Efficient inter-kernel communication in a heterogeneous multikernel operating system

Ajithchandra Saya

Report submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Master of Engineering
In
Computer Engineering

Binoy Ravindran
Patrick Schaumont
Chao Wang

06/26/2015
Blacksburg, Virginia

Keywords: Inter-kernel communication, messaging layer, Heterogeneous architectures, ARM, x86, operating system, Linux, power-performance benefits, PCIe interface

Copyright 2015 @ Ajithchandra Saya

Efficient inter-kernel communication in a heterogeneous multikernel operating system

Ajithchandra Saya

ABSTRACT

This report primarily focuses on the essential components needed for creating Heterogeneous Popcorn, first truly heterogeneous operating system for ARM-x86. It supports 64 bit ARM (aarch64) and x86_64 architectures. Heterogeneous Popcorn is created by using Popcorn Linux, a replicated kernel operating system based on Linux. The basic building blocks of Popcorn Linux like messaging layer, task migration and memory consistency subsystem had to be redesigned or modified to create heterogeneous Popcorn.

The report primarily discusses the new design of inter-kernel messaging layer using PCIe interface. Non-transparent bridging (NTB) technology in PCIe is used to provide remote memory access capabilities. A multi-threaded parallel messaging layer framework is designed to provide multiple channels for data transfer. Different design considerations are discussed and analyzed in the report. The messaging layer provides very deterministic and low message passing latencies and high bandwidth capabilities.

The support for seamless migration of application threads across ARM and x86 is provided in heterogeneous Popcorn. A new application binary format is defined and a compiler framework is developed to create such heterogeneous binaries. The Linux ELF loader is also modified to load such new binary. The register information passing and mapping is handled during task migration.

To My Parents: S.N.Bhat and Veena

Acknowledgements

I wish to extend my sincerest gratitude to Dr. Binoy Ravindran for his guidance and support throughout my work. I have had great discussions and brain storming sessions with him which have filled me with inspiration and broadened the scope of my thinking. I am also very grateful to Dr. Patrick Schaumont who greatly served as my interim advisor and helped with my official work whenever needed. I am extremely thankful to Dr. Chao Wang for agreeing to be on my committee and providing necessary support whenever I approached him.

I would also like to thank Dr. Antonio Barbalace who has been a source of inspiration and constant support. I have learnt from him immensely, especially on the technical front. He has been the first ‘go-to’ person for all my queries and he has always responded by helping me out. I would also like to extend my gratitude to the rest of my lab mates and friends at Virginia Tech who have been by my side on countless occasions.

Finally, I would like to thank my parents and sister for their perennial love and motivation. Their encouragement and support have been essential for the successful completion of my degree requirements.

Table of Contents

List of Tables & Figures

Chapter 1: Introduction

Computer architectures are evolving in the direction of heterogeneity to provide best power-performance benefits for a varied mix of workloads both in embedded and general purpose computing space. With the hitting of power wall and the emergence of multicore processors, power has become a primary design constraint [1]. Even in servers and data centers, energy efficiency is a major concern.

As computer architects strive to provide the best possible solution for different mix of workloads, having specialized cores with different power-performance characteristics is an attractive solution. Today, heterogeneity can be found in the form of micro architectural features within a single-ISA as in ARM big.LITTLE and Nvidia Tegra 3 for energy efficiency, or a set of overlapping ISAs with some additional specialized instructions for better performance as in Intel Xeon-Xeon Phi. The other form of heterogeneous architecture is the one in which specific accelerators like GPU act as slaves to CPU cores. Whereas these architectures try to exploit heterogeneity in a constrained way, a truly heterogeneous design with different ISA cores and micro architectural features will be able to fully utilize the benefits of heterogeneity and can provide greater power and performance benefits as shown in [2]. Different parts of an application can be divided into different phases based on the kind of instructions being executed. These different phases can then be mapped to the best suited ISA to gain power-performance benefits within the same application. Having multiple processors and accelerators with different capabilities is quite common in today's SOCs. AMD is working on a socket compatible ARM64 and x86_64 processor chips [3] and eventually can lead to a shared memory ARM-x86 processor. Heterogeneity clearly seems the future of computer architecture.

The prospects of a heterogeneous ISA system is exciting as demonstrated in [2] but providing a single operating system and the necessary software support to enable such heterogeneous systems is a daunting task and opens a whole new arena of research. As of today, there is no single operating system that supports multiple ISA architectures. Traditional SMP operating systems like Linux will need a major redesign to support such systems. Even though today there are no research operating systems designed to accommodate heterogeneity, multikernel based research operating

systems like Barrelfish [4] and Popcorn Linux [5] can be extended to support these emerging heterogeneous architectures.

The multikernel research operating systems usually run one kernel per core or a cluster of cores. The different kernels communicate with one another using explicit message passing to provide the user with a single operating system interface. We will briefly discuss Popcorn Linux, a research operating system from System Software Research Group at Virginia Tech. In the current work, we extend Popcorn Linux to run on ARM64 and develop essential building blocks needed to create an ARM-x86 heterogeneous Popcorn Linux, the first truly heterogeneous research operating system.

1.1 Popcorn Linux

Popcorn Linux is a replicated-kernel operating system based on Linux. The main aim of Popcorn Linux is to provide scalability by overcoming the scalability issues of traditional operating systems like Linux running on many-cores architectures. Also it aims to improve programmability by providing consistent shared memory interface to the developer under wide variety of processor cores with coherent/non-coherent shared memory and distributed memory hardware architectures. It provides standard Linux interface such that numerous applications currently running on Linux can seamlessly reap the benefits of Popcorn Linux. Therefore Linux was chosen as the base operating system instead of a micro-kernel which seems more suitable for a multikernel approach.

The basic design of Popcorn Linux is shown in Figure 1.1. Each core or a cluster of cores of run a Linux kernel. Different kernels running on same/different ISA cores communicate with one another using an inter kernel messaging layer to provide the user with a single operating system interface. Popcorn namespace provides a unified memory and processor view thereby creating a single system image. Popcorn also provides a number of other useful services. It provides a transparent and efficient task migration between kernels. A unified consistent view of virtual memory is provided across kernels for running applications thus providing the application developer with a standard SMP like coherent shared memory programming interface.

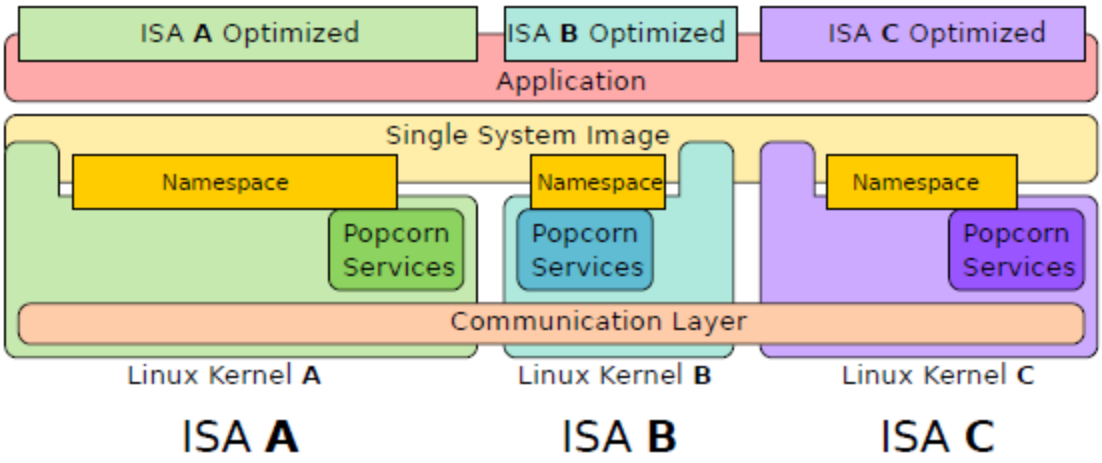

Figure 1.1: Popcorn Linux Architecture

The important services provided by Popcorn Linux are briefly discussed below. A detailed explanation can be found in [5].

1.1.1 Single system Image

The single system image is a consistent single interface for multiple kernels running in Popcorn Linux. It provides a unified view of processors and memory across kernels using the underlying messaging layer. It also provides `/proc` filesystem consistency and signaling across different kernels. A distributed futex algorithm based on a server-client model is implemented to enable futex services in *libc* across multiple kernels to provide user-space applications with standard POSIX synchronization APIs.

1.1.2 Task Migration

Popcorn's task migration service enables seamlessly migration of thread/processes across different kernels. When a thread wants to migrate from *kernel A* to *kernel B*, the thread state information (register values, thread context, signals, futex etc.) is sent from *Kernel A* to *kernel B* using the messaging layer. In each kernel, a pool of threads is created upfront to be used during migrations. In *kernel B*, a thread is chosen from this pool, all the information is restored and the thread is scheduled. The thread on *kernel A*, which is now a shadow thread is put to sleep. When the task

returns from *kernel B* to *kernel A*, the shadow thread is woken up, the updated task state is restored and it is scheduled. The thread on *kernel B* now becomes a shadow thread. Successive migrations of this thread is just a thread state update to the shadow thread and hence is faster than the first migration.

1.1.3 Memory consistency

For application threads to seamlessly run across multiple kernels, there is a need to provide memory consistency across kernels. This means consistency in virtual memory area (VMA) and virtual to physical page mappings. VMA is updated lazily in different kernels running the application threads. If a VMA mapping is modified in any kernel, it is kept consistent across all the kernels having the VMA mapping by using message passing. The virtual to physical page mappings are kept consistent by using a page-replication algorithm [6].

The page-replication algorithm is similar to the cache coherency protocol but works at page level granularity. It uses three states: modified, shared and invalid to ensure that all the threads of an application across different kernels have a coherent view of memory pages. When a migrated thread causes a page fault to occur, the protocol communicates with other kernels using the messaging layer to find the kernel having the corresponding VMA and the latest copy of that page. The VMA information is lazily obtained during this event and the corresponding VMA mapping is created on that kernel. If the page is not found in other kernels, the VMA is mapped and we proceed with normal page fault handling in the local kernel, thereby installing the page locally. If the page fault is due to a read and another kernel has a copy of the page, then copy on write (COW) mechanism is used. The protocol is described in greater detail in [6].

1.2 Heterogeneous Popcorn

Having created Popcorn Linux, a multikernel research operating system on homogeneous x86 SMP machines and also on Xeon-XeonPhi overlapping ISA architecture [7], the next step is to explore the full potential of an operating system like Popcorn by creating a truly heterogeneous operating system running on two entirely different ISAs.

In this work, we create heterogeneous Popcorn, the first truly heterogeneous operating system running on ARM and x86 ISA architectures. Today, ARM and x86 are the two most widely used ISAs with different focuses: power efficiency and performance. ARM has primarily focused on embedded space targeting battery operated devices where energy efficiency is a primary constraint. x86 has focused on providing complex instructions and microarchitecture to improve the performance of desktop and servers. So having a single operating system that can work seamlessly on both ISAs would provide a great power-performance spectrum to explore for applications, where different parts of an application can run on different architectures to provide the best possible power-performance benefits.

This report is organized as follows. In Chapter 2, we discuss the design and implementation of messaging layer, the most critical piece which enables inter-kernel communication in Heterogeneous Popcorn. Chapter 3 discusses design changes needed in Popcorn services for Heterogeneous Popcorn. We mainly discuss the application binary format and the task migration mechanism in such a heterogeneous setting.

Chapter 2: Related Work

In this chapter, we will discuss the common inter-core communication techniques that exists in the current SMP hardware. We will also discuss how these technologies have been utilized to create message passing mechanisms in some well-known multikernel operating systems.

2.1 Commodity Hardware

Using messaging and notification to provide inter-core communication at both operating system and application level has been a well-studied technique.

At application level, message passing is provided by a standard set of API called MPI (Message Passing Interface) [8]. MPI is supported over a wide range of commodity hardware like shared memory between two cores in a multicore machine, through TCP/IP between multiple nodes in a cluster or specially built high speed interconnects like infiniband. At operating system level, messaging is used in a microkernel based operating systems to communicate between different services running as processes using an IPC (Inter-Process Communication). Also distributed operating systems have used messaging to maintain a coherent system state across different nodes.

From a theoretical point of view, it is important to differentiate between messaging and notification. Messaging is the mechanism of moving information from CPU A to CPU B. Notification is the mechanism used to inform CPU B when a message is received. In this section, we will discuss the hardware support available in current x86 systems for messaging and notification.

2.1.1 Memory Copy

Commodity multicore x86 machines do not provide any special hardware support for message passing. The message sender needs to copy data to a specified address in a shared memory window and the receiver copies the data out from that shared memory location into a local buffer.

There is a fundamental source of inefficiency in this method. The sender as to copy the data in to the shared memory buffer and the receiver has to copy out the message in to a local buffer. This is called Copy in/Copy Out technique and involves two costly memory copies especially if the message size is larger.

There are many techniques in which this problem has been addressed. The receiver process can pre allocate the buffer and pass the virtual address to the kernel. The kernel can set up a shared mapping for the buffer between the sender and the receiver. So when the sender sends a message, it is directly copied to the receiver allocated buffer and there is no need to copy out the message from a shared buffer. This technique is used in KNEM, a high performance intra-node MPI communication technique. In current hypervisors like Xen, this problem is addressed using page flipping to provide high speed networking between virtual machines. In this technique, the memory page to which the sender writes is moved from page tables of sender to receiver using some mechanism such that it becomes available in the virtual address space of the receiver.

2.1.2. Notification

In this sub section, we will discuss the three major primitives used for notification on x86 hardware today – Polling, Inter-processor interrupts and MONITOR/WAIT instructions.

2.1.2.1 Polling

Polling is a technique in which a value in memory is repeatedly checked until an expected value is received. It is also referred to as spinning. The greatest advantage of polling is that it gives the lowest possible latency from hardware. As soon as a message is made available by the cache coherency protocol, the CPU will come out of spinning and continue with execution. Also, the application gets the greatest possible throughput as we don't need to wait on any notification to act upon. Also polling can occur in user space without requiring kernel mode switching, which eliminates the unnecessary overhead.

However it has its own disadvantages. In polling, the CPU has to keep spinning on a memory location wasting processor cycles while it could have been doing something more useful. If the processor is dedicated to this one application thread, then this is not a problem as the processor

would have been idle anyways (other than additional power consumption) but most of today's commodity hardware are designed for multi-tasking and hence polling can be used in only those application which have the resources to dedicate for it.

2.1.2.2 Inter-Processor Interrupts

Inter-processor interrupts are used for synchronizing processors in x86 SMP machines [9]. They use the APIC/LAPIC infrastructure between CPUs which on modern day machines use the same cache coherency protocol for message passing.

Within the Linux kernel SMP implementation, IPI is used for following reasons –

1. Coordinating system management operations like restart and shutdown
2. A CPU which steals a task from other CPUs run queues sends them an IPI to coordinate scheduling
3. It is also used to shootdown the TLB of other CPUs. When another thread modifies the physical pages associated with a process invalidating the mapping on other CPUs, a IPI is sent to remove the TLB mapping from the other CPUs.

The main advantage of IPI is that there is no need for the CPU to spin on a memory location. So the CPU can do some useful work. However there are several disadvantages to it. IPI carries the overhead of executing a interrupt handler and the cache pollution that comes with it. Also an IPI has to be sent from kernel space. So in order to send it from user space, we need to switch mode adding to the overhead involved.

2.1.2.3 Monitor/Wait

The SSE instruction set introduced the monitor/wait instructions in x86-64 architecture [9]. The instruction pair allows a core to put to sleep waiting on a write to a particular memory location. The monitor instruction allows to set the memory location to wait on and the wait instruction waits for a write on the set address. Even though it is a proprietary implementation, under the hood it is believed to work using the cache coherency protocol. Whenever the cache line containing the memory location monitored is invalidated, the core is awakened.

This has several advantages. It avoids the core from spinning on a memory location there by reducing power consumption. Also, the spinning core is put to sleep thereby freeing up the resources to be used by another core.

2.2 Multikernel operating systems

In this section we will discuss certain research operating systems which use multiple OS on a single machine but provide user with a single system image and familiar shared memory programming model for application. These OS fall under the category of multikernel OS to which popcorn and heterogeneous popcorn work discussed in this report belongs. We will discuss some such popular multikernel OS like Barrelfish, FOS etc. and see how inter-kernel communication is performed in those systems.

2.2.1 Barrelfish

Barrelfish is the first multikernel operating system [4] developed by ETH Zurich in collaboration with Microsoft research. The authors define a multikernel OS as one in which inter-core communication is explicit, the OS structure is hardware agnostic and the state is replicated instead of being shared. The motivation behind such a design is to make software more closely match the current and future multicore machines and also the future heterogeneous systems. The authors strongly argue that such a OS, which is fully distributed under the hood should still provide programmer with familiar programming interface as available today for current SMP machines. They provide an implementation of OpenMP shared memory threading library using remote thread creation technique and show respectable performance.

The inter-core messaging in barrelfish is innovative on current multicore machines. Each message is allowed to take up 64 bytes or one cache line on x86 machines. Each message has a sequence number at the end of it. The sender writes the message to the shared memory location and the receiver spins on the sequence number. When the expected sequence number is seen, the message has arrived and is processed.

In order to avoid wasting CPU cycles polling on the memory location for a message, barrelfish uses a hybrid notification approach using both IPI and polling. They first spin on the memory location for a certain amount of time and fall back to IPI if the message is not available within that time. The authors mathematically show that setting the spinning time to the time needed to service the IPI provides a good results.

2.2.2 Factored Operating System

Factored Operating System (FOS) was introduced by Wentzlaff et al. from MIT in 2009 [10]. The basic idea of FOS is to run different OS services pinned to specific cores. The user space processes will send messages to these cores to access the services rather than running the service on different cores as and when needed as stock operating systems. According to [11], the overhead incurred due to message passing is roughly equivalent to making a syscall to the OS. Still FOS achieves performance gains through improved cache behavior as a result of pinning OS services to specific cores.

FOS messaging channels are dynamically allocated at runtime by hashing an application provided messaging channel ID. Each authorized process should provide an authentication code, which is a unique 64-bit "capability value" before putting the message on the channel. FOS provides support for both use space and kernel space message. User space messaging is realized using shared memory between processes and kernel space messaging is achieved through kernel assisted copying of message from current to destination process's heap. The user and kernel space messaging is used in a hybrid approach: initial messages are sent over kernel. If a certain number of messages are sent within a specified time, then it switches over to user space messaging mode.

2.2.3 Corey

Corey advances the argument of "applications should control sharing" [12]. It is like a thin monolithic kernel (termed "exokernel") than a multikernel, but shares the idea of replicated not shared states between cores. It was developed by MIT. The authors observe that shared state within the kernel is responsible for scalability bottlenecks and argue that scalability gains could be made

if OS had knowledge at fine-grained level, if a particular resource used by the application like a virtual memory or file descriptor is strictly local to one core or needs to be shared between cores.

This goal is accomplished by three OS mechanisms –

- Address Ranges – The applications can provide the OS information about whether a data structure is strictly private or shared between multiple cores during its allocation. The private objects are mapped by local core only and the shared objects are mapped by multiple cores
- Kernel Cores – Certain kernel functionality and the necessary data structures can be pinned to a dedicated core to avoid sharing and cache invalidations
- Sharing – The OS provides primitives using which the application can define whether a resource is local or shared. This can be used to define the kernel resources locally which avoids the unnecessary overhead of using the shared lookup tables for these resources.

Chapter 3: Messaging Layer

Messaging layer is the heart of Popcorn Linux. It enables communication between different kernels running on different cores or cluster of cores. The latency of message passing is one of the most critical aspects that determines the overall performance of Popcorn since every other service is built on top of it. The design of messaging layer highly depends on the underlying hardware architecture. In a cache coherent shared memory multiprocessor architecture, the message passing can be built using cache coherency while in a distributed memory architecture, the messaging layer needs to use some kind of interconnect like PCIe, Infiniband or Ethernet.

3.1 Setup

There is no single chip with coherent/non-coherent shared memory containing ARM and x86 architecture available as of today. ARMv8 was the first ARM architecture (aarch64) to implement 64 bit processors. Chips based on ARM64 started coming into market in late 2014 with ARM introducing its first development platform for aarch64 called Juno development board [8]. However it does not provide a generic PCIe or other interface through which we can connect to a x86 machine to create a high speed interconnect based distributed system. In late 2014, Applied Micro (APM) working with ARM came out with the first aarch64 based server chip and released its development platform called APM X-gene Mustang [9]. As a server market product, it provides PCIe interconnect on the motherboard which facilitated in creating our ARM64-x86 distributed system setup.

Our setup consists of an APM X-gene Mustang board containing eight aarch64 cores connected to a 4 core x86_64 AMD machine. These two machines provide PCIe slots through which we can establish a high speed interconnect. However, since both of them were standalone motherboards with PCIe slots, there was a need to find a solution which could connect two PCIe slots by using some kind of hardware interconnect. We used Dolphin interconnect solution's IXH610 host adapter [10] on the PCIe slots of both ARM and x86 machines and connected them through Dolphin's custom cables to create a Non-Transparent Bridge (NTB) based solution as shown in Figure 2.1 below.

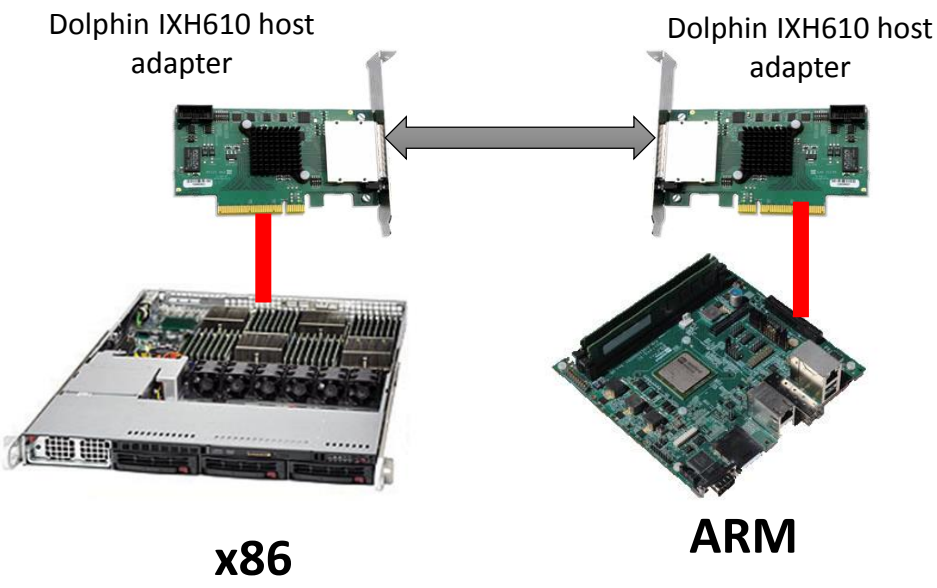

Figure 2.1: ARM-x86 hardware setup

The details on Dolphin interconnect solution is provided in the following section.

3.2 Dolphin interconnect

3.2.1 Hardware

Dolphin IXH610 host adapter is a PCI Express 2nd generation adapter which supports both transparent and non-transparent bridging architecture, providing high speed connectivity of up to 40Gbps using a standard PCIe external cabling system. In transparent bridge architecture, the device can operate using standard PCIe drivers without the need for any special software support. However in non-transparent bridging (NTB) architecture, the device can provide Remote Direct Memory Access (RDMA) using the software stack provided by Dolphin interconnect. It provides support for both CPU assisted programmed IO (PIO) based data transfer and direct memory access (DMA). More details on the Host adapter can be found at [10].

The Dolphin Express IX cables which support standard PCI Express x8 iPass cables is used to connect the host adapters on ARM and x86 machines.

3.2.2 Software

Dolphin interconnect provides a complete software stack with multiple interfaces like SISCI, supersockets, standard TCP/IP network interface etc. The software architecture of the Dolphin software solution is given in Figure 2.2.

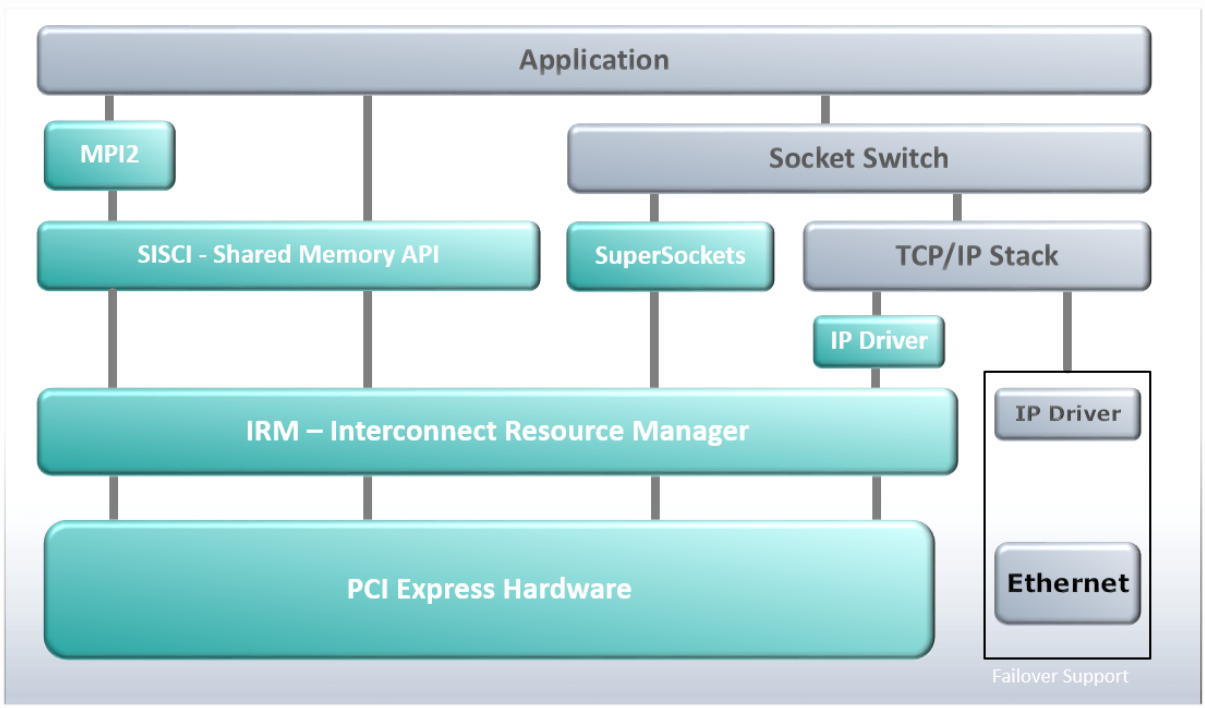

Figure 2.2: Dolphin software stack overview

Interconnect Resource Manager (IRM) is the main software component which manages resources like mapped memory regions, DMA and interrupts. It is responsible for bringing up the devices during power up and also implements a heartbeat mechanism which is used to detect and manage the network of nodes. It provides a common kernel space interface called GENIF which can be used to provide multiple different user space interfaces. Interfaces like SISCI (Software Infrastructure for Shared-memory Cluster Interconnect) [11], supersockets [12] and the standard TCP/IP interface are developed as clients on top of the GENIF interface.

3.2.3 GENIF Interface

The GENIF interface is a kernel space application programming interface provided by IRM software module of the Dolphin software stack .The GENIF interface is a common interface used

by all other modules to provide different interfaces for user space applications. Messaging layer is a kernel module which necessitates the use of a kernel space interface. As GENIF provides a common kernel space interface, it is used as the interface to communicate with the host adaptor and our kernel module is built as a client on top of it.

The sequence of events needed to establish a channel between two machines connected by Dolphin PCIe host adapter using GENIF interface is shown in Figure 2.3. The flow is explained briefly below

- **Create:** Before creating any channel, SCI interface needs to be opened. Then an SCI instance for each channel can be created to obtain a handle. Using the instance handle, a local memory segment and an interrupt to associate with the memory segment can be created. The local memory segment is exported and made visible to all remote nodes.
- **Initialize:** Once the remote memory segment is visible, a connection is established with the remote memory segment and mapping of it is created in the local memory space. Also, a connection is established with the associated remote interrupt to create a local interrupt handle for the remote interrupt. If DMA is to be used for data transfer, the DMA module needs to be initialized. DMA initialization and usage is explained later.
- **Usage:** After creating and initializing a channel, direct processor assisted copy (PIO) or DMA can be used to copy data from local segment to the remote segment. Once the data copy from local to remote memory segment is completed, the interrupt handle associated with the remote segment is raised to notify the remote node about the new data available in its memory segment. The remote node then copies the data from its memory segment to some a local memory location and raises the interrupt mapped by it, which is associated with our local segment to notify that the remote memory segment is available to be used again for data transfer.
- **De-initialize:** Once the channel is no longer needed, the local mapping of the remote memory segment is removed and the remote memory segment is disconnected. If initialized, the DMA channel is also de-initialized as explained later.

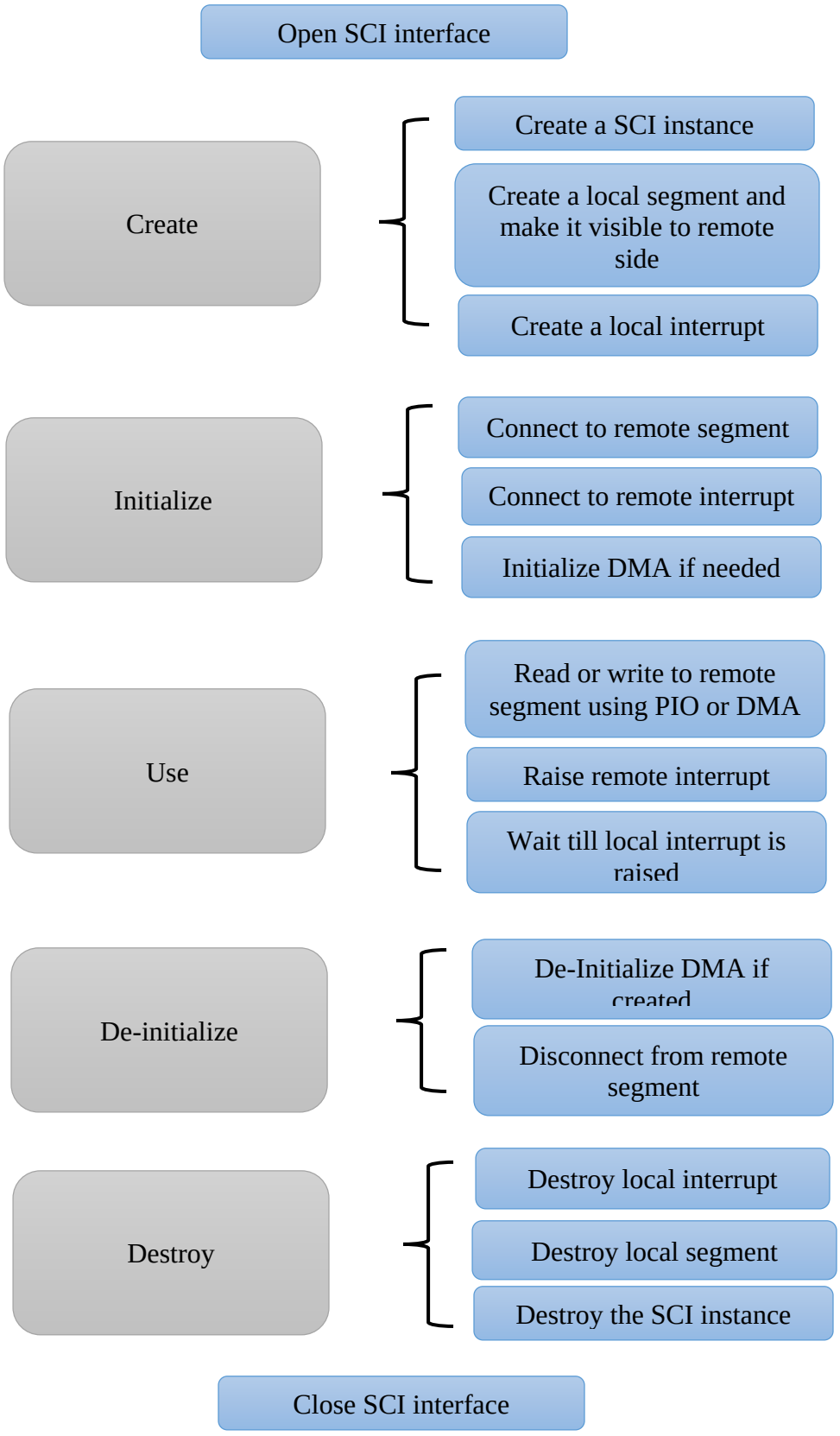

Figure 2.3: GENIF interface – Sequence of events

- **Destroy:** The local interrupt is destroyed. The local memory segment is first made unavailable for connection to the remote nodes and then destroyed. Finally the SCI instance is also removed to completely destroy the channel. The SCI interface is closed once all channels are destroyed.

3.2.3.1 DMA

If DMA is used for data transfer, we need to first initialize the DMA module. The flow of events for using the DMA module is detailed in Figure 2.4 below.

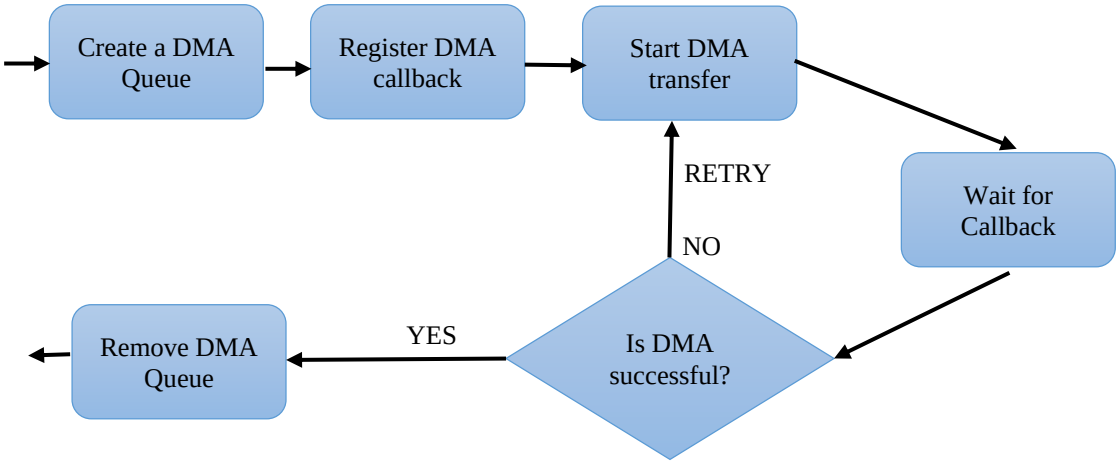

Figure 2.4: DMA module usage flow chart

During DMA initialization, for each channel, a DMA queue is created. A callback function is registered per queue. DMA transfers are asynchronous. So a DMA transfer request is issued by starting the DMA transfer. Then we need to wait on the DMA module to complete the DMA transfer from local memory segment to the remote memory segment. Once DMA transfer is completed successfully or if any errors are encountered, the DMA module calls the callback function associated with the queue with the status of the transfer. Now, if the status shows that the DMA has failed due to any reasons, we simply retry the transfer. If not, the data transfer is successful. During de-initialization, we need to free the DMA queue that was associated with the channel.

3.3 Design

In this section, we discuss the design of the Popcorn messaging layer for an ARM-x86 system. The messaging layer is a multi-threaded pluggable kernel module which provides inter kernel communication in Popcorn Linux. It enables message passing from one kernel to another kernel. Each message has an associated header with a message type field. During boot process, each kernel registers different callback functions that need to be executed for different types of messages. When a message is received at a kernel, based on the type of message, the associated callback function is executed. The primary design objective is to provide a messaging layer which can send messages with as minimum latency as possible. The design of the messaging layer is shown in Figure 2.5 below.

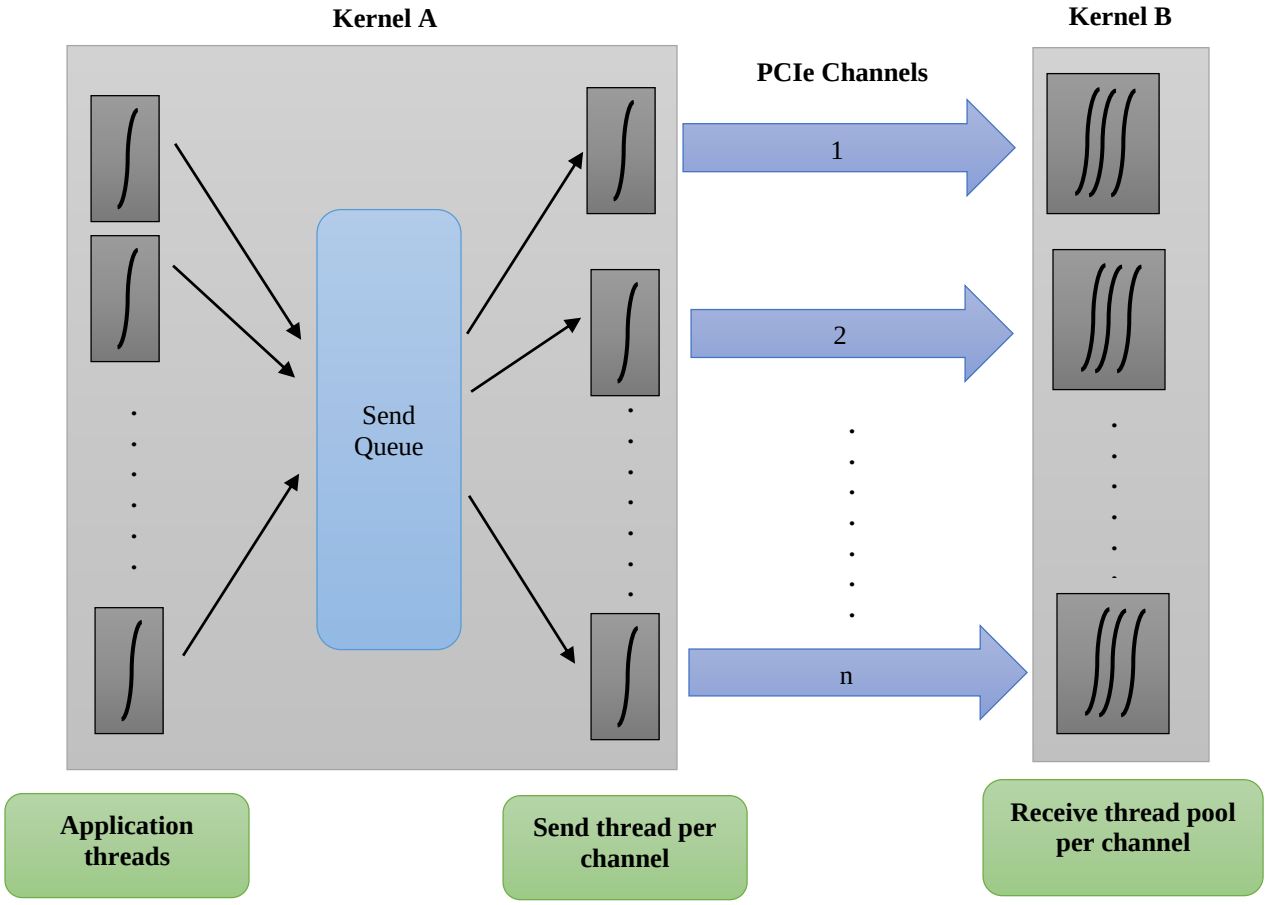

Figure 2.5: Messaging layer architecture

3.3.1 Architecture

As seen from the Figure 2.5, the messaging layer communicates between kernel A and kernel B by creating N PCI Express channels between them. Each kernel consists of a send and a receive side.

Send:

Any thread that wants to send a message using the messaging layer is called an application thread from the messaging layer point of view. However this thread can be a user space application thread or a kernel space thread used to manage any Popcorn Linux service. These application threads that want to send a message from one kernel to another, create a message with appropriate information in the message header and put it into the send queue. The send queue is a global queue which contains all the messages from different application threads.

Channels are established between different kernels using the GENIF interface APIs provided for the PCIe interconnect as explained in the previous section. There is one send thread associated with each channel on the send side. The send threads are usually blocked waiting on a message to be available in the send queue. Whenever a message is present in the send queue, the send thread picks the message from the send queue and sends it across the channel using DMA or CPU assisted copy (PIO) as explained before. The send thread then waits for interrupt from the remote side which signals that the data is copied out of the channel and it is free to be used again. Once the channel becomes free, the send thread picks the next message in the queue if available, else it starts waiting for the next message.

Receive:

On Receive side, there is a pool of receive threads associated with each channel. These threads are waiting on the message to be available on their channel. Whenever a message is received, the remote side triggers the interrupt associated with the channel on which the message is available. A thread from the receive pool of that channel is woken up to copy the data from the channel to a local memory location. This thread then raises the interrupt to the remote side to signal the completion of data transfer. After signaling, the thread parses the message to find the associated

callback function using message type information and executes the callback function. Once the callback function is executed, the message is processed completely and the receive thread returns to its pool and starts waiting on the next message to be available on the channel for processing.

3.3.2 Design Considerations

Some of the design choices considered and the decisions made in the architecture described above is deliberated in this section.

- **Need for send queue:** The application threads instead of directly sending data through the channel puts it into a send queue. This is necessary because the application threads might send large number of messages at the same time or in a short interval and the messaging channel might be slower to handle such high number of requests. In the above scenario, if application thread is itself directly sending the message, then it has to wait till the channel becomes available, potentially wasting its time waiting on the channel. So, as in any typical producer-consumer design model, there is a need of a buffer/queue to handle the difference in rate of production and consumption.
- **No receive queue:** Using the same producer-consumer model, we might argue that there is a need to have a receive queue as the rate of receiving messages may be higher than the rate of processing them, as processing depends on message type. Even though it is true, the rate mismatch will result in the message being left in the send queue on the sending side instead of lying in the receive queue waiting to be processed. As the message has to anyway wait to be processed, it does not make any difference whether it is waiting in the send or receive queue. This is mainly because the application thread sending the message is not blocked waiting on the message once it puts it on to the send queue.
- **eceive thread pool:** We need a pool of receive threads per channel instead of a single thread because the receive thread processes the message itself by executing its callback function. Since the callback function is message/application specific and not defined by the messaging layer, the callback function may be long and might take significant amount of time. If there is a single receive thread per channel, the channel will be blocked and unavailable until the message is processed. However with a receive thread pool, the channel

is available for use again as soon as the data is copied from the memory segment of the hannel.

- **Send buffer pool:** The application thread has to put the message to be sent to the send queue by allocating a new message buffer and copying the message data on to it. Once the message is sent by the send thread, the message buffer is freed. This results in allocating and freeing of messages using vmalloc and vfree which can potentially increase the latency as these operations can internally block. We can overcome this problem by creating a send buffer pool, which is a pool of pre-allocated buffers of maximum message size. The application thread just needs to find a free buffer from the pool, copy the message to the buffer and set it to be in use. Once the message is sent, the buffer is released back to the pool and is available to be used again.
- **Global send buffer:** The contention on the global send buffer can be very high in systems with high core count. This can result in varying latencies for message passing which is undesirable. In such cases, we can use a distributed send queue where the application threads can insert messages to randomly chosen queue. Each send thread can be associated with a send queue so that send threads don't contend on the same queue. However we need to ensure that the messages are not getting delayed by queuing in a single send queue rendering other send threads unusable. In a low core count case like our setup, the contention is very low and negligible. So distributed send queue is not needed.

3.3.3 Thread priorities

Another important factor that effects the speed/latency of the messaging layer is the number of threads and their thread priorities.

We need to ensure that the send and receive threads are having higher priority compared to the application or any other kernel threads. This is needed because messaging layer is a critical component of the system. When a message is sent across from one kernel to another, we need to ensure that it reaches its destination as soon as possible. Under high load condition, if the send and receive threads have normal priorities, it might not be scheduled immediately due to other kernel

threads, thereby delaying the messages. So we set the send and receive threads to use SCHED_FIFO scheduling policy with a very high priority value of 10.

Another factor that is critical for performance is the number of receive threads per pool and the number of channels to be used. We create the number of channels to be equal to the number of processors per node. So in our setup, as the x86_64 machine has 4 cores, we have used 4 channels. We use two receive threads per pool per channel. Hence there are a total of 4 send threads and 2 receive threads per channel, 8 in total. This is calculated based on the average time taken by the callback functions in the system. Having too many threads per pool can result in increased number of thread switching which is undesirable. Also, we pin the send and receive thread pools of each channel to a separate processor core by using core affinity settings. This improves thread locality thereby improving the performance.

3.3 Results and Analysis

The performance of the messaging layer can be measured in terms of two factors – latency and throughput. Latency is the amount of time taken for a message from the time it is sent from an application thread on one kernel to the time its callback function is executed on the other kernel. Throughput is the number of messages that can be sent through the messaging layer per second. Throughput measures the bandwidth of the messaging layer, i.e. the capacity of the messaging layer to handle messages.

The Hardware details of the x86 and ARM machines used is given in Table 2.1 below.

Feature	x86_64	Aarch64
Processor	AMD Phenom	APM X-gene (ARM v8)
# of cores	4	8
CPU frequency	2.3 GHz	2.4 GHz
RAM	2 GB	16 GB
L1 data cache	128 KB	32 KB
L2 cache	512 KB	256 KB
LLC	2 MB	8 MB

Table 2.1: Hardware specifications of setup

Before we present the latency and throughput values for the entire messaging layer, it is important to understand the latency numbers for raw data transfer over PCIe provided by the GENIF interface of Dolphin interconnect solution. The graphs in Figure 2.6 and Figure 2.7 below show per channel data send latencies on x86 and ARM side respectively. The latencies are measured with varying data sizes. In order to get an idea about the latency numbers, it is compared against local memory to memory copy using *memcpy* and the user space latency numbers using the SISCI interface. The user space latency numbers are measured using the sample applications provided by Dolphin solutions.

As shown in the figures, the data send latency is very less, in the order of few micro seconds. The latency numbers are comparable to a local *memcpy*. The latencies non-linearly increase with increase in data size. In general, the latency in kernel space is less than that in user space. This is mainly due to context switch overhead from user space to kernel space for user space applications.

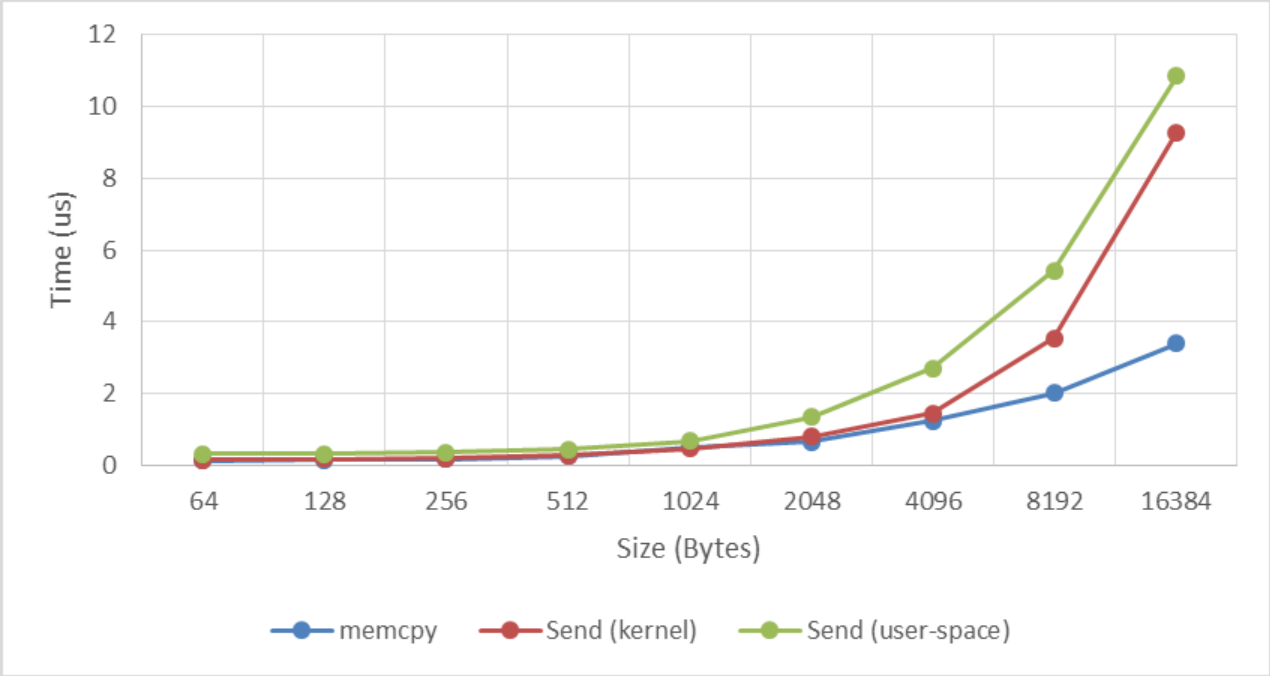

Figure 2.6: x86 PCIe data transfer latency

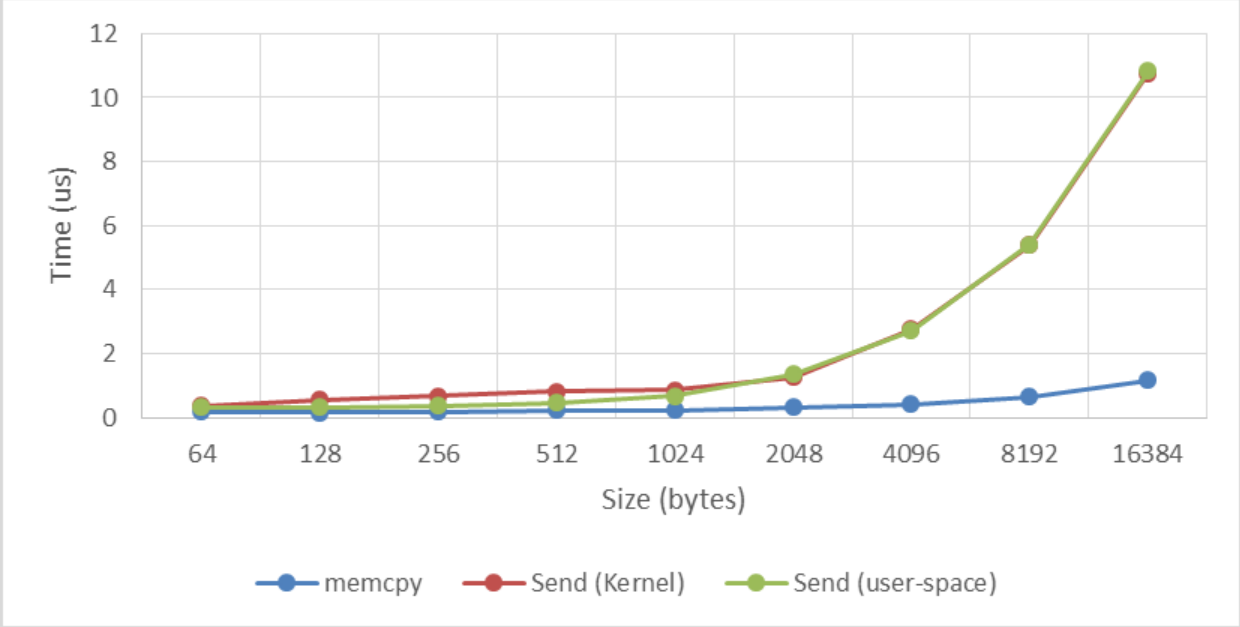

Figure 2.7: ARM PCIe data transfer latency

3.4.1 Latency

The round trip latency for a message to be transferred from x86 to ARM and back to x86 using PIO and DMA for data transfer across 100 messages is shown in Figure 2.8 below. Each message is equal to the size of a page, i.e. 4kB. The average and the standard deviation (SD) for the latency values is tabulated in Table 2.2.

The average round trip latency for a 4kB message using PIO is 28.54us, which is very less. Also the standard deviation is very less which indicates that the latency of messages using the messaging layer is highly deterministic with very little variation. Thus the messaging layer provides a highly reliable communication channel with deterministic low data transfer latencies.

The latency numbers for DMA transfer is more than that for PIO transfer as seen from the figure. This is expected because DMA transfer involves initialization of DMA controller for every data transfer. So, if the size of the data to be transferred is not significantly large, the overhead of DMA controller initialization is not negligible which results in higher latency numbers. So DMA is a viable option only when large amount of data needs to be transferred. Since in the Popcorn use case, most of the data transferred is of size 4kB or around, PIO is primarily used for message passing.

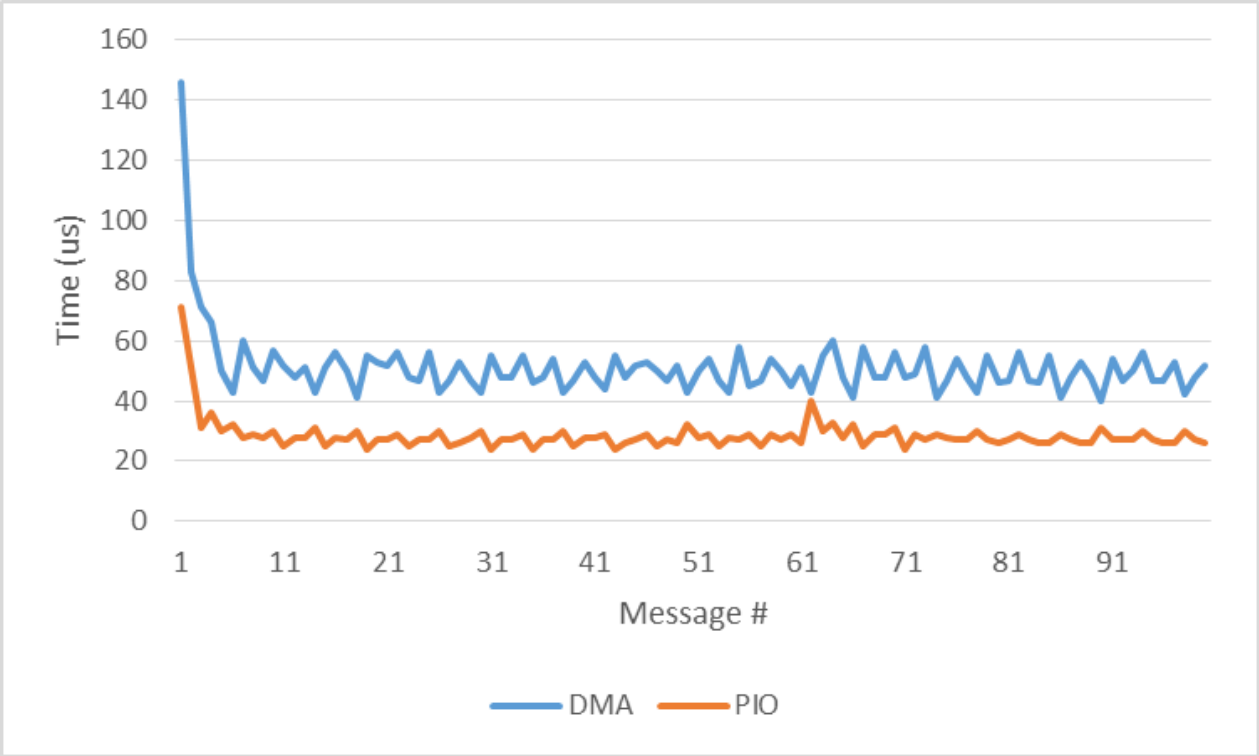

Figure 2.8: x86 PIO and DMA data transfer latency

	Mean (us)	Standard Deviation
PIO	28.54	5.463229
DMA	51.22	11.50018

Table 2.2: x86 PIO and DMA latency

Similarly, the round trip latency for a message to be transferred from ARM to x86 and back to ARM is shown in Figure 2.9 below. The average and standard deviation values are tabulated in Table 2.3. The observations are similar to the observations made for messages originating from x86 side.

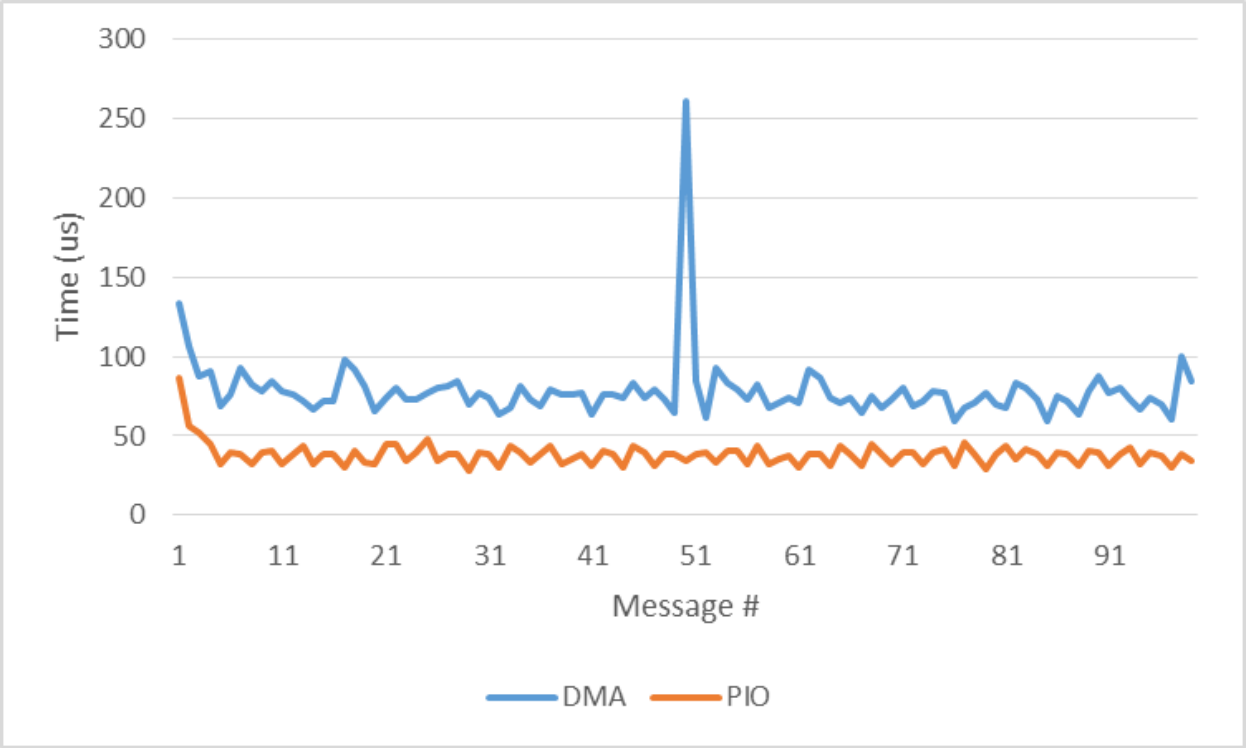

Figure 2.9: ARM PIO and DMA latency

	Mean (us)	Standard Deviation
PIO	38.20202	7.260432
DMA	78.50505	21.24222

Table 2.3: ARM PIO and DMA latency

3.4.2 Throughput

Throughput is the maximum capacity of the messaging layer. It is usually defined as the amount of data that can be transferred per second. For the messaging layer, it can also be measured in terms of the time taken by the messaging layer to send a fixed number of messages.

The time taken by the messaging layer for round trip transfer of 100,000 messages using PIO with varying number of channels and for different message sizes is shown in Figure 2.10 and Figure 2.11 for x86 and ARM respectively.

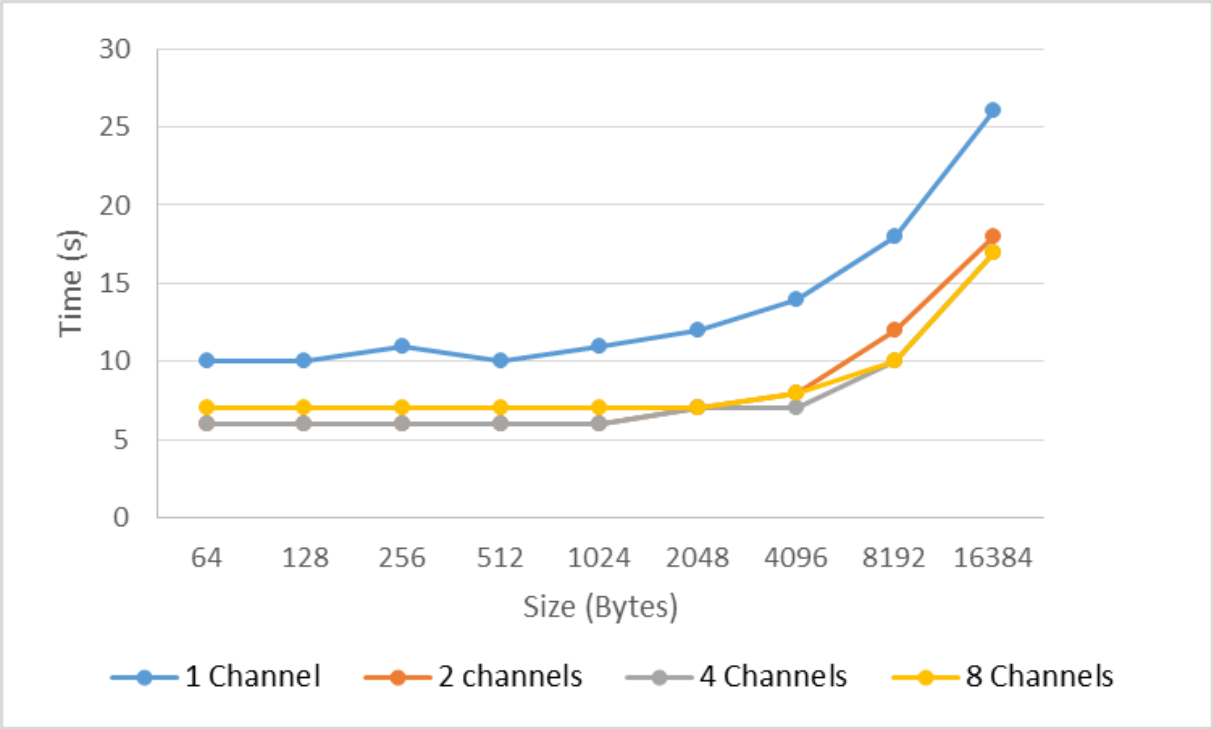

Figure 2.10: x86 PIO throughput

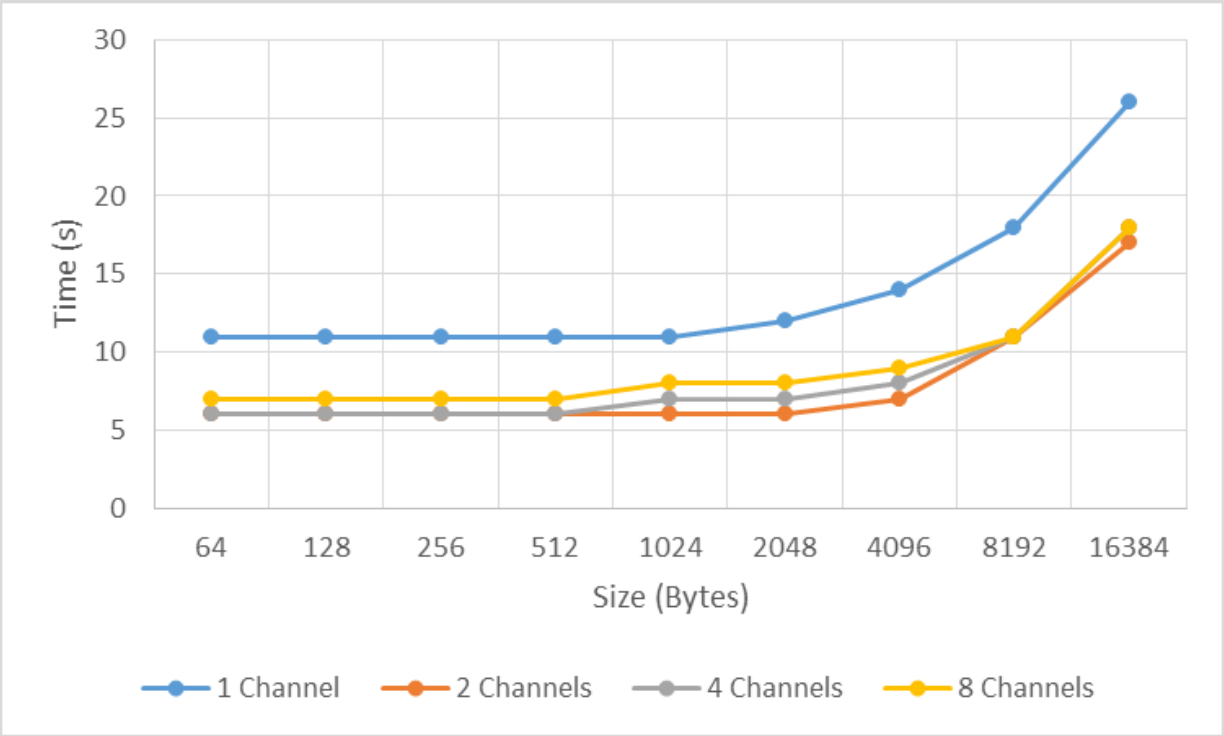

Figure 2.11: ARM PIO throughput

Similar graphs for throughput using DMA assisted data transfer is shown in Figure 2.12 and Figure 2.13 for x86 and ARM respectively.

From the throughput graphs, the following observations can be made:

- The throughput increases or the time to send 100,000 messages decreases as the number of channels increases
- The time taken to send 100,000 messages increases exponentially as the size of the message increases from 64B to 16kB
- The throughput for PIO assisted data transfer is more than the DMA assisted data transfer. This is due to the additional initialization cost as explained earlier. So the DMA should potential be faster than PIO when the data size is very high

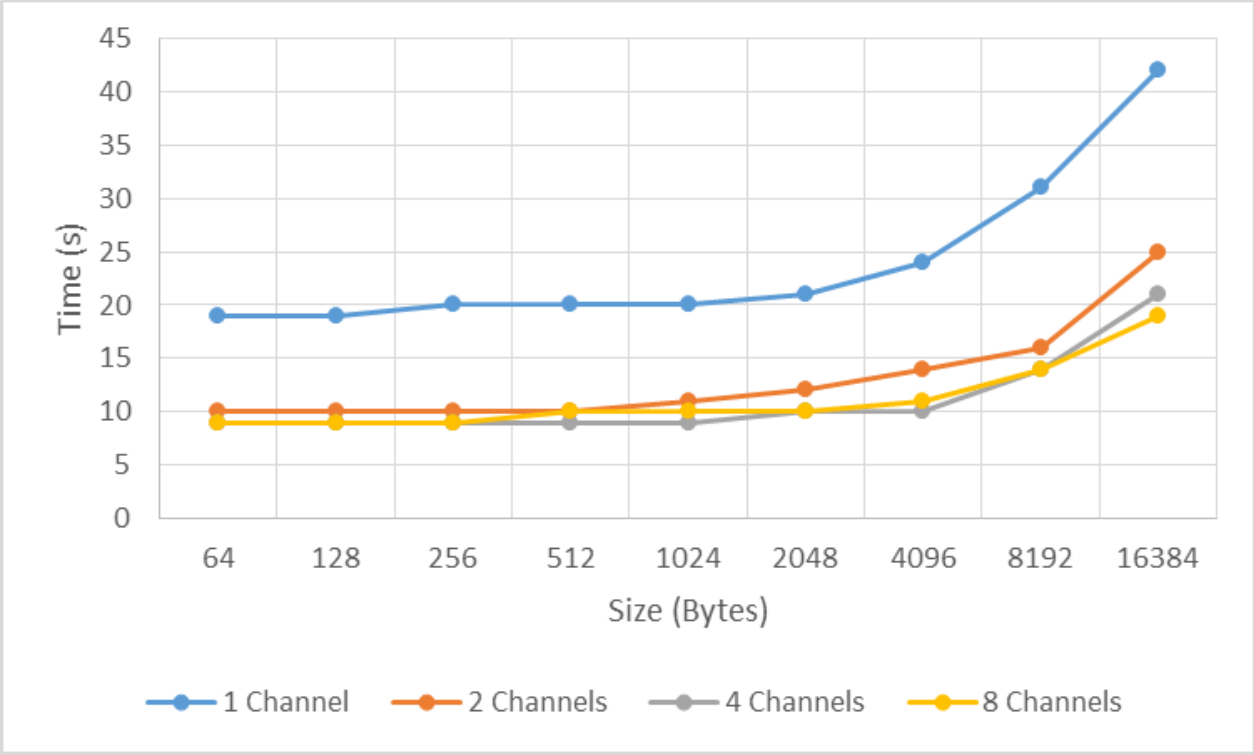

Figure 2.12: x86 DMA throughput

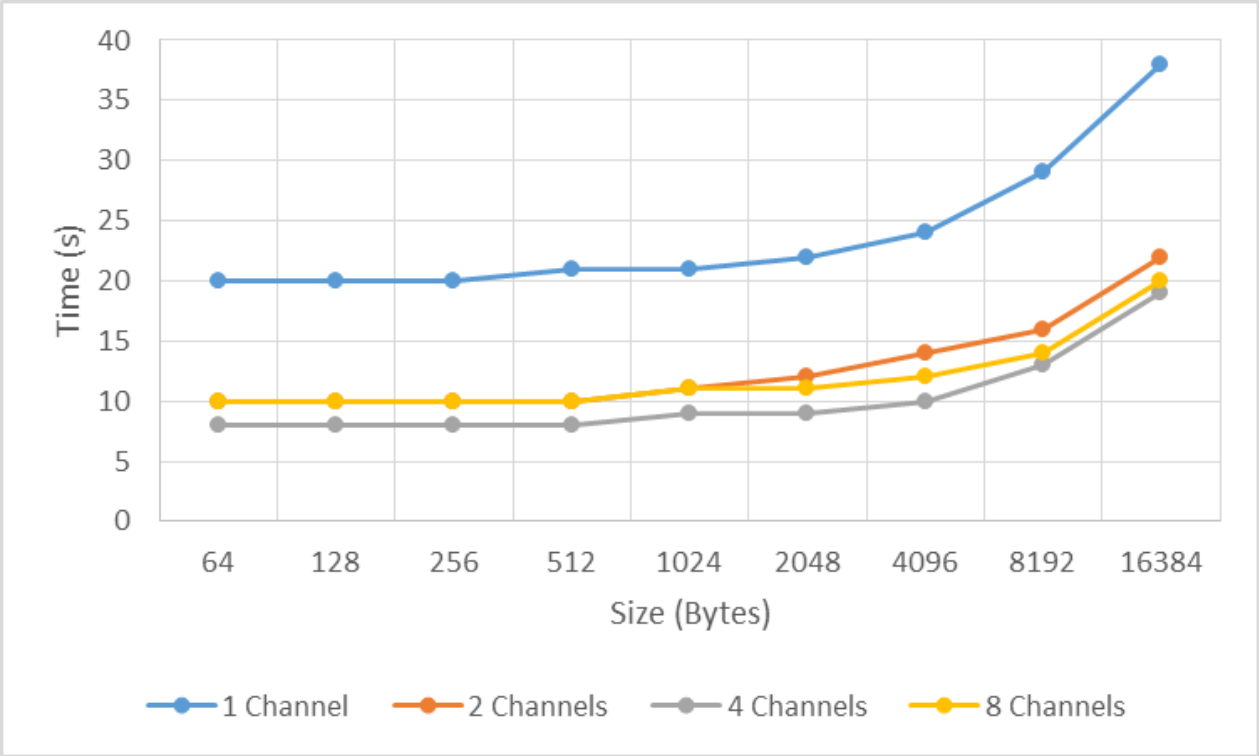

Figure 2.13: ARM DMA throughput

3.4.3 Thread Priority

The impact of thread priority on the latency of messages is analyzed in this section. Figure 2.14 shows the round trip latency for 100 messages sent using PIO data transfer from x86 to ARM at varying send and receive thread priorities. In "With priority" case, the threads are assigned a priority of 10 with SCHED_FIFO while without priority is using normal thread priority. Table 2.4 shows the average and standard deviation for latency numbers with and without thread priority.

	With priority	Without Priority
Mean (us)	28.54	55.91
SD	5.463228779	96.10222829

Table 2.4: Thread priority impact: Mean and SD

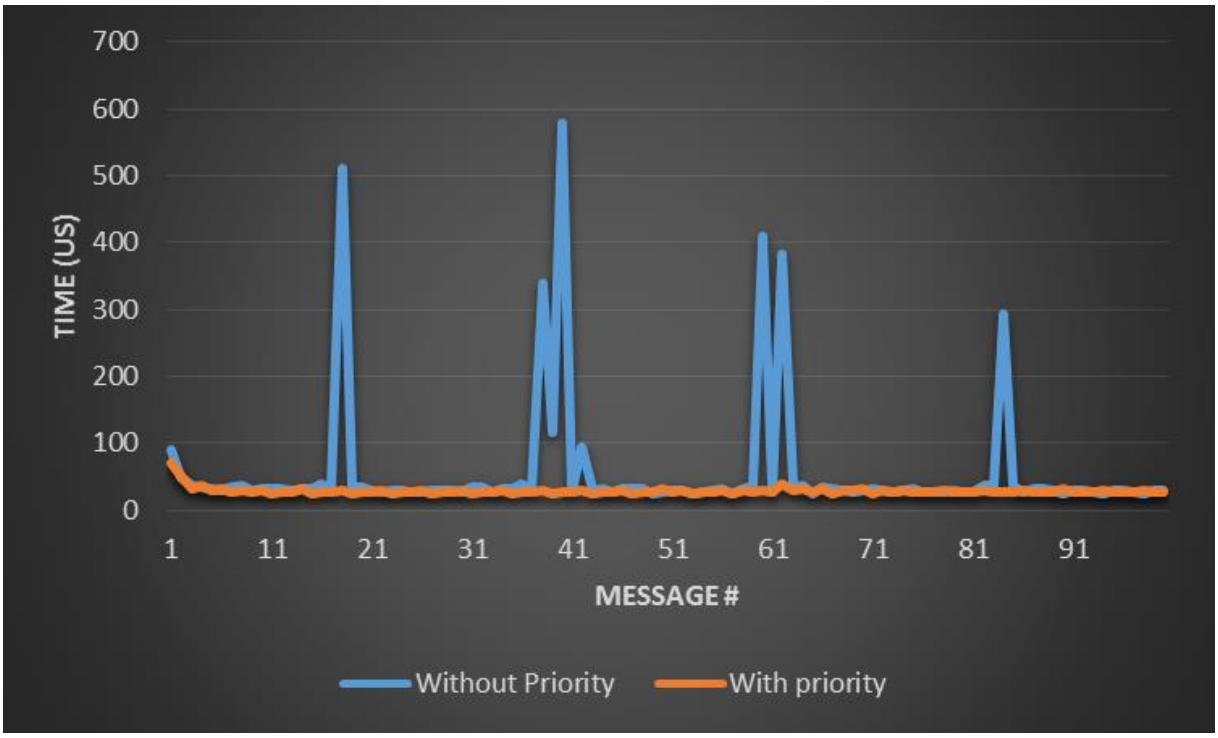

Figure 2.14: Impact of thread priority

As shown in Figure 2.14, when the thread priority of send and receive threads is not set to a higher priority level, the latency values are not deterministic and a large number of spurious values are observed. As seen from figure, sometimes latency values can go up to 600us. This increases the average latency as well as the standard deviation for the case with normal thread priorities as seen from Table 2.4. So assigning proper thread priorities for the messaging layer threads is very important to ensure deterministic latency numbers.

3.4.4 Queuing latency

Another scenario for the messaging layer is the condition where multiple application threads continuously insert messages into the messaging layer thereby overloading the messaging layer. The latency or round trip time for the message in this case depends on both channel latency and the amount of time for which the message has to wait in the send queue before messaging layer threads are scheduled or channel becomes available. This scenario is simulated here.

Figure 2.15 shows the latency or round trip time for 100 messages inserted continuously on x86 side by 4 application threads. Each application thread inserts 25 messages.

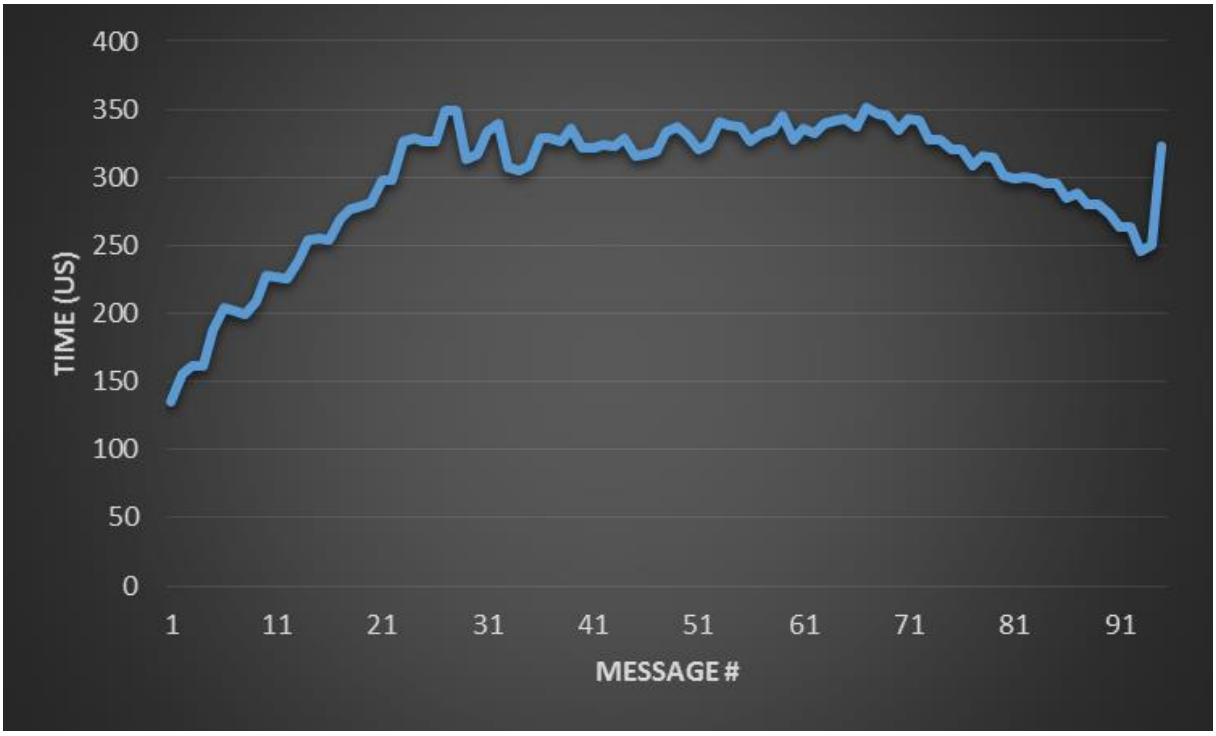

Figure 2.15: Latency numbers under overloading

As seen from the figure, initially the latency of messages keeps on increasing due to the time spent by messages waiting in the send queue. However, after some time, the messaging layer threads schedule the application threads out as they are of higher priority and process all the messages present in the queue at once. So it can be seen that the latency numbers become consistent after the initial increase. However, the initial increase is observed due to the latency involved in scheduling out the application threads. For the above scenario, the average and standard deviation for the latency numbers is 285.57us and 73.18 respectively.

3.4 Conclusion

The messaging layer is one of the most critical parts of Popcorn Linux operating system. Design of the messaging layer is highly dependent on the underlying hardware architecture. In our heterogeneous ARM-x86 distributed system setup, the PCIe host adapter from Dolphin solution

with non-transparent bridging (NTB) technology is used to build the messaging layer. The messaging layer is designed as a multi-threaded framework with send and receive threads using a typical producer consumer design model. It consists of 4 channels using PIO for data transfer.

The latency and throughput numbers of the messaging layer are acceptable. There is no difference observed in values from x86 and ARM sides. Also, determinism provided by the messaging layer is a very critical part, as the latency is guaranteed to be bound within a particular value. The priority levels of the messaging layer threads has high impact in ensuring the determinism in latency values. Overall, the messaging layer designed above provides deterministic latency and desirable bandwidth values.

Chapter 4: Heterogeneous-ISA Popcorn

In this chapter, we briefly discuss the design changes and other modifications done to Popcorn Linux to create the ARM-x86 heterogeneous operating system, Heterogeneous ISA Popcorn. This work involved first porting the entire Popcorn Linux to work on aarch64 architecture and making design changes in Popcorn services like task migration and memory consistency protocol subsystem to get applications seamlessly migrating across ARM and x86 cores. Also, a new binary layout was created by the compiler team within Popcorn project. A new compiler framework was developed for creating the application binaries that can run on such heterogeneous architecture.

In the following sections, first we describe the application binary layout followed by the task migration technique that allows threads to migrate across ARM and x86 cores in Popcorn Linux. This is followed by a discussion on other design changes.

4.1 Binary Layout

The traditional binaries for applications are compiled for a particular architecture and contain the machine code for that architecture. So this binary cannot be executed on a heterogeneous ISA system. In order to run an application across multiple architectures, we need to ensure the following:

1. The application segment running on a particular architecture has the machine code or *.text* section data corresponding to the same architecture
2. The global variables and other global symbols are present at the same virtual memory address on both architectures

The compiler team in Popcorn project has created a new compiler framework which creates a binary layout that can be used to execute applications on ARM-x86 heterogeneous Popcorn Linux. The binary created has the following characteristics:

1. Two separate binaries, one for ARM and x86 architecture are created by the compiler for each application.

2. All the symbols in the global data section are aligned on the binaries for both ARM and x86 architectures. This means, all the global symbols in *.data*, *.bss* and all other global data sections are present at the same virtual address on both ARM and x86 binaries.
3. All the function symbols in the text section of the binary is aligned in ARM and x86 binaries. This allows the application to migrate between ARM and x86 architectures at function boundaries.
4. The same alignment rules are followed for any supporting libraries like glibc etc. which are linked to the application. All required libraries are linked statically and dynamic linkage is prohibited.

These binaries for ARM and x86 should have the same file name and must be kept at same location on both ARM and x86 file systems.

4.2 Thread migration

In this section, the migration of an application thread from x86 to ARM in Heterogeneous Popcorn Linux is explained in detail. The application binaries are created as explained in the previous section. The migration of an application thread is primarily handled by the process server module of Popcorn Linux.

4.2.1 Glue code

At the beginning of the compilation, the Popcorn compiler inserts a pragma on top of the functions which need to be migrated from one architecture to another. Let the function to be migrated be named as *foo*(). During compilation, the compiler adds code to transform the function *foo()* to create a wrapper function *wrap_foo()* which encapsulates the real *foo()* function within it. The wrapper function wraps the real function along with the glue code needed for migration. The glue code does the following:

- It adds the code to pack arguments needed for the function *foo()* into a particular memory location in the global data section. This is needed because the function arguments are usually passed using registers during function calls. Since x86 and ARM do not have similar register banks and the Application Binary Interface (ABI) for ARM and x86 are

different, the calling convention for passing function arguments is different between ARM and x86. So, the function arguments are passed in a fixed global memory location.

- It also adds the code to find the state of execution of the wrapper function to find if it is packing or unpacking the function arguments
- While unpacking, the wrapper function has the logic to get the function arguments packed at a fixed global memory location and put those values into the registers according to the calling convention for the current architecture: ARM or x86.

4.2.2 Binary Loading

The ELF binary format loading in Popcorn is modified to create a new ELF loader. When binary is being loaded into the memory for execution, we extract the information on the different sections of the binary like text, data, BSS etc. This information is used while creating the virtual memory areas (VMA) during binary loading.

For global data sections like *.bss*, we need the memory consistency protocol to pitch in to ensure that the global memory is consistent across different threads of the application running across different kernels but for .text section, we need to ensure that the memory consistency protocol does not pitch in and the machine code for the current architecture is picked locally. In order to ensure this, during ELF loading, the section information is extracted for all the different sections in the binary. This is used to set a flag: VM_FETCH_LOCAL in the VMA for particular sections which need to be locally fetched. This will inform the memory consistency protocol not to act for these sections of VMA thus making those information to be fetched from the local binary.

Also, the file offset for the data corresponding to a particular virtual memory location is stored in the VMA. Since the VMA is created on a particular architecture (either x86 or ARM), the corresponding file offset value will be stored. Now, if the data is to be fetched on the other architecture, the file offset in the VMA structure will be wrong thereby fetching incorrect data. So we have implemented a function *get_file_offset()* which can be called at any point in time from the memory consistency protocol to find the right file offset in the local binary corresponding to that particular virtual memory location.

4.2.3 Migration

The process of migration for an application from x86 to ARM side is shown in Figure 3.1 below.

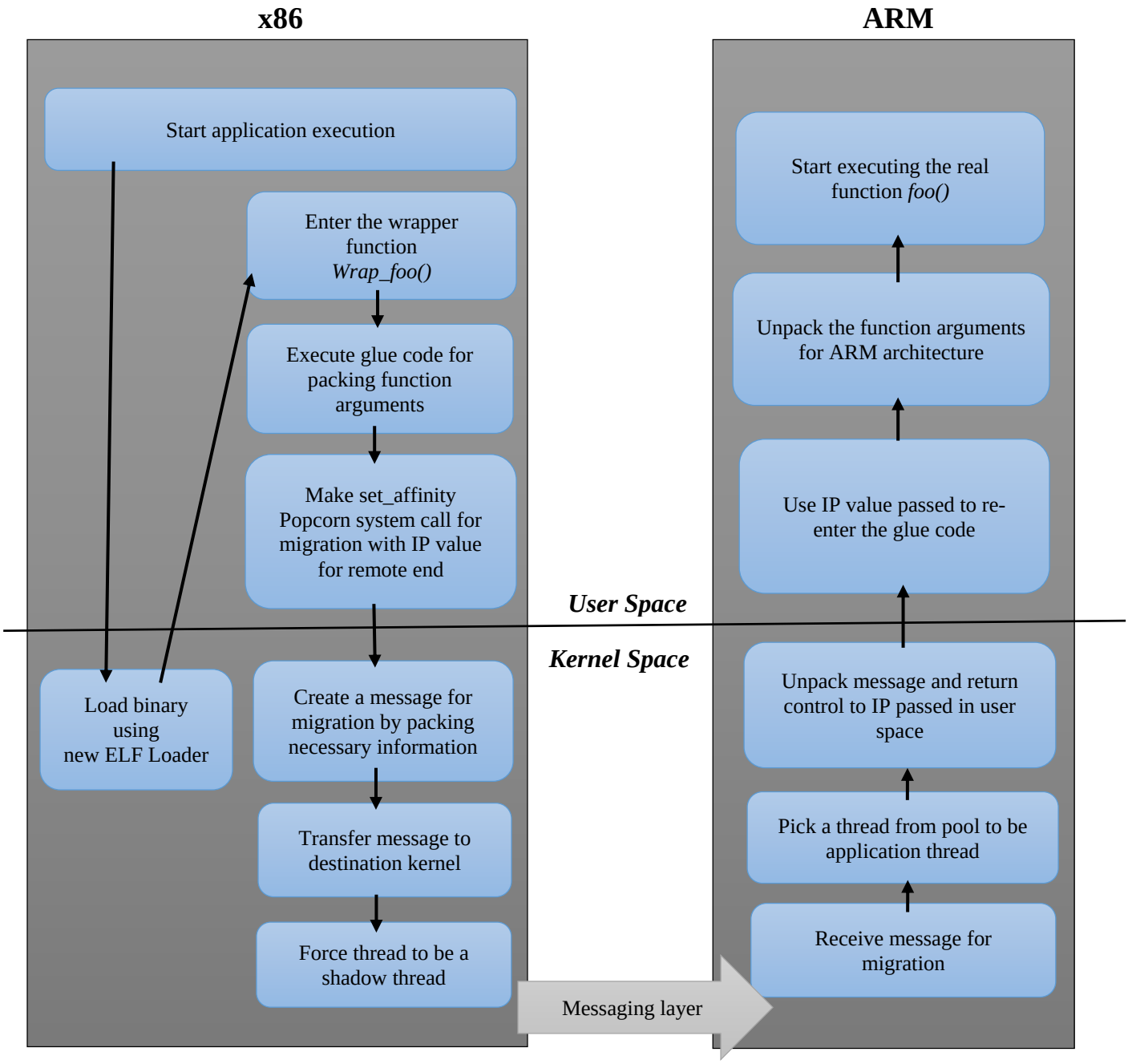

Figure 3.1: Task migration flow - x86 to ARM

When the application execution is initiated on the x86 side, the new ELF binary loader loads the application binary on the x86 kernel and the execution is started. When the application is about to execute the function *foo()*, it hits the wrapper function *wrap_foo()*. The application now executes the glue code as explained in section 3.2.1 to setup the function arguments in the global memory.

Now the application makes a system call to migrate from x86 to ARM side. A new system call *sched_Setaffinity_for_popcorn()* is created which can take the IP address to which the instruction pointer should be set before resuming execution on the other side. The IP address passed is usually the beginning of the wrapper function as the wrapper function is used to unpack the registers on the migrated kernel. When this system call is made, the Popcorn task migration service pitches in.

The task migration service packs the register, thread state information and other needed virtual memory related information corresponding to the thread/application into a message and sends it across to the destination ARM kernel to which the threads intends to migrate. In x86 to ARM migration, the *pt_reg* structure which holds the register values cannot be sent as is because *pt_reg* structure is architecture specific. So important information like stack pointer (SP), Instruction pointer (IP), thread local storage (TLS) information are mapped from x86 to ARM before passed ii in the message.

On ARM side, when the message is received, it is processed to unpack the information. Task migration service creates a pool of shadow threads to assist in migration when the service is started at kernel boot. A thread is picked from this thread pool to act as a shadow thread for the original thread on x86 and all the thread state and other information from the message is updated to the shadow thread. The threads protection state is set correctly and its execution is resumed using the IP value passed in the system call on x86. This resumes the execution at the wrapper function *wrap_foo()* where the function arguments are unpacked according to the calling convention. Now the actual function *foo()* is executed using the arguments present in the registers.

The memory consistency protocol is used to keep the virtual global address space consistent. It uses the VMA flag to decide which section needs to be kept consistent as explained earlier.

References

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Power Challenges May End the multicore era," *Communications of the ACM,* pp. 93-102, Februaury 2013.

[2] A. Venkat and D. M. Tullsen, "Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, Minneapolis, Minnesota, USA, 2014.

[3] AMD , "AMD Unveils Ambidextrous Computing Roadmap," AMD, 05 May 2014. [Online]. Available: http://www.amd.com/en-us/press-releases/Pages/ambidextrous-computing-2014may05.aspx.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, New York, NY, USA, 2009.

[5] A. Barbalace, B. Ravindran and D. Katz, "Popcorn: a replicated-kernel OS based on Linux," in *Ottawa Linux Symposium (OLS '14)*, Ottawa, Canada, July 2014.

[6] M. Sadini, A. Barbalace, B. Ravindran and F. Quaglia, "A Page Coherency Protocol for Popcorn Replicated-kernel Operating System," in *Many-Core Architecture Research Community (MARC) Symposium, ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, Indianapolis, Indiana, USA, October 2013.

[7] A. Barbalace, M. Sadini, S. B. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray and B. Ravindran, "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms," in *ACM European Conference on Computer Systems (EuroSys 2015)*, Bordeaux, France, April 2015.

[8] A. N. Laboratory, "The message passing interface (mpi) standard," [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/.

[9] Intel, "Intel 64 and IA-32 architectures software developers manual," Intel Corporation, August 2012.

[10] A. A. David Wentzlaff, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review,* vol. 43, no. 2, pp. 76-85, April 2009.

[11] A. Belay, "Message passing in a factored OS," Master's thesis, Massachusetts Institute, 2011.

[12] H. C. R. C. Y. M. F. K. R. M. A. P. S. Boyd-Wickizer, "Corey: An Operating System for Many Cores," *In proceedings of 8th USENIX Symposium on Operating Systems Design and Implementation,* pp. 43 - 57, 2008.

[13] "Juno ARM Development Platform," ARM , [Online]. Available: https://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php.

[14] "X-C1 Development Kit Basic," Applied Micro Circuits Corporation, [Online]. Available: https://www.apm.com/products/data-center/x-gene-family/x-c1-development-kits/x-c1-development-kit-basic/.

[15] "IXH610 Host Adapter," Dolphin Interconnect Solutions, [Online]. Available: http://www.dolphinics.com/products/IXH610.html.

[16] "Dolphin SISCI Developer's Kit," Dolphin Interconnect Solutions, [Online]. Available: http://www.dolphinics.com/products/embedded-sisci-developers-kit.html.

[17] "Dolphin Supersockets," Dolphin Interconnect Solutions, [Online]. Available: http://www.dolphinics.com/products/dolphin-supersockets.html.