SACJ 32(2) December 2019
Research Article

Design and evaluation of bulk data transfer extensions for the NFComms framework

Sean Pennefather, Karen Bradshaw, Barry Irwin

Department of Computer Science, Rhodes University, South Africa

ABSTRACT
We present the design and implementation of an indirect messaging extension for the existing NFComms framework that provides communication between a network flow processor and host CPU. This extension addresses the bulk throughput limitations of the framework and is intended to work in conjunction with existing communication mediums. Testing of the framework extensions shows an increase in throughput performance of up to 268× that of the current direct message passing framework at the cost of increased single message latency of up to 2×. This trade-off is considered acceptable as the proposed extensions are intended for bulk data transfer only while the existing message passing functionality of the framework is preserved and can be used in situations where low latency is required for small messages.

Keywords:

Categories: • **Computer systems organization ∼ Heterogeneous (hybrid) systems • Software and its engineering ∼ Cooperating communicating processes • Software and its engineering ∼ Concurrency control • Software and its engineering ∼ Message passing**

Email:
Sean Pennefather pennefather.sean@gmail.com (CORRESPONDING),
Karen Bradshaw k.bradshaw@ru.ac.za,
Barry Irwin b.irwin@ru.ac.za

Article history:
Received: 15 Mar 2019
Accepted: 16 Oct 2019
Available online: 20 Dec 2019

1 INTRODUCTION

The rise in availability of heterogeneous computing architectures such as the Graphics Processing Unit (GPU) (Mittal & Vetter, 2015) and Field-Programmable Gate Array (FPGA) (Andrews et al., 2004; Tsoi & Luk, 2010) has meant that in many application domains, the strengths of the respective architectures can be leveraged to improve performance. Applications that operate on such heterogeneous systems can potentially distribute work according to these strengths resulting in better performance. As the field of heterogeneous computing continues to grow (Zahran, 2017), utilising alternative architectures such as Network Processing Units (NPU) for computation has attracted increased interest (Bos & Huang, 2006; Shuhui, Rongxing, & Xuemin, 2013). Given that most NPUs (Netronome, 2018; Wheeler, 2013) consist of multiple independent general processing units, being able to leverage such processing capabilities in more general processing applications warrants further investigation.

Traditionally, network processors were largely intended to help servers facilitate the increasing demand for higher network throughput while supporting more complex and adaptive network protocols (Wheeler, 2013). Such devices often exist in advanced network routing engines or embedded as PCIe network cards to be interfaced with directly by a host. These devices are generally designed to operate in computing environments requiring low latency and high throughput (Ahmadi & Wong, 2006). Introducing a NPU based PCIe network card into an existing host does allow for the development of heterogeneous systems however, NPUs such as the Network Flow Processor (NFP) (Netronome, 2018), developed by Netronome[1], usually execute applications independently of the host, acting as a pre- or post-processing stage for network traffic to reduce host workload.

As described by Lastovetsky and Dongarra (2009), heterogeneous systems always consist of multiple processing elements and a communication framework interconnecting such elements. When considering heterogeneous platforms involving architectures such as an NPU, the communication framework provided is usually designed for explicit use with domain specific applications, often being represented as virtual network interfaces (Netronome, 2018). Though well suited to communicating network traffic, this framework does not provide an efficient interface for communicating generic data between the component architectures that make up a heterogeneous CPU-NPU system. To mitigate this weakness, a prototype communication framework was designed and implemented (Pennefather, Bradshaw, & Irwin, 2018a) to allow processes operating on the NFP to communicate with host processes at runtime. Though functional, the observed throughput speeds were considered unsuitable for viable use in any meaningful application.

This work is an extension of previously published work (Pennefather, Bradshaw, & Irwin, 2018b) which builds on the existing framework to provide additional functionality for use in bulk transfer operations. The extension includes further application testing and an updated performance evaluation with the results differing slightly from prior testing due to the inclusion of additional error detection routines to help maintain a reliable communication medium.

The remainder of the paper is structured as follows. Section 2 provides an overview of the current framework as well as a brief summary of the existing functionality and throughput capabilities. A discussion on the requirements and design of the framework extensions is given in Section 3 and is followed by a review of their implementation in Section 4. Testing of the extensions is presented in Section 5. The paper concludes in Section 6.

2 FRAMEWORK OVERVIEW

To provide a medium for communication between the host and the NFP, a message passing framework was designed and a prototype implemented (Pennefather, Bradshaw, & Irwin, 2017, 2018a). These implementations focused on providing a reliable framework for transmitting messages between the two architectures at runtime with an average message latency of 14.66 μs. During conceptualisation of the framework, three basic requirements that the implementation would be expected to achieve were identified.

The first requirement is that communication between the host and the NFP be performed via synchronous message passing conforming to the CSP formalisms as described by Hoare (1978). The

[1] https://www.netronome.com/

motivation behind this design requirement was largely to give the proposed system a foundation based on a well understood and tested formalism. During the conceptualisation stage, it was still uncertain as to whether the NFP architecture would even be able to support the mechanisms necessary to implement such a framework. By basing the design on CSP, it was possible to distil the message passing transactions into a series of basic communication components and prove that the equivalent components either existed or could be implemented on the NFP architecture (Pennefather, 2017).

The second requirement is that the framework be scalable. The number of channels that could be established between the two architectures should not be limited by the number of threads reserved for the message passing framework. By making this a requirement at conceptualisation, the framework could be designed with scalability in mind. This is of particular importance when considering that the resources available for the framework are finite and so a mechanism for handling a non-finite number of blocking messages without introducing deadlocks or livelocks would need to be considered.

A third requirement is that the framework be designed to interface with the Go programming language developed by Google. This was put forward as an initial requirement due to interest from external parties as well as the Go programming language having native support for concurrent communication via channels inspired by CSP (Kappler, 2016).

Before the implementation of the initial prototype framework can be discussed, it is important to present a brief overview of the NFP architecture to help contextualise many design decisions in the development of both the initial framework and the subsequent extensions.

2.1 NFComms Framework

In previous work (Pennefather et al., 2017, 2018a) the conceptual model of the NFComms communication framework was explored. As discussed earlier, this exploration initially focused on confirming that such a model could feasibly be implemented on the NFP architecture provided it conformed to the CSP formalism. This feasibility study was then used to design a communication scheme that handled the differences in symmetry and naming between the two component architectures. As one of the initial requirements was to allow the framework to interface with the Go programming language, the endpoint interfaces to the framework were presented as channel constructs that could interact with internal managers, which in turn would be responsible for handling the translation and transmission of the messages over the PCIe.

Though covered in previous work, a brief overview of the current framework is presented in Figure 1 to help contextualise the current research. The framework can be broken into four distinct regions with two residing on the host, and two residing on the NFP. The host based regions are user space and kernel space with the user space component being responsible for interfacing with the host component of the application. This interaction occurs through the use of either the NFComms library designed for the Go programming language, or a low level C API. The NFComms library provides the handlers required to select an NFP PCIe device, create connections targeting specific contexts operating on the NFP architecture, and initialise the host component of the framework.

Conceptually the communication between the architectures occurs through the use of mailboxes that are duplicated on both the host and NFP device. The underlying approach taken for communication

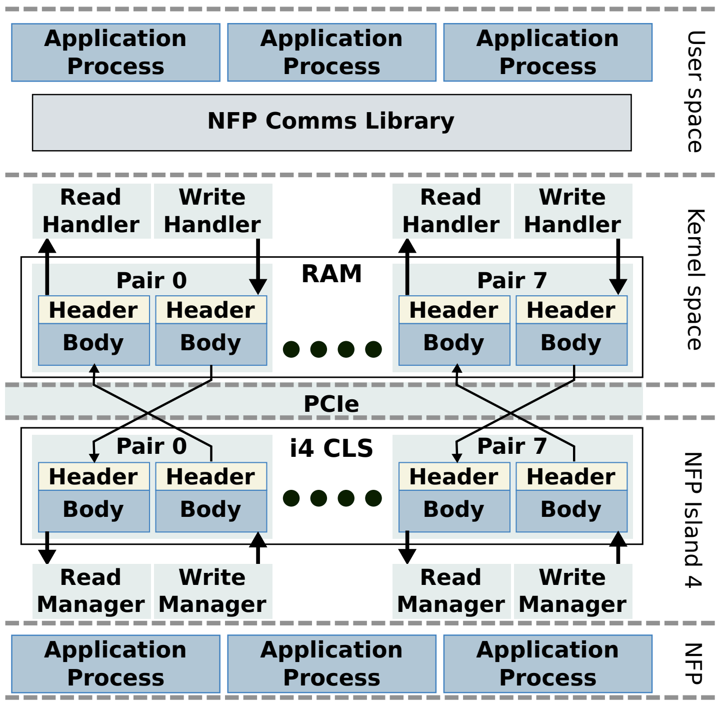


Figure 1: Overview of the NFComms framework

is thus moving any messages inserted into a mailbox to its corresponding counterpart on the opposing architecture.

2.2 Performance Concerns

Though the initial implementation of the framework was shown to be functional, an analysis of the maximum achievable throughput via the framework recorded a peak sustained transmission rate of 114.3 Mb/s which was considered unsuitable for any bulk transfer operations. In previous work (Pennefather et al., 2018b) it was found that the poor performance was chiefly due to the very restrictive 56 byte message size enforced by the 14 register transfer limit discussed. 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 needed.

3 EXTENSION DESIGN

At present, all communication between the two architectures occurs through the synchronous transmission of messages where each message is passed using a single reflect operation. Given that PCIe operates by passing transaction layer packets (TLP) that can support a payload of up to 4096 bytes (Lawley, 2014), the current implementation is only able to use approximately 1.4% of the available bandwidth for a single message. This is further compounded by the synchronous

Table 1: Memory regions and their respective characteristics.

Memory Type	Capacity	Latency (in execution cycles)
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

nature of these messages, requiring the back propagation of an acknowledgement signal to indicate completion.

Though a workaround to the reflect operation limitation could be investigated such as allowing for multiple reflect operations to occur within a message transaction, the limited resources available to a single context imply that such a modification may, in the best case, double current throughput resulting in a bulk transfer speed that is still largely unsuitable for applications. Furthermore, all communication from any memory engines or any other dedicated processing units within the NFP architecture must occur via these transfer registers and thus they should be reserved sparingly. It is however acknowledged by the authors that being able to transmit a large body of data from or to the NFP would be beneficial despite the data associated with these bulk transactions not being able to be stored in the limited memory of a single context. It is thus proposed that such transactions interact with data stored in one of the bulk memory regions on the NFP card.

3.1 Conceptual Design

Given that an initial goal of the message passing framework is for all communication to occur through synchronous events, these bulk memory transactions should adhere to the same restriction. Memory within the NFP does not conform to standard models supported by most common computing devices. Outside of the scope of a single microengine[2], there are four separate types of memory addressable on the NFP, listed in Table 1.

Each memory type supports a different latency that is inversely proportional to its capacity. For this component of the framework, the three memory locations of particular interest are CTM, IMEM, and EMEM as they have capacities large enough to host bulk data transactions. To perform an indirect transaction, an address for the intended memory region is required. The address can then be supplied as part of a DMA descriptor, informing the DMA engine where either the source or destination memory region is located.

To enforce synchronisation as well as acquire the correct memory address for a transaction, the intent of the proposed indirect message passing extension is to have the transaction involve an NFP process rather than statically assigning address locations on the NFP device. Though this adds an additional step to the transaction, it allows the NFP process to be responsible for supplying the address associated with the intended source or destination of a transfer event. Synchronisation can

[2]A microengine advertises a memory model that conforms to the standard Harvard architecture.

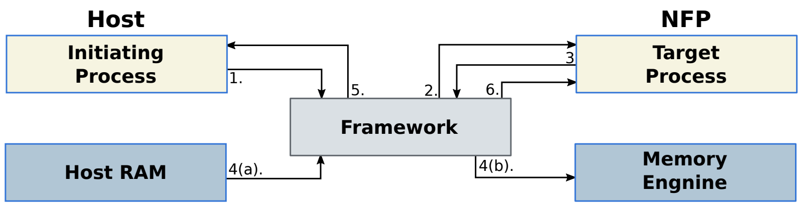


Figure 2: Overview of the events involved in an indirect message transmission event

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

Conceptually, a write event from the host to the NFP follows the abstracted sequence presented in Figure 2. This transaction begins with a message being submitted to the framework in event ①. The framework then 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 2 except that the originating process resides on the NFP device. For these transactions however, there is an additional message pass event that moves the destination address for the message from the host to the card. This is because in both instances the actor responsible for moving the actual data is the DMA engine of the NFP device. For transactions originating from the host, this address is submitted as part of the original message. 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 pass event could impact the time taken to resolve the transaction, but should become relatively insignificant when transmitting suitably large messages.

4 BULK TRANSFER IMPLEMENTATION

An important consideration when implementing this extension was for the resulting framework to have minimal impact on its current functionality, allowing for backwards compatibility with existing applications. To allow applications to utilise the extensions however, two additional library functions for both the NFP and the host were added. For the NFComms library these are `sendInd()` and `ReceiveInd()`, while for the NFP the equivalent functions conform to the existing naming scheme and are `channel_write_ind()` and `channel_read_ind()`. In all cases the parameters these functions take in are the same as their direct communication counterparts with the exception of an additional size field to indicate the size (in bytes) of the message to be transmitted. Beyond the inclusion of these additional functions, no changes to the framework should be visible to or affect

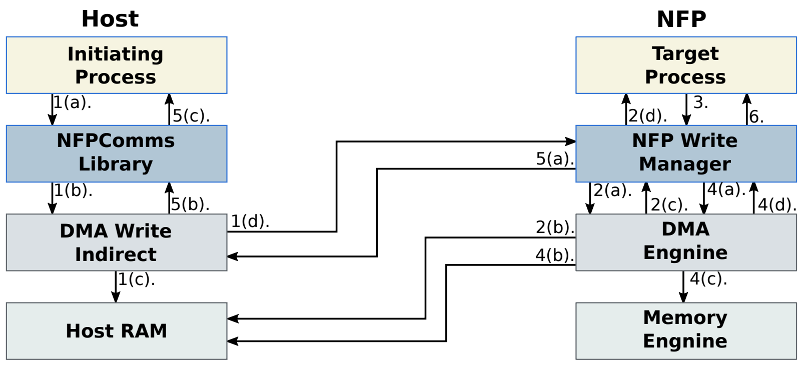


Figure 3: Detailed overview of the events and actors involved in an indirect message transmission event

existing applications utilising it.

Rather than detailing the modifications necessary to extend the existing framework to support indirect message passing, the implementation is better described in conjunction with Figure 3 that details the sequence of events necessary to complete a single transition originating from the host. The implementation can thus be better presented by discussing what operations are necessary to resolve each stage in the event. For messages originating from the NFP, only differences between the two directions are highlighted as care was taken to make many of the underlying mechanisms compatible with both directions. To aid in the description, the extensions are divided into two parts: the host and the NFP.

4.1 Host Library Interactions

Within the host, the initiation of an indirect message pass event begins with a `sendInd()` call to the NFComms library as shown by event (1a) in Figure 3. This function requires a valid target ID that has previously been subscribed to the active `NfpComms` object and a reference to a declared object with a static size. This object contains the data that the message pass 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. Details relating to the how the target ID is defined are discussed in (Pennefather et al., 2018a).

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

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

message pass 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 Go types. 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 can be created.

Once a reference to the memory location is established and submitted to the underlying API, a call is made to the NFComms driver via the IOCTL interface as presented by event (1b). For the next stage of the transaction, the existing message passing framework is used as the underlying medium to submit data to the NFP device. The information submitted however, is not the actual message data, but rather a series of small page lists containing references to where the data can be found within the host memory.

4.2 DMA transactions

As the NFP device does not support scatter-gather operations when interfacing with host memory[4], the proposed approach is to implement a software level variant of this operation. Algorithm 1 provides a simplified overview of the core algorithm responsible for achieving this, represented by events (1c) and (1d) in Figure 3. For the sake of brevity, all error checking in both the initial data and transmission events has been omitted and operations are only described broadly.
The first notable point of this algorithm is that the gather list described in step **(1)** can only contain six elements due to the size limitations of the direct message passing framework discussed previously. Each list entry contains the basic information needed by the NFP DMA controller to move a single page of memory from the host to the NFP device which includes:

- number of bytes within the page containing the message data
- page offset where the message data begins
- physical address of the page compatible with the IOMMU.

The goal of this algorithm is to acquire all pages associated with the message body and then, iteratively pin each page so that it cannot be moved as well as mark it for a DMA originating from the NFP device. As each page is acquired and pinned in event (1c), it is inserted into the gather list until either the whole message has been pinned or the gather list is full. Once the internal loop containing steps **(6 - 11)** completes, the scatter list is packed as a message to be submitted to the NFP device in event (1d). 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 existing message passing functionality of the framework.

This function blocks until an acknowledgement message for the operation is received from the NFP at which point if there are still outstanding pages to submit, the process is repeated. As part of

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

```
1. Create a gather list for holding six page entries;
2. Fetch user-space message struct via copy_from_user();
3. Calculate page offset of supplied user address;
4. Use copy_from_user() to fetch underlying pages;
while there are outstanding pages to copy do
    5. Record the gather list size to be full (six);
    for each element in the gather list do
        if there are no more outstanding pages then
            6. Update gather list size with current loop iteration;
            7. Exit the current loop;
        end
        8. Use dma_map_page() to map next outstanding page for DMA and update current gather list
          entry address with the returned physical address;
        9. Update the current gather list entry with the outstanding page offset;
        10. Update the current gather list entry with the outstanding page size;
        11. Update outstanding page count;
    end
    12. Pack gather list into a message struct with the gather list size;
    13. Add current outstanding pages count and outstanding bytes count to message;
    14. Send message to NFP using direct messaging framework;
    15. Sleep until message is acknowledged;
    16. Check and handle errors;
end
```

Algorithm 1: Pseudocode for performing a bulk data transfer to or from the NFP device

the acknowledgement, the NFP submits information about its current status regarding the transaction such as the number of bytes it has processed and the number of bytes outstanding. The state of the NFP, which can indicate if an error has been detected in the transaction, is also submitted to the host as part of the acknowledgement.

This information is used by the host on subsequent iterations of the transaction to either continue as normal, repeat the previous iteration of the algorithm, or characterise the error and propagate it back to the user-space application.

Once all pages have been sent to the NFP, the host sleeps, waiting for an acknowledgement message from the NFP as part of event 5a, indicating that all DMA operations have completed successfully. Sleeping is performed using wait queues (Sovani, 2005) with the host component of the indirect message transaction subscribing to the queue after each submission to the NFP. The acknowledgement of a message submission triggers the wakeup condition on the queue at which point the host thread servicing the transaction returns to the running state. This approach allows the host to service multiple indirect message transactions at once while sharing a single MSIX number. Once the acknowledgement has been received, the host de-registers and unpins all the pages allocated to the NFP for DMA. The indirect write routine then returns, unblocking the calling routine associated with the NFComms library. Once unblocked, the NFComms library signals the host application of the successful transmission by unblocking the initial `sendInd()` call. At this point (provided the `sendInd()` returned without error), the host component of the application can safely assume that the data have been transmitted to the NFP architecture.

4.3 NFP Manager Transactions

Continuing on from the point where the first instance of event $\textcircled{1d}$ occurs, one of the eight NFP read managers described in Figure 1 is signalled as part of step **(13)**. At this point, given that the message was received over the direct message passing framework, the message could be either a direct or an indirect message and thus, the procedure of acquiring the message from the host is the same for both cases.

To distinguish the start of an indirect transfer from a direct message event, the message header is extended to support a type field, currently used to indicate if the message is part of a direct or indirect transfer. On receipt of a message, the managing thread uses this field's value to either forward the message to the destination process as a direct transaction or initiate the indirect transaction routine. Algorithm 2 provides an overview of how the indirect read routine handles a message transaction.

When the read routine begins, the destination address for the message body within the NFP architecture is still unknown. The read routine thus begins by requesting the address from the destination process. If the destination process is not ready to partake in the transaction, the routine blocks until either the destination is read 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)**. 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 Figure 3, for each occurrence of events $\textcircled{1c}$ and $\textcircled{1d}$ within the host component of the transaction, events $\textcircled{2a}$ - $\textcircled{2c}$ occur to complete each intermediate transfer. Within the NFP, these events are effectively represented by steps **(8-14)** in Algorithm 2 and involve relaying DMA read requests to the NFP DMA engine for each of the supplied host page addresses.

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 completed successfully. At this point, the transaction is considered complete and the indirect read routine returns, allowing the read manager to facilitate a new transaction.

4.4 Target Process Transactions

From the perspective of the target process, a host indirect read request begins when the process reaches the corresponding `channel_read_ind()` call. This function takes in a 40-bit memory address, a size (in bytes), and a register signal pair associated with the connection. When called, function blocks until the transaction is complete and begins by converting the supplied memory reference into a CPP target address to conform with the format defined in the NFP data book

```
if the request is for a bulk write 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 2: Pseudocode executed to perform a bulk read transaction on the NFP

(Netronome, 2016). Following this, a signal address that the calling manager can assert to conclude the transaction, is built and stored with the memory address.

With setup complete, the function sleeps on the supplied signal that is asserted by the manager when the message pass event begins. When woken this function submits a request to have the constructed memory and signal addresses as well as a destination address sent to the requesting read manager. Once sent, this function again sleeps but this time on the signal that was submitted to the manager. Once the signal is asserted, this function simply returns, unblocking the calling context and completing the message transaction event from the perspective of the destination process.

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 section focuses on recording the maximum achievable throughput of the implemented extensions for varying message sizes and process counts. The final section of the testing attempts to evaluate performance of the extensions using a previously published example (Pennefather et al., 2018a) employing direct message passing as a benchmark.

As noted in Section 3.1 the framework was designed to target a range of memory types however, for the testing of the extensions, all message transactions are 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, and proves the ability of the bulk transfer operations.

5.1 Latency Testing

Latency testing was performed using a basic application implementing a simple producer and 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 start-up 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 less than 56 bytes produced no significant variation in latencies. Testing was performed by transmitting 10000 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

Table 2: Average recorded message latencies in microseconds using direct and indirect message passing over 10000 runs

	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

overlap with the initial stage of the transmission being initiated by the producer. This would result in the timings recorded not correctly 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.

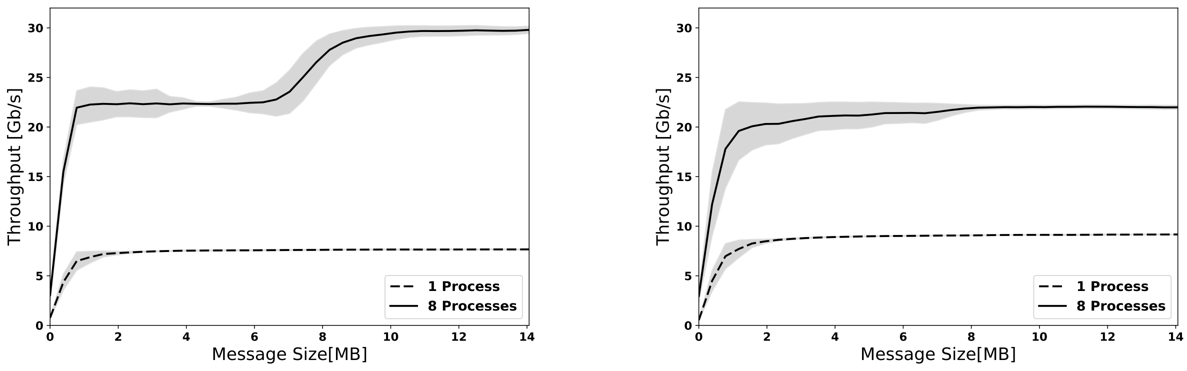
The test was repeated four times, recording latencies for each architecture and each messaging medium. The results of these four tests are presented in Table 2 as average latencies observed per test. Though care was taken to keep the tests uniform between the four configurations, tests involving indirect 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 2, the average latency for a direct message originating from the host is approximately double that of the equivalent operation using the indirect extensions. A similar pattern can be seen for transaction traversing in the opposite direction with the indirect operation recorded to be 1.73 time slower the the direct counterpart. These results differ slightly from the latencies for indirect communicaion recorded in (Pennefather et al., 2018b). This is due to the a collection of minor alterations being added to the framework to improve stability and error handling. The modifications are expected to reduce through performance slightly, but their impact to latency is minimal.

5.2 Throughput Testing

Given that the focus of the extensions described in Section 4 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: recording 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 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



(a) Host to NFP (b) NFP to Host

Figure 4: Recorded throughputs for indirect messaging transmissions

opposing architectures and all data transmitted from the source process were 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 4. 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.

Test configurations involving the eight NFP processes operated by having the host-based process either read or write 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 pass events were processed concurrently.

In Figure 4(a) the recorded throughput for sending data from the host to the NFP is recorded. In situations where only one message transmission operation was preformed, the throughput exhibited some minor fluctuations until the individual message payloads reached approximately 2 MB in size. From the 2 MB mark, increasing message payload sizes resulted in relatively negligible changes to throughput, settling at an average of about 7.8 Gb/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 at the 2 MB mark of approximately 20 Gb/s. However, the throughput began to further increase after the message payload size exceeded 8 MB.

Figure 4(b) shows the equivalent data as Figure 4(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 transfers originating from the NFP exhibit a slightly increased average throughput of 9 Gb/s for message sizes exceeding 8 MB.

Considering the throughput of the test configurations utilising eight processes, the variance in recorded timings shows a large degree of fluctuation when the message payload sizes less than 8 MB. Message sizes beyond the limit exhibit an average throughput of 22 Gb/s with very little variance. As noted during the latency testing, these results differ to the equivilent test presente previously (Pennefather et al., 2018b). The sustained throughput of the bulk transactions has dropped from the initially report 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 testing.

5.3 Application Testing

To explore the usability of the proposed system as well as the implemented extensions, two basic application examples are provided. These examples have been selected to help highlight both the strengths and weaknesses of using a CPU-NPU heterogeneous system for processing.

5.3.1 Travelling Salesman

The first application example is an implementation of the Travelling Salesman problem is explored. The goal of the problem is to find the shortest cycle that connects all nodes in a 2D plane (Rocki & Suda, 2012) where 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 in a bounded time. The result may not be the optimal solution, but may still suitable for 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 & Norvig, 2009).

The goal of the IHC is to identify the shortest local path (the local maxima) for a given starting path. This local maxima is identified by iteratively evaluating all nodes in the path and determining if the connections between two pairs of connected nodes overlap on a 2D plane. Once identified, the pairs are recorded and the evaluation continues. Once all nodes have been evaluated, the overlap that exhibits the longest length is then optimised. The above process is then repeated on the revised path until no candidates can be found, implying a local maxima.

In this implementation the 2-opt algorithm was used to optimise the path. This algorithm swaps the position two nodes in the path, one from each pair to efficiently 'untwist' the overlapping vertices as shown in Figure 5. The non-overlapping vertices between nodes are maintained and so their order between the swapped nodes must be reversed. (Burtscher, 2014; Rocki & Suda, 2012)

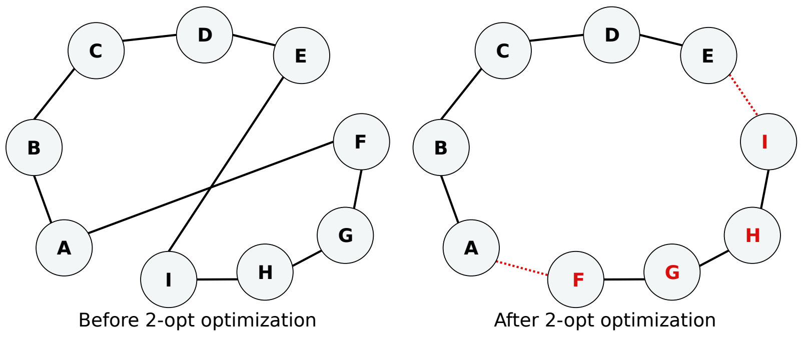


Figure 5: Application of the 2-opt swapping optimization on a candidate path.

For comparative purposes, the implementation was designed to closely follow the approach taken by O'Neil (O'Neil, Tamir, & Burtscher, 2011) in terms of how the candidate paths are identified. The representation of nodes mirrors that of the implementations described by O'Neil with each node being stored as a set of coordinates from which distances to other nodes can be calculated.

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 accepted a city file containing the coordinates of each node within the graph to be minimised. This data was fed 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 climbers the application should apply. 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.

The NFP component of the application contains a single manager process and 59 climber processes spread across NFP islands 32-36. Initially, 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 *result buffers* 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* consists of a series of candidate buffers where a 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 before assuming the responsibility of collecting results from climber processes and recording the optimal cycle found. This process 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 to where it is stored in the *results buffer* 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*. Buffers 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 climbs has been completed, the manager process submits the minimum recorded cycle along with its length to the host process.

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 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 maxima for a given configuration of the nodes can be preformed as described in Algorithm 3. This algorithm will continue to execute until the manager process announces that enough climb events have been processed.

```
1. nodeList ←ReadFromDefault ();
while Attempts still needed do
    2. nodeList ←ShuffleNodes ();
    3. reduction ← TryReduce (*nodeList);
    while reduction > 0 do
        4. reduction ← TryReduce (*nodeList);
    end
    5. bufferIndex ← FindFreeBuffer ();
    6. buffer [bufferIndex ] ← nodeList;
    7. tourCost ← TourCost (nodeList);
    8. SubmitToManager (tourCost,bufferIndex);
end
```

Algorithm 3: Pseudo-code 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 using *2-opt* optimisation to determine if it is possible to reduce the current length of the path. This process of identifying candidate nodes and performing the *2-opt* optimisation is repeated until no single 2-opt optimisation will result in a shorter path. At this point the Climber process has found a local maxima which is a candidate for the shortest identifiable path.
The climber process then searches the *buffer list* for a free buffer to reserve and current node configuration is then written to it. The buffer index and the cost of the current path are then submitted to the manager process as a work element in the circular queue. The Climber process can then immediately return to attempting to discover a new candidate cycle. The fact that the recently discovered candidate will not be registered by the manager before the climber begins a new iteration is not a concern as excess candidate cycles can simply be ignored.

For testing the application, an existing dataset of 100 nodes was used (Zuse Institute Berlin, 1995) and the application was run for 10000 random restarts. An optimized CPU implementation of the algorithm was also tested along with the GPU implementation used by O'Neil (O'Neil et al., 2011). The results of these tests are depicted in Table 3 which shows the average recorded time taken by each application. From these results it is clear that the GPU implementation greatly out performs both the CPU and NFP with the NFP implementation requiring the most time to compute.

5.3.2 PCSA

As a final example, probabilistic counting with stochastic averaging (PCSA) was designed and implemented to use the NFComms framework. PCSA has been selected to provide and example of an

Table 3: Time taken in seconds per implementation to process 10000 random climbs

	NFP	**Host**	**GPU**
Min	26.82	13.36	0.32
Avg	27.53	13.37	0.33
Max	29.13	13.38	0.35
STD	0.74	0.01	0.01

application better suited to the heterogeneous NFP-CPU framework with a heavier emphasis on data streaming rather than batched computation. PCSA is a sketching algorithm designed to estimate the Cardinality of a a stream of data while using a fixed amount of memory. It has subsequently been replaced by hyperloglog (Flajolet, Fusy, Gandouet, & Meunier, 2007) which uses less memory to provide the same level of accuracy, however PCSA has been selected over hyperloglog as it can more easily be implemented on the NFP without logarithmic or divide operators.

PCSA was proposed in 1985 by Flajolet and Martin as a heuristic approach to determining the cardinality of a data stream using fixed memory (Flajolet & Martin, 1985). 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 number of trailing '1's present in the binary representation of the hash are counted to produce the value $r(x)$, this value is then used to compute the $R(x)$ value where $R(x) = 2^{r(x)}$. The resulting $R(x)$ value is then *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 will generate a separate sketch and the cardinality estimate C can then be computed using equations (1) and (2). The relative accuracy of the PCSA algorithm is approximately $0.78/\sqrt{M}$.

$$Mean = \frac{\sum_{i=1}^{M} r(sketch_i)}{M} \tag{1}$$

$$C = \frac{2^{Mean}}{0.77531} \tag{2}$$

A PSCA algorithm containing 2048 sketches was implemented on the NFP. This would allow the implementation to estimate the cardinality with an average error of 1.78%. The hashing function used in this implementation is a standard CRC which produces 32 bit hash values for the update operation. To account for multiple sketches, the 9 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 PC and SA components of the algorithm can use the same hash.

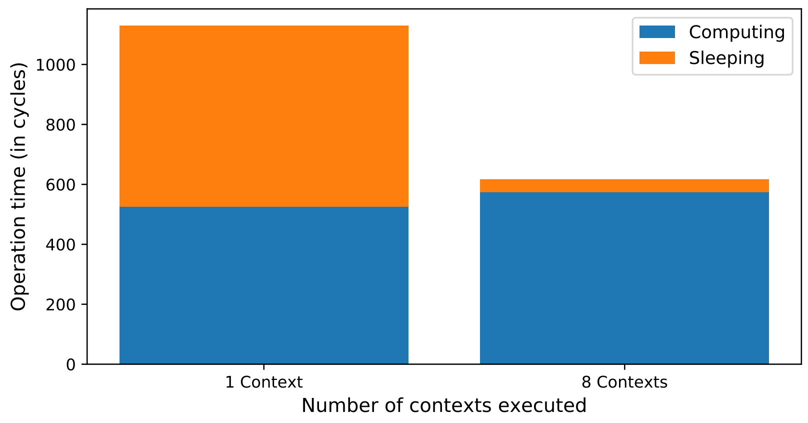


Figure 6: Comparison of time taken to execute 8 PCSA update operations on a single microengine.

As the PSCA operation does not require the same element to be hashed into multiple rows, both the 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 be scaled easily across the NFP architecture.

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 2 would be resolved on the host. To facilitate this, the host handler process sends the sum of all trailing 1's in the sketches and the size of the sketch to the host upon request.

To test the implementation, a large pre-recorded network capture was replayed through the NFP. Once complete, the details required to estimate the cardinality were requested from the NFP. To provide a comparison, the dictionary based solution used was implemented on the host to record the number of unique IPs seen. This host based solution was fed the same network capture to produce the true cardinality for the stream. Comparing the results of the host based solution and the cardinality estimate, it was found that the estimate had an error of 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 would take 124 cycles. The operation was then partitioned into time spend on computation and time spent sleeping with the active context spending 75 cycles, just over 60% of the time sleeping. Only 49 cycles involved computation with 39 being used in the calculation of the CRC hash. This wait time is due to updating the PCSA sketch requiring two memory operations involving CLS, each of which incur 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 total time to compute for this test was 1129 cycles, of which 526 where used in computation as indicated in Figure 6. The total number of cycles does not exactly match the 124 × 8 predicted count as the test required additional cycles during the start-up and competition states.

The microengine was then allowed to use all eight available contexts and the same test was performed with the results also recorded in Figure 6.The computation component of this test required 48 cycles more than the single context implementation, however it only spend 42 cycles sleeping, resulting in the test only taking 616 cycles, or 45% fewer cycles. This is due to the multiple contexts effectively hiding the access latencies with only a few extra cycles needed for the context switching overheads. 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 CONCLUSION

Considering the recorded results from latency testing, it is clear that the indirect messaging framework is significantly slower for transmitting individual messages compared to the direct messaging framework. 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 Gb/s when servicing a single transaction, and up to 30 Gb/s when servicing eight transactions.

Section 5.3 presents two application examples which implement the indirect communication extensions to show the functionality of the framework. The first application is an solution to the travelling salesman problem through the use of IHC while the second is an implementation of PCSA for estimating the cardinality of IP addresses in a network stream. Though functional, the solution to the travelling salesman problem is shown to not be as effective as the equivalent solution targeting the CPU and GPU, indicating that heavy computation is not a strength of the CPU-NPU heterogeneous system. The final application example shows that the system is much better suited to stream based algorithms, with PCSA implementation able to estimate the cardinality of IP addresses in a network stream at wire speeds of up to 10 Gb/s.

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 114.3 Mb/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.

In its current state, all communication via the NFComms framework is designed to interface with a single NFP device. Future work will involve extending this framework to provide a communication medium capable of supporting multiple NFP devices in a single application. In addition to adding support for multiple devices, investigations into extending the framework to allow for asynchronous functions will also be performed.

References

Ahmadi, M., & Wong, S. (2006). Network processors: Challenges and trends. In *17th Annual Workshop on Circuits, Systems and Signal Processing* (pp. 223–232).

Andrews, D., Niehaus, D., Jidin, R., Finley, M., Peck, W., Frisbie, M., … Ashenden, P. (2004). Programming models for hybrid FPGA-CPU computational components: A missing link. *IEEE Micro*, *24*(4), 42–53.

Bos, H., & Huang, K. (2006). Towards software-based signature detection for intrusion prevention on the network card. In A. Valdes & D. Zamboni (Eds.), *Recent Advances in Intrusion Detection* (pp. 102–123). Berlin, Heidelberg: Springer Berlin Heidelberg.

Burtscher, M. (2014). A High-Speed 2-Opt TSP Solver for Large Problem Sizes. Online. Presentation slides. Last accessed 09 Dec 2019. Texas State University. Texas State University. Retrieved from http://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf

Flajolet, P., Fusy, É. F., Gandouet, O., & Meunier, F. (2007). 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., & Martin, G. N. (1985). Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, *31*(2), 182–209.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, *21*(8), 666–677.

Kappler, T. (2016). Package CSP. Package Documentation. Last accessed 8 Mar 2019. Go Documentation. Retrieved from https://godoc.org/github.com/thomas11/csp

Lastovetsky, A., & Dongarra, J. (2009). *High performance heterogeneous computing*. Hoboken, New Jersey, USA: John Wiley & Sons, Inc.

Lawley, J. (2014). Understanding performance of PCI Express systems. White Paper. Last accessed 09 Dec 2019. Xilinx. Retrieved from https://www.xilinx.com/support/documentation/white_papers/wp350.pdf

Mittal, S., & Vetter, J. S. (2015). A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, *47*(4), 69:1–69:35. 10.1145/2788396

Netronome. (2016). *Netronome Network Flow Processor NFP-6xxx-x C preliminary draft databook*. Netronome. Confidential Property. USA.

Netronome. (2018). Netronome NFP-4000 Flow Processor. Product brief. Last accessed 09 Dec 2019. Netronome. Retrieved from https://www.netronome.com/m/documents/PB_NFP-4000.pdf

O'Neil, M., Tamir, D., & Burtscher, M. (2011). A parallel GPU version of the traveling salesman problem. Online. Accessed on: 14 September 2018. Retrieved from https://userweb.cs.txstate.edu/~mb92/papers/pdpta11b.pdf

Pennefather, S. (2017). *CSP compliance testing of a message passing framework for the NFP NPU*. Rhodes University. Rhodes University.

Pennefather, S., Bradshaw, K., & Irwin, B. (2017). 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)* (pp. 178–183). SATNAC. South Africa.

Pennefather, S., Bradshaw, K., & Irwin, B. (2018a). 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)* (pp. 46–56). IPDPS. Canada.

Pennefather, S., Bradshaw, K., & Irwin, B. (2018b). Extending the NFComms framework for bulk data transfers. In *South African Institute of Computer Scientists and Information Technologists (SAICSIT) 2018*. SAICSIT, South Africa.

Rocki, K., & Suda, R. (2012). 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)* (pp. 489–495). 10.1109/CCGrid.2012.133

Russell, S., & Norvig, P. (2009). *Artificial intelligence: A modern approach* (3rd). Upper Saddle River, NJ, USA: Prentice Hall Press.

Shuhui, C., Rongxing, L., & Xuemin, S. (2013). SRC: A multicore NPU-based TCP stream reassembly card for deep packet inspection. *Security and communication networks*, *7*(2), 265–278.

Sovani, K. (2005). Kernel korner—Sleeping in the kernel. Online article. Last accessed 09 Dec 2019. Linux Journal. Retrieved from https://www.linuxjournal.com/article/8144

Tsoi, K. H., & Luk, W. (2010). Axel: A heterogeneous cluster with FPGAs and GPUs. *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 115–124. 10.1145/1723112.1723134

Wheeler, B. (2013). *A new era of network processing*. The Linley Group. Accessed on: 27 May 2017. Retrieved from https://www.linleygroup.com/uploads/ericsson_npu_white_paper.pdf

Zahran, M. (2017). Heterogeneous computing: Here to stay. *Communications of the ACM*, *60*(3), 42–45. 10.1145/3024918

Zuse Institute Berlin. (1995). MP-TESTDATA—The TSPLIB symmetric traveling salesman problem instances. Retrieved from http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/