Design of a Message Passing Model for Use in a Heterogeneous CPU-NFP Framework for Network Analytics

Sean Pennefather*, Karen Bradshaw†, Barry Irwin‡
Department of Computer Science, Rhodes University, Grahamstown
*g10p0016@campus.ru.ac.za, †k.bradshaw@ru.ac.za, ‡b.irwin@ru.ac.za

***Abstract*—Currently, network analytics requires direct access to network packets, normally through a third-party application, which means that obtaining realtime results is difficult. We propose the NFP-CPU heterogeneous framework to allow parts of applications written in the Go programming language to be executed on a Network Flow Processor (NFP) for enhanced performance. This paper explores the need and feasibility of implementing a message passing model for data transmission between the NFP and CPU, which is the crux of such a heterogeneous framework. Architectural differences between the two domains are highlighted within this context and we present a solution to bridging these differences.**

***Index Terms*—Heterogeneous Computing, Network Processors, Network Analytics**

I. Introduction

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 processors to handle the network specific computations. Network processors, which can exist in advanced routing engines or as daughter cards to be interfaced with directly by a host, are designed to operate in computing environments requiring low latency and high throughput [1]. 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.

This research explores the development of a framework that combines the host and the network processor as a single heterogeneous platform for the execution of applications. Our approach is similar to that used by NVidia in providing its CUDA runtime SDK and 'nvcc' compiler to enable the execution of a single application to be split between the GPU and CPU. Such a framework would allow a variety of network analytics to be performed by leveraging the extensive functionality of a high-level concurrent language like Go, while having direct hardware access to the network packets themselves. The benefits of this kind of architecture include integration of real-time network analytics into existing toolsets for visualisation and processing as well as providing an enriched development environment and support dynamic memory allocation through the CPU architecture.

As described by Lastovetsky in the taxonomy of heterogeneous platforms [2], such systems are always comprised of multiple processing elements and a communication framework interconnecting such elements. This paper focuses on the latter by evaluating existing approaches to communication provided by Go and the Network Flow Processor (NFP) architecture to determine their compatibility. Thus, the main aim of this research is to implement communication within the NFP-CPU framework in the form of message passing, similar to that described by Hoare [3] and implemented as channels in various concurrent programming languages [4]–[6].

For the prototype implementation, we focus on interfacing with the NFP-400 [7] developed by Netronome, which can be programmed with a variation of C-89 referred to as MicroC. The NFP is coupled with a standard AMD64 processor to produce a CPU-NFP heterogeneous system that forms the underlying hardware for this research.

The NFP-400 consists of 60 32-bit flow processing cores (FPC), also referred to as microengines, each of which can run up to eight contexts [7]. These microengines do not contain a stack, limiting some functionality such as recursive calls, but do contain a large number of general purpose registers. Each microengine supports an instruction register of approximately 8000 operations in which to store an application however, instruction space must be split between all eight contexts if they are not intending to execute the same application code.

To support these microengines, the NFP-400 contains a hierarchical memory structure based on size and access latency. Access to the different tiers of memory is done through memory engines, or memory units that manage the memory and handle the resolution of all read/write requests. Another important feature of the NFP-400 in the context of this research is the inclusion of a PCIe Gen3 interface with support for eight lanes and allowing up to 8 GT/s.

For programming the heterogeneous system, we have chosen the programming language Go, developed in 2007 as a language targeting multiprocessor development [8]. Go has been designed with a heavy focus on concurrency, which is natively supported through the inclusion of asynchronous functions referred to as goroutines. Go was selected as the target language because of this concurrency support and its message passing communication model.

The remainder of the paper is structured as follows. Section II discusses some of the theory related to synchronous message passing, including the concepts behind naming and symmetry. Section III and IV discuss how message passing is handled by each of the two candidate architectures comprising the heterogeneous platform. This discussion is followed by

Section V which aims to highlight the differences between the two approaches to message passing and help establish the need for a message passing fabric to handle communication between the two architectures. Section VI describes a potential implementation of the message passing fabric through the introduction of a message manager pair which is responsible for resolving the symmetric and naming differences between the two platforms. Initial results of this approach and concluding comments are presented in Section VII.

II. Communication Between Concurrent Processes

Concurrent programs are generally designed to incorporate two or more cooperating processes, which are expected to be executed in a parallel manner. During execution, these processes may be required to interact at specific stages, implying the need for both a communication and synchronisation primitive [9].

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 1. Any information transmitted in the opposite direction is represented as a separate event.

A constraint that the system must enforce is 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 threads, both threads must have reached the message passing event state in their respective execution paths for it to be resolved. 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 2, the first interaction is the transmission of the message from the transmitting thread while the second is the acknowledgement message from the receiving thread. Should the transmitting thread enter the message passing event before the receiving thread, message transmission from the context of the receiving thread occurs. After transmission however, the transmitting thread is forced to wait for the acknowledgement message from the receiving thread. When the receiving thread finally reaches the message passing event, it is able to acknowledge the receipt of the message by sending an acknowledgement message after which both threads can continue executing normally.

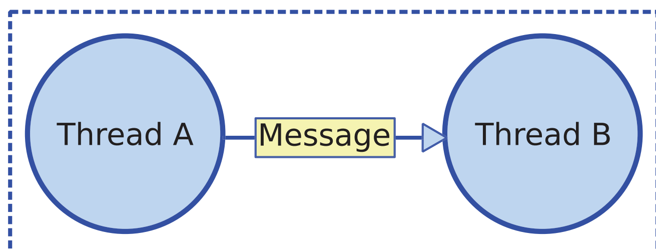

Fig. 1. Abstract representation of a message passing event between threads A and B (adapted from [10]).

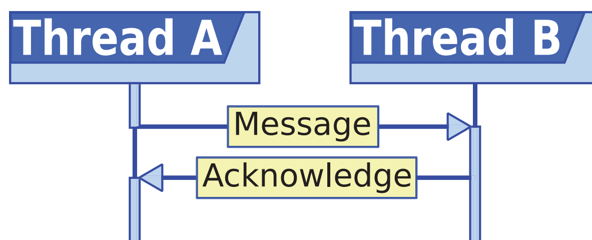

Fig. 2. Abstract representation of a message passing event between threads A and B.

```
Source: [send message to destination]
Destination: [receive message from
source]
```

Fig. 3. Abstracted syntax for message passing.

A. Naming

In order to implement synchronous communication through message passing such as depicted in Figure 2, an important attribute that must be considered is how the relevant parties are identified. Considering an abstracted implementation of message passing, a message could be sent using syntax similar to that shown in Figure 3 [11] .

In this implementation, the variable `message` represents the information being passed 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 Davis [12], 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 and described further in this section.

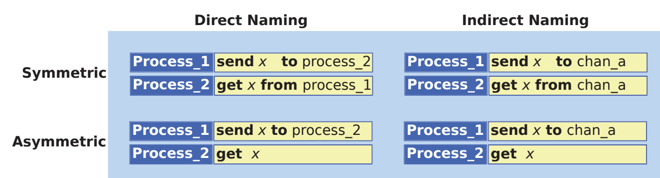

Fig. 4. Diagram showing the four types of communication considering naming and symmetry aspects.

For a message transmission event to be resolved correctly, the transmitting process must provide a description of the destination so that the system can determine which processes is expected to be involved in the event. The simplest method of archiving this is to use direct naming where some unique identifier of the target process is provided for the message transmission event [13], [14]. The abstracted syntax shown in Figure 3 is thus an example of direct naming where both communicating parties identify their counterpart processes by name.

The 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 [15]. 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.

The symmetry of a communication event describes the relationship between the participating processes. For this 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 [16]. 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 [17].

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. Alternatives include a selective wait [12] 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 reducing the multiplicity of parters 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 relayed to the correct process.

III. Message Passing in GO

Concurrency and synchronous communication are core attributes of the Go programming language [18]. To support this, Go includes a basic primitive for concurrency referred to as a goroutine, which is a play on the more traditional term coroutine, formally described by Conway in 1963 [19]. As with a coroutine, a goroutine is intended to allow independent functions or routines that can be executed asynchronously to be multiplexed so that they can progress in a concurrent manner [8], [20].

A. Goroutines vs. Threads

As discussed in the language documentation [21], 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 included 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 then there are threads available. Should a goroutine be assigned to a thread block while waiting on an event, the non-blocked goroutines multiplexed on that thread are migrated to another thread, allowing them to continue executing [22].

B. Goroutine Channel Communication

As noted by Chrisnall [8], 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 Communicating Sequential Processes (CSP) proposed by Tony Hoare [23] in 1978 [24].

In Go, a channel is a first class object that must be typed so as to specify the kind of information than can be passed through it [20], [25]. 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.

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 and 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 [25]. Goroutines can also perform the equivalent of a selective wait [12] by using the `select` statement and specifying multiple channels on which to wait. The `select` statement can optionally assign conditions or guards to channels, which must be satisfied before they can be considered eligible for selection.

IV. NFP Communication Primitives

The NFP architecture has been designed to achieve high throughput in network traffic processing. As discussed in Section I, this is facilitated by the use of FPCs or microengines, each of which executes independently, supporting up to eight contexts or threads.

To allow for communication between the NFP-400 and the host, the network processor includes a PCIe module that

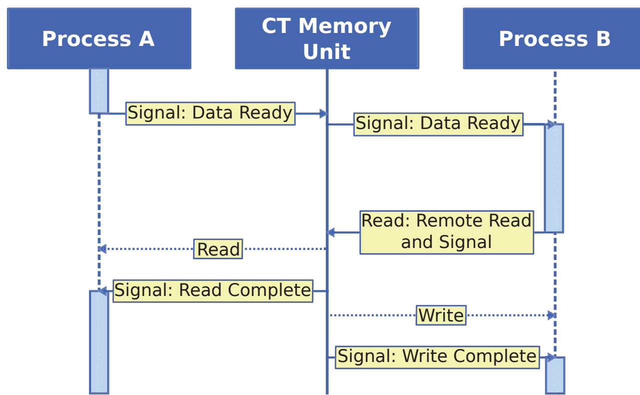

Fig. 5. Breakdown of synchronous message passing within the NFP.

provides an interface conforming to the PCI Express Gen3 standard. As described in the databook [26], this module is capable of supporting up to eight lanes at 8 GT/s. The module is also capable of acting as either a PCIe endpoint or a Root Complex. When configured to act as an endpoint, the PCIe controller allows visibility of the system bus and its connected targets such as memory units and the internal microengines. When configured to act as a Root Complex, the PCIE controller can generate the necessary transactions to perform endpoint discovery and configuration .

From the context of applications operating on the microengines it is possible to use the PCIe module to submit MSIX interrupts to the host provided the PCIe module is in endpoint mode [26]. With appropriate modifications to the NFP driver, the host OS can be equipped to listen for such interrupts and generate a software interrupt for user space applications.

A. Reflect Operations

Though each microengine executes independently, the architecture does provide support for moving data between microengines. This transfer operation is a subset of the reflect operation set and is performed by one of two types of memory units present on the device [26], [27]. Read and write reflect operations occur by first allocating a subset of transfer registers, either read or write, and then sending a reflect command to one of the compatible memory units, requesting that it performs the transfer operation on behalf of the microengine. Details such as target microengine, direction of transfer, size of transfer, and signal requirements are also sent as part of the command.

B. Implementing Message Passing

Using the communication tools supported by the NFP architecture, a message passing framework has been designed and implemented. The framework has been shown to conform to the synchronous message passing described in CSP proposed by Tony Hoare [23] and to be scalable across multiple contexts and microengines without introducing deadlocks due to resource contention.

The framework operates using reflect operations, which allow signals and data to be relayed between two microengines as depicted in Figure 5. A context wishing to communicate, writes the relevant data into a write transfer register and then constructs the ID of the target process based on its context number, microengine ID, and island number. The communicating process then sends an inter-thread signal to a known signal number on the target process indicating that it is ready to participate in a message passing event before sleeping. This message is actually addressed to the memory unit, which performs the reflect operation on behalf of the sending process.

The receiving process goes to sleep upon entering the message passing event until the appropriate signal is received. At this point the receiving process wakes up, performs a reflect read on the predetermined transfer registers, again via the memory engine, and informs the engine to signal the initiating process after the transfer is complete. This resolves the message transfer event. It must be noted that the receiving process needs to know the process ID and write transfer registers of the transmitting process as such information cannot be relayed within the initial inter-thread signal operation.

V. Comparison of Message Passing in Go and NFP

Considering Figure 4, it is clear that message passing in Go is positioned in the bottom right quadrant as channels are used to identify parties involved in a message passing event. These channels however enforce no restrictions on the number of functions that can read or write to them and so asymmetric message passing is possible.

MicroC on the other hand is clearly situated in the top left quadrant as the ID of each process involved in the message passing event must be directly named. 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 major 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 are transmitted between the CPU and NFP. The NFP-400 is connected to the Host via the PCIe bus and interactions between the NFP and the PCIe fabric must go through the PCIe block [26]. For message passing, this means that all data in transmission will converge before being transmitted over the PCIe bus.

VI. Design of an Intermediate Manager Pair

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 managers are responsible

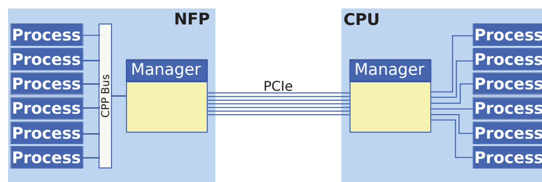

Fig. 6. Channel manager overview.

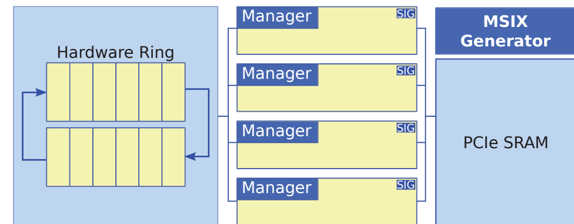

Fig. 7. Layout of the NFP channel manager components.

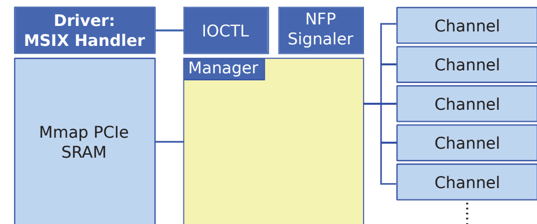

Fig. 8. Layout of the host channel manager components.

for relaying messages between the two architectures as well as bridging the semantic divide in both naming schemes and symmetry as depicted in Figure 6.

From the perspective of the NFP architecture, the managers act as the required intermediary that 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. The details of the metadata are resolved at compile time by a preprocessor responsible for building the managers.

For a goroutine executing on the CPU, interacting with the manager is largely abstracted as communication takes place through a channel. From the perspective of the goroutine, the channel is 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 manager is responsible for receiving, encapsulating and transmitting the message to its actual destination.

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

A. NFP Channel Manager

For receiving messages from processes operating on the NFP device, the NFP channel manager provides a circular buffer or hardware ring which such processes can address directly. Writes into this ring can be considered atomic as the underlying hardware resolves simultaneous writes from multiple processes to ensure both operations are committed. Each message written to the ring does not actually contain any message data but instead contains the encapsulating information which is used to construct the message header and allow the manager to retrieve the associated message data. This header includes the ID of the writing process, the message size and destination, an address of the transfer registers where the data is stored, and a signal that must be set when the transmission is complete.

For consuming these messages, up to four manager engines are subscribed to the ring and are woken up when work is available to consume as depicted in Figure 7. These engines fetch the message body associated with the consumed message and build the message packet, which is then transferred into the PCIe SRAM. To inform the host channel manager that a message is available for processing, the MSIX generator is signalled to send the interrupt to the host CPU. When a specified signal on the active manager engine is set by the host, the signal is forwarded to the initiating process, completing the message transfer.

B. Host Channel Manager

The host manager is largely configured by a preprocessor as it is intended to support a channel interface for every process executing on the NFP device. Messages destined for the NFP device are received over one of the known channels and encapsulated in a message packet containing the ID of the channel over which it was received, its length, and the destination NFP process ID associated with that channel. The message packet is then written to a shared memory location shown in Figure 8 before the NFP channel manager is signalled indicating that a new message is available to be processed. A goroutine is spawned to wait on an MSIX interrupt specified in the message packet which indicates that the target process has received the message. When the interrupt occurs, the contents of the channel over which the message was initially transmitted is consumed, allowing the transmitting process to continue operation.

Unfortunately, due to a channel operation being an atomic event in Go, it is currently not possible to view the channel contents without consuming them. As a consequence, a single channel transaction can no longer be used to guarantee a synchronous event. The currently implemented solution to this is to enforce that any channel communication with processes operating on the NFP must write a second message onto the channel, which is consumed by the manager after the message transmission event has completed.

VII. Results and Conclusion

Preliminary testing of the proposed channel manager pair shows the successful transmission of messages between a producer process operating on the NFP and a consumer

process written in Go. Some performance limitations were however identified, most of which are attributed to two aspects of the communication.

The first aspect which inhibited throughput is the use of shared memory through which the body of the message is transmitted. In the current implementation, the shared memory for both managers is 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 [28], 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 of the data.

The proposed solution to this memory access limitation is to allocate memory on the host for the host channel manager and have the NFP device write the message bodies to the reserved memory via DMA transfers prior to synchronising with the host channel manager.

The second limitation is due to the signalling of the NFP channel manager from the host. This is continually done using an API supplied with the device which is capable of resolving an address to a specific signal within the NFP channel manager. Currently this operation has a recorded latency of approximately 3 μs which can introduce a significant bottleneck when message payloads are relatively small. An evaluation of the operation to determine how the latency is introduced will be performed in an attempt to mitigate this limitation.

Though still in its infancy, the proposed message passing model has been shown to resolve the naming disparity between the two message passing paradigms, thereby providing an effective communication fabric for the heterogeneous NFP-CPU framework. This means that network analytic applications can leverage the benefits of the high-level programming features provided by Go, as well as wire speed processing available through the NFP.

Future work relating to the proposed message passing model entails resolving the noted performance limitations and developing a a full scale implementation of the communication management system. Other aspects to evaluate include the feasibility of supporting asynchronous communication between the two architectures and introducing support for bulk data transfers which may be better implemented as single operations rather than a series of message passing events.

ACKNOWLEDGEMENTS

This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP and NRF SA (TP13070820716). The authors acknowledge that opinions, findings and conclusions or recommendations expressed here are those of the authors and that none of the above mentioned sponsors accept liability whatsoever in this regard. Furthermore the author would like to thank Netronome for providing the necessary hardware and support to make this research feasible.

REFERENCES

[1] M. Ahmadi and S. Wong, "Network Processors: Challenges and Trends," in *17th Annual Workshop on Circuits, Systems and Signal Processing*, 2006, pp. 223–232.

[2] A. Lastovetsky and J. Dongarra, *High Performance Heterogeneous Computing*. Hoboken, New Jersey, USA: John Wiley & Sons, Inc., 2009.

[3] C. A. R. Hoare, *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985 [2004].

[4] R. Prabhakar and R. Kumar, "Concurrent programming with Go," Google, Tech. Rep., June 2011.

[5] D. Watt, *Programming XC on XMOS Devices*. XMOS Limited., 2009.

[6] D. May, "OCCAM," *ACM SIGPLAN Notices*, vol. 18, no. 4, pp. 69–79, Apr. 1983.

[7] Netronome, "NFP-4000 Intelligent Ethernet Controller Family," Netronome, Santa Clara, USA, Product Brief, 2016, accessed on 20 March 2017.

[8] D. Chisnall, *The Go Programming Language Phrasebook*. Michigan, USA: Addison-Wesley, March 2012.

[9] G. Andrews, *Concurrent Programming: Principles and Practice*. Redwood City, CA, USA: Benjamin-Cummings Publishing Company, 1991.

[10] D. Hyde, "Introduction to the Programming Language Occam," Department of Computer Science Bucknell University, Lanugage Manual, March 1995, accessed on: February 2016.

[11] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys (CSUR)*, vol. 15, no. 1, pp. 3–43, Mar. 1983.

[12] A. Burns and G. Davies, *Concurrent Programming*. Redwood City, CA, USA: Addison Wesley Publishing Company, 1993.

[13] A. Dinning, "A Survey of Synchronization Methods for Parallel Computers," *IEEE Computer Society*, vol. 22, no. 7, pp. 66–77, Jul. 1989.

[14] A. Silberschatz, P. Galvin, and G. Gagne, *Operating Systems*, 7th ed., B. Zobrist, Ed. John Wiley and Sons Inc., 2005.

[15] I. Bertolotti and T. Hu, *Embedded Software Development: The Open-Source Approach*. CRC Press, 2016.

[16] R. Koymans, *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1992.

[17] P. Treleaven and M. Vanneschi, Eds., *Proceedings of an Advanced Course on Future Parallel Computers*. London, UK: Springer-Verlag, 1987.

[18] N. Kozyra, *Mastering Concurrency in Go*. Birmingham, UK: Packt Publishing, July 2014.

[19] M. E. Conway, "Design of a Separable Transition-diagram Compiler," *Commun. ACM*, vol. 6, no. 7, pp. 396–408, Jul. 1963.

[20] Google, "The Go Programming Language: FAQ," Google, Language documentation, 2017, Build version: Go 1.8. Accessed on: 20 February 2017. [Online]. Available: https://golang.org/doc/faq#goroutines

[21] ——, "The Go Programming Language: Effective Go," Google, Language documentation, 2017, Build version: Go 1.8. Accessed on: 20 February 2017. [Online]. Available: https://golang.org/doc/effective_go.html

[22] GO devlopment team. (2015, August) The Go programming language specification. Online. The Go Programming Language. Accessed on: 3 Feb 2016. [Online]. Available: https://golang.org/ref/spec#Introduction

[23] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[24] M. Todorova, M. Nisheva-Pavlova, G. Penchev, T. Trifonov, P. Armyanov, and A. Semerdzhiev, "The Go Programming Language: Characteristics and Capabilities," in *Annual of "Informatics" Section Union of Scientists in Bulgaria*, vol. 6. Sofia, Bulgaria: Faculty of Mathematics and Informatics, Sofia University, 2013, pp. 76–85.

[25] I. Balbaert, *The Way To GO: A Thorough Introduction to the Go Programming Language*. Bloomington, India: iUniverse, May 2012.

[26] *Netronome Network Flow Processor NFP-6xxx-xC Preliminary Draft Databook*, Netronome, Propietry and Confidential.

[27] Netronome, "Netronome Network Flow Processor 6xxxx Flow Processor Core Programmer's Reference Manual," Netronome, Users Guide, 2006, Accessed on 1 June 2016, Confidential Property.

[28] J. Lawley, "Understanding Performance of PCI Express Systems," Xilinx, White Paper, 2014.