately, by not forcing the supplied data to be of a specific type, the Go application is required to perform a runtime evaluation of the supplied object using the reflect package[3] to determine its type. This operation is time consuming which in turn impacts the performance of the message passing event. To help mitigate this, a series of assertions are first run against the supplied data object to test if it is one of the built-in types supported by Go. A subset of the currently supported assertions are presented in Listing 5.2 which describes how the data type is evaluated. If an assertion passes, the application can safely continue knowing the data type of the supplied object and acquire a pointer that can be supplied to the NFP API. If the assertion fails, the fallback is to use the reflect package to determine the type of the object and convert it into a byte array from which a pointer

[3]For more information see `https://golang.org/pkg/reflect/`

can be created.

```
switch asserted := data.(type) {
  case *byte:
    address = unsafe.Pointer(asserted)
  case []byte:
    address = unsafe.Pointer(&asserted[0])
  case byte:
    address = unsafe.Pointer(&asserted)
  case *uint32:
    address = unsafe.Pointer(asserted)
  case []uint32:
    address = unsafe.Pointer(&asserted[0])
  case uint32:
    address = unsafe.Pointer(&asserted)
  //Other assertions omitted
  default:
    //Reflect operation
```

Listing 5.2: Subset of type assertions made by the NFComms runtime.

This message is then passed to the Host Write Handler which calls `WriteToDMA` (discussed in Section 4.3.3) to handle submission to the driver. To account for indirect operations, `WriteToDMA` is extended to first check the *type* field of the supplied DMA message structure and determine if the supplied message is an indirect message. If not, the function behaves as previously described. If the message is an indirect one, a message is built using the new address field in addition to the size, source ID, target ID, and buffer number parameters. A call is made to the NFComms driver via the IOCTL interface with a reference to the DMA message structure and buffer number supplied as a parameter. This request is then processed by the DMA Indirect Write Handler which accepts and checks the supplied parameters before calling the DMA Indirect Handler, discussed in Section 5.2.3.

For the reception of an indirect message from the NFP, `ReceiveInd()` is used; a high-level overview of the host-based transactions involved is presented in Figure 5.5. As with all other communication functions, this is a blocking operation that waits until either the operation has completed successfully, or an error has occurred. This function takes in the same parameters as the `SendInd()` previously discussed with the initial stages of input validation and construction of a C-compatible pointer occurring in the same manner. Once a valid request has been confirmed and a pointer to the destination memory has been created, `ReceiveInd()` blocks until it receives a message from Host Read Engine, represented by event ③ in Figure 5.5.

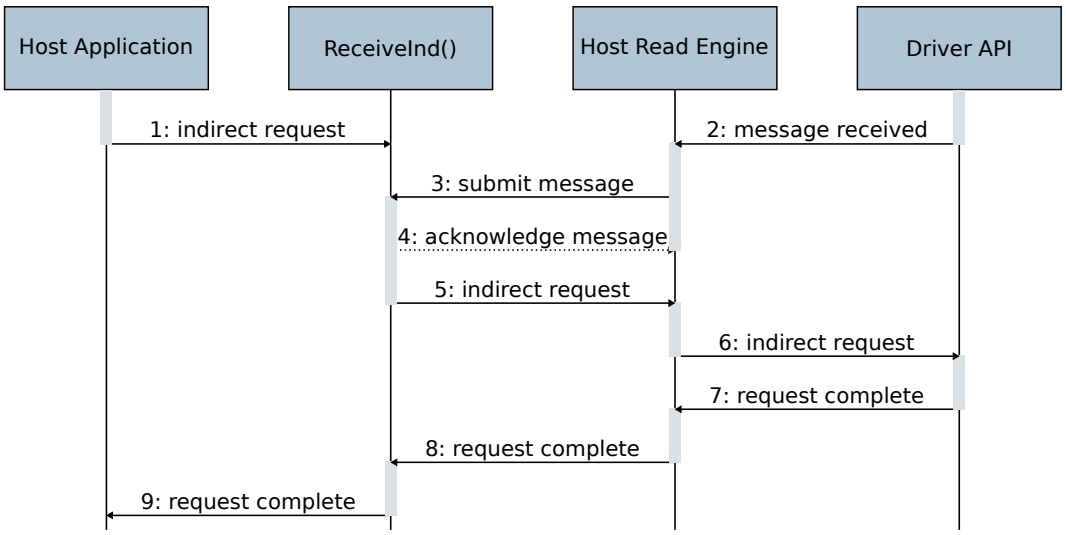


Figure 5.5: Indirect read transaction facilitated by the host Read Engine.

The *status* field of the received message is checked to confirm it is an indirect transfer and if not, an error is returned to the caller. If the message is an indirect transfer, it implies that the source NFP process has initiated the transfer event and the relevant NFP Write Engine is ready to begin the transfer of the message body. For this transaction to occur however, the destination address for the message body must be supplied to the NFP. Before this is done, the *size* field of the received message is checked to confirm that the message size submitted from the NFP matches the size of the allocated memory region on the host. If not, this is considered an error and the calling application is informed accordingly.

Provided no error has occurred, `ReceiveInd()` acknowledges the received message, informing the Host Read Engine that it is ready to partake in the transaction. The *address* field for the received message is then updated to contain the address of the destination memory location before being relayed back over the associated message channel for the Host Read Engine (discussed in Section 4.4.3) to receive. This operation is represented by event (5) in Figure 5.5 and informs the Host Read Engine that it can begin transferring the message body. When the Host Read Engine receives the indirect read request, it can immediately call `ReadFromDMA` to handle the transmission of the request to the driver where it can in turn be forwarded to the NFP Read Engine actively handling this request. As with `WriteToDMA`, `ReadFromDMA` has been extended to used the *type* field to distinguish between direct and indirect messages with indirect requests being passed to the DMA Indirect Handler discussed in Section 5.2.3.

Once the message body has been moved from the NFP to the supplied memory location

on the host, an acknowledgement is relayed to the active Host Read Engine. This acknowledgement is forwarded to the calling `ReceiveInd()` which in turn relays it back to the calling host process. The resources reserved for the event can then be released as the transaction is complete.

5.4.2 NFP Library Interactions

Considering the NFP device, the supporting communication library was also extended to support two additional functions: `channel_write_ind()` and `channel_read_ind()`. The extended library set for interfacing with the NFComms framework from a microengine is presented in Listing 5.3. Unlike the NFComms runtime extension however, functions `channel_write_ind()` and `channel_read_ind()` do not accept the same parameters as their direct communication counterparts. This is largely to do with the source and destination locations of messages residing outside of the microengine. Instead of requiring a set of transfer registers for the transaction, `channel_write_ind()` and `channel_read_ind()` require a 40-bit memory address and the size of the message body in bytes. Additionally, the `count` parameter seen in the direct messaging functions has been replaced with a `size` parameter. This change is again due to how data is handled. When referring to transfer registers, it is better to specify the number of registers involved, with each register containing four bytes. For memory regions, the memory is addressed in bytes and as such, specifying the size of an object must be done at the byte resolution. Finally, with respect to the overall framework, no changes beyond the inclusion of these additional functions was performed and as such these extensions should not affect existing applications utilising it.

```
1  int channel_read(__xread unsigned int *xfer, unsigned int count,
2                   __xread unsigned int *xfer_sync, SIGNAL *sig_sync);
3  void channel_write(__xwrite void *xfer, unsigned int count,
4                   unsigned int target_addr);
5  void channel_write_ind(void __addr40 __mem *pTarget, unsigned int size,
6                   unsigned int target_addr);
7  int channel_read_ind(void __addr40 __mem *pTarget, unsigned int size,
8                   __xread unsigned int *xfer_sync, SIGNAL *sig_sync);
9  void channel_write_sig(__xwrite void *xfer, unsigned int count,
10                  unsigned int target_addr, SIGNAL *client_signal);
```

Listing 5.3: Extended NFP library functions.

5.5 Testing

Evaluation of the implemented extensions was performed by dividing the testing into three distinct components. The first section focuses on timing individual messages to determine the latency impact of sending messages using the framework extensions compared with the original direct messaging approach. The second component focuses on recording the maximum achievable throughput of the implemented extensions for varying message sizes and process counts. The final component of the testing attempts to evaluate performance of the extensions using the fractal generation example presented in Section 4.6.1 but modifying it to utilise indirect communication.

As noted in Section 5.1.1, the framework was designed to target a range of memory types however, for the testing of the extensions, all message transactions were performed using EMEM to make the tests comparable. This memory region was selected as it supports memory capacities large enough to accommodate the data volumes used in throughput testing to confirm the ability of the bulk transfer operations.

5.5.1 Latency Testing

Latency testing was performed using a basic application implementing a simple producer-consumer model. The architecture on which each component executed was alternated between tests to monitor the impact the origin of a message could have on latency. To help improve testing accuracy and minimise the potential performance impact of startup procedures, timing results were only recorded for message batches transmitted after at least one successful batch transmission had already occurred.

Both the producer and consumer consisted of a single process running for the duration of the test. The messages transmitted between the two processes were limited to 56 bytes which coincides with the maximum size of a single direct message event. Tests with messages sizes smaller than 56 bytes produced no significant variation in latencies. Testing was performed by transmitting 10,000 message batches from consumer to producer with timing performed on every batch.

Though synchronous, the act of consuming a message is an atomic event from the perspective of the consumer. Once consumed, the acknowledgement is propagated back to the producer at which point the transfer of a new message is initiated. Should timing be performed on individual transactions, the overhead of setup and storage relating to each timing event could potentially overlap with the initial stage of the transmission being initiated by the producer. This would result in the timings recorded not correctly

Table 5.3: Average recorded message latencies (in ms) using direct and indirect message passing over 10,000 iterations.

	Min	**Max**	**Average**	**Standard Deviation**
NFP to Host (Direct)	3.21	159.42	30.04	7.43
Host to NFP (Direct)	7.57	37.09	17.99	3.25
NFP to Host (Indirect)	24.38	118.97	51.96	8.48
Host to NFP (Indirect)	28.95	61.12	39.42	3.45

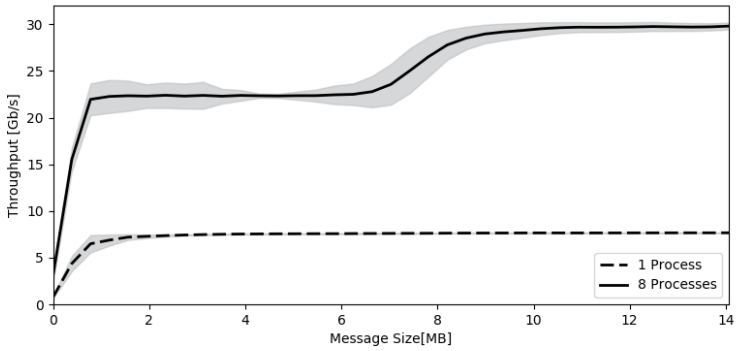
observing the full message transaction latency. To help mitigate this issue, timing was instead performed over the transmission of a single batch containing 100 messages.

The test was repeated four times, recording latencies for each architecture and each messaging medium. The results of these four tests are presented in Table 5.3 as average latencies observed per test. Though care was taken to keep the tests uniform between the four configurations, tests involving indirect communication required an additional step to move data between the memory region targeted by the message transaction and a location local to the NFP process. This additional step was included so that the NFP source or destination location could be kept uniform for all tests. Though included, explicit testing showed that this additional step introduced a latency increase of less than 3% on indirect message passing.

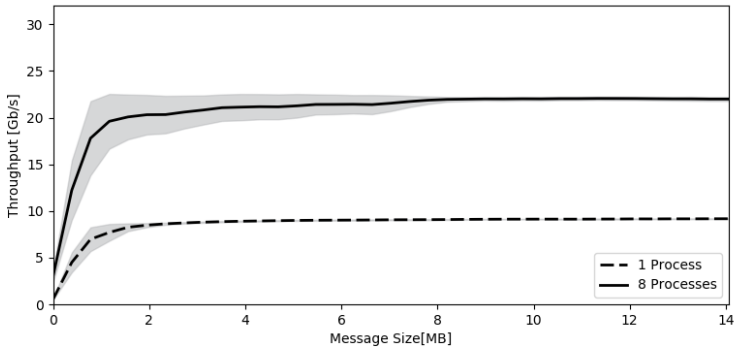
Comparing the results recorded in Table 5.3, the average latency for a direct message originating from the host is approximately half that of the equivalent operation using the indirect extensions. A similar pattern can be seen for transactions traversing in the opposite direction with the indirect operation recorded to be 1.73× slower than its direct counterpart. These results differ slightly from the latencies for indirect communication documented in (Pennefather *et al.*, 2018b). This is due to minor alterations added to the framework to improve stability and error handling. The modifications are expected to reduce throughput performance slightly, but their impact on latency is minimal.

5.5.2 Throughput Testing

Given that the focus of the extensions was to improve the throughput of the NFComms communication framework, an evaluation of the bulk transfer capabilities associated with these extensions was required. This evaluation was performed by using a simple test application executed on four different configurations that records bulk throughput for transmission of data between the two architectures in both directions. For each direction, two transmission tests were performed: one using a single source or destination NFP



(a) Host to NFP



(b) NFP to Host

Figure 5.6: Recorded throughputs for indirect messaging transmissions.

process to send or receive the message payload, and one using eight source or destination NFP processes to perform the same task.

The test application consisted of two processes communicating a single 100MB payload in one direction. For each test configuration, both the source and destination processes were executed on opposing architectures and all data transmitted from the source process was checked for integrity at the destination. All timings reported exclude the integrity check which recorded no corruption throughout testing. Each test consisted of repeatedly performing the bulk transmission, using incrementally larger message payload sizes on each iteration up to a size of 20 MB. The time taken to complete each transfer was recorded from which the average throughput for each message transmission was calculated.

For each of the four test configurations, 150 tests were performed and the collected throughput results for the first 14 MB are presented in Figure 5.6. Results for the remaining 6 MB have been omitted as no variation was noted. The average recorded throughput has been plotted with the shadows representing one standard deviation for each test point.

In the test configurations involving the eight NFP processes, the host-based process either read or wrote chunks of the 100MB payload to the different NFP processes in a controlled fashion. Each NFP process was responsible for handling explicit chunks of the message so that the source or destination of each chunk within the message block could be known. The reasoning for including this configuration test was to evaluate the peak throughput achievable by the NFComms framework including situations where multiple indirect message passing events were processed concurrently.

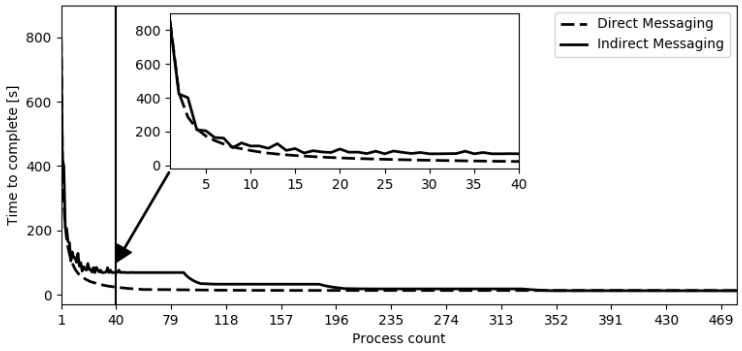
In Figure 5.6(a) the throughput for sending data from the host to the NFP is recorded. In situations where only one message transmission operation was performed, the throughput exhibited some minor fluctuations until the individual message payloads reached approximately 2 MB in size. From 2 MB, increasing message payload sizes resulted in relatively negligible changes to throughput, settling at an average of about 7.8 Gbit/s for the remainder of the test points. The host-to-card configuration involving eight processes did not exhibit the same pattern in throughput performance. As with the single process configuration, throughput performance for the eight process configuration increased to a peak average throughput for 2 MB of approximately 20 Gbit/s. However, the throughput began to further increase after the message payload size exceeded 8 MB.

Figure 5.6(b) shows the equivalent data as Figure 5.6(a) except that messages occur in the opposite direction. Comparing the single process throughput for either direction, we see that both test configurations show a similar shape except that transfers originating from the NFP exhibit a slightly increased average throughput of 9 Gbit/s for message sizes exceeding 8 MB.

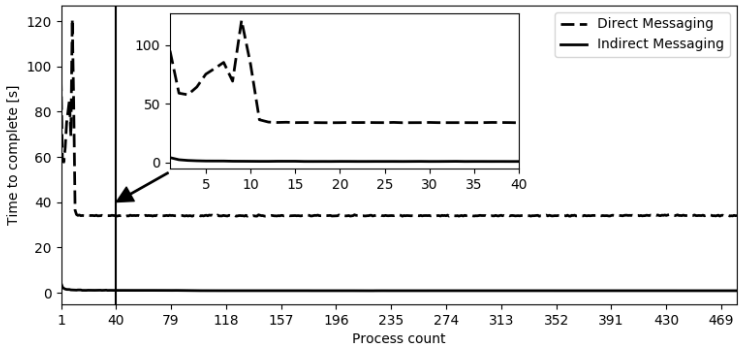
Considering the throughput of the test configurations utilising eight processes, the variance in recorded times shows a large degree of fluctuation when the message payload size is smaller than 8 MB. Message sizes beyond this limit exhibit an average throughput of 22 Gbit/s with very little variance. As noted during the latency testing, these results differ from those of the equivalent test presented previously (Pennefather *et al.*, 2018b). The sustained throughput of the bulk transactions has dropped from the initially reported peak throughput of 35Gb/s. However, in exchange for the reduction in performance, the stability of the bulk throughput operations has improved significantly with fewer fluctuations noted during testing.

5.5.3 Application Testing

Section 4.6.1 described an application example using the NFP to assist in the generation of fractal shapes pertaining to the Mandelbrot set (Mandelbrot, 2004). This application



(a) 300x400 image to depth 0xFFFFF



(b) 1200x1600 image to depth 0xFF

Figure 5.7: Execution times for the mandelbrot set using different communication mediums.

worked by treating the NFP processes as a pool of workers to which work could be allocated until all work associated with the fractal had been processed. The host component of the application was responsible for setting up the work units as well as collecting results and managing the pool.

The original tests associated with the 32-bit implementation of the fractal application were rerun using the updated framework first without any changes to the original application. Thereafter, the original application was modified to increase the size of the fractal to render while also reducing the maximum depth. The observed timings for these tests are presented in Figure 5.7(a,b).

To observe the performance impact of using the extensions presented in this chapter, the mandelbrot application was modified to use indirect message passing. As the original application was designed to make optimal use of the communication medium, all work was moved to the NFP card only when needed. Simply replacing the direct communication calls with indirect communication calls resulted in a greatly reduced performance given the

small workload size and the larger latency associated with indirect messaging. Instead, the application was modified so that the work pool resided on the NFP device itself. Indirect messaging was used to move raw data to the card and results from the card once processing had been completed. The tests run for the original application were repeated with the new results given in Figure 5.7.

Considering Figure 5.7(a), it is clear that once the number of NFP processes exceeds 300, the choice of which communication medium should be used is immaterial. For instances where fewer than 300 processes are used, the direct messing approach shows better performance. In this configuration, the time taken to complete each unit of work is long enough that interleaving the messaging with the processing effectively hides the communication latency cost.

Figure 5.7(b) on the other hand shows a large number of work units, all with relatively small workloads. In this situation, the communication overhead of the direct messaging approach exceeds the time taken to process individual work elements and so time to complete the application quickly converges to 34 s using more than 12 processes. In comparison, using the indirect communication approach results in the application completion times dropping to 1.4 s with more than four processes and converging at under 1 s if more than 14 processes are utilised.

5.5.4 Performance Discussion

Considering the recorded results from latency testing, it is clear that the indirect messaging framework is notably slower for transmitting individual messages compared with the direct messaging one. This is to be expected given that the extensions presented here utilise the direct messaging component as the underlying communication medium. As a result, when transmitting messages smaller than 56 bytes in size, the direct messaging functions should be used, especially if latency is a consideration for the target application. In terms of throughput however, testing showed that the implemented extensions are capable of achieving an average throughput of 7.8 to 9 Gbit/s when servicing a single transaction, and up to 30 Gbit/s when servicing eight transactions.

By comparing the performance times of the two variants it can seen that in some situations, such as when processing times for individual work units exceed communication latency, using direct messaging results in a faster completion time. In situations where the volume of data to be moved is very large, or where communication cannot be interleaved with processing, it is more beneficial to use indirect messaging. This indicates that the indirect communication extensions do have value while not overriding the existing

communication functionality of the NFComms framework.

Section 5.5.3 describes the testing performed on two variants of a simple heterogeneous application: one using direct messaging for communication and the other, indirect messaging. The initial goal of the indirect messaging extensions was to improve the maximum throughput that the NFComms framework is capable of achieving. Considering that the original maximum throughput achievable by the framework was 102.4 Mbit/s, the indirect messaging extensions represent an improvement of up to 268× when using the indirect messaging extensions. Though this comes at the cost of increased latency on individual messages, this trade-off is considered acceptable as the framework can still facilitate direct messaging transactions in situations where small, low latency transactions are required.

5.6 Summary

This chapter presented the design and implementation of the bulk throughput extensions necessary to mitigate the poor throughput performance of the NFComms framework. The source of the throughput limitations was identified in Section 5.1 where concerns relating to the reflect operation limiting the number of transfer registers available for communication were noted. In addition to these concerns, a discussion relating to the storage locations of larger memory objects which would justify supporting higher throughput operations was also presented. The identified problems were followed by the proposed solution where the concept of using a separate set of indirect messaging operators for indirect operations was introduced. A brief discussion on zero-copy operations concluded the section.

Section 5.2 presented the changes implemented to the driver to facilitate indirect communication. This presentation also included changes to the DMA message struct and the supported status codes that were necessary to realise the extensions. After an evaluation of the operations, it was concluded that both the indirect read and write operations would occur in the same manner from the perspective of the host. As a result, a single indirect handler function was introduced that would handle the acquisition, presentation, and maintenance of pages for the NFP architecture to interact with. In order for the NFP architecture to participate in an indirect operation, the NFP Read and NFP Write Engines were also extended. The modifications made to these engines were documented in Section 5.3.

To allow end-user functions to utilise the indirect message routines, both the NFComms runtime and the NFP communication library were extended accordingly. The required

parameters needed to enable the new functions were discussed and operation of the indirect routines was compared with that of the original direct ones. Finally, testing and a preliminary evaluation of the introduced extensions were presented in Section 5.5 which includes a short discussion on the findings in terms of performance and latency.

Although a communication framework has been developed, the testing presented both in this chapter and Chapter 4 is not complete and thus it is difficult to draw any meaningful conclusions. The subsequent chapter (Chapter 6) provides a more in-depth evaluation of the performance and functionality of both the NFComms framework and the overarching heterogeneous CPU-NFP system.

6

Testing

This chapter presents a series of application examples implemented to test both the functionally and performance capabilities of the NFComms framework. Although Chapter 5 concluded with a performance analysis of the revised framework, testing the resulting heterogeneous CPU-NPU system is still required to better evaluate its viability. This chapter has three goals which it attempts to achieve through the presentation of worked examples.

The first goal is to provide a set of solutions for known problems that can be used to test the performance of the heterogeneous system. The second goal is to provide a qualitative comparison between the heterogeneous system and either a CPU-only solution or a GPU accelerated solution. The third and final goal is to highlight both the strengths and weaknesses of the CPU-NFP heterogeneous system in terms of computation, memory, and communication latency. To achieve these three goals, five application examples have been selected and implemented spanning two domains: non-networking related (Sections 6.1 and 6.2) and networking related (Sections 6.3 through 6.5). For all tests, the same hardware configuration was used for the computing platform. The NFP endpoint device was the NFP-400, used in previous testing (see Sections 5.5 and 4.6) while the specifications of the host are given in Table 6.1.

Table 6.1: Host configuration.

Product Type	**Product Name**
Motherboard	H170 PRO GAMING
Processor	Intel Core i5-6400 CPU @ 2.70 GHz
Memory	DDR4 2133 MHz 16 GB
Network	On-board Intel Ethernet interface 1 Gbit/s

As introduced in Section 1.2, one of the objectives of the research is to explore the viability of introducing a heterogeneous CPU-NPU system for use in general computation. Now that a functional message passing framework for runtime communication between the architectures has been established, the viability of initialising such a heterogeneous system can be explored. The goal of this exploration is to evaluate the feasibility of using the NFP as a coprocessor in general computation. To perform this exploration, two of the five application examples present solutions to computation problems: the travelling salesman presented in Section 6.1 and k-mismatch, presented in Section 6.2. These problems have been extensively explored with solutions that are well known and can easily be replicated (O'Neil *et al.*, 2011; Rocki and Suda, 2012; Clifford *et al.*, 2016). Thus, existing solutions can act as suitable benchmarks for comparison with the heterogeneous CPU-NFP solution.

As the NFP is an architecture designed to be used in the field of network processing, the remaining three applications have been selected to leverage this feature. The intent is to show how the NFComms framework can be used to enrich their functionality by allowing for runtime communication between the NFP processes and the host, enabling real-time interactions for data transfer, and offloading computation poorly suited to the NFP.

Section 6.3 begins by presenting an application designed to estimate the cardinality of IP addresses within a network stream, using the NFComms framework to allow a host process to perform the floating point operations necessary for the estimate. The cardinality estimation is followed by a simple sketching example in Section 6.4 where a Count-min sketch is used to record the top 50 heavy hitters in a network stream. In both examples, the NFComms framework allows a host process to interact with the NFP application, providing an interface for requesting the current results. Finally, as an extended example, Section 6.5 provides an application for performing GeoIP lookup operations on IP addresses within the network stream during execution. This application operates across both the NFP and the CPU with the CPU responsible for performing the actual lookup operations while the NFP handles the collection and transmission of network traffic.

Each application discussed in this chapter is presented as a self-contained section introducing necessary background for the relevant problem before detailing its implementation.

This is followed by a testing and evaluation subsection where correctness of operation and performance is explored. The performance evaluation is based on either benchmarking the application against comparable implementations, or recording achievable operation speeds. The results of these individual testing subsections are collected and reflected upon in Section 6.6. This reflection considers the performance metrics of each implementation and identifies common strengths and weaknesses characteristic of the heterogeneous system under evaluation. A brief summary of some common limitations associated with GPUs is presented in Section 6.7 to help contextualise the state of the CPU-NFP heterogeneous system and associated issues. This is followed by a final decision regarding the viability of using the NFP as a coprocessor presented in Section 6.8 after which, the chapter concludes with a short summary.

6.1 The Travelling Salesman Problem

The first problem evaluated using the NPU as a coprocessor is a common routing problem referred to as the 'Travelling Salesman' problem (Rocki and Suda, 2012; O'Neil *et al.*, 2011). This problem belongs to the set of NP-complete problems with the goal being to find the shortest cycle that connects all nodes in a 2D plane (Rocki and Suda, 2012). Finding the optimal path is a combinatorial optimisation problem with a search space that rapidly increases with larger node counts.

As the node count increases, attempting to find the optimal solution through an exhaustive search of all candidate cycles quickly becomes infeasible. The commonly adopted approach in such situations is to rather use heuristic techniques to find a candidate solution that may not be the optimal solution, but suitable for the specific implementation requirements. For this implementation, the *2-opt* pairwise exchange technique was used along with the random restart Iterative Hill Climbing (IHC) algorithm (Russell and Norvig, 2009). This approach was selected as it would allow the resulting solution to be comparable with existing work (O'Neil *et al.*, 2011).

6.1.1 2-Opt Iterative Hill Climbing

The goal of the IHC algorithm is to identify the shortest local path, the local maximum, for a given starting path. This local maximum is identified by iteratively evaluating all nodes in the path and determining if the connections between two pairs of connected nodes cross on a 2D plane. Once identified, the pairs are recorded and the evaluation continues. Once all nodes have been evaluated, the pair with an overlapping connection

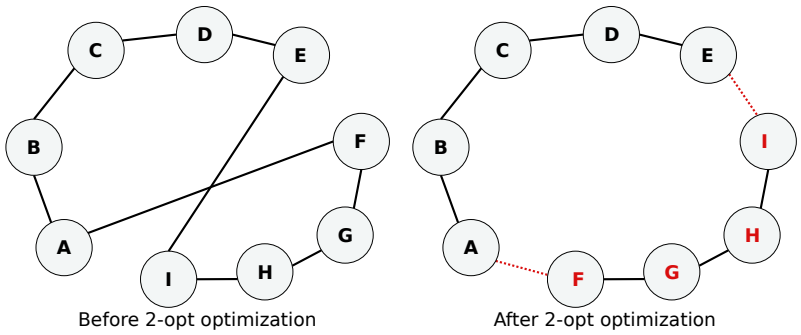


Figure 6.1: Application of the *2-opt* swapping optimization on a candidate path.

that exhibits the greatest length is optimised. The above process is then repeated on the revised path until no candidates can be found, implying a local maximum.

In this implementation the optimisation process used for IHC is the *2-opt* algorithm. This algorithm swaps the positions of two nodes within the path, one from each pair to efficiently 'untwist' the overlapping vertices as shown in Figure 6.1. The non-overlapping vertices between nodes are maintained and so the order between the swapped nodes must be reversed (Burtscher, 2014; Rocki and Suda, 2012).

6.1.2 NFP Implementations

Two variants of the proposed algorithm were implemented to run on the NFP with both implementations closely following the approach taken by O'Neil *et al.* (2011) in terms of how the candidate paths are identified. The approach taken in the first implementation (compute-based) represents the nodes in a manner that mirrors that of the implementations described by O'Neil *et al.* (2011) with each node being stored as a set of coordinates. The distances between nodes can be calculated using standard Cartesian coordinate geometry. The second approach (memory-based) explores whether pre-computing the distances results in an optimisation.

Compute-based Implementation: To interface with the NFP, a host component of the application was designed to handle the submission of work and collection of results. This component accepts a city file containing the coordinates of each node within the graph to be minimised. The data are input as an array to the NFP using the indirect messaging routine along with the number of nodes present in the file, and the number of restarts the application should attempt. After this, the host component can simply wait for the NFP to compute the result and return a sequence of coordinates representing the shortest path found.

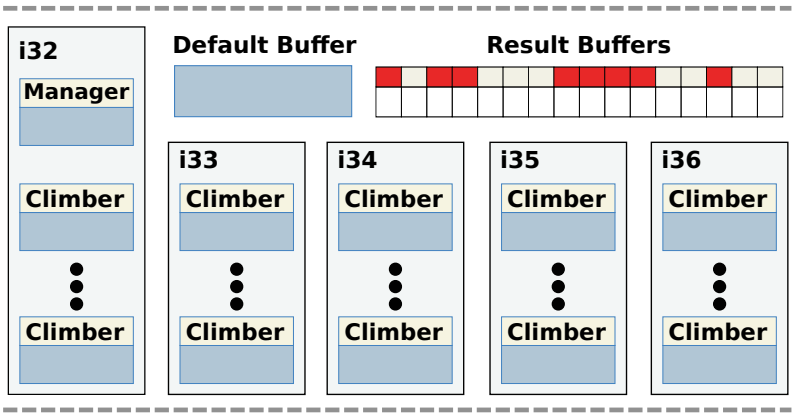


Figure 6.2: Application components of the NFP Travelling Salesman implementation.

The NFP component of the application consists of two process types, a manager and a climber. Given that the NFP device under test contains 60 microengines spanning the general processing islands, the current implementation contains a single manager process and 59 climber processes as shown in Figure 6.2.

In this implementation, the goal of the manager process is to receive a sequence of nodes from the host after which it can set up the *default buffer* and configure the *results buffer list* for processing. The *default buffer* holds the initial configuration of nodes as an array of coordinates which the climber processes can use as a starting reference. The *results buffer list* acts as an array where each climber thread can store a potential minimum cycle for review. Associated with each of these buffers is a flag indicating if the associated buffer is in use.

After the initial configuration is complete, the manager process signals all climber processes to begin executing through the use of a barrier. Once complete, the manager process then assumes the responsibility of collecting results from climber processes and recording the minimum cycle found. This is achieved through the use of a circular queue which is set up during the initial stages and shared with all climber processes. Whenever a climber process finds a candidate cycle, the cycle cost and index where it is stored in the *results buffer list* is inserted into this queue as a work element. The manager thread acts as a consumer of the queue and iterates through each work element, recording the lowest cost and the index of the sequence in the *results buffer list*. Indices in the *results buffer list* associated with all candidates aside from the minimal candidate are marked as free so they can be used to store future candidates. Once the required number of restarts has been completed, the manager process submits the minimum recorded cycle along with its length to the host process for display.

The climber process is responsible for performing the actual processing component of the application. Before attempting to find a candidate path, each climber instance first reads in the node coordinates from the *default buffer* into local memory. This greatly reduces the memory access latencies during processing as interactions with local memory do not involve events through the CPP bus. Once the node coordinates are locally stored, searching for the maximum for a given configuration of nodes can be performed as described in Algorithm 14. This algorithm continues to execute until the manager process announces that enough climb events have been processed.

```
nodeList ←ReadFromDefault ();
while Attempts still needed do
    nodeList ←ShuffleNodes ();
    reduction ← TryReduce (*nodeList);
    while reduction > 0 do
        reduction ← TryReduce (*nodeList);
    end
    bufferIndex ← FindFreeBuffer ();
    buffer [bufferIndex ] ← nodeList;
    tourCost ← TourCost (nodeList);
    SubmitToManager (tourCost,bufferIndex);
end
```

Algorithm 14: Pseudocode representing the Climber process.

Each iteration of the climb event begins by shuffling the list of nodes into a new random configuration from which the climb can begin. The configuration is then evaluated to determine if it is possible to reduce the current length of the path. The approach taken in this application is to identify candidate nodes for a *2-opt* optimisation, discussed in Section 6.1.1. The procedure followed is described in Algorithm 15 and is based on the approach taken by O'Neil *et al.* (2011) for both the CPU- and GPU-based solutions. This process of identifying candidate nodes and performing the *2-opt* optimisation is repeated until no single *2-opt* optimisation results in a shorter path. At this point the Climber process has found a local maximum which is a candidate for the shortest identifiable path.

The climber process then searches the *results buffer list* for a free buffer, using an atomic operation to claim it. The current node configuration is then written to the reserved buffer element for the manager process to access. The buffer index and the cost of the current path are then submitted to the manager process as a work element in the manager queue, while the Climber process can immediately continue attempting to discover a new candidate cycle. The fact that the recently discovered candidate is not registered by the manager before the climber begins a new iteration is not a concern as excess candidate

cycles can simply be ignored.

A limitation of Algorithm 15 which should be noted is that the maximum distances between nodes that can be accurately recorded is limited to 65,536. This limitation arises from the *Distance* routine used to calculate the Euclidean distance between two nodes being implemented without the use of floating point, divide, or *square-root* operations due to architecture limitations of the NFP-400.

```
input  : A reference to array nodelist
output: The size of the reduction found
best-reduction ← 0;
pre-i ← length(nodelist)−1;
for i ← 0 to length(nodelist)−2 do
    a ← nodelist[i];
    pre-a ← nodelist[pre-i ];
    pos-j ← i + 3;
    for j ← i + 2 to length(nodelist) do
        if pos-j == 100 then
            pos-j ← 99;
        end
        a ← nodelist[j];
        pos-b ← nodelist[pos-j ];
        pre-cost ← Distance(pre-a, a) + Distance(a,pos-b);
        pos-cost ← Distance(pre-a, a) + Distance(a,pos-b);
        reduction ← pre-cost − pos-cost;
        if best-reduction < reduction then
            best-reduction ← reduction;
            best-a ← a;
            best-b ← a;
        end
        pos-j ← pos-j +1;
    end
    pre-i ← i;
end
if best-reduction > 0 then
    TwoOptSwap (best-a,best-b);
end
return best-reduction;
```

Algorithm 15: The *TryReduce* algorithm used by the Climber process.

Memory-based Implementation: In an attempt to account for the distance limitation between nodes and to evaluate if pre-computing distances could result in a speedup, an alternative memory oriented approach was investigated. This implementation would also attempt to account for another shortcoming of the implementation, that is, the poor use of multiple contexts, due to Algorithm 15 not including any operations that would justify

releasing the active context on a single microengine. As a result, only a single context executes for the majority of the climb operation with context switches only occurring during the *FindFreeBuffer()* and *SubmitToManager()* events.

Although Algorithm 15 can easily be modified as a concurrent variation, such a variation does not result in improved performance unless executed on a parallel system as the memory access overhead is lower than the context switching overhead. Given that the contexts within a single microengine share the same ALU, it is suspected that executing the application on multiple contexts will result in a minimal improvement when compared to instances where a single context is used.

The approach taken for allocating and collecting work in this implementation operates in a very similar fashion to the compute-based implementation with the host component responsible for handling the submission and reception of work. A notable difference is that, in this implementation the host is initially used to generate a 2D array containing all distances between each pair of nodes in the dataset. Once computed, this array is submitted to the host in place of the set of node coordinates. As the NFP no longer works with the individual coordinates, it instead returns a list of indices indicating the order in which the original coordinates should be reordered.

The manager component of this variant contains one additional task compared to the compute approach. Once the dataset has been collected from the host component, it is duplicated into the CLS on all islands within the NFP that contain climber processes. By placing the lookup array in CLS, the latencies associated with read operations which are expected to occur often in this implementation can be reduced. CLS has been selected as the distance array exceeds the capacity of the local memory associated with each process. Considering the remaining memory types discussed in Section 2.1.1, CLS exhibits the lowest access latency. Once distributed, the manager process then operates in the same manner as the compute approach, waiting for candidate cycles to be added to the processing queue.

The climber processes follow the same sequence of events described in Algorithm 14 with the only difference reflected in how the `TryReduce()` routine identifies candidate nodes to swap. In this memory-based implementation, the `Distance()` functions are replaced with indexing operations into the 2D distance array stored in CLS.

By implementing the search component of the algorithm in this manner, the implementation can handle distances between nodes of up to $2^{32} - 1$. Furthermore, as operations involving the CLS memory region require the submission of a request over the CPP bus,

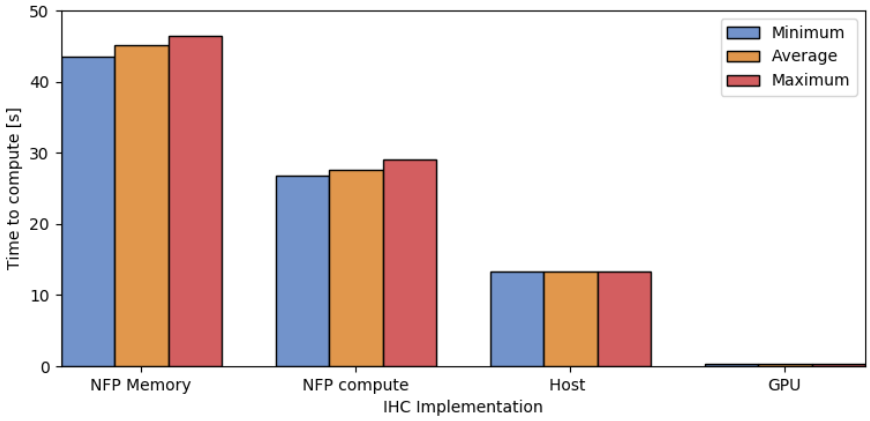


Figure 6.3: Time taken per implementation to process 10,000 random climbs.

the active context can release control of the ALU while the read operation is taking place, allowing other contexts within the microengine to execute. This allows the application to better utilise the available resources.

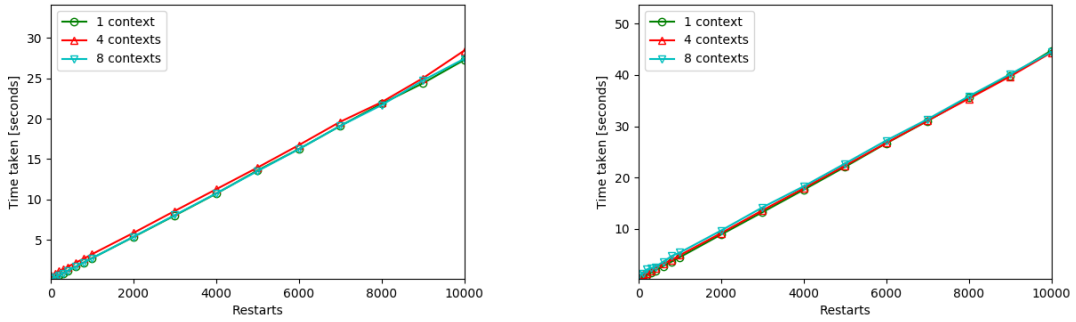
6.1.3 Application Testing

For testing the application, an existing dataset of 100 nodes was used[1] and both variations of the application were run for 10,000 random restarts. An optimized CPU version of the algorithm was also implemented and tested along with the GPU implementation provided by O'Neil *et al.* (2011). The results of these tests are depicted in Figure 6.3, which shows the average time taken by each application. From these results it is clear that the GPU implementation greatly outperforms all other implementations with both NFP implementations requiring the most time to compute. This comparatively poor result could be due to the relativity slow clock rate of the NFP compared to the CPU in conjunction with latencies associated with memory operations.

Comparing the two NFP implementations, it is clear that the memory-based implementation takes significantly longer to complete when compared to the compute-based approach. It is suspected that this is due to the memory latencies associated with fetching each precomputed weight for evaluation in the climbing algorithm. Though each microengine is capable of running up to eight contexts, designing the climbing algorithm to utilise this functionality does not necessarily result in improved performance. To better highlight this and help compare the two NFP implementations, both were tested using the same dataset of 100 nodes, but with iteratively increasing restart counts. This test was re-

[1] Accessible from: `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/`

peated 20 times to produce an average execution time per implementation. These tests were then repeated for a varying number of contexts available on each microengine per implementation with the results presented in Figure 6.4.



(a) NFP compute-based implementation for different restart counts

(b) NFP memory-based implementation for different restart counts

Figure 6.4: Average execution times for NFP implementations of the Travelling Salesman problem.

The first observation is that both graphs depict a distinctly linear curve, with the computation times for the memory-based implementation increasing more rapidly for increasing restart counts than the compute-based one. This implies that the overhead of memory transfer events exceeds the time taken to compute the distances between nodes. Furthermore, increasing the number of contexts does not allow the NFP to hide these transfer latencies as all tests of the memory-based implementation recorded similar performance times.

A final characteristic to consider is the impact that the number of microengines can have

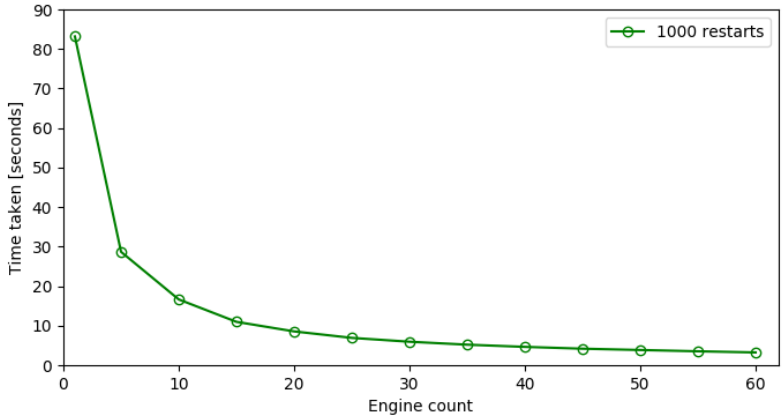


Figure 6.5: Time taken to process 1,000 random climbs vs. microengine count.

on performance. To test this, the compute-based implementation of the application was run on the same dataset of 100 nodes with 1,000 random restarts. This test was then repeated with a iteratively increasing number of microengines enabled to run the climber process (see Figure 6.5). From these tests it is clear that the number of microengines used impacts performance as expected. An interesting characteristic however, is the degrading improvement in performance for an increasingly large number of microengines. This is also expected as the overhead associated with memory transfer events increases with larger numbers of climber processes.

6.2 K-mismatch String Matching

Partial pattern matching is the second problem implemented to evaluate the capabilities of both the NFP and the proposed framework in the domain of general computing. To reduce the scope of the problem domain, the initial evaluation focused on the k-mismatch algorithm. The k-mismatch is a string matching algorithm that attempts to find occurrences of a pattern within a larger text that do not differ by more than some constant k (Levenshtein, 1966; Clifford *et al.*, 2016).

For this evaluation the following restrictions were observed. Firstly the application takes in two strings p and $n(n \geq p)$ where p is the pattern and n is the text that is searched for instances of p. The application also requires a constant $k(k \leq p)$. The goal of this application is to compute the Hamming distance between p and $n' = n_m...n_{m+i}$ where $m \geq 0$ and $i > 0$ (Amir *et al.*, 2004; Clifford *et al.*, 2016). The Hamming distance is the number of mismatches observed between p and n'. If the Hamming distance exceeds the threshold value k, then p and n' do not match. No preprocessing of the text n was performed to help identify candidate sub-strings, instead a naive approach was taken where each index in the text was evaluated. It is acknowledged that this is not necessarily the optimal approach; however, provided that all implementations compared conform to this approach, the results should be comparable.

To evaluate the applicability of the NFP for solving this problem, two solutions employing the k-mismatch algorithm were implemented. The first solution (called the worker pool implementation) attempts to count the number of times a given pattern occurs in the text by treating the contexts on the NFP as a pool of processors to which work can be subscribed. As the k-mismatch algorithm can run for varying lengths depending on the threshold value and the similarity between the pattern and the region of the text, the completion time of each test is not deterministic. By treating the contexts as a pool

of processors, work can be consumed on a first-come first-serve basis whereby contexts completing early can resubscribe and receive new work before longer running threads complete. Given that each set of eight contexts share an ALU, a pool approach may not be as effective as having the contexts execute in true parallel and so an alternative partitioning solution was also explored. This second solution (referred to as the static partitioning implementation) partitions the text being evaluated according to the number of contexts available, assigning each context to a region of the text to be tested. This approach bypasses the need to subscribe work to processors but includes other drawbacks such as not optimising process use for variable completion times.

Finally, for both the solutions described, an equivalent solution was designed to run exclusively on the CPU, acting as a benchmark to contextualise the performance of the NFP solutions. The performance characteristics tested are the time taken for a solution to complete for a given text size, and power in watts required to perform the computation.

6.2.1 Worker Pool Implementation

The worker pool implementation involves reserving two processors to arbitrate distribution and collection of data for a group of worker processors. Rather than consider the work processors individually, this solution constructs a queue of work elements that must be processed. The assignment of a work element to a processor is not a concern of the distributing process as it is only responsible for managing the queue and inserting work elements when space becomes available. All remaining contexts or threads can then be treated as a pool of consumers for the queue with work being consumed by the first available processor. Once a work element has been processed, the processor can rejoin the pool, waiting for more work.

The overall approach to processing taken by this variant of the solution, is described in Algorithm 17. This algorithm iterates through text n to produce multiple work elements such that $C(n') \leq C(p)$ where $C(x) = \sum_{i=1}^{n} \delta(x, s_i)$. As this is a naive approach, all potential sub-strings with $C(n') < k$ must be considered, resulting in $C(n) - (C(n) \bmod k)$ work elements to process. These elements are handed to a pool of workers which are responsible for performing the k-mismatch algorithm and announcing the position in the array, as well as Hamming distance if the match is achieved below the threshold k.

The actual k-mismatch algorithm, presented in Algorithm 16, has also been kept simple. For this implementation, the Hamming distance is computed by iterating through a sub-text n' and comparing each index against the equivalent position in the pattern p. If there is a mismatch, a counter initially set to the threshold k, is decremented. If the counter

```
input : Index, Threshold, Pattern,
        SubText
output: Hamming distance
l ←Index;
match ←Threshold;
p ←Pattern;
st ←SubText;
for i ← 0 to length(st) do
    if p [i] != st [i] then
        match −−;
    end
    if match ≤ 0 then
        return Threshold;
    end
end
return Threshold - match;
```

Algorithm 16: Basic k-mismatch.

```
input : Pattern, Text, Threshold, Worker
        Count
p ←Pattern;
t ←Text;
k ←Threshold;
setupWorkPool(Worker Count);
for i ← 0 to length(t)−length(t)%k
 do
    if (i+length(p)) > length(t) then
        addWork(i, k, p, t[i :]);
        ts ← t[i :];
    else
        ts ← t[i : i + p];
    end
    addWork(i,k,p,ts);
end
waitUntilPoolEmpty();
```

Algorithm 17: Dispatching work elements.

reaches 0, the threshold has been exceeded, implying that the pattern p and sub-text n' do not match. If the loop resolves, the Hamming distance is returned.

NFP implementation: For the NFP implementation, the application is divided into three processes, namely, the global dispatcher, the island dispatcher, and the pattern matcher (see Figure 6.6). The global dispatcher is responsible for receiving the pattern and text from the CPU and storing it in a globally accessible memory location. The global dispatcher then divides the text into sub-strings called windows, each of which can be up to 64 KB in size. Owing to how memory is handled on the NFP (see Section 2.1.1), the globally accessible memory region where the text is initially stored (either IMEM or EMEM depending on required capacity), has a large access latency. To reduce the memory access footprint, the text should be moved to regions local to the clusters or islands where the microengines reside. Each island contains 12 microengines and the tested NFP device contains five islands. Accessing island local memory such as CTM boasts a notably shorter access latency compared to global memory as discussed in Section 2.1.1, but only has a capacity of 64 KB, dictating the maximum window size. Each window constructed by the dispatcher process is added to a work queue along with the pattern to be tested against and the k-threshold. The goal of this processing step is to break the initial workload up into chunks that are small enough to fit in the island specific memory.

Associated with this work queue are the five island dispatch processes, one for each island.

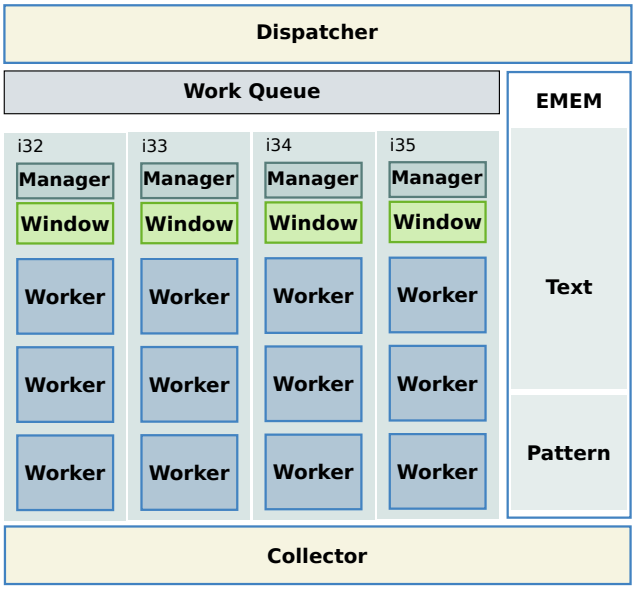


Figure 6.6: NFP k-mismatch process configuration.

These processes receive work from the queue and become responsible for evaluation of the associated window. Whenever work is consumed by an island dispatch process, that process begins by moving text entries from global memory to the CTM memory unit local to the current island. The volume of data copied matches the supplied window size plus the size of the pattern. Once complete, each process acts as an island specific dispatcher and dispatches each index within the window to a work queue, to which worker elements local to the island have subscribed. Once all work has been dispatched and processed for the given window, the island dispatch process resubscribes to the dispatcher work queue for a new window to process.

The worker elements constitute the application component that handles the processing of each work element added to an island specific queue. The approach here is that all available NFP contexts on the five general processing islands execute an instance of the worker. These processes subscribe to the work queue local to their respective islands on startup and sleep until work is assigned to them. When work is received, the worker fetches a substring of text up to the size of the pattern starting from the supplied index. Each worker then works through the pattern according to Algorithm 16. The result is submitted to the collector process after which the current worker resubscribes itself to the work queue for more work.

CPU implementation: The CPU implementation, written in Go, follows a very similar approach to that taken in the NFP implementation, except that data does not need to be migrated from its initially stored location to alternative memory regions before processing.

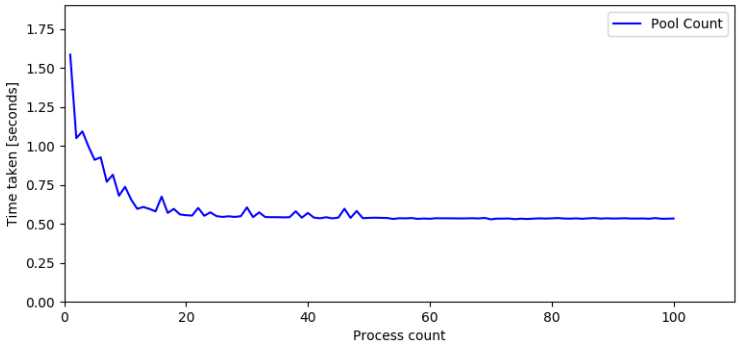


Figure 6.7: Average execution times of k-mismatch algorithm on the CPU for varying process counts and fixed text size.

Table 6.2: Power consumption (in watt seconds) of the k-mismatch solution for tests involving a text size of 100 MB.

No.	Test Count	Test	Average	Standard Deviation
1	12	CPU Idle	65.01	0.94
2	12	CPU Test	90.58	0.29
3	12	CPU+NFP Idle	76.85	0.14
4	12	CPU+NFP Test	79.90	0.96

The CPU implementation includes a dispatcher thread and a set of worker threads. The dispatcher thread is responsible for partitioning the text body into a set of work elements in the manner described in Algorithm 17. The workers operate in the same manner as the NFP implementation, executing Algorithm 16 to process each work unit. To determine the number of worker threads to spawn for the process pool, the execution time of the k-mismatch algorithm with a fixed k-threshold was tested for varying pool sizes. This test was run 20 times with very little variation observed. The average results of this test are presented in Figure 6.7 which indicates that beyond a pool size of 50 processes, very little performance variation was recorded for a fixed text size. The number of worker threads for the CPU implementation was set to 100 as the testing recorded very little fluctuation for this pool size.

Testing: To determine the power requirements of the CPU and NFP implementations, power characteristics associated with four tests using a text size of 100 MB were recorded (see Table 6.2). For power testing the CPU implementation, the NFP device was removed from the host system. The host was then powered on and current draw was monitored to determine power draw of the host while idle (No. 1). The CPU-only variant k-mismatch algorithm was then tested with a text size of 100 MB and power draw was monitored throughout the test (No. 2). The NFP device was inserted back into the host and the

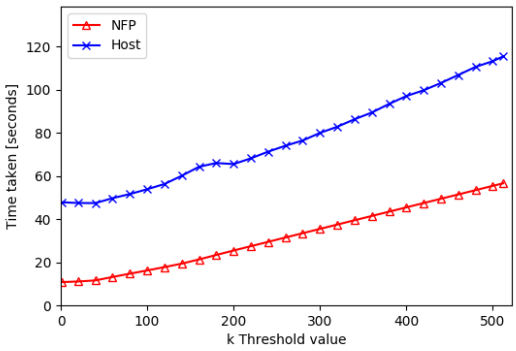


Figure 6.8: Bulk transfer from host to NFP.

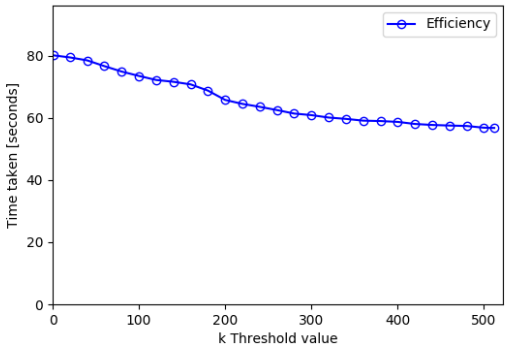


Figure 6.9: Bulk transfer from NFP to host.

power draw was monitored while the system remained idle (No. 3). The NFP variant of k-mismatch algorithm was then tested with the same text size (No. 4).

Very little fluctuation in power consumption was recorded with the STD of all tests being under one watt. Considering the idle power draw of both solutions, it can be seen that the presence of the NFP introduces an average increase in consumption of 11.84 watts. During testing however, the NFP solution requires an average of 79.90 watts, indicating an increase of 3.05 watts while the CPU-only implementation depicts an average increase of 25.57 watts when compared to idle.

In addition to recording the observed power consumption while performing the testing, the time taken to complete the k-mismatch problem for iteratively increasing k-threshold values was also recorded. Figure 6.8 presents these results by plotting the performance curves of both architectures for the given text and pattern configuration. For these results it is clear that the NFP implementation exhibited a shorter execution time for all test iterations. Approximating the curves to a linear plot, the gradient of the NFP is greater than that of the CPU, implying that the execution time of the algorithm on the CPU is less heavily impacted by the k-threshold value when compared with that of the NFP.

An alternative view of these results is to consider the power consumption of the two solutions. Taking the power consumed during testing on each architecture and multiplying it by the time taken to complete each iterative test results in the total work performed per solution. Using the work done per architecture per test iteration, the efficiency of the NFP relative to the CPU can be determined by looking at the difference in work performed per test iteration. The resulting efficiency curve is presented in Figure 6.9 which better describes how the k-threshold impacts the performance of each implementation. These results indicate that the NFP implementation starts out being 80% more efficient that the

CPU for the given pattern and text configuration. As the k-threshold increases however, the relative efficiency of the NFP begins to decline, reaching 56% more efficient once the k-threshold is at 512 KB. This implies that a larger pattern size could eventually lead to the CPU implementation matching the NFP implementation in terms of efficiency and potentially becoming more efficient.

6.2.2 Static Partitioning Implementation

As an alternative solution, the static partitioning approach allocates each context a set amount of work dependent on the number of contexts available and the size of the text to be searched. Whereas the focus of testing in the previous implementation was on the efficiency of the implementations obtained by recording the power consumption during testing, for this solution it lies in evaluating the impact that the different memory types available on the NFP have on the performance. To test this, CPU and NFP implementations of the static partitioning solution were developed.

NFP implementation: As with the worker pool solution, this solution accepts the pattern and text from the CPU and stores it in a globally accessible memory location. The text is then iterated through, comparing each iteration with the supplied pattern to determine if the difference between the pattern and the region of the text differs by less than the supplied k-threshold. As this approach does not use a pool of processes, the implementation does not include a dispatcher but instead comprises a collection of workers. To help manage the collection of memory and the submission of processing results, a single coordination process is added. This process is responsible for collecting the pattern and text data from the host and signalling the worker processors once the initial transfer operation is complete.

All remaining contexts available on the NFP are used as workers, each executing the same k-mismatch algorithm with the text and pattern made globally accessible. To allow for partitioning, each thread is able to identify its relative index in a manner similar to the `threadIdx` and `blockIdx` seen in languages such as CUDA. These functions are `__ctx()` for context number and `__meid()` for both microengine and island numbers. This relative index is used to mark which region of the text the current context is responsible for. As highlighted in Section 6.2.1, the location of memory relative to the processing element can have a notable impact on performance. For the previous solution this was accounted for by having a set of dispatchers that would migrate the pattern and text to each island, positioning it closer to the relevant worker. In this solution, the lack of a dedicated dispatcher processes results in this operation being the responsibility of the

workers themselves.

Initially both the text and pattern were stored in either IMEM or EMEM as these regions boast the necessary capacity for storing the text body. By assigning each worker a continuous segment of text to process rather than utilising striding techniques, it is still possible to have memory migrated closer to the worker elements for processing. To achieve this, the island local memory types CLS and CTM can be partitioned into windows or buffers, one for each processing thread on that island. For CTM, after reserving memory for the operation of the solution, the remaining capacity available for the text is 250 KB. Given that each island contains 96 workers, each processor can be allocated up to 2.6 KB of buffer capacity. For CLS, the remaining available capacity is 50 KB, effectively limiting the CLS buffer per context to 533 bytes. By introducing buffers, each worker can move a subset of their respective window partition into a memory type that has a lower access latency for processing. This however obviously introduces a data transfer step to all worker elements which could impact performance negativity. As an alternative, the use of buffers can be bypassed entirely with the data stored in IMEM or EMEM which can be interacted with directly by the workers. This approach should result in longer execution times when compared to the buffered approach due to increased memory access latency, but does not require the additional data transfer events.

CPU implementation: To allow the two architectures to be comparable, the CPU implementation of the above solution follows the same format as the NFP implementation. The application is written in Go and designed to iteratively search through the supplied text for a given pattern. The text to be searched has been statically partitioned into four chunks, each of which is handled by a separate thread. Testing the application using four threads yielded the best performance as it maps directly to the number of hardware threads advertised by the host CPU. Testing configurations utilising two and three threads resulted in longer processing times and monitoring CPU usage during these tests indicate that some threads were idling. For the two thread configuration, the number of idling threads was two; while the three thread implementation recorded one idling thread. As the host CPU supports four hardware threads these results are as expected. Increasing the number of threads beyond four did not produce any improvement to performance which is again as expected given that the number of software threads now exceeds the available hardware threads resulting in over-subscription.

6.2.3 Test Suite Design

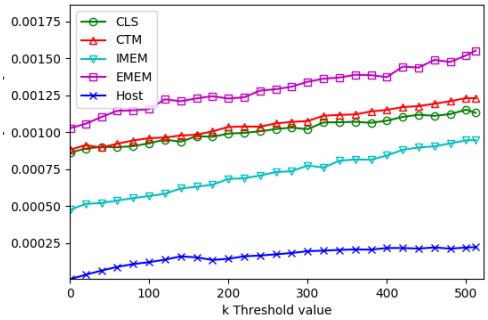
To quantify the impact that memory can have on performance as well as compare the CPU and NFP implementations for various text sizes, a test suite was designed to test three variables, namely, the k-threshold, the memory region utilised on the NFP, and the text size. To see the impact the k-threshold value can have on both the NFP and CPU implementations, each test involved multiple iterations of the k-mismatch algorithm while using the same text, pattern, and memory region but with the k-threshold value iteratively increasing. To quantify the impact different memory regions could have on performance, testing of the NFP implementations was broken into four separate tests, one for each of the following memory regions: CLS, CTM, IMEM, and EMEM. By keeping the text and pattern fixed, these tests allow the performance of each memory region to be compared. The pattern size used for all iterations was fixed at 512 bytes so that the smallest described buffer could store enough text to perform a k-mismatch evaluation.

A characteristic of the different memory regions which must be considered during testing however is the relative capacity of IMEM. Both EMEM and IMEM are global memory regions with only one instance of each present on the NFP device under test. For IMEM, the capacity available for storing text data is 3 MB, thus for text sizes smaller than this, IMEM can hold the entire text from which the worker elements can operate. For text sizes greater than 3 MB, the capacity of IMEM is no longer suitable and must be subdivided into buffers for each of the 480 worker elements, resulting in a buffer size of 6.4 KB per context but also requiring an access latency of 150 to 200 cycles (see Table 2.1). To compare the impact that different text sizes can have on the different memory regions, each of the above tests was repeated for five different memory sizes: 512 bytes, 2 KB, 1 MB, 10 MB, and 100 MB.

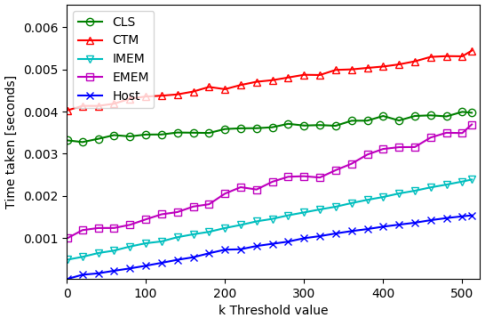
As an additional performance metric, the efficiency of each solution in terms of power consumption was explored. Although not particularly relevant to small scale applications such as the solutions described here, power consumption in large systems is a important factor to consider. By evaluating the power costs associated with the provided solutions, some basic observations regarding energy requirements associated with the NFP during processing can be made.

6.2.4 Results

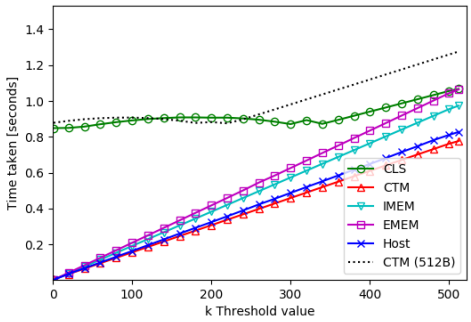
The results of the tests have been classified by text size and are presented in Figure 6.10 (a)-(e). Each figure presents the time taken to iteratively search the text body for a pattern using the k-mismatch algorithm and a given k-threshold. The results for the NFP



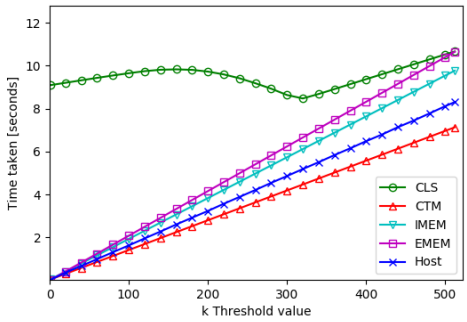
(a) Execution times using a text size of 512 bytes



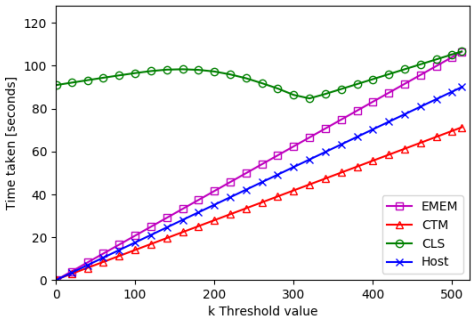
(b) Execution times using a text size of 2 KB



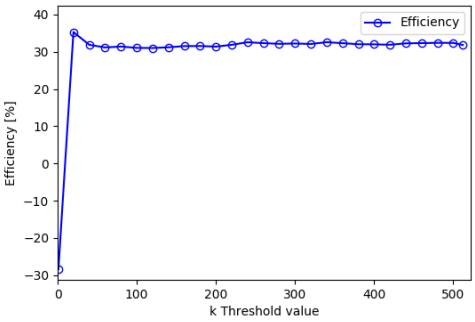
(c) Execution times using a text size of 1 MB



(d) Execution times using a text size of 10 MB



(e) Execution times using a text size of 100 MB



(f) Efficiency of NFP implementation relative to CPU implementation

Figure 6.10: Average results for static partitioning tests using varying text sizes.

solutions are additionally presented according to the memory region used to perform the test. Each iteration of each test was performed 50 times with the average being recorded in the respective sub-figure.

Memory performance: For tests utilising a text size exceeding 2 KB in size, almost no variance in the results was observed except for tests involving EMEM or the CPU. Although care was taken to limit the number of threads executing on the CPU, the

variance in the host is expected to be due to the inclusion of some core routines forming part of the Linux kernel. The variance in EMEM is largely due to the PCIe operations required to fetch and submit data stored in the host memory. For tests utilising a text size of 512 bytes and 2 KB, the increased variance is assumed to be due to the shorter completion times. This allows non-deterministic elements such as communication and memory access events to contribute more to the execution time than tests involving larger text sizes.

When the pattern size is comparably small such as in Figure 6.10(a), it is immediately clear that the CPU implementation outperforms the NFP in terms of computation time, being on average 4.2× faster than the fastest NFP implementation. Another observation for this graph is that the NFP solution utilising only EMEM for the storage of the text required the longest time to complete. These observations correspond with the initial assumptions based on the access latencies associated with the EMEM memory region. An unexpected observation is the performance of the NFP solution using IMEM. As the memory size of the text is small enough to reside fully in IMEM, no windows or buffers are required to process the tests and thus this implementation works in exactly the same manner as the EMEM test with the difference in computation times being solely due to the reduced access latency. As both CTM and CLS tests require that the worker processors copy the text from EMEM or IMEM into a buffer local to their respective islands, these tests did not perform as well as the IMEM solution for the given text size.

In Figure 6.10(b), the text size has been increased from 512 bytes to 2 KB. The significance of this test is that the NFP solution using the CLS implementation is no longer able to store the entire text body in a worker buffer and thus the buffer must be refilled four times during execution. This CLS implementation required more time to compute than both the IMEM and EMEM implementations; however, the rate at which the computation time increases relative to the k-threshold values is notably lower than any other implementation. This implies that once the buffers are filled, the time to compute the k-mismatch algorithm is faster than other implemented solutions due to the lower access latencies associated with CLS.

An observation of particular interest in this graph is that the NFP solution using CTM performs slower than any other solution for the given text size. As the CTM solution supports a buffer size of 2.6 KB per context, the buffer is only filled once at the start of each worker. It was initially suspected that the increased latency may be due to the larger transfer process not allowing the workers operating on a single microengine to better interleave data transfer and computation. To test this, the CTM buffer size was reduced to

512 bytes, equal to the CLS buffer, and the tests were repeated with the results included in Figure 6.10(b). From this test, it is immediately clear that for text sizes of 2 KB, breaking the CTM into smaller buffers improves performance. An interesting observation is that the time taken to fill the CTM buffers is less overall than the time taken to fill the CLS buffer; however, the time taken to compute iteratively greater k-threshold values increases at a higher rate than for the CLS implementation. This observation coincides with the greater access latencies associated with CTM when compared to CLS.

Increasing the text size to 1 MB as presented in Figure 6.10(c) results in very different performance characteristics when compared with the previous two tests. It is apparent that the NFP solutions using buffer sizes of 512 bytes are no longer efficient. Both the CLS and CTM solutions now include a very large offset in execution times when compared to the other solutions. As with the previous test, it is noted that the CLS solution grows at a slower rate than all the other tested implementations, but due to the large offset, this only becomes a benefit when the k-threshold matches the pattern size. In addition to the large offset in execution times, the CTM solution supporting a 512-byte buffer size also has the disadvantage of longer access latencies than the CLS solution, making it the worst performing solution for the given text size. Considering the other solutions however, it is observed that the host implementation is no longer the best performing solution as the CTM implementation using the 2.6 KB buffer size records the lowest execution times for all k-threshold values tested.

When considering the tests using a text size of 10 MB and 100 MB as presented in Figure 6.10(d) and 6.10(e), respectively, the same pattern as presented in Figure 6.10(c) is observed. For these two tests, the CTM solution using a 512-byte buffer has been omitted as it provides no meaningful information. The CLS implementation has been retained to show that it has the same performance as the EMEM solution once the k-threshold equals the pattern size.

To compare the results of the static partitioning solution with those of the process pool implementation, the power consumption for the tests involving a text size of 100 MB was recorded.

Power consumption: To determine power requirements, the sequence of tests described in Section 6.2.4 were repeated. The observed results mirrored those previously recorded in Table 6.2, with very little fluctuation in power consumption. Using the power consumed by the two implementations during the k-mismatch test with a text size of 100 MB, the relative efficiency of the NFP when compared to the CPU implementation is approximately 32.2%, as indicated in Figure 6.10(f).

Summary: Reflecting on the results recorded for the k-mismatch implementations, the most notable observations are as follows. Firstly, for pattern sizes of 1 MB and greater, an NFP-based solution using a CTM buffer can outperform a CPU implementation. For pattern sizes less than 1 MB, the host is the better candidate for this implementation of the k-mismatch algorithm. Secondly, the use of CLS memory for computation of the k-mismatch would be the optimal solution as the lower access latency results in a more gradual increase in computation time for larger k-thresholds. This however, would only be feasible if the capacity of CLS were larger as the overhead of transferring text data to the CLS buffers clearly exceeds the computation times. As the host outperforms the NFP for small text sizes, the current capacity of the CLS does not provide a good solution to any of the tested text sizes. When considering power consumption, the NFP is shown to use less power than a CPU-only implementation, however this is only true while the the application is being executed. When idle, inclusion of the NFP device does increase overall power consumption which is an important factor to consider if such a device were to be used for general processing.

6.2.5 K-difference Extension

As an extension to the k-mismatch algorithm, the k-difference algorithm is explored on the NFP. Both the k-mismatch and k-difference algorithms are used to find all occurrences of a given pattern within a larger text. As previously discussed, the k-mismatch algorithm achieves this by iterating through the text and attempting to compute the Hamming distance associated with a given pattern and each location in the text. The Hamming distance is simply the number of characters by which the pattern and current subregion of the text differ. If the calculated difference is below some threshold value k, the subregion of the text is considered a match for the given pattern.

The k-difference algorithm also operates by iterating through the text; however, it attempts to compute the Levenshtein distance (originally proposed by Levenshtein (1965)) between the pattern and each location in the text. While the Hamming distance records the number of symbol differences between the pattern and a subregion of the text, Navarro (2001) notes that the Levenshtein distance records the minimum number of edits that would need to be performed to a subregion of the text for it to be an exact match of the supplied pattern. Each edit can be one of three possible operations, namely, a substitution, an operation, or a deletion. Substitution is the only operation considered when calculating the Hamming distance as it simply involves replacing a symbol in a subregion of the text with the symbol residing at the same position in the pattern. The insertion and deletion operations indicate the inclusion of an additional symbol or removal of the

current symbol from the region of the text, respectively.

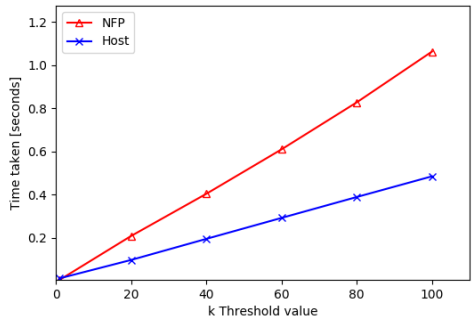
```
input  : Index, Threshold, Pattern, SubText
output: Levenshtein distance
p ←Pattern;
st ←SubText;
for i ← 0 to length(st) do
    buffer [i] = i;
end
for i ← 0 to length(st) do
    prev_iteration ← buffer [0];
    buffer [0]++;
    for j ← 0 to length(st) do
        min_change ← min(buffer [i + 1]+1, buffer [i]+1);
        if p [i] != st [i] then
            min_change ← min(buffer [i + 1], buffer [i]+1);
        else
            min_change ← min(buffer [i + 1], buffer [i]);
        end
        prev_iteration ← buffer [j];
        if min_change > Threshold then
            return 0;
        end
        buffer [j] ← min_change;
    end
end
return Threshold - buffer [length(st)−1];
```

Algorithm 18: Basic k-difference algorithm.

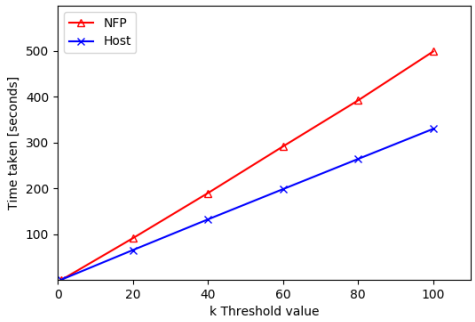
There are three main approaches to computing the Levenshtein distance. The first is a recursive approach, however given that the NFP does not support recursion, an implementation using this approach was not attempted. The second approach is to construct a matrix where every row maps each symbol in the pattern, and the columns represent each symbol in a sub-region of the text. This matrix, once computed, holds all Levenshtein distances between all sub-regions of the text and sub-patterns with the Levenshtein distance between the pattern and a region of the text being the value stored in the last cell of the matrix. The initialisation time for this algorithm is $O(n)$ where n is the size of the pattern, while the processing time of this approach is $O(mn)$ where m is the size of the region of the text being compared against. The space usage is also $O(mn)$ to accommodate the matrix.

The third approach also has a processing time of $O(mn)$ where m is the size of the text and n is the size of the pattern, however this approach has a space usage of $O(m+n)$. The

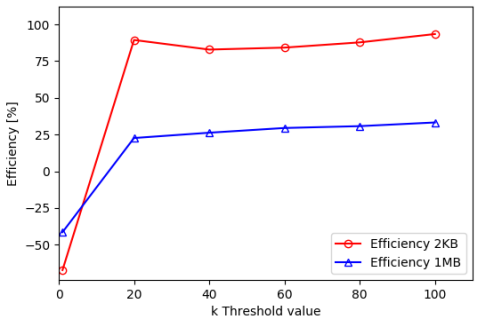
idea behind this approach is the same as that where a matrix for all Levenshtein distances is calculated, but in this case only the most recent column and row are recorded. Given that this approach has the same time complexity as the full matrix approach but with a reduced space complexity, it was selected as the candidate for implementation (see Algorithm 18). To test this solution, the k-mismatch implementation using static partitioning was extended to calculate the Levenshtein distance instead of the Hamming distance. For comparative purposes, the CPU implementation was also modified accordingly by replacing the evaluation function with the function described in Algorithm 18.



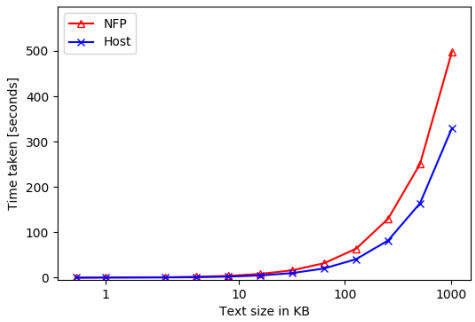
(a) Execution times for a text size of 2 KB



(b) Execution times for a text size of 1 MB



(c) Power efficiency of the CPU relative to the NPU for the recorded tests



(d) Execution times for a fixed k-threshold and varying text sizes

Figure 6.11: Average results for k-difference tests using varying text sizes.

Testing: For the evaluation of the k-difference solution, the goal is to compare the relative performance capabilities of the implementation to the previously described k-mismatch solution using static partitioning. Due to the increased complexity of the application and the lack of optimisations, only two variants of the test suite discussed in Section 6.2.2 were used. In the first variant the text size was set to 2 KB and the pattern size to 100 bytes. The second variant increased the text size to 1 MB but kept the pattern size at 100 bytes. As the goal of this test was to provide a comparable metric for the CPU and NFP implementations, the same testing suite was used for both CPU and NFP

implementations. As the overall efficiency of the NFP compared to the CPU is of interest, the power consumption during testing was also recorded.

To perform the tests on the CPU implementation, the same test sequence followed for both variants of the k-mismatch algorithm was repeated but only for text sizes of 2 KB and 1 MB (see Figure 6.11(a) and (b)). In terms of power consumption, the results recorded no variance when compared against previous power evaluations with the same metrics presented in Table 6.2 being observed (see Figure 6.11(c)). As an additional test, the k-difference algorithm was run multiple times on both architectures with a fixed k-threshold value of 100. The text size was then iteratively increased from 512 bytes up to 1 MB (see Figure 6.11(d)).

From these results it is clear that changing the algorithm from an O(n) complexity to an O(nm) causes the execution time to increase greatly for increasing values of k. This is better highlighted when comparing the recorded results in Figure 6.10(c) and Figure 6.11(b), particularly for values of 40 and 100 for the k-threshold. In Figure 6.10(c), if just the NFP implementation using a CTM buffer is considered, the execution time with a k-threshold of 40 is 0.065 seconds, and 0.156 s with a k-threshold of 100. The equivalent results presented in Figure 6.11(b) record an execution time of 189.97 s and 498.21 s for the respective thresholds. Using these figures, the execution time for the k-mismatch implementation can be approximated to 0.0015 s per k-threshold increment while the k-difference implementation can be approximated to 5.14 s per k-threshold increment. For the host implementation, the equivalent k-mismatch implementation can be approximated to 0.0016 s per k-threshold increment while the k-difference implementation can be approximated to 3.28 s per k-threshold increment. Although computation times of both implementations of the k-difference algorithm increase at a significantly greater rate when compared to the k-mismatch implementations, the CPU variant clearly outperforms the NFP one.

This performance comparison is better illustrated in Figure 6.11(c) which depicts the efficiency of the CPU compared with the NPU for the two test variants. In both cases the efficiency clearly favours the CPU due to differences in execution times. For the test involving a text size of 2 KB, the CPU is 93% more efficient that the NFP with a k-threshold of 100. For the 1 MB test, the CPU is 33% more efficient using the same threshold.

As a final comparison, the performance of the two k-difference implementations for varying text sizes is presented in Figure 6.11(d). Each iteration of the test doubles the text size and as such, a logarithmic graph has been used to represent the results. These results

show how the two implementations react to increased text sizes with both exhibiting a linear relationship between text size and execution time of the k-difference algorithm.

6.3 PCSA

As the first networking related problem, Probabilistic Counting with Stochastic Averaging (PCSA) was designed and implemented on the NFP. PCSA has been selected as it provides an example of an application better suited to the heterogeneous CPU-NFP system with a heavier emphasis on data streaming instead of batched computation. PCSA is a sketching algorithm designed to estimate the cardinality of a stream of data while using a fixed amount of memory. It has subsequently been replaced by hyperloglog (Flajolet *et al.*) which uses less memory to provide the same level of accuracy. PCSA however was selected over hyperloglog as it can more easily be implemented on the NFP without logarithmic or division operators.

6.3.1 Implementation

PCSA was proposed by Flajolet and Martin (1985) as a heuristic approach to determining the cardinality of a data stream using fixed memory. The probabilistic counting component of the implementation uses a sketch which is a single 32- or 64-bit number where each element to be stored in the sketch is first hashed to produce a number of the appropriate bit width. To update the sketch with a received value, the value is hashed and the number of trailing 'ones' present in the binary representation of the hash are counted to produce the value $r(x)$ which is used to compute the $R(x)$ value where $R(x) = 2^{r(x)}$. The resulting $R(x)$ value is subsequently *or*'d with the sketch, completing the update operation. The cardinality estimate can then be produced by: $R(sketch)/0.77351$ where 0.77351 is referred to as the unbiased statistical estimator.

To improve the accuracy of the algorithm, stochastic averaging is also used by dividing the input stream into M sub-streams and processing these independently. Each sub-stream generates a separate sketch and the cardinality estimate C can be computed using Equations (6.1) and (6.2). The relative accuracy of the PCSA algorithm is approximately $0.78/\sqrt{M}$.

$$Mean = \frac{\sum_{i=1}^{M} r(sketch_i)}{M} \tag{6.1}$$

$$C = \frac{2^{Mean}}{0.77531} \tag{6.2}$$

A PSCA algorithm containing 2048 sketches was implemented on the NFP. This would allow the implementation to estimate the cardinality of IP addresses with an average error of 1.72%. The hashing function used in this implementation is a standard cyclic redundancy check (CRC) which produces 32-bit hash values for the update operation. To account for multiple sketches, the nine most significant bits of the hash are used to index which sketch will be updated to account for the current hash. Provided the hashed values are uniformly distributed, this approach for both the probabilistic counting and stochastic averaging components of the algorithm can use the same hash.

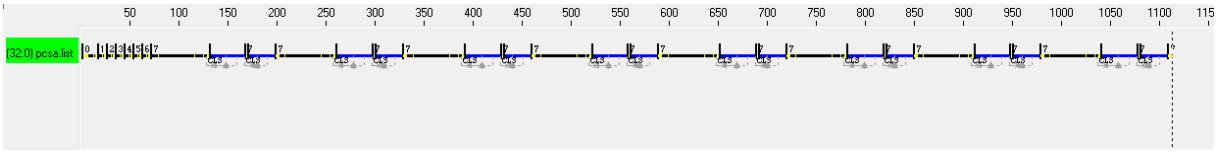
As the PSCA operation does not require the same element to be hashed into multiple rows, both packet handling and updating can occur on the same context. Furthermore, as the order in which elements are received and the sketches updated does not matter, the application can easily be scaled.

To acquire the estimate from the PCSA, a host handler process was implemented which would simply wait for a request from the host before responding. As the NFP does not support the division operator, the calculation described in Equation (6.2) is resolved on the host. To facilitate this, the host handler process sends the sum of all trailing 'ones' in the sketches and the size of the sketch to the host upon request.

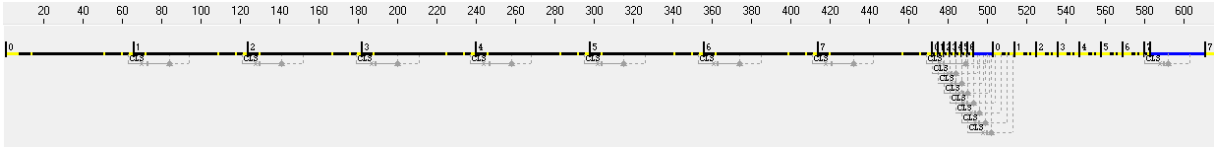
6.3.2 Testing

To test the implementation, a pre-recorded network capture containing 1.7 GB of traffic in 1,698,526 packets was replayed through the NFP. Once complete, the details required to estimate the cardinality were requested from the NFP. To provide a comparison, a dictionary based solution was implemented on the host to record the number of unique IPs seen. The same network capture was input to the host based solution to produce the true cardinality for the stream. Comparing the results of the host-based solution and the cardinality estimate by the NFP, it was found that the error in the estimate was 1.02%.

Using the NFP simulator, the performance of PCSA on the NFP was further evaluated and it was found that a single update operation took 124 cycles. The operation was then partitioned into time spent on computation and time spent sleeping with the active context spending 75 cycles, just over 60% of the time sleeping. Only 49 cycles were required for computation of which 39 were used in the calculation of the CRC hash. This wait time is due to updating the PCSA sketch which required two memory operations involving CLS,



(a) Eight update operations using one context.



(b) Eight update operations using eight contexts

Figure 6.12: Execution history of PCSA update operations for different context counts.

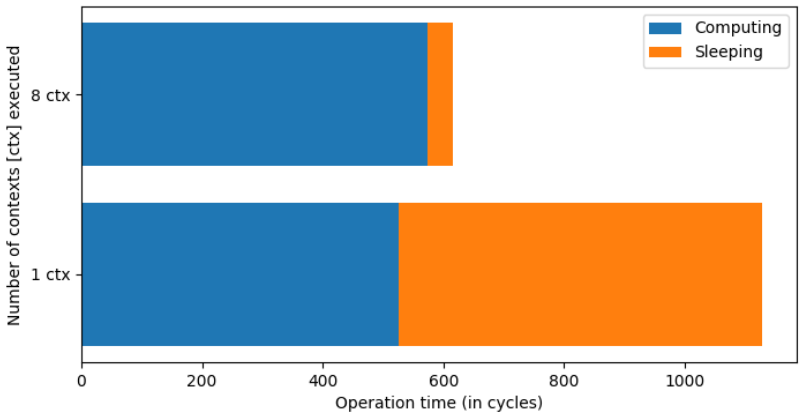


Figure 6.13: Comparison of time taken to execute eight PCSA update operations on a single microengine.

each of which incurs an access latency of 25 - 50 cycles. For the architecture used in this investigation, the clock speed of a single microengine is 600 MHz, thus each cycle requires 1.6 ns to execute.

For comparative purposes, eight update operations were performed on a single microengine limited to executing one context. The execution history is presented in Figure 6.12(a). The total time to compute this test was 1129 cycles, of which 526 were used for computation as indicated in Figure 6.13. The total number of cycles does not exactly match the 124×8 predicted count as the test required additional cycles during the startup and completion states.

The microengine was then extended to use all eight available contexts and the same test was performed with the results also recorded in Figure 6.13. The computation component of this test required 48 cycles more than the single context implementation; however, it only

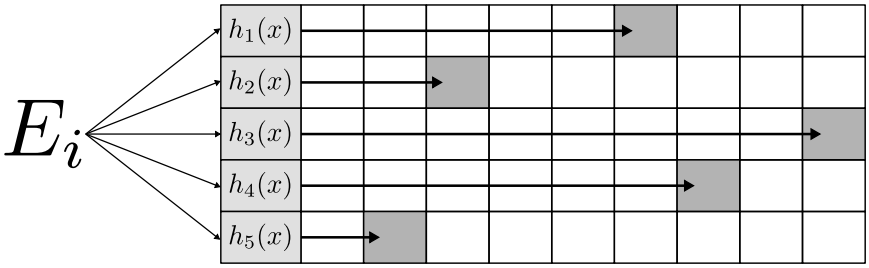


Figure 6.14: Basic update operation of the *count-min* sketch.

spent 42 cycles sleeping, resulting in the test only taking 616 cycles, or 45% fewer cycles. This reduction is due to the multiple contexts effectively hiding the access latencies as can be seen in Figure 6.12(b), with only a few extra cycles needed for the context switching overhead. In terms of the overall application, these results imply an average execution time of 77 cycles or 128 ns per update operation on a single microengine. This low latency, coupled with the ability to deploy the application on 60 microengines in parallel, allows the NFP-400 to maintain this PCSA implementation at wire speeds of up to 10 Gb/s.

6.4 Count-min sketch

Following the PCSA solution, the viability of deploying alternative sketch based applications on the NFP and CPU was explored. The *count-min* sketch (Cormode and Muthukrishnan, 2005) is an evolution of the Bloom filter that was originally designed to solve the membership problem associated with a stream of data. Rather than determining whether an element has been seen, the *count-min* sketch is a data structure that can be used to estimate the frequency of occurrence for elements within the stream.

The *count-min* sketch comprises a set of d rows with each row being divided into a series of k buckets. Associated with a row is a hashing function that can resolve an input element to a bucket within that row. The theory of operation for the *count-min* sketch is that each element received is hashed, and the bucket corresponding to the hash is incremented. To account for the cardinality of the input stream being greater than the number of buckets in a given row, multiple rows are implemented, resulting in the 2D array structure of the sketch. The idea behind having multiple rows is that, while each row boasts the same number of buckets, the hashing functions used on each row are pairwise independent. Each element to be recorded by the *count-min* sketch is thus hashed by each of the row specific functions and the corresponding bucket of the given row is incremented as depicted in Figure 6.14. This results in a single element being hashed to a different bucket for each row.

At any point during the execution of the *count-min* sketch, it can be queried to get an estimate of the frequency of occurrence of an element within the stream. To get an estimate for a given element, the element is first hashed by each of the row specific hashing functions and the value in the corresponding bucket is returned. The guessed frequency is the minimum of the returned values. The update time complexity of the *count-min* sketch is $O(d)$ where d is the number of rows or depth of the sketch while the space complexity is $O(dk)$. Although the *count-min* sketch is a probabilistic data structure, it is possible to determine the error and confidence of estimates based on the dimensions of the sketch. The space required by the sketch can be determined by the formula $\frac{e}{\epsilon} ln\frac{1}{\delta}$ where $\frac{e}{\epsilon}$ indicates the number of buckets per row, and $ln\frac{1}{\delta}$ represents the number of rows required. Here, ϵ represents the error in our estimation while δ indicates the uncertainty that the error will be adhered to.

Implementation

For this implementation, the goal is to use a *count-min* sketch to record the top 50 IP addresses seen in a network stream passing through the NFP. To achieve this, the application needs to both build a sketch and maintain a heap containing the top 50 IP addresses. For the sketch, the desired uncertainty is 0.1 % and error is 1%. Using Equations (6.3) and (6.4), the desired width of the sketch is 272 and the depth is 7.

$$width = \frac{e}{\epsilon} \tag{6.3}$$

$$depth = ln\frac{1}{\delta} \tag{6.4}$$

In order to maintain a heap of the top 50 IP addresses recorded, every update operation to the sketch also needs to be followed by a query operation. The proposed implementation is to allocate responsibility for each row of the sketch to a row handler process operating on a separate microengine on the NFP. These microengines receive an IP address and a frequency as input from an upstream microengine. The row handler is responsible for updating its row with the received IP address and also recording the current value of the bucket to which the received IP address was hashed. The row handler can then forward the IP address and the smaller of the input count value and queried count value to the next microengine, forming a chain. Once the last microengine executing a row handler process has completed the update and query operation, the resulting count value can be considered the sketch estimate and can be checked against a heap of top addresses.

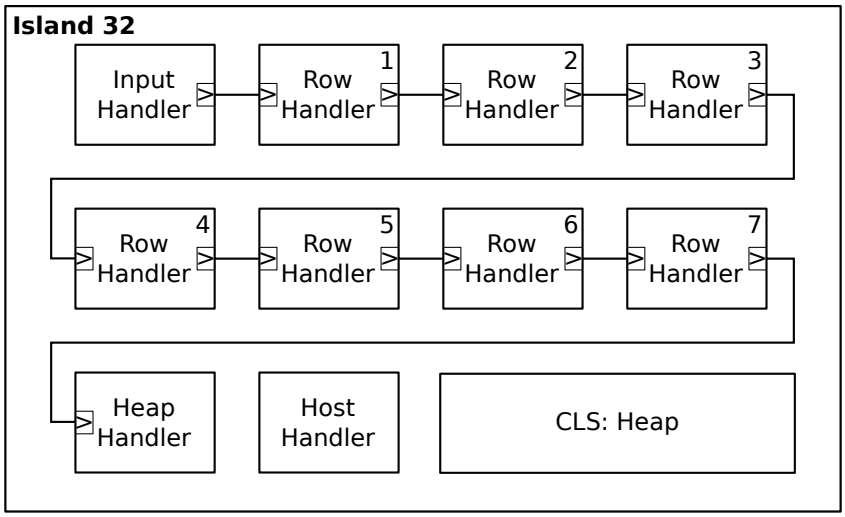


Figure 6.15: Connection layout for *count-min* handlers on a single island.

The last microengine in the chain is the heap handler which, like the row handler engines, receives an IP address and count value as input. This count value is the sketch estimate for the given IP. The heap handler can iterate through the heap, checking each element to see if the IP is already in the heap, and recording the location of the smallest element recorded. If the IP in question is not found in the heap, the smallest record in the heap is compared against the received count value. If the new value is larger, the smallest element in the heap is replaced with the received IP and count.

To handle the reception of traffic, an input handler process is implemented on the first microengine in the chain. This process simply waits for a packet to handle. Once received, the reader process extracts the source IP address and pushes it to the first row handler process in the chain along with a count of 0xFFFFFFFF. Once submitted, the input handler can hand the packet to the outbound channel to continue propagation. This application was designed to be scalable, either by increasing the number of row handler processes or by duplicating the entire layout across multiple islands. Communication between the engines is implemented using the *NFComms* framework which supports intra-architecture communication. By using this framework, the physical location of the engines is less important and a single chain could easily be extended to span multiple islands or even processes operating on the CPU if a sufficiently large depth is required.

Finally, to read the top 50 IP addresses and the associated estimates, a host handler process is added. This process waits for a request signal from the host and then replies with the current top 50 IP addresses and their associated estimates. The connection layout for all the handlers on a single island is illustrated in Figure 6.15.

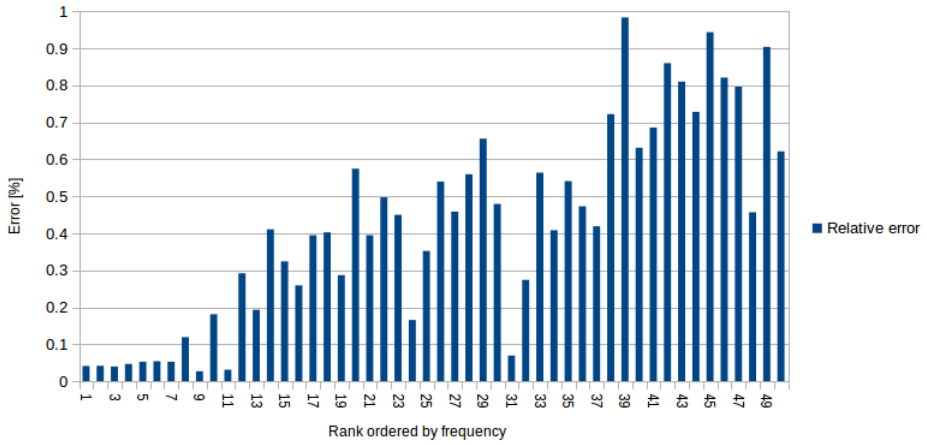


Figure 6.16: Estimation errors for top 50 recorded IPs.

Testing and evaluation

To test the implemented *count-min* sketch, a pre-recorded network capture was used as the input stream. This capture was replayed to the NFP interface and once complete, the host requested and recorded the top 50 IP addresses and their associated estimates. For comparison, a dictionary based approach to recording the top 50 IP addresses was implemented on the host and supplied with the same network capture as input.

In terms of space requirements, the dictionary based solution used 8.13 MB of memory while the sketch based solution only used 218.6 KB of memory, a space reduction of 38×. It should also be noted that the memory consumed by the sketch is fixed, regardless of the size of the stream while the dictionary based solution grows so that it can accommodate new unique elements during execution.

The advantage of using the dictionary based solution is that it provides the true frequencies for all IPs recorded in the capture which allows it to act as a benchmark for the test. Comparing these true values against the estimates provided by the sketch, the accuracy of the top 50 IP accesses can be determined (see Figure 6.16). From this distribution it is clear that the relative error between estimated frequency and true frequency is comparability small, never exceeding 1% in the top 50 addresses.

6.5 GeoIP Application

As a final implementation, an extended example using the NFComms framework was designed and implemented. This application has two main goals which it attempts to achieve. The first goal is to present a functional use case example of a system in the

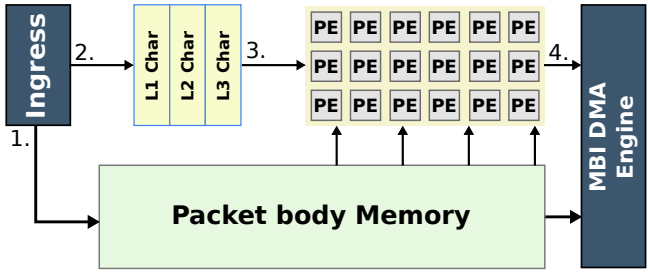


Figure 6.17: Packet pre-classification by the NBI.

domain of network processing, utilising both the host CPU and the NFP device for computation. The second goal is to determine the feasibility of developing a monitoring system capable of performing complex operations such as geotagging network traffic in real-time without heavily impacting network throughput. To achieve these goals, a network processing application spanning both the host and NFP architectures is presented. The work presented in this section has been published in (Pennefather *et al.*, 2018c).

6.5.1 Packet Pre-Classification

Before describing the design of the proposed application, a dedicated processing unit present on the NFP architecture that is relevant to this solution must first be introduced. The unit is the Network Block Interface (NBI) which forms part of the packet processing subsystem. This component can be interfaced with to request and submit traffic to the underlying components of the NFP responsible for interfacing with the physical medium (Netronome, 2016).

When a network packet is received on a physical interface, it is first handled by the NBI pre-classifier associated with that physical port. The pre-classifier assigns the packet to a buffer pool and prepares it for processing. After the pre-classification phase, received packets are run through a pool of packet classification engines, referred to a picoengines, which construct a stream of packet descriptors for the provided packets, as loosely depicted in Figure 6.17. These descriptors include meta information relating to the packet such as the size and physical location where the packet is stored within the NFP architecture. Once packet classification is complete, the pre-processed packets are ready to be moved from the NBI pre-classifier memory into a region more accessible to engines running the application code.

In a bid to minimise access latencies due to interacting with the pre-processed packets, the packets should be stored as close as possible to the microengines responsible for further processing. Unfortunately, due to the limited size of such memory regions, it is not

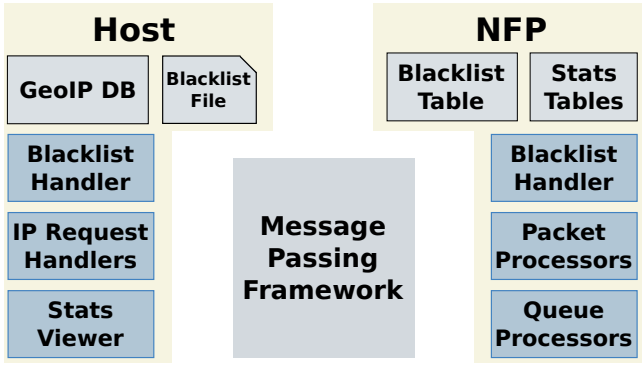


Figure 6.18: Overview of the monitoring application.

feasible to store the entirety of every packet in these regions so instead, packets larger than a specified threshold must be split with only the upper part of the packet and the metadata stored in these regions. The remainder of the packet is then stored elsewhere and a reference to its location is appended to the packet metadata. This approach allows packet processing to take advantage of having relatively low access latencies for most operations involving the packet header, and only requesting the full body of the packet when it is needed.

6.5.2 Application Overview

Continuing on with application design, the proposed solution is divided into two components, a host-based component and an NFP-based component as depicted in Figure 6.18. The host component is largely responsible for providing an interface for the user to interact with the system during operation via the `Stats Viewer` and `Blacklist Handler` sub-processes. For network processing, the host component also plays a key role as it is responsible for performing the actual geolocation of IP addresses seen by the network as well as issuing and revoking entries for blacklisted countries.

The NFP component of this application handles the collection, processing and retransmission of all network traffic with the bulk of this operation being handled by a collection of `Packet Processor` units. These units are distributed across the flow processing islands associated with the NBI as discussed in Section 6.5.1. The collection and maintenance of country based statistics is the responsibility of the `Queue Processor` units that operate on a work queue where the `Packet Processor` units can dump packet statistics before continuing to monitor the NBI queue for new traffic. Finally, as with the host component, the NFP component also supports a blacklist handler, which interacts with its host counterpart to synchronise the blacklist table.

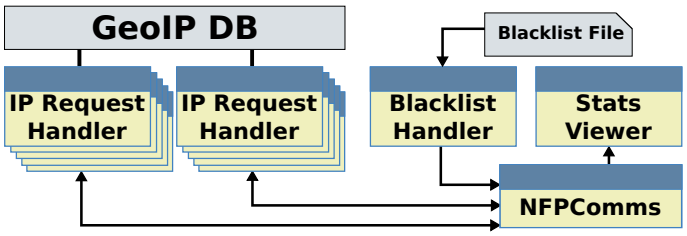


Figure 6.19: Overview of the host component of the application.

Host overview: Focusing on the host component of the application, Figure 6.19 presents the major sub-processes involved in the monitoring system. To utilise the NFComms framework, this component has been implemented in Go with each sub-process depicted in Figure 6.19 operating as a separate goroutine. The `Blacklist Handler` currently operates by simply reading the contents of a file at a set interval and processing its contents into a list of countries. Each country is associated with a country ID so that it can easily be identified using a single integer. This standard is adhered to by all components of the application with the translation of the country ID to an actual country name only occurring during the parsing of the blacklisted country file and the presentation of monitoring statistics. The `Stats Viewer` routine currently operates as a simple terminal application, allowing the users to specify a country for which he/she wishes to view the current statistics. Once specified, the current statistics are requested from the statistics tables stored on the NFP and presented to the user in a human readable format.

The remaining components of the host component are devoted to the translation of an IP address to a country ID. As latency is a major concern when implementing this component of the application, a new instance of `IP Request Handler` routine is created whenever a request from a packet processor running on the NFP architecture is received. This approach takes advantage of a strength of the Go programming language, namely the ease with which it can create, maintain and destroy lightweight routines that execute concurrently (Kozyra, 2014). Using this approach, the round trip latency for servicing an IP lookup request originating from the NFP was recorded to be between 55-80 μs.

The actual operation of converting a supplied IP address into a country is handled through an external GeoIP2 library[2]. This library interacts with the database containing GeoLite2 data as provided by MaxMind[3].

NFP overview: The application components presented in Figure 6.20 are distributed across multiple regions within the NFP architecture. To help hide some of the latencies

[2]The library source can be found at: `www.github.com/rainycape/geoip`
[3]Accessible from: `https://dev.maxmind.com/geoip/geoip2/geolite2/`

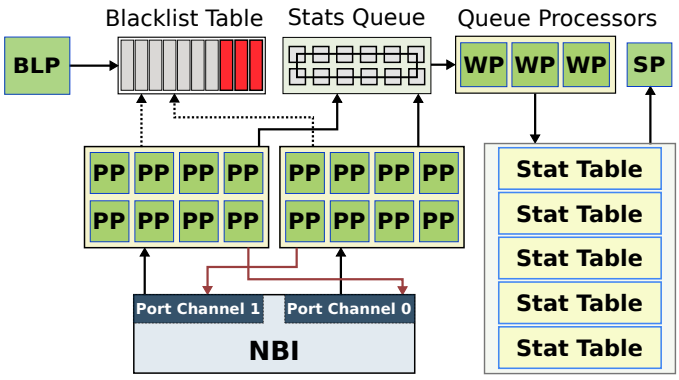


Figure 6.20: Overview of the NFP component of the application.

associated with packet processing, the NFP component of the application is designed to support a scaled number of work units for all components except the `Blacklist Handler` (BLP). For this implementation, there is only one blacklist table that must be managed making multiple instances of its management unit unnecessary.

The `Blacklist Handler` is responsible for providing the host component of the application with an interface through which the blacklist table can be modified. The host achieves this by submitting a message containing up to 26 records to the NFP `Blacklist Handler`. The handler then sequentially works through the message payload and updates the appropriate locations in the blacklist table. It is important to note that, while updating a single table entry is atomic, the process of parsing the complete message is not and the updates submitted by the host may not be immediately available to the `Packet Processor` (PP) units. As the NFP architecture does not support dynamic memory allocation, this table is declared to contain 249 elements to account for all countries defined in ISO 3166[4]. Within this table each element contains a flag which is set if traffic associated with that country should be dropped. By statically assigning the size of the table, a `Packet Processor` unit does not need to sequentially search the table for a particular country ID but rather it can use the country ID as an index resulting in the lookup being performed as a single operation.

As discussed in Section 6.5.1, the acquisition and submission of network traffic from the perspective of the application is performed by the NBI, which in turn is responsible for managing the pre-classification and submission subroutines. To interface with the NBI, the `Packet Processor` units leverage some of the supporting libraries supplied by Netronome (2018a) to request a packet for processing and modification before requesting that the packet be routed out of a specified port or dropped. Given that the NFP

[4]`https://www.iso.org/iso-3166-country-codes.html`

Table 6.3: Recorded IP datagram attributes.

Recorded Attributes
Country ID
IP Protocol
Length
Ingress Interface
(SYN, ACK, FIN, RST) flags^{+}

$^{+}$ If present in the packet being processed.

device used in this implementation only advertises two physical interfaces, routing can be performed by simply requesting the NBI to submit a processed packet over the alternative port queue to which it was received.

During initial testing, it was shown that the primary bottleneck with the packet processing component of the application was expectedly the operation of requesting the country ID associated with a given IP address. To help partially alleviate this, the `Packet Processor` units were extended to include a small amount of cache capable of storing up to 15 IP addresses and their associated country IDs. This extension was limited to 15 entries due to the remaining capacity of local memory associated with each processing element. The intent of this cache is that it can be treated as a rolling buffer local to a specific `Packet Processor`. Whenever an IP address is extracted from the acquired header, the processing unit first runs through this buffer to check whether the address has recently been serviced by the active unit. If it has been, the `Packet Processor` can simply use the associated country ID rather than requesting the IP to be translated by the host.

The other responsibility of the `Packet Processor` units is to collect some basic statistics on each network packet received, provided it does not have a source or destination IP associated with a blacklisted country. As this is done largely as a proof of concept, the types of statistics recorded are limited with only the attributes specified in Table 6.3 being logged if present in the packet being processed. This information is collected and used to build a packet record structure which is packed and inserted into a work queue. Once submitted, the structure is no longer the responsibility of the current `Packet Processor` and it can return to waiting until a new packet arrives for processing.

Servicing of the circular buffer is the responsibility of a different set of processing units depicted as `Queue Processor` (WP) units in Figure 6.20. During the initialisation phase of the application, one of these `Queue Processors` begins by initialising a work queue into which the `Packet Processor` units can deposit the statistical data relating to processed

packets. Once initialised, all `Queue Processor` units subscribe to the work queue as workers where they sleep until assigned a packet record by the queue.

The other responsibility of the initialising process is to set up the statistics tables to be populated. As with the blacklist table, this table consists of 249 entries, one for each country with each entry consisting of two records: one for ingress traffic and one for egress traffic. The ingress record list holds statistical information relating to traffic received over a specific physical interface while the egress record list holds the equivalent data for the alternate interface.

During operation, when a `Packet Processor` unit inserts a packet record into the work queue, the next available `Queue Processor` unit is woken up and assigned the work. This record is then parsed to identify the country ID and the originating interface over which the original packet was received. This data is used to identify the table and record which should be updated with the record body. Updating these records is achieved through atomic increment operations which serialise all modification commands to the memory in the case where more than one `Queue Processor` unit is attempting to update a single country record. Once a `Queue Processor` has completed processing a packet record, it resubscribes itself to the work queue.

Finally, to access the data stored in the country tables, a basic `Stats Interface` (SP) unit is provided. This process simply waits for a read request from the host component `Stats Viewer` routine executing on the host which specifies a country ID. The `Stats Interface` unit then reads the statistical data associated with that country and submits it to the calling routine where it can be viewed by the user.

6.5.3 Application Testing and Results

To evaluate the validity and performance of the implemented system, two separate test suites were designed and implemented. The first suite focused on throughput and general performance of the application components while the second focused on confirming the correctness of the logging and filtering subsystems.

Performance Evaluation

For performance, focus was on evaluating the impact the implemented application could have on throughput by initiating bulk transfers through the network being monitored and recording the resulting transfer speeds. This test suite comprised multiple test instances where each instance performed a bulk transfer between two nodes on opposing sides of

the monitored network. This transfer attempted to move 1024 MB of data through the network using the *iperf* tool[5] and the time taken to complete this task was recorded. In addition to this bulk transfer, each test instance would also inject traffic from an increasingly wide range of network addresses into the network stream, all targeting the destination node. The intent of this traffic injection is to potentially highlight the impact that increasing the range of source IPs could have on the throughput of the transfer.

To provide a baseline, the NFP device was initially disconnected from the test environment and the test suite was run, the results of which are presented as Baseline A in Figure 6.21(a). Following this, the NFP device was reconnected to the test bench but programmed with a basic forwarding application that simply passed through network traffic without performing any processing. The test suite was again run to produce Baseline B which highlights any performance impact introduced by simply inserting the NFP device into the network stream. Finally, the NFP device was flashed with the implemented application and the test suite was rerun.

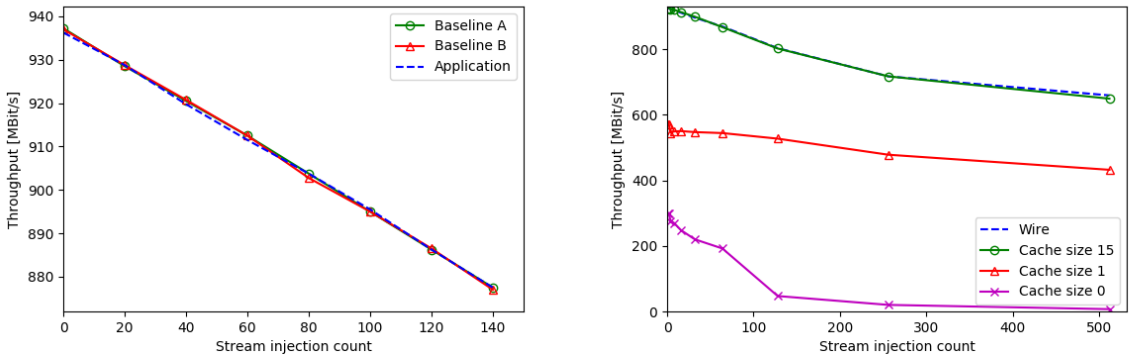
The results of the testing are shown in Figure 6.21(a) which depicts the recorded throughput for the baseline tests and the application test. Though multiple tests were performed, the overall throughput recorded for both baselines and the application indicated almost no marked variation in throughput. On average, Baseline B and the application both exhibited a throughput degradation of 0.15 Mbit/s relative to Baseline A, however, this is within the error margin of the recorded results.

A drop in performance was initially expected after more than 15 artificial IP streams were injected into a network stream as this would be the point where the caches discussed in Section 6.5.2 could begin to overflow, resulting in a large increase in GeoIP lookup requests occurring throughout the remainder of the test. The recorded results however did not reflect this. Reviewing the output from the traffic statistics sub-process for this test suite confirmed correct operation.

As incoming traffic is handled by the next available processing engine, engines blocked by a lookup operation are effectively removed from the pool of available engines until the lookup resolves. During this time however, there are still enough processing engines available to service network traffic associated with known country IDs. As a consequence, the latencies associated with the lookup operations are effectively hidden from traffic with cached addresses.

To better evaluate the impact that different cache sizes could have on performance, the

[5]Accessible from: `https://iperf.fr/en/`



(a) Results from the throughput test suite (b) Results of testing on varying IP cache sizes

Figure 6.21: Recorded throughput for different cache sizes.

cache size was altered and the test suite rerun (see Figure 6.21(b)). Testing with cache sizes capable of holding between 4 and 15 IP addresses resulted in no observable performance degradation, but reducing the cache size to a single IP address or eliminating the cache entirely resulted in a notable drop in performance. For a cache size of zero, the performance dropped as low as 7 MBit/s when the number of concurrent streams was increased to 512. This is significantly lower than the recorded throughput of 649 Mbit/s for a cache size of 15 IP addresses.

Correctness Evaluation

For this application, operation can be broken down into three distinct components: GeoIP lookup, dropping of blacklisted traffic, and correct output generated by the `Stats Viewer` routine. Confirming the correctness of each of these components was achieved by creating a set of test simulations with controlled input and expected output. To evaluate both the GeoIP lookup and statistics subsystems, a set of both UDP and TCP traffic was generated, spanning a range of IP addresses associated with a specific country. This dataset was iteratively extended to include traffic from an increasing number of countries on each iteration run through the monitored network. The results for each iteration were recorded and compared against the expected output to confirm correctness.

The generated traffic from the last iteration of the GeoIP and statistics correctness evaluation was used as the basis for evaluating the blacklisting application component. For this test suite, the source data supplied to the network under test remained fixed while the set of blacklisted counties was varied. Running this test suite but applying a short delay between when a modification to the range of blacklisted countries was performed and when the test data were supplied to the network resulted in the expected operation.

Omitting the delay resulted in traffic from blacklisted countries initially being incorrectly forwarded until the new rule set had been propagated to the monitoring system.

6.6 Reflection

In the context of the five application examples provided, some general observations relating to both the NFComms framework and the concept of a CPU-NFP heterogeneous system can be made. The aim of these observations is to summarise the results of the different examples and provide an informed opinion on the strengths and weaknesses of the system under test. These observations have been divided into three categories: communication, computation, and memory.

6.6.1 Communication

As discussed in Section 1.2, one of the goals of this research was to provide a reliable communication medium to allow processes on the host and NFP to communicate. In Chapter 4 this was realised with the presentation of the NFComms framework, which was refined in Chapter 5 to include bulk data transfers. Both chapters provide functionality and performance testing on the framework itself but utilise very basic examples and do not consider the concept of a heterogeneous compute platform. This chapter presents the intended use of the framework to produce a heterogeneous system in five scenarios. Sections 6.1 and 6.2 show use of the framework purely for the submission of work and collection of results. From these tests we observe that the NFComms framework operates as intended, providing a reliable communication medium for data transfer and coordination between the two disparate architectures. For these examples, the communication latency is not crucial as communication is limited to before and after the processing stages. In Section 6.4, the NFComms framework was used for intra-architecture communication, allowing the components of the *count-min* sketch to communicate with other processing elements without being physically aware of their location within the architecture.

Section 6.5 presents an application example where communication events between the CPU and NFP must occur during processing. This requirement is implemented by having part of the processing event involve the CPU in a manner that cannot be interleaved. Figure 6.21(b) describes the observed throughput of the application example with various cache sizes. From these results, two observations can be made regarding communication for offloading computation to the CPU. Firstly, the latency associated with synchronous communication and subsequent processing can have a very negative impact on application

performance. This is clearly seen when reviewing the performance of the application with a cache size of zero. In this situation, every network frame received generates a blocking communication event, stalling its categorisation and potential retransmission. The second observation is that the inclusion of a simple buffer can greatly reduce the impact of communication events. By allowing the communicating processes to store a single address, the performance increased from 7 Mbit/s to 432 Mbit/s, a 61× improvement.

When utilising the NFComms framework to enable the use of the NFP as a coprocessor, care must be taken to optimise how and when communication occurs. Owing to the inherent latency associated with the message passing events, the use of these in time critical components of an application should be minimised. Applications requiring regular communication events for a sequential solution do not map well to the NFP. Synchronous communication does, however, allow for a message transmission event to act as a point of synchronisation between the communicating parties which can be used to coordinate processes operating on the disparate architectures.

6.6.2 Computation

In terms of computation, there are three important factors to consider: instruction set, clock speed, and core count. As detailed in Section 2.1.1, the processing capabilities of a microengine have been optimally designed for the domain of network processing and as a result, have a limited set of processing features when compared with more general processing units. Capabilities such as floating point operations and the division operator are, for example, omitted in the NFP architecture (Netronome, 2018b) while other functions such as the modulo (%) operator are present, but are not supported by any dedicated hardware (Netronome, 2016). This reduced instruction set quickly becomes a limiting factor in the types of tasks that can be offloaded to the NFP. An example of this is the application examples presented in Section 4.6 where prime number and fractal generation solutions are implemented to run on the NFP. In both cases, the processing performance was considered very poor, largely because of the limited instruction set. For the prime number generation, this was due to the lack of hardware support for the modulo operator, restricting the solution to use a software variant. In the case of the fractal generation, the lack of support for floating point operations meant that the solution had to be implemented using fixed point arithmetic (Barina, 2014). Although functional, the resolution that can be stored in a 32-bit value when using fixed point arithmetic is limited. For applications that do not require floating point operations such as those presented in Sections 6.1, 6.2, and 6.4, the use of the NFP as a coprocessor is feasible.

The next factor to consider is the clock speed of the processing elements. The NFP-400 which was used in this research has a clock speed of 600 MHz, 4.5× slower than the host processor used for testing. As discussed in Sections 1.3 and 2.1.1, the NFP-400 was selected at the outset of the research due to its availability; however, newer revisions of this architecture, such as the NFP-6000 are now available and boast a speed of 1.2 GHz which could help improve the performance of applications presented in this chapter.

The final topic to discuss in terms of computation is the impact of core count. Although not explicitly stated, a factor considered when selecting the application examples presented in this chapter was their scalability. The proposed solution to the travelling salesman problem investigates the impact multiple microengines can have on the proposed solution by recording the execution time for 1000 random restarts for an increasing number of microengines (see Figure 6.5). This plot helps depict the impact that increasing the number of microengines can have on performance. The observed speedup is expected as each microengine executes in parallel having its own dedicated resources and ALU.

Each microenigne can however support up to eight contexts that execute concurrently but share the same ALU. The impact utilising multiple contexts can have on performance is highly dependent on the application. In a situation where a large number of memory operations external to the microengine must occur, the use of multiple contexts can introduce a speed-up. This can be seen in the PCSA solution provided in Section 6.3 where by enabling multiple contexts, the time spent waiting for a memory transaction to resolve can be used by another context, optimising ALU use. Referring back to the travelling salesman problem however, this was found not to be the case. Figure 6.4 shows performance times of each solution using different context counts where no significant difference in performance was recorded.

6.6.3 Memory

Considering the application examples, it is clear that minimising memory latency is an important factor to consider. In the travelling salesman implementation, it was shown that performing lookup operations on pre-computed distances between nodes is less efficient than simply recomputing the values as required (see Figure 6.3). Section 6.2 provides a more detailed evaluation of memory performance for the k-mismatch solution from which it is clear that memory selection is highly dependent on the application. Based on the results observed, CLS is the most efficient memory type to use for computation as it exhibits the lowest access latency. For the k-mismatch solution however, it was found to be one of the poorest performing memory types due to its limited capacity.

Both IMEM and EMEM were found to be good candidates due to their large capacity with IMEM being the better option provided the source data was under 4 MB in size. Interestingly, CTM resulted in the best performance overall requiring slightly longer access times than CLS, but advertising a larger capacity. For the implementations under test, this capacity/latency trade-off resulted in the best execution times for texts over 1 MB in size.

6.7 Contextualising the Limitations of the Framework

Before closing the chapter, it is worth drawing a comparison between the limitations of the CPU-NFP heterogeneous platform presented and those of the CPU-GPU heterogeneous platform in its early days. In both systems, the coprocessor is a multi-core PCIe endpoint originally intended to operate in a domain outside of general computing. Today, with the advances in the Compute Unified Device Architecture (CUDA)[6] and its unified memory, General Processing on a GPU (GPGPU) has matured to provide a large speed-up to many types of problems in the general processing domain. This however, was not always the case. Early approaches such as the work by Buck *et al.* (2004) and the first versions of CUDA[7] helped pioneer the concept of general processing on a GPU, although not without limitations.

Originally, GPUs were designed exclusively for processing graphics data before it was rendered on the host display (Barlas, 2014). Processing typically involved handling a large volume of graphical data within a limited time frame and the GPU architecture was designed to facilitate this. The general approach was to introduce a large number of relatively simple ALUs, all with very small independent caches, that could execute in parallel. This allowed the architecture to take advantage of the data parallelism seen in graphical data where ALUs could work on independent data segments in parallel. At this point, GPUs were in a similar state to many current NPUs: they were designed for a specific domain and optimised accordingly (Barlas, 2014).

During the earlier exploration stages of general computing on a GPU, some research observed that certain tasks, assumed to be optimal for the GPU, actually performed worse than on existing CPU implementations. For example, Fatahalian *et al.* (2004) present their findings on performing dense matrix-matrix multiplication on the GPU where it is recorded to be less efficient than the CPU. As noted by the authors, the key cause of this limitation is the access latencies associated with cached data on the GPU. Vuduc

[6] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`
[7] `https://developer.nvidia.com/content/cuda-toolkit-11-june-2007`

et al. (2010) performed a similar investigation of three different applications and observed similar results, with an optimised CPU solution outperforming the equivalent GPU implementations.

Other research however, such as that presented by Ohshima *et al.* (2007) showed how heterogeneous solutions utilising GPUs could result in improved application performance. In this situation the authors were able to utilise the GeForce 7800 GTX[8] as a coprocessor for matrix multiplications, reducing the execution time of the test application to 40.1%. Compared with today's standards, this improvement may seem very minor, but given that support for GPGPU was still in its infancy, such results were considered significant. A similar investigation was performed by Charalambous *et al.* (2005) where components of the RAxML[9] bioinformatics program were ported to a GPU. This solution yielded a small improvement, with the heterogeneous solution introducing an overall speed-up of 20% when compared to a CPU-only implementation.

Although no longer relevant, Owens *et al.* (2007) provide a good summary of some of the limitations associated with the GPU in the domain of general processing at the time. As early GPUs were effectively massively parallel vector processors, the lack of integer and logical operations was a notable limitation. Other issues such as memory limitations (echoed by Fatahalian *et al.* (2004)) were also a concern. Furthermore, programming GPUs was by no means simple. At this point the GPU architecture had not been adapted to support general processing and so such applications had to be carefully adapted before they could be interpreted within the graphic processing pipeline. Prior to CUDA, programming models like Brook (Buck *et al.*, 2004) were used to help facilitate this process.

Considering the limitations observed for the NFP under test in Section 6.6, it can be seen that early instances of general processing on the GPU suffered from similar drawbacks. Just as the early GPUs did not support integer and logical operations, the NFP suffers from lack of floating-point operations. A similar trend can be seen in memory with early GPUs being restricted in terms of cache size, while testing in this research concluded that limited low latency memory capacity is a major drawback of the NFP. The architectural limitations also reflect a critical applicability of the relevant architecture to the domain of general computing. Both architectures have observed applications that perform poorly and both have recorded heterogeneous solutions (see Section 4.6.1) that exhibit a relatively small improvement in performance compared to a CPU-only approach.

[8] `https://www.nvidia.com/page/geforce_7800.html`
[9] `https://cme.h-its.org/exelixis/web/software/raxml/index.html`

6.8 Concluding Remarks

Although use of the NFP for generic computation is viable, beyond very specific use cases, it appears that the performance improvement in this domain would not justify the inclusion of an NFP as a coprocessor. This conclusion is largely driven by the reflections discussed in Section 6.6 along with the performance results observed for the application tests in Section 4.6.

This does not mean that the proposed framework or the concept of a heterogeneous CPU-NFP system should be discarded as such a system does have use in the fields of network and stream processing. For example, as the the NFP-6 architecture boasts a faster clock rate, it is speculated that the performance of presented applications could improve if implemented on the revised architecture.

Furthermore, considering the limitations observed in early GPUs and their subsequent evolution, the viability of using a NFP as a coprocessor could be realised if the issues raised in Section 6.6 are addressed. Such improvements should include larger memory capacities, memory management, or provide support for floating point operations.

6.9 Summary

This chapter explored the viability of the implemented NFComms framework as well as the resulting heterogeneous CPU-NFP computing system in the domains of general purpose and network processing. This exploration was achieved through the presentation of five application examples that provide a qualitative comparison between the heterogeneous system and either a CPU-only solution or a GPU accelerated solution.

To better explore the domain of general computing, solutions to the travelling salesman problem and k-mismatch problem were presented in Sections 6.1 and 6.2, respectively. These applications explored the performance of well known solutions when using the NFP as a coprocessor. Section 6.1 focused on exploring the impact that multiple processing elements could have on an application, both in terms of truly parallel elements and those that execute concurrently but share resources. Section 6.2 presented two approaches to the k-mismatch problem and evaluated them accordingly. The focus of this application example was to explore the impact memory can have on processing by testing multiple variants of the solutions, each targeting a different memory type.

The remaining three examples focused on the domains of network and stream processing with the first two applications being approximation models or sketches. These algorithms

were used to estimate a characteristic of an unbounded stream of data in bounded memory. The first example is PCSA, presented in Section 6.3, utilised the NFP as a coprocessor for estimating the cardinality of an IP addresses in a stream of network traffic. In addition to providing an example better suited to the NFP, this solution also helped highlight the impact memory latency can have on processing and how the use of multiple contexts can effectively be used to hide this. Section 6.4 followed the PCSA implementation with the presentation of a *count-min* sketch used to detect the top 50 most frequently seen IP addresses in a network stream. The final example, presented in Section 6.5, demonstrated how both the CPU and NFP could be used to enrich a network-based application with additional functionality. This application presented a network filter that identifies IP addresses based on their country of origin which is achieved by making the NFP responsible for performing the filtering operations while the geolocation operations were offloaded to the CPU.

Following the presentation of application examples and subsequent testing, a reflection on the results was presented in Section 6.6. This reflection discussed the performance characteristics observed throughout testing in terms of communication, computation, and memory. This reflection observed that the NFComms framework works as expected but could have a notably negative impact on performance in communication dominant applications. In terms of computation, the reduced instruction set of the NFP restricts the types of applications that can be run on the NFP, however those that can be adapted or are unaffected by this restriction are viable. Finally, the impact of memory on the presented application was discussed, highlighting the trade-off between capacity and latency and how it effects the performance of the applications under evaluation.

To help contextualise the architectural limitations of the NFP, a brief overview of the issues surrounding GPUs during the early research into GPGPU was presented. This section highlighted processing and memory limitations which initially resulted in the GPU performing poorly, particularly when compared to the current state of the research domain. This discussion was used to help formulate a final decision on the viability of using the NFP as a coprocessor in Section 6.8. In its current state, using the NFP-400 as a coprocessor for general computing is not considered viable although, this observation could easily change should the limiting factors identified be addressed.

Chapter 7 follows this chapter by summarising the work presented in this thesis. A reflection on the goals initially presented in Chapter 1 as well as a discussion on the contribution this work makes to the fields of network processing and heterogeneous computing. The thesis concludes with a discussion on future work.

7

Conclusion

The research documented in this thesis investigated the feasibility of introducing a generic framework for runtime communication between host processes and applications executing on an NFP. The result of this investigation is NFComms, a CSP compliant prototype framework that allows for synchronous message passing between applications written in Go and applications operating on the NFP. The feasibility of both the NFComms framework and the resulting CPU-NFP heterogeneous system has been empirically evaluated through the use of problems spanning the domains of both general computation and network specific processing.

This chapter concludes the thesis by presenting a summary of the research conducted in Section 7.1. The objectives of the research are reflected on in Section 7.2 before the contributions of this work are presented in Section 7.3. Finally, the chapter concludes with a discussion on potential avenues for further research that could be undertaken to extend the work presented.

7.1 Summary of Research

This research began with a feasibility study into describing a communications framework between an NFP and a user-space application operating on the host system. To facilitate this research, an investigation into process algebras was first performed, concluding with a summary of two prominent formal notations. This was followed by a summary of different communication models categorised by how communication is achieved.

After reviewing both the formal notations of CCS and CSP as well as existing communication models, it was concluded that the formal notation CSP would be used to underpin this initial evaluation. This decision was largely driven by the investigation concluding that the host language, Go, is already considered capable of implementing models defined in CSP with examples described by Hoare (1978) being implemented in Go (Kappler, 2016). The capability of the NFP to represent communication as synchronous and atomic events was shown similarly with a subset of the examples described by Hoare (1978) being replicated on the architecture. Additional mechanisms of CSP were also investigated, confirming compliance of the NFP with the CSP model.

As the feasibility study concluded that a communication framework between the CPU and NFP was viable, design and implementation of such a framework could be undertaken. Framework development began with an investigation into another important communication characteristic, that is, naming and symmetry. This quickly became a concern when it was found that the proposed hardware, the NFP, and the supported programming language, Go, conformed to different communication semantics. However, a solution was identified through the use of supporting actors referred to in the text as engines. This solution was elaborated on to produce an overview of how synchronous message passing would be realised within the framework, including the formalisation of a message structure understood by all engines.

With the communication mechanism identified and concerns relating to symmetry and naming resolved, development on the framework could begin. First, an approach to handling communication between the kernel and the NFP architecture was established by extending the existing NFP driver to support outstanding mechanisms essential for synchronous communication. This included adding support for MSIX interrupts, establishing a message-box system for handling multiple channels independently, and a set of handler functions to interact with requests made from user-space applications. Both the Go runtime and the engines associated with the NFP were then defined and developed, resulting in the first implementation of the NFComms framework.

A series of simple tests were performed to confirm correctness of the prototype and to evaluate performance. The framework was shown to operate as intended with latency tests recording a minimum latency of 15.5 μs when submitting messages over the framework originating from the host and 18 μs for messages originating from the NFP. However, the throughput capabilities of the prototype were shown to be unacceptably poor. For communication originating from the host, a peak throughput of 102.4 Mbit/s was recorded, while only 41 MBit/s throughput was recorded for transfers originating from the NFP. Although functional, these performance results were considered unsuitable for practical applications.

To resolve this issue, it was decided that a second revision of the framework was required. First, the source of the throughput limitations was identified. An investigation into resolving this bottleneck was undertaken and included an evaluation of storage locations for larger memory objects which would justify supporting higher throughput operations. The proposed solution was to introduce a separate set of messaging operators for indirect operations. These additional operators would be available for bulk data transfer while the existing communication channels could remain available to allow the revised framework to be backwards compatible with applications designed for the original implementation.

To implement the proposed solution, extensions to the NFP driver were introduced, allowing for the use of zero-copy techniques to reduce data movement between the kernel and user-space applications. The approach taken for bulk data transfer was to construct a gather list of all pages where data should be read from or stored. This list would then be submitted to the NFP and the NFP PCIe engine could handle the collection of the actual data associated with the pages on behalf of the framework. Rather than introduce a completely new communication medium, the existing communication framework was used for migrating lists from the host kernel to the NFP.

Once implemented, a second round of testing was performed. This was done firstly to confirm functionality, and secondly to evaluate performance in terms of latency and throughput. The latency tests indicated that that extensions to the framework are between 1.7× and 2× worse than the original implementation. In terms of throughput however, testing showed that the implemented extensions are capable of achieving an average throughput of 7.8 to 9 Gbit/s when servicing a single transaction, and up to 30 Gbit/s when servicing eight transactions. Considering that the original maximum throughput achievable by the framework was 102.4 Mbit/s, the indirect messaging extensions represent an improvement of up to 268×.

Finally, with the extension of the NFComms framework complete, an exploration could

be undertaken into the viability of using the implemented framework as well as the resulting heterogeneous CPU-NFP computing system in the domains of general purpose and network processing. This exploration was achieved through the presentation of five application examples that provide a qualitative comparison between the heterogeneous system and either a CPU-only solution or a GPU accelerated solution.

To better explore the domain of general computing, solutions to the travelling salesman and k-mismatch problems were presented with the focus on exploring the impact that multiple processing elements and different memory types can have on the performance of an application. The remaining three examples focus on the domains of network and stream processing with the first two applications being approximation models or sketches.

The first of these applications was PCSA which utilises the NFP as a coprocessor for estimating the cardinality of an IP addresses in a stream of network traffic. This was followed by an implementation of a *count-min* sketch, used to detect the top 50 most frequently seen IP address in a network stream. The final example presented a network filter that identifies IP addresses based on their country of origin which is achieved by making the NFP responsible for performing the filtering operations while the geolocation operations are offloaded to the CPU.

Testing was followed with a reflection of the results which discussed the performance characteristics observed in terms of communication, computation, and memory. This reflection concluded that the NFComms framework works as expected but could have a notably negative impact on performance in communication dominant applications. An investigation into issues surrounding GPUs during the early stages of the GPGPU era was undertaken to help contextualise the limitations observed with the NFP. The results of this investigation were used to help formulate a final decision on the viability of using the NFP as a coprocessor. It was concluded that, in its current state, using the NFP-400 as a coprocessor for general computing is not viable, although this observation could easily change should the limiting factors identified be addressed in future iterations of the architecture.

7.2 Achievement of Research Objectives

As presented in Section 1.2, the overarching objective of this research was to provide a CPU-NPU heterogeneous system. More specifically, the primary aim of this work was to evaluate the viability of introducing a generic communication framework between a host process and an NPU. To achieve this primary aim, four objectives were defined as

summarised in this section. For each of these we reflect on the degree to which it was achieved.

1. The first objective was to perform an exploratory evaluation of the two architectures and determine whether a communication medium could be established. This goal was achieved largely in Chapter 3 where it was shown that a framework, conforming to synchronous message passing as described in CSP, would be compatible with the NFP architecture. This was achieved by implementing a subset of the problems presented in Hoare (1978) on the NFP in a similar manner as performed by Kappler (2016).

2. The second objective was to implement a prototype communication framework to allow processes executing on the disparate architectures to communicate at runtime. This goal was achieved by developing the NFComms framework, a prototype communications framework that allows communication between processors operating on the host and NFP architecture. This communication is achieved through synchronous message passing, conforming to the communication mechanisms of CSP. The development of the NFComms framework is documented in Chapter 4 and Chapter 5.

3. To allow for interaction with processes operating on the host, the third objective was to provide an interface to allow applications, written in the Go programming language, to interface with the communication framework. This objective was achieved by introducing a Go runtime that would allow applications to subscribe to synchronous channels within the NFComms framework for interacting with the NFP (see Chapter 4 and Chapter 5).

4. The fourth objective involved carrying out an evaluation of the feasibility of using an NPU as a coprocessor in the domains of general and network computation. To satisfy this goal, the heterogeneous CPU-NPU compute platform using the NFComms framework and a NFP-400 as a coprocessor (as presented in Chapter 6) was used to explore two general purpose problems, evaluating computation speeds, memory impact, and power consumption. Additionally, three network orientated applications were also tested on the compute platform. The results of these five tests were used to formulate the opinion that the NFP-400 is currently not suitable as a coprocessor for general computation, largely owing to the limited instruction set, memory capacity, and processing speed.

7.3 Research Contributions

This research has contributed the following three novel aspects:

- The first contribution involves two parts. Firstly, a study into the compliance of the NFP to CSP formalisms was achieved by mapping CSP primitives to the NFP architecture. Based on this, the second part of the contribution relates to the confirmation that a communication framework between the Go programming language and the NFP is feasible through the use of synchronous communication as defined in CSP.

- Secondly, this research presents the NFComms framework, a platform that enables communication between processes operating on the NFP and CPU architectures through the use of synchronous message passing. The NFComms framework can also be used to facilitate intra-communication in a similar fashion, allowing tasks to be transparently distributed across both architectures and still communicate. Resources required by the framework have been kept minimal, requiring only two microengines and one hardware ring to operate. To enable interactions with user-space applications, the framework introduces a runtime for the Go programming language, enabling interactions with the framework. In addition, the NFComms framework supports channels for bulk data transfers, allowing for data transmission of up to 30 Gbit/s while maintaining its synchronous behaviour.

- As the third contribution, an evaluation into the feasibility of utilising a NFP-400 device is empirically presented with problems spanning both general computing and network processing having been implemented and benchmarked on the resulting heterogeneous platform. These results were used to conclude that employing the NFP-400 for general processing is not currently feasible. However, with the developments already underway in later versions of the NFP architecture, such general processing may soon become feasible. As noted in Chapter 6, an example of this is the NFP-6 architecture which boasts a 1.2 GHz clock speed instead of the 600 MHz clock associated with the NFP-400.

7.4 Future Work

Considering the work presented in this thesis, there are two clear aspects of the research that can be pursued further. Firstly, the current implementation of the NFComms framework was designed as a prototype, and as a result, is very limited. One avenue of research

would be to investigate extending the prototype into a more mature implementation. The second aspect is to extend the feasibility study on evaluating the use of an NPU as a coprocessor for general computing.

Extending the NFComms framework: The first aspect of this research that requires further extension is the NFComms system. Considering the current state of the framework, there are four distinct components which can be improved upon.

- The focus of the work presented in Chapter 3 was to determine if a communication framework between the NPU and CPU was feasible. To perform this evaluation, the well known formal notation CSP was selected as the basis of communication. Given that a functional framework has been implemented, the next step is to explore adapting it to support alternative communication models. One such model is asynchronous communication which would allow the NFComms framework to decouple the submission and reception of messages, hiding the communication latency in applications that do not require synchronous operations.

- Another limitation of the current prototype is that is can only establish a platform for interacting with a single NFP device. As mentioned in Section 4.3.5, when instantiating an interface to the NFComms runtime, the device number associated with the NFP is required. Although currently unused, the intent is for this device number to allow for interfacing with multiple NFP devices through the framework from a single host application. Further research into this is required to support this functionality. As part of this future work, the feasibility of allowing NFP devices residing within a single host to communicate without the involvement of the CPU also warrants exploration.

- While the previous two points focus on extending the NFComms framework for interfacing with the NFP device, an alternative avenue of research is to consider supporting alternative network processors. This research would require an extensive review of the communication layout and implementation to determine compatibility, potentially requiring a separate feasibility study on communication. Should this prove successful however, it would greatly improve the existing implementation, allowing it to become more generic.

- Finally, as noted in Section 1.3 one of the limitations imposed on this work is that the framework only supports Debian Linux. This limitation was introduced due to the focus of the research largely being on designing an initial prototype to perform the feasibility study. Now that the prototype has been established, future work can

explore porting the NFComms driver to alternative operating systems.

Extended feasibility study: As stated in Chapter 1, this research is largely exploitative in nature. Although it is the opinion of the authors that the NFP under test is not currently feasible for general computing, this most certainly does not mean the same conclusion can be made for other NPUs or future revisions of the NFP. One avenue of future work would be to expand the range of network processing architectures tested to present a more comprehensive study. As noted in Section 7.3, limitations associated with processing speed on the NFP-400 may already have been resolved due to the improved clock speed. Other issues such as the capacities of low latency memory regions or the reduced instruction set may not be present on alternate network processing architectures, potentially making them better candidates unless other limitations are also discovered.

References

Abelson, H., Dybvig, R., Haynes, C., Rozas, G., Adams, N., Friedman, D., Kohlbecker, E., Steele, G., Bartley, D., Halstead, R., Oxley, D., Sussman, G., Brooks, G., Hanson, C., Pitman, K., and Wand, M. Revised report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. doi:10.1023/A:1010051815785.

Aceto, L., Larsen, K. G., and Ingolfsdottir, A. An introduction to Milner's CCS. Supporting documentation, BRICS, Department of Computer Science, Aalborg University, 2004. Accessed on: 5 March 2019.
URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.132.1261`

Adiletta, M., Rosenbluth, M., Bernstein, D., Wolrich, G., and Wilkinson, H. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6:6–18, 2002. ISSN 1535766X.

Ahmad, I., Gulati, A., and Mashtizadeh, A. vIC: interrupt coalescing for virtual machine storage device IO. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11. USENIX Association, Berkeley, CA, USA, 2011.

Ahmadi, M. and Wong, S. Network processors: Challenges and trends. In *17th Annual Workshop on Circuits, Systems and Signal Processing*, pages 223–232. 2006.

Aimonetti, M. *Go Bootcamp*. Published Online, April 2015. Accessed on: 2 February 2019.
URL `http://www.golangbootcamp.com/book`

AMD. Introduction to OpenCL programming. Training guide, AMD, 2010. Accessed on: 26 May 2019.

URL `http://www.site.uottawa.ca/~mbolic/ceg4131/AMD-Introduction%20to%20OpenCL%20Programming%20(1).pdf`

Amir, A., Lewenstein, M., and Porat, E. Faster algorithms for string matching with k mismatches. *Journal of Algorithms*, 50(2):257 – 275, 2004. ISSN 0196-6774. SODA 2000 Special Issue.

Andrews, G. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing Company, Redwood City, CA, USA, 1991. ISBN 0805300864.

Andrews, G. R. and Schneider, F. B. Concepts and notations for concurrent programming. *ACM Computing Surveys (CSUR)*, 15(1):3–43, 1983. ISSN 0360-0300. doi:10.1145/356901.356903.

Baeten, J. C. M. A brief history of process algebra. *Theoretical Computer Science - Process Algebra*, 335(2-3):131–146, May 2005. ISSN 0304-3975. doi:10.1016/j.tcs.2004.07.036.

Baiardi, F. and Vanneschi, M. Parallelism issues in multistyle computers. In *Future Parallel Computers*, pages 1 – 34. Springer-Verlag, London, UK, 1987. doi:10.1007/3-540-18203-9_1.

Bala, V., Bruck, J., Cypher, R., Elustondo, P., Ho, A., Ho, C.-T., Kipnis, S., and Snir, M. CCL: a portable and tunable collective communication library for scalable parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):154–164, Feb 1995. ISSN 1045-9219. doi:10.1109/71.342126.

Bala, V. and Kipnis, S. Process Groups: a mechanism for the coordination of and communication among processes in the Venus collective communication library. In *Proceedings 7th International Parallel Processing Symposium*, pages 614–620. April 1993. doi:10.1109/IPPS.1993.262809.

Balbaert, I. *The Way To Go: A Thorough Introduction to the Go Programming Language*. iUniverse, Bloomington, India, May 2012. ISBN 1469769166.

Barina, D. Fixed-point arithmetic. Technical report, Brno University of Technology, 2014. Accessed on: 6 October 2018.
URL `http://www.fit.vutbr.cz/~ibarina/pub/fixed-point.pdf`

Barlas, G. *Multicore and GPU Programming: an Integrated Approach*. Morgan Kaufmann, 1st edition, 2014. ISBN 0124171370.

Barney, B. Message passing interface (MPI). Website, Lawrence Livermore National Laboratory, 2019. Accessed on: 26 May 2019.
URL `https://computing.llnl.gov/tutorials/mpi/`

Bertolotti, I. and Hu, T. *Embedded Software Development: The Open-Source Approach*. CRC Press, 1st edition, 2015. ISBN 146659392X.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014. ISSN 0146-4833. doi:10.1145/2656877.2656890.

Bottomley, J. Dynamic DMA mapping using the generic device. Kernel Documentation, Linux Kernel Organization, 2017. Accessed on: 22 August 2018.
URL `https://www.kernel.org/doc/Documentation/DMA-API.txt`

Brose, E. ZeroCopy: techniques, benefits and pitfalls. 2008. Accessed on: 17 April 2018.
URL `https://www.researchgate.net/publication/237632895_ZeroCopy_Techniques_Benets_and_Pitfalls`

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH*, 23(3):777–786, August 2004. ISSN 0730-0301.

Budruk, R., Anderson, D., and Solari, E. *PCI Express System Architecture*. Pearson Education, 2003. ISBN 0321156307.

Burch, C. Tail recursion and iteration. Technical report, Hendrix College, September 2012. Accessed on: 22 May 2019.
URL `http://www.toves.org/books/tail/`

Burns, A. and Davies, G. *Concurrent Programming*. Addison Wesley Publishing Company, Redwood City, CA, USA, 1993. ISBN 0-201-54417-2.

Burtscher, M. A high-speed 2-Opt TSP solver for large problem sizes. Presentation slides, Texas State University, 2014. Accessed on: 9 September 2018.
URL `http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf`

C++ development team. C++ keywords. Language documentation, C++ development team, 2019. Accessed on: 26 May 2019.
URL `http://en.cppreference.com/w/cpp/keyword`

Cavium. OCTEON III CN70XX and CN71XX single to quad-core embedded processors with hardware virtualization. Technical report, Cavium, 2014. Accessed on: 28 January 2019.
URL `https://www.marvell.com/documents/u3nve6oglfv9pxok1qqb/`

Cavium. Cavium LiquidIO II network appliance smart NIC. Technical report, Cavium, 2017a. Accessed on: 28 January 2019.
URL `https://www.marvell.com/documents/konmn48108xfxalr96jk/`

Cavium. Cavium OCTEON multi-core processor. Technical report, Cavium, 2017b. Accessed on: 28 January 2019.
URL `http://www.cavium.com/octeon-mips64.html`

Charalambous, M., Trancoso, P., and Stamatakis, A. Initial experiences porting a bioinformatics application to a graphics processor. In *Proceedings of the 10th Panhellenic Conference on Advances in Informatics*, PCI'05, pages 415–425. Springer-Verlag, Berlin, Heidelberg, 2005. doi:10.1007/11573036_39.

Chin, A. Reusable OpenCL FPGA infrastructure. Masters thesis, Graduate Department of Electrical and Computer Engineering, University of Toronto, 2012. Accessed on: 27 May 2019.
URL `https://tspace.library.utoronto.ca/handle/1807/32567`

Chisnall, D. *The Go Programming Language Phrasebook*. Addison-Wesley, Michigan, USA, March 2012. ISBN 0321817141.

Cisco. The Cisco flow processor: Cisco's next generation network processor. White paper, Cisco, USA, 2014. Accessed on: 26 May 2019.
URL `https://www.cisco.com/c/en/us/products/collateral/routers/asr-1000-series-aggregation-services-routers/solution_overview_c22-448936.html`

Clifford, R., Fontaine, A., Porat, E., Sach, B., and Starikovskaya, T. The K-mismatch problem revisited. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '16, pages 2039–2052. ACM, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2016. ISBN 978-1-611974-33-1.

Clinger, W. D. Proper tail recursion and space efficiency. *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, 33(5):174–185, May 1998. doi:10.1145/277652.277719.

Coleman, J. Reducing interrupt latency through the use of message signaled interrupts. White Paper, Intel Corporation, January 2009. Accessed on: 8 May 2019. URL `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/msg-signaled-interrupts-paper.pdf`

Conway, M. E. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963. ISSN 0001-0782. doi:10.1145/366663.366704.

Corbet, J., Rubini, A., and Kroah-Hartman, G. *Linux Device Drivers. Where the Kernel Meets the Hardware.* O'Reilly Media, 3rd edition, February 2009. ISBN B0026OR2XQ.

Cormode, G. and Muthukrishnan, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58 – 75, 2005. doi:10.1016/j.jalgor.2003.12.001.

Cowles, J. and Gamboa, R. Contributions to the theory of tail recursive functions. Internal article, University of Wyoming, 2004. Accessed on: 8 May 2019. URL `http://www.cs.uwyo.edu/~ruben/static/pdf/tailrec.pdf`

De Nicola, R. A gentle introduction to process algebras. Supporting notes, MT - Institute for Advanced Studies Lucca, 2014. Accessed on: 26 May 2019. URL `http://tiny.cc/rrta7y`

Deremer, F. Practical translators for LR(k) languages. Doctoral thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, October 1969. Accessed on: 26 May 2019.

Dinning, A. A survey of synchronization methods for parallel computers. *IEEE Computer Society*, 22(7):66–77, July 1989. doi:10.1109/2.30733.

Fatahalian, K., Sugerman, J., and Hanrahan, P. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 133–137. ACM, New York, NY, USA, 2004. doi:10.1145/1058129.1058148.

Flajolet, P., Fusy, Ã., Gandouet, O., and Meunier, F. HyperLogLog: the analysis

of a near-optimal cardinality estimation algorithm. In *DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms, (AofA 07)*.

Flajolet, P. and Martin, G. N. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182 – 209, 1985. ISSN 0022-0000.

Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2 edition, 2013. ISBN 9780124058941.

Gelado, I., Stone, J. E., Cabezas, J., Patel, S., Navarro, N., and Hwu, W. W. An asymmetric distributed shared memory model for heterogeneous parallel systems. *SIGPLAN Not.*, 45(3):347–358, March 2010. ISSN 0362-1340. doi:10.1145/1735971.1736059.

George, L. and Blume, M. Taming the IXP Network Processor. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 26–37. ACM, New York, NY, USA, 2003. doi:10.1145/781131.781135.

Golang. Setting up and using gccgo. The Go Programming Language, 2015. Accessed on: 8 May 2019.
URL `https://golang.org/doc/install/gccgo`

Google. The Go programming language specification. Language documentation, Google, August 2015. Accessed on: 3 November 2018.
URL `https://golang.org/ref/spec#Introduction`

Google. The Go programming language: Effective Go. Language documentation, Google, 2017a. Build version: Go 1.8. Accessed on: 20 February 2017.
URL `https://golang.org/doc/effective_go.html`

Google. The Go programming language: FAQ. Language documentation, Google, 2017b. Build version: Go 1.8. Accessed on: 20 February 2017.
URL `https://golang.org/doc/faq#goroutines`

Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014. ISBN 9780262527392.

Gropp, W., Lusk, E., and Thakur, R. Intermediate MPI. Presentation slides,

Argonne National Laboratory, 2015. Accessed on: 7 May 2019.
URL ftp://ftp.idsa.prd.fr/local/ascii/hleroy/Cours-Mpi-1-argonne.pdf

Harris, M. Unified memory for CUDA beginners. Developer blog, NVIDIA, 2017. Acccessed on: May 2019.
URL https://devblogs.nvidia.com/unified-memory-cuda-beginners/

Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi:10.1145/363235.363259.

Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. ISSN 0001-0782.

Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. ISBN 0-13-153271-5.

Hoefler, T., Di Girolamo, S., Taranov, K., Grant, R. E., and Brightwell, R. sPIN: high-performance streaming processing in the network. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, pages 59:1–59:16. ACM, New York, NY, USA, 2017. doi:10.1145/3126908.3126970.

Hwu, W. and Stone, J. The OpenCL programming model. Presentation slides, UPCRC Illinois, 2010. Accessed on: 9 May 2019.
URL https://www.ks.uiuc.edu/Research/gpu/files/upcrc_opencl_lec1.pdf

Hyde, D. Introduction to the programming language Occam. Technical Report, Department of Computer Science Bucknell University, 1995. Accessed on: 9 May 2019.
URL http://www.cs.otago.ac.nz/cosc441/occam.pdf

Intel. Intel IXP2800 Network Processor for OC-192/10 Gbps network edge and core applications. Product Brief 1, Intel, Santa Clara, CA. USA, 2002. Accessed on: 26 May 2019.
URL http://www.ic72.com/pdf_file/i/587106.pdf

Intel. Introduction to the auto-partitioning programming model. accelerating custom application development on Intel IXP2XXX network processors. Technical report, Intel, USA, October 2003. Accessed on: 23 August 2016.
URL http://www.intel.com/design/network/papers/25411401.pdf

Intel. Intel IXP28XX Network Processors. Hardware Design Guide 309192-002US, Intel, August 2005. Accessed on: 26 May 2019.

URL `https://datasheet.octopart.com/RPIXP2850BB-Intel-datasheet-11898136.pdf`

Intel. Intel virtualization technology for directed I/O. Architecture Specification, Intel, 2017. Accessed on: 26 May 2019.
URL `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf`

Jacobsen, M., Richmond, D., Hogains, M., and Kastner, R. RIFFA 2.1: a reusable integration framework for FPGA accelerators. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(8):22:1–22:23, October 2015. doi: 10.1145/2815631.

Jain, N. and Jain, M. K. Current status of network processors. *International Journal of Computer Applications (0975-8887)*, 98, July 2014. doi:10.5120/17239-7573.

Johnson, E. and Kunze, A. *IXP2400/2800 Programming. The Complete Microengine Coding Guide*, volume 1 of *Intel Press*. Rich Bowles, Chicago, USA, 1st edition, April 2003. ISBN 0-9717861-6-X.

Johnson, E. J. and Kunze, A. *IXP-1200 Programming*. Intel Press, 2002. ISBN 097128878X.

Kahn, G. The semantics of a simple language for parallel programming. In *IFIP Congress on Information Processing. North-Holland*, volume 74, pages 471–475. North Holland, Amsterdam, 1974.

Kappler, T. Package CSP. Package Documentation, Go Documentation, February 2016. Accessed on: 8 March 2019.
URL `https://godoc.org/github.com/thomas11/csp`

Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., and Krishnamurthy, A. High performance packet processing with FlexNIC. *SIGPLAN Not.*, 51(4):67–81, March 2016. ISSN 0362-1340. doi:10.1145/2954679.2872367.

Khronos OpenCL Working Group. The OpenCL specification. Technical specification, Khronos, 2012. Version 1.2, Revision 19.

Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *SIGOPS Oper. Syst. Rev.*, 18(3):263–297, August 1999. ISSN 0734-2071. doi:10.1145/354871.354874.

Koymans, R. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1992. ISBN 978-3-540-56283-2.

Kozyra, N. *Mastering Concurrency in Go*. Packt Publishing, Birmingham, UK, July 2014. ISBN 1783983485.

Landaverde, R., Zhang, T., Coskun, A. K., and Herbordt, M. C. An investigation of unified memory access performance in CUDA. *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014. doi:10.1109/HPEC.2014.7040988.

Lastovetsky, A. and Dongarra, J. *High Performance Heterogeneous Computing*. John Wiley & Sons, Inc., Hoboken, New Jersey, USA, 2009. ISBN 0470040394.

Lawley, J. Understanding performance of PCI express systems. White Paper, Xilinx, 2014. Accessed on: 26 May 2019.
URL `https://www.xilinx.com/support/documentation/white_papers/wp350.pdf`

LeBlanc, T. J. and Markatos, E. P. Shared memory vs. message passing in shared-memory multiprocessors. In *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 254–263. Dec 1992. doi:10.1109/SPDP.1992.242736.

Levenshtein, V. Binary codes capable of correcting spurious insertions and deletions of ones. *Russian Problemy Peredachi Informatsii*, 1:12–25, 1965.

Levenshtein, V. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics*, pages 707–801, 1966.

Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., and Chen, E. ClickNP: highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 1–14. ACM, New York, NY, USA, 2016. doi:10.1145/2934872.2934897.

Li, F. and Wan, J. Network processor architectures, programming models, and applications. Technical report, University of Massachusetts Lowell, 2003. doi:10.1.1.103.5775.

Li, K. IVY: a shared virtual memory system for parallel computing. In *International Conference on Parallel Processing*, volume 2, pages 94–101. 1988.

Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. *ACM*

Transactions on Computer Systems (TOCS), 7(4):321–359, November 1989. doi:10.1145/75104.75105.

Li, L., Huang, B., Dai, J., and Harrison, L. Automatic multithreading and multiprocessing of C programs for IXP. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 10, pages 132–141. New York, NY, USA, 2005. doi:10.1145/1065944.1065963.

Lin, X., KrishnaMurthy, A., Peter, S., and Gupta, K. iPipe: a framework for building datacenter applications using in-networking processors. Research report, Paul G. Allen Center for Computer Science and Engineering, 2018. Accessed on: 26 May 2019.
URL `https://www2.cs.uic.edu/~brents/cs494-cdcs/papers/ipipe-preprint.pdf`

Liu, Y. A. and Stoller, S. D. From recursion to iteration: What are the optimizations? In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '00, pages 73–82. ACM, New York, NY, USA, 1999. doi:10.1145/328690.328700.

Long, G. *C Language Interfaces*. Prentice Hall, 1989. ISBN 9780131096615.

Love, R. *Linux Kernel Development*. Addison-Wesley, 3rd edition, 2010. ISBN 9780768696974.

Lu, H., Dwarkadas, S., Cox, A. L., and Zwaenepoel, W. Message passing versus distributed shared memory on networks of workstations. In *Supercomputing 95: Proceedings of the 1995 ACM-IEEE Conference on Supercomputing*, pages 37–37. Dec 1995. doi:10.1109/SUPERC.1995.241387.

Mandelbrot, B. *Fractals and Chaos: the Mandelbrot Set and Beyond*. Springer, 2004. ISBN 978-1-4757-4017-2.

Marbach, P. Priority service and max-min fairness. *IEEE/ACM Transactions on Networking*, 11(5):733–746, October 2003. doi:10.1109/TNET.2003.818196.

Marvell. Marvell Xelerated X11 family of network processors. Product brief, Marvell, 2012. Accessed on: 26 May 2019.
URL `https://www.electronicsdatasheets.com/download/52529f1be34e242f50ea9d73.pdf`

Marvell. Marvell Xelerated HX300 family of network processors. Product brief, Marvell,

2013. Accessed on: 27 May 2019.
URL https://www.electronicsdatasheets.com/download/52529f1be34e242f50ea9d71.pdf

Marvell. OCTEON III CN7XXX family of multi-core MIPS64 processors. Product overview, Marvell, 2019. Accessed on: 11 May 2019.
URL https://www.marvell.com/embedded-processors/infrastructure-processors/octeon-multi-core-mips64-processors/octeon-iii-cn7xxx/octeon-cn77xx/

McCarthy, J. A basis for a mathematical theory of computation, preliminary report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), pages 225–238. ACM, New York, NY, USA, 1961. doi:10.1145/1460690.1460715.

Meijer, S., Kienhuis, B., Walters, J., and Snuijf, D. Automatic partitioning and mapping of stream-based applications onto the Intel IXP Network Processor. In *Proceedings of the 10th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '07, pages 23–30. ACM, New York, NY, USA, 2007. doi:10.1145/1269843.1269847.

Mellanox. Mellanox IndigoNPS-400. Product Brief, Mellanox, Sunnyvale, USA, March 2017a. Accessed on: 27 May 2019.
URL http://www.mellanox.com/related-docs/prod_npu/PB_NPS-400.pdf

Mellanox. TILEncore-Gx36 intelligent application adapter. Product brief, Mellanox, 2017b. Accessed on: 26 May 2019.
URL http://www.mellanox.com/related-docs/prod_multi_core/PB_TILEncore-Gx36.pdf

Miller, D., Henderson, R., and Jelinek, J. Dynamic DMA mapping guide. Kernel documentation, Linux Kernel Organization, USA, 2017. Accessed on: 7 May 2019.
URL https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

Milner, R. Calculus of communicating systems. *Lecture Notes in Computer Science*, 1980. ISSN 0387102353.

Milner, R. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, pages 267–310, 1983. doi:10.1016/0304-3975(83)90114-7.

Milner, R. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0131149849.

Mittal, S. and Vetter, J. S. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, 47(4):69:1–69:35, 2015. doi:10.1145/2788396.

Mochel, P. and Murphy, M. sysfs - the filesystem for exporting kernel objects. Technical Document, Linux Kernel Organization, August 2011. Accessed on: 27 May 2019. URL `https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt`

National Instruments. PCI express: An overview of the PCI express standard. White paper, National Instruments, November 2014. Accessed on: 14 August 2017. URL `ftp://ftp.ni.com/pub/newsimages/2005/Introduction%20to%20PCI%20Express.pdf`

Navarro, G. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, March 2001. doi:10.1145/375360.375365.

Netronome. Netronome Network Flow Processor 6xxxx flow processor core programmer's reference manual. Users Guide, Netronome, USA, 2006. Confidential Property.

Netronome. Netronome Network Flow Processor 6xxx: network flow C compiler user's guide. Users Guide, Netronome, USA, 2008. Confidential Property.

Netronome. Netronome launches data plane hardware and software for SDN and NFV designs. Online document, Netronome, 2014. Accessed on: 2 August 2018. URL `http://tiny.cc/dpta7y`

Netronome. Netronome network flow processor NFP-6xxx-x C preliminary draft databook. Technical Manual, Netronome, USA, 2016. Confidential Property.

Netronome. Datapath programming tools. Product page, Netronome, 2018a. Accessed on: 9 May 2019. URL `https://www.netronome.com/products/datapath-programming-tools/`

Netronome. Netronome NFP-4000 flow processor. Product brief, Netronome, 2018b. Accessed on: 27 May 2019. URL `https://www.netronome.com/m/documents/PB_NFP-4000.pdf`

Nickolls, J., Buck, I., Garland, M., and Skadron, K. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, March 2008. doi:10.1145/1365490.1365500.

Null, L. and Lobur, J. *Computer Organization and Architecture*. Jones & Bartlett Learning, 4th edition, 2015. ISBN 0-13-607373-5.

NVIDIA. CUDA C best practices guide. Design guide, NVIDIA, 2018. Accessed on: 8 May 2019.
URL `https://www.cs.unc.edu/~prins/Classes/633/Readings/CUDA_C_Best_Practices_Guide%20v9.2.pdf`

Ohshima, S., Kise, K., Katagiri, T., and Yuba, T. Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In **Daydé, M., Palma, J. M. L. M., Coutinho, Á. L. G. A., Pacitti, E., and Lopes, J. C.**, editors, *High Performance Computing for Computational Science - VECPAR 2006*, pages 305–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-71351-7.

O'Neil, M., Tamir, D., and Burtscher, M. A parallel GPU version of the traveling salesman problem. Online, 2011. Accessed on: 14 September 2018.
URL `https://userweb.cs.txstate.edu/~mb92/papers/pdpta11b.pdf`

Open-MPI. Supported systems. Technical report, Open-MPI, 2019. Accessed on: 8 May 2019.
URL `https://www.open-mpi.org/faq/?category=supported-systems`

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krueger, J., Lefohn, A. E., and Purcell, T. J. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26:80–113, 2007. doi:10.1111/j.1467-8659.2007.01012.x.

Pennefather, S., Bradshaw, K., and Irwin, B. Design of a message passing model for use in a heterogeneous CPU-NFP framework for network analytics. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, pages 178–183. SATNAC, South Africa, September 2017.

Pennefather, S., Bradshaw, K., and Irwin, B. Exploration and design of a synchronous message passing framework for a CPU-NPU heterogeneous architecture. In *IEEE 32nd International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 46–56. IPDPS, Canada, 2018a.

Pennefather, S., Bradshaw, K., and Irwin, B. Extending the NFComms framework for bulk data transfers. In *South African Institute of Computer Scientists and Information Technologists (SAICSIT) 2018*. SAICSIT, South Africa, 2018b.

Pennefather, S., Bradshaw, K., and Irwin, B. Real-time geotagging and filtering of network data using a heterogeneous NPU-CPU architecture. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC) 2018*, pages 230–236. 2018c. ISBN 978-0-620-81022-7.

Pennefather, S., Bradshaw, K., and Irwin, B. Design and evaluation of bulk data transfer extensions for the NFComms framework. *South African Computer Journal (SACJ)*, 2019a. Under review.

Pennefather, S., Bradshaw, K., and Irwin, B. A synchronous message passing framework for the CPU-NPU heterogeneous architecture. *Concurrency and Computation, Practice and Experience*, 2019b. Under review.

Petersson, K. Syntax and semantics of programming languages. Technical report, Department of Computer Science, University of Goteborg Chalmers, 1997. Lecture Notes. Accessed on: 8 December 2018.
URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.4077&rep=rep1&type=pdf`

Petri, C. A. Kommunikation mit Automaten. Ph.D. thesis, Universitat Hamburg, 1962. PhD Thesis.

Pike, R. Go at Google: language design in the service of software engineering. In *SPLASH 2012 Conference*. Tucson, Arizona, USA, October 2012. Modified vesion of presented keynote. Accessed on: 7 May 2019.
URL `https://talks.golang.org/2012/splash.article`

Pinchart, L. Mastering the DMA and IOMMU APIs. Conference presentation, Embedded Linux Conference, San Jose, 2014. Accessed on: 7 May 2019.
URL `https://elinux.org/images/4/49/20140429-dma.pdf`

Prasad, R., Jain, M., and Dovrolis, C. Effects of interrupt coalescence on network measurements. In **Barakat, C. and Pratt, I.**, editors, *Passive and Active Network Measurement*, pages 247–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24668-8.

Pugmire, D., Monroe, L., Connor Davenport, C., DuBois, A., DuBois, D., and Poole, S. NPU-based image compositing in a distributed visualization system. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):798–809, July 2007. doi:10.1109/TVCG.2007.1026.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, June 2014. ISSN 0163-5964. doi:10.1145/2678373.2665678.

Rashti, M. J., Green, J., Balaji, P., Afsahi, A., and Gropp, W. Multi-core and network aware MPI topology functions. In **Cotronis, Y., Danalis, A., Nikolopoulos, D. S., and Dongarra, J.**, editors, *Recent Advances in the Message Passing Interface*, pages 50–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-24449-0.

Ravoori, R., Muthuraj, S., and Biddika, M. A deeper look at PCIe 3.0 protocol. White paper, SION Semiconductors, February 2017. Accessed on: 27 May 2019.
URL `http://www.sionsemi.com/whitepapers/pcie-overview.html`

Rocki, K. and Suda, R. Accelerating 2-Opt and 3-Opt local search using GPU in the travelling salesman problem. In *the 2012 International Conference on High Performance Computing and Simulation (HPCS 2012)*, pages 489–495. July 2012. doi:10.1109/CCGrid.2012.133.

Roscoe, A. W. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 0136744095.

Roscoe, A. W. *Understanding Concurrent Systems.* Springer Science & Business Media, Oxford, UK, 2010. ISBN 978-1-84882-258-0.

Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. ISBN 0136042597, 9780136042594.

Salah, K. To coalesce or not to coalesce. *AEU - International Journal of Electronics and Communications*, 61(4):215 – 225, 2007. doi:10.1016/j.aeue.2006.04.007.

Sandia National Laboratories. Portals 4.0. Website, Sandia National Laboratories, 2019. Accessed on: 8 April 2019.
URL `http://www.cs.sandia.gov/Portals/portals4.html`

Schneider, S. *Concurrent and Real Time Systems: the CSP Approach*. John Wiley & Sons, Ltd., 1999. ISBN 978-0471623731.

Schroeder, T. Peer-to-peer and unified virtual addressing. Presentation slides, NVIDIA, 2011. Accessed on: 8 May 2019.
URL `https://developer.download.nvidia.com/CUDA/training/cuda_webinars_GPUDirect_uva.pdf`

Scott, D. and Strachey, C. Toward a mathematical semantics for computer languages. Technical monograph, Oxford University Computing Laboratory, Programming Research Group, June 1971. Accessed on: 26 May 2019.
URL `https://www.cs.ox.ac.uk/files/3228/PRG06.pdf`

Sellitto, M. and Schaa, D. Opencl. Presentation slides, Northeastern University Computer Architecture Research Lab (NUCAR), 2012. Accessed on: 9 May 2019.
URL `http://www.ece.neu.edu/groups/nucar/Analogic/Class5-B-Events.pdf`

Shah, N., Ravindran, K., Plishker, W., and Keutzer, K. NP-Click: a programming model for the Intel IXP1200. In **Crowley, P.**, editor, *Proceedings of the 2nd Workshop on Network Processors (NP-2), 9th International Symposium on High Performance Computer Architectures (HPCA)*, pages 181–201. 2003. doi:10.1.1.101.847.

Shah, N., Ravindran, K., Plishker, W., and Keutzer, K. NP-Click: a productive software development approach for network processors. *IEEE Micro*, 24:45–54, 09 2004. doi:10.1109/MM.2004.53.

Shanley, T. and Anderson, D. *PCI System Architecture*. Addison-Wesley Longman Publishing, Boston, MA, USA, 4th edition, 1999. ISBN 0201309742.

Silberschatz, A., Galvin, P., and Gagne, G. *Operating System Concepts*. John Wiley and Sons Inc., 7th edition, 2005. ISBN 0471694665.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, second edition, 1998. ISBN 0262692155.

Solomon, R. PCI express basics and background. Presentation, PCI SIG, 2014. From PCIe Technology Seminar. Accessed on May 2019.
URL `https://pcisig.com/sites/default/files/files/PCI_Express_Basics_Background.pdf#page=26`

Stallings, W. *Computer Organization and Architecture, Design and Performance*. Pearson Education, 8th edition, 2010. ISBN 0-13-607373-5.

Stark, G. Introduction to the NFP architecture. Presentation, Netronome, San Jose, November 2015. Accessed on: 8 May 2018.
URL `http://open-nfp.org/media/pdfs/P4DevCon_NFPArchIntro.pdf`

Stefanov, T., Zissulescu, C., Turjan, A., Kienhuis, B., and Deprettere, E. System design using Kahn process networks: the Compaan/Laura approach. In *Proceedings of the Conference on Design, Automation and Test in Europe* , volume 1. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2085-5.

Stone, J., Gohara, D., and Shi, G. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010. doi:10.1109/MCSE.2010.69.

Stuart, W. The Joy of Micro-C. Netronome, Online, December 2014. Accessed on: 14 September 2017.
URL `https://open-nfp.org/m/documents/the-joy-of-micro-c_fcjSfra.pdf`

Taylor, L. Gccgo in GCC 4.7.1. Electronic, Google, July 2012. Accessed on: 7 February 2019.
URL `http://blog.golang.org/gccgo-in-gcc-471`

The Go Programming Language. How to write Go code. The Go Programming Language, 2015. Accessed on: 8 May 2019.
URL `https://golang.org/doc/code.html`

Todorova, M., Nisheva-Pavlova, M., Penchev, G., Trifonov, T., Armyanov, P., and Semerdzhiev, A. The Go programming language: Characteristics and capabilities. In *Annual of "Informatics" Section Union of Scientists in Bulgaria*, volume 6, pages 76–85. Faculty of Mathematics and Informatics, Sofia University, Sofia, Bulgaria, 2013. Accessed on: 8 May 2019.
URL `http://old.usb-bg.org/Bg/Annual_Informatics/2013/SUB-Informatics-2013-6-076-085.pdf`

Tsoi, K. H. and Luk, W. Axel: a heterogeneous cluster with FPGAs and GPUs. *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 115–124, February 2010. doi:10.1145/1723112.1723134.

Tsoi, K. H., Tse, A. H., Pietzuch, P., and Luk, W. Programming framework

for clusters with heterogeneous accelerators. *ACM SIGARCH Computer Architecture News*, pages 53–59, September 2010. doi:10.1145/1926367.1926377.

University of Kent. C++ CSP barrier class reference. API Documentation, University of Kent, 2007. Accessed on: 26 May 2019.
URL `https://www.cs.kent.ac.uk/projects/ofa/c++csp/doc/classcsp_1_1_barrier.html`

USC center for high-performance computing. Message passing interface (MPI). Website, University of Southern Calafornia (USC), 2018. Accessed on: 26 May 2019.
URL `https://hpcc.usc.edu/support/documentation/message-passing-interface/`

Vuduc, R., Chandramowlishwaran, A., Choi, J., Guney, M., and Shringarpure, A. On the limits of GPU acceleration. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 13–13. USENIX Association, Berkeley, CA, USA, 2010. Accessed on: 27 May 2019.
URL `https://www.usenix.org/presentation/limits-gpu-acceleration`

Walker, D. Standards for message-passing in a distributed memory environment. Conference, Open, 8 1992. Accessed on: 7 May 2019.
URL `https://www.osti.gov/biblio/7104668`

Welch, P. and Austin, P. CSP for Java (JCSP). API Documentation, University of Kent, 2009. Accessed on: 27 May 2019.
URL `https://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/org/jcsp/lang/Barrier.html`

Wheeler, B. A new era of network processing. White paper, The Linley Group, October 2013. Accessed on: 27 May 2017.
URL `http://www.linleygroup.com/uploads/ericsson_npu_white_paper.pdf`

Xilinx. Xilinx PCI express interrupt debugging guide. Guide, Xilinx, 2014. Accessed on: 27 May 2019.
URL `https://www.xilinx.com/support/answers/58495.html`

Zhou, S., Stumm, M., Li, K., and Wortman, D. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel & Distributed Systems*, 3(5):540–554, September 1992. doi:10.1109/71.159038.

Appendices

A

Code Listings

Throughout this dissertation, code listings are used to describe application functionality and to highlight features and errors identified during development. The majority of these listings are suitably small to be presented in the main text without disrupting document flow. Some listings crucial to the understanding of the research were considered too large to be placed inline and are instead presented in this appendix and referred to from the relevant parts of the main text.

A.1 CSP Compliance

This section consists of the listings describing the source code used to produce the NFP implementations of the sample problems described by Hoare (1978) and further discussed in Section 3.3. Listings A.1 and A.2 represent the source code for the *west* and *east* processes respectively. Each of these was implemented as a separate process on the same island (island 33) within the NFP architecture. Process *west* was deployed to ME 0 while process *east* was deployed to ME 2. To provide a workspace in which to implement the CSP examples, an additional process described in Listing A.3 was included. This process, the application process, was deployed to ME 1.

```
1  //This is process West to be used in implementing the CSP examples. Process
2  //West acts as the producer in most examples provided by Hoare.
3  /*************************************************************************/
4  #include <nfp.h>
5  #include <nfp_cls_reflect.h>
6  //Address location of the applicaion ME and its relevant registrs.
7  #define APP_LOC 5
8  #define S_APP 15
9  #define R_APP 1
10
11 __declspec(write_reg)        unsigned int r_west;
12 __declspec(visible)          SIGNAL       s_west;
13 //-------------------------------------------------------------------------
14 void produce(void) {
15     __assign_relative_register((void*)&s_west, 15);
16     __assign_relative_register((void*)&r_west, 1);
17      //r_west is the outpout register for the producer model. The value here
18      //is altered for the different exercises.
19      r_west = '*';
20      cls_reflect_write_sig_remote(&r_west, APP_LOC, R_APP, S_APP, 1);
21      __wait_for_all(&s_west);
22 }
23 //-------------------------------------------------------------------------
24 int main(void) {
25     if (__ctx() == 0) {
26         while (1) produce();
27     }
28   return 0;
29  }
30  /*************************************************************************/
```

Listing A.1: Source code for process *west* used in the initial investigation of the communication framework.

```
1  //This is process East to be used in implementing the CSP examples. Process
2  //East acts as the consumer in most examples provided by Hoare.
3  /*************************************************************************/
4  #include <nfp.h>
5  //Address location of the applicaion ME and its relevant registrs.
6  #define APP_LOC 5
7  #define S_APP 14
8  #define APP_ISLAND 33
9
10 __declspec(visible read_reg)          unsigned int r_east;
11 __declspec(visible)                   SIGNAL       s_east;
12 __declspec(ctm export scope(global)) int          east_store;
13 //-------------------------------------------------------------------------
14 void consume(void) {
15     unsigned int s_west = APP_ISLAND<<24|APP_LOC<<9|S_APP<<2;
16     __assign_relative_register((void*)&s_east, 15);
17     __assign_relative_register((void*)&r_east, 1);
18
19     wait_for_all(&s_east);
20     //east_store is a memory locaion where the last value recvieved is
21     //stored. This location can be polled to confirm functionality of the
22     //system.
23     east_store = r_east;
24     __asm ct[interthread_signal, --, s_west, 0, 1]
25 }
26 //-------------------------------------------------------------------------
27 int main(void) {
28
29   if(__ctx() == 0) {
30     while (1) consume();
31   }
32   return 0;
33  }
34  /*************************************************************************/
```

Listing A.2: Source code for process *east* used in the initial investigation of the communication framework.

```
1  //The connection process which will mediate messages between East and West.
2  //Exercises described bt Hoare and goCSP will be presented here.
3  /*************************************************************************/
4  #include <nfp.h>
5  #include <nfp_cls_reflect.h>
6  __declspec(visible read_reg)          unsigned int r_frm_west;
7  __declspec(visible read_reg)          unsigned int r_frm_east;
8  __declspec(visible)                   SIGNAL       s_frm_west;
9  __declspec(visible)                   SIGNAL       s_frm_east;
10 /*************************************************************************/
11 //Function waits for west ME to assert signal s_frm_west and then reads and
12 //returns value from register r_frm_west
13 int get_from_west(void) {
14   wait_for_all(&s_frm_west);
15   return r_frm_west;
16 }
17 //Function construsts the address for signal s_west on the west ME and
18 //issues an interthread signal.
19 void ack_west(void) {
20   unsigned int s_west = 33<<24|4<<9|0<<6|15<<2;
21   __asm ct[interthread_signal, --, s_west, 0, 1]
22 }
23 //Function insets the supplied value into a write register and then issues a
24 //reflect write request targeting the east ME.
25 void send_to_east(unsigned int value) {
26   __declspec(write_reg) unsigned int val = value;
27   cls_reflect_write_sig_remote(&val, 6, 1, 15, 1);
28 }
29 //Function simply waits for the east ME to assert signal s_frm_east
30 void ack_east(void) {
31   __wait_for_all(&s_frm_east);
32 }
33 /*************************************************************************/
34 //Problem: Write a process X to copy characters output by process west to
35 //process, east.
36 void COPY(void) {
37   volatile unsigned int val;
38   while(1) {
39       val = get_from_west();
40       ack_west();
41       send_to_east(val);
42       ack_east();
43     }
```

```
44 }
45 //Problem: Adapt the previous program [COPY] to replace every pair of
46 //consecutive asterisks '**' by an upward arrow '\^'.
47 void SQUASH(void) {
48   unsigned int val, val2;
49   while(1) {
50     val = get_from_west();
51     ack_west();
52     if (val != '*') {
53       send_to_east(val);
54       ack_east();
55     } else {
56       val2 = get_from_west();
57       ack_west();
58       if (val2 != '*') {
59         send_to_east(val);
60         ack_east();
61         send_to_east(val2);
62         ack_east();
63       } else {
64         val = '^';
65         send_to_east(val);
66         ack_east();
67       }
68     }
69   }
70 }
71 /*****************************************************************************/
72 //Main program loop
73 //-----------------------------------------------------------
74 int main(void) {
75   if (__ctx() == 0) {
76     __assign_relative_register((void*)&s_frm_west, 15);
77     __assign_relative_register((void*)&s_frm_east, 14);
78     __assign_relative_register((void*)&r_frm_west, 1);
79     __assign_relative_register((void*)&r_frm_east, 2);
80     //COPY();
81     //SQUASH();
82   }
83   return 0;
84 }
85 /*****************************************************************************/
```

Listing A.3: Source code for the intermediary engine between the *east* and *west* processes.

A.2 Barrier Implementation

Source code relating to the barrier implementation approach undertaken in Section 3.3.4 is described here. The source core is split into two components with Listing A.4 containing the source code for the barrier related functions called by the application presented in Listing A.5 which was executed on the two microengines under test.

```
1  #include "barrier.h"
2  #include <nfp.h>
3
4  Barrier init_barrier(volatile __declspec(mem addr40) void * addr) {
5      Barrier b;
6      b.addr_hi = ((unsigned long long int)addr >> 8) & 0xFF000000;
7      b.addr_lo =  (unsigned long long int)addr       & 0xFFFFFFFF;
8      return b;
9  }
10 void barrier_block(Barrier barrier) {
11     __declspec(read_reg) int b_val = 1;
12     SIGNAL_PAIR return_signal;
13 //Decrement the barrier, and get the current barrier value.
14     __asm {
15         mem[test_subsat, b_val, barrier.addr_hi, << 8, barrier.addr_lo, 1],
16             sig_done[return_signal];
17         ctx_arb[return_signal];
18     }
19 //If we are not the last process, spin until the barrier is depleted (is 0).
20     if((b_val-1) != 0) {
21         do {
22           __asm {
23               mem[read32, b_val, barrier.addr_hi, << 8, barrier.addr_lo, 1],
24                   ctx_swap[return_signal];
25           }
26         } while(b_val != 0);
27     }
28 }
```

Listing A.4: Source code for the barrier related functions.

```
1  #include <nfp.h>
2  #include "barrier.h"
3
4  __declspec(ctm32 export scope(global)) int barrier_one = 2;
5
6  int main()
7  {
8      if(__ctx() == 0)
9      {
10         Barrier b = init_barrier(&barrier_one);
11         barrier_block(b);
12     }
13     return 0;
14 }
```

Listing A.5: Source code executed on the two microengines tested.