High Performance Inter-kernel Communication and Networking in a Replicated-kernel Operating System

B M Saif Ansary

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

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Antonio Barbalace
Roberto Palmieri

September 21, 2015
Blacksburg, Virginia

Keywords: Filesystem, Heterogeneous Architecture, Messaging Layer, High-performance Networking

High Performance Inter-kernel Communication and Networking in a Replicated-kernel Operating System

B M Saif Ansary

(ABSTRACT)

Modern computer hardware platforms are moving towards high core-count and Instruction Set Architecture (ISA) heterogeneous processors to achieve improved performance as single core performance has hit its performance limit. These trends put the current monolithic SMP operating system under scrutiny in terms of scalability and portability. Proper pairing of computing workloads with computing resources have become increasingly arduous with traditional software architecture.

One of the most promising emerging operating system architectures is the Multi-kernel. Multi-kernels not only address scalability issues, but also inherently support heterogeneity. Furthermore, provide an easy way to properly map computing workloads to correct type of processing resources in presence of heterogeneity. They do so by partitioning the resources and running independent kernel instances and co-operating amongst themselves to present a unified view of the system to the application. Popcorn is one the most prominent multi-kernels today, which is unique in the sense that it runs multiple Linux instances on different cores or group of cores, and provides a unified view of the system i.e., Single System Image(SSI).

This thesis presents four contributions.First, it introduces a ?lesystem for Popcorn, which is a vital part of SSI. Popcorn supports thread/process migration that requires migration of file descriptors which is not provided by traditional filesystems as well as popular distributed file systems, this work proposes a scalable messaging based file descriptor migration and consistency protocol for Popcorn. Second, multi-kernel OSs rely heavily on a fast low latency messaging layer to be scalable. Messaging is even more important in heterogeneous systems where different types of cores are on different islands with no shared memory. Thus, a second contribution proposes a fast-low latency messaging layer to enable communication among heterogeneous processor islands for Heterogeneous Popcorn.

With advances in networking technology, latest Ethernet technologies are out there which are able to support up to 40 Gbps bandwidth, but due to scalability issues in monolithic kernels number of connections served per second does not scale. Third is Snap Bean a virtual network device and fourth is Angel an opportunistic load balancer for Popcorn network system.

With the messaging layer Popcorn gets over 30% performance benefit over OpenCL and Intel Offloading technique [4]. And with NetPopcorn we achieve over 7 to 8 times better performance over vanilla Linux and 2 to 5 times over state-of-the-art Affinity Accpect [39].

This work is supported in part by the US Office of Naval Research under contract N00014-12-1-0880.

Acknowledgments

I am incredibly grateful to all the people who helped in this endeavor, I would like to thank the following people from the bottom of my heart:

Dr. Binoy Ravindran, for providing me direction and guidance. His inspiration made me more focused towards my work.

Dr. Antonio Barbalace, for his expert opinions and wizardry with Linux kernel.

Dr. Roberto Palmieri, for his kind advices and inspirations.

Marina Sadini for her wonderful, cheerful presence even in the most tensed situations and help with NetPopcorn.

Yuzhong Wen for his help with the messaging layer.

Dr. Sachine Hirve, Akshay Ravichandrand, Ajitchandra Saha, Sharath Bhat, Rob Lyerly, Christopher Jelesnianski for being wonderful colleagues.

All my friends in Virginia Tech cricket team for giving me such a wonderful time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As single core performance of microprocessors reach a performance limit, modern microprocessors are moving towards multicore solutions. In recent times, increasingly high core-count multiprocessors have been released. The proliferation of such high-core count machines have raised question about scalability issues in current monolithic kernel OSs [5]. Some research works have shown that monolithic design can still scale [11] but others point out that solving one scalability issue exposes more issues [12] hence an iterative process must be followed. Lock contention in Symmetric Multiprocessor (SMP) Kernels is one of the primary bottlenecks of monolithic design. Although, they work well for a small core count SMPs but they do not scale well for high-core count SMPs. SMP kernels suffer from poor isolation of data, as they have many shared data structures accessed by multiple cores. As a consequence cache-coherency protocol struggle to keep cache consistent which prevents scalability.

Nowadays, not only core count in a single chip is increasing but also Instruction Set Architecture (ISA) heterogeneity in the same system is emerging. Traditional SMP design breaks down with such heterogeneity as code sharing is not possible between different types of cores, systems have to run separate OS in separate cores.

Qualcomm MSM family of System on Chips (SoC) are such chips. Some other chips have similar ISA cores but with different capabilities. For example some cores are inclined towards performance while others are power-frugal. Arm big.LITTLE [47] is one such chip. It is used in smart phones and other mobile computing platforms. The architecture has some powerful ARM cores coexisting with low-performance power-frugal cores. The highly powerful cores are used for CPU intensive workloads and background processing is handled by power-frugal cores. Heterogeneity can also be found in workstation based solution such as Xeon-Xeon Phi platforms. Intel combines a Xeon server processor with Xeon Phi co-processor to enhance processing of parallel applications. But even on this system the usual shared memory programming model cannot run. The programs must be rewritten to take full advantage of the heterogeneity.

Many emerging systems concepts have been proposed to properly pair computation with comput-

ing resources. A new operating system design, called Multi-kernel, has emerged from researchers at Microsoft Research and ETH Zurich. In their work in [5] they show the scalability problem in monolithic SMP kernels and propose a solution based on a design orthogonal to current OS designs. The design runs multiple replicated OS instances on different cores and uses message passing to keep kernel state consistent. They provide a logically consistent userspace so that legacy software can run seamlessly. The kernel data structures are not shared, they are replicated and they are synchronized through kernel level messaging. This allows kernels to run independently and allows them to provide better scalability. Moreover, this design can be expanded to support heterogeneous systems, where kernels specific to ISAs run independently on those cores but provide a unified system view using an agreed upon messaging interface.

The Popcorn Linux project is a research effort that aims at building scalable systems software for future generations of heterogeneous and homogeneous multi/many-core architectures [30]. Popcorn Linux is a replicated multi-kernel operating system where every kernel instance is a full-fledged modified Linux kernel. The system is based on Linux kernel 3.2 (later updated to 3.12 for x86-Arm effort). Many instances of Popcorn kernels co-exist in the same hardware platform, stitched together with a messaging framework, it provides an unified userspace where the applications run as if they were running in a shared memory environment. Different kernel instances are allocated separate memory resources to ensure better locality of data and cache performance. Popcorn has been expanded to support heterogeneous systems and the work [4] shows that it is able to take advantage of heterogeneous platforms and achieve better performance. Work is ongoing to port Popcorn to x86-ARM system.

In order to provide a unified system view, Popcorn requires a filesystem that supports thread and process migration. Traditional, distributed filesystems do not support thread/process migration as they cannot migrate file descriptors(FD). This thesis aims to provide an initial filesystem that provides file descriptor migration for thread and processes. Also this thesis aims to provide a low latency, pluggable messaging framework for heterogeneous (Xeon-Xeon Phi) Popcorn. Popcorns good scalability in high count multicores compared to Linux is shown in [30]. Taking advantage of Popcorn, this thesis also proposes a high performance networking framework (NetPocorn) that scales lot better in terms of request serviced per second, request service time and connection failure compared to previous work. The Linux kernel network stack has many locks and shared data structures, hence it does not scale well in high core count servers. NetPopcorn scales better on high core count servers and also provides similar performance on low core count machines. NetPopcorn achieved up to 8 times greater performance than Linux.

Figure 1.1 shows this scalability of Linux on varying number of cores. We run apache web server [16] in both process based pre-fork model and thread based worker model to show that with the increasing core count, the scalability is not linear. We use Receive Flow Steering (RFS) which tries to localize network flows to be processed on the same core. We can see although it shows some performance improvements, but it still does not scale linearly. We also compare our work with Affinity Accept which provides better scalability than Linux by replicating kernel data structures. File system work has been vetted by running microbenchmarks as well as real applications. Messaging layer work has been verified by microbenchmarks and variety of benchmark

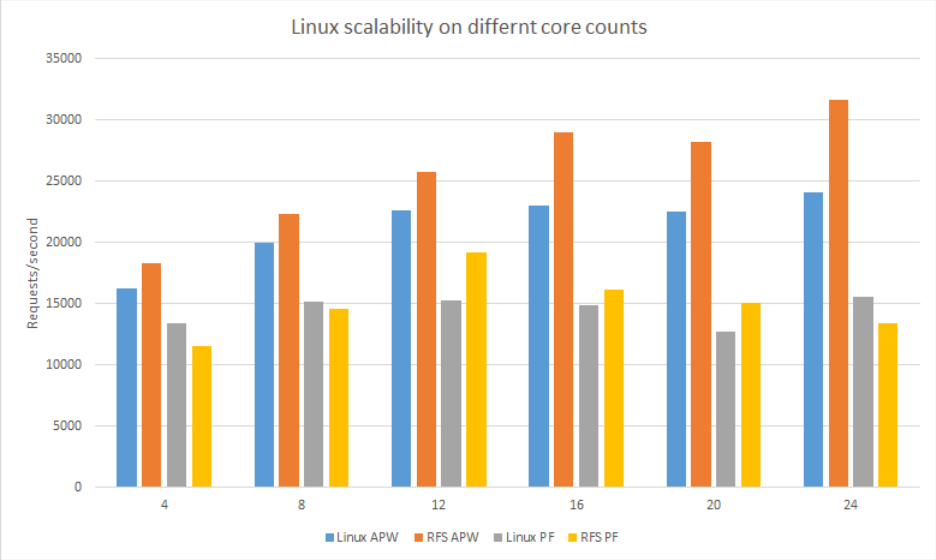

Figure 1.1: Linux networking performance on varying number of cores.

applications. Real world applications have been successfully run over this layer and has shown performance gains with accurate results.

1.1 Research Contributions

This thesis embodies four contributions that enhance Popcorn Linux OS:

1. Popcorn filesystem: Popcorn filesystem and file descriptor migration: The Popcorn filesystem strives to provide a unified filesystem among multiple kernel instances. So all processes and threads must see the same filesystem from every kernel instances. And as processes and threads migrate to different nodes, file descriptors must be migrated as well. This contribution allows this feature and allows Popcorn to migrate process/thread seamlessly to different nodes.

2. Messaging Layer: This is the heart of Popcorn heterogeneous effort. It provides inter kernel communication for Popcorn kernels. The messaging layer allows co-operations of the kernel instances and allows Popcorn to function. This contribution enabled Popcorn to expand to heterogeneous platforms. The previous work was focused on shared memory based messaging where this contribution assumes no shared memory between the processor islands. We show iterative evolution of the design and improvements in performance up to 4.5x from the initial version. We also compare benchmark performance against other techniques like OpenCL and Intel-Offloading and show with the final version of messaging layer that Popcorn achieves up to 30% better performance.

3. Snap Bean: A network device driver model that exposes a single network interface card to multiple Popcorn kernels. This is one of the two components of NetPopcorn. Snap Bean acts as virtual network device for kernel instances of Popcorn to allow access to multiple queues of network interfaces. The driver allows similar functionalities as the actual device i.e., interrupts, direct Tx/Rx, addition of filters etc. We show through microbenchmarks and applications that this adds minimal overhead (less than 1%) while comparing with the original device driver. It requires very little modification on the real device driver and functions independently. Which allow excellent scalability.

4. Angel : An opportunistic network traffic balancer for Popcorn to better utilize processor resources in case of unbalanced load. This is an indispensable part of NetPopcorn. It allows kernels which are under lot of pressure to redistribute the load to other kernels to achieve better performance. We show a detailed analysis of Angel. It show minimal overhead but achieves significant performance in case of unbalanced workloads. Using this two components NetPopcorn achieves 7 to 8 times better performance than vanilla Linux and 2 to 5 times over Affinity accept over Affinity accept in terms of requests served per second. It reduces average request serving time and number of failed requests. We show with these metrics NetPocorńs superiority to state-of-the-art.

1.2 Scope

The contributions of this thesis cover several areas of operating systems. However, the problem space for each area is vast and was narrowed for the research involved. In the following we describe in which way each problem was narrowed down.

The first limitation is that filesystem uses NFS as its base filesystem. NFS can suffer from performance issues when compared with native filesystems like EXT4 [18]. It becomes a source of in-efficiency as it has to runs on Popcorn inter-kernel network system.

Second, Messaging layer is based on library provided by Intel called Symmetric Communications Interface (SCIF). SCIF provides commutation functionality Although, it was modified but still in some cases it can be a source of in-efficiency. But our work focuses on a higher level of abstraction rather than going at a lower data transfer level. Messaging layer design is also constrained by demands of the upper-layer protocols of Popcorn(e.g Page consistency) which expect complete compliance with the previous messaging layer and its assumptions.

1.3 Thesis Organization

This thesis is organized as follows:

Chapter 2 provides information about different relevant works. Work related to different filesystems, messaging framework, network systems and other multi-kernel OS are discussed.

Chapter 3 provides a more detailed discussion about various Popcorn services and functionality. In order to understand the contributions it is important to understand how Popcorn functions.

Chapter 4 presents the design and implementation of filesystem support for Popcorn. The methods used to achieve Linux like consistency on Popcorn and other techniques used to reduce the number of messages between kernels.

Chapter 5 presents NetPopcorn, a highly scalable network software stack and system software

Chapter 6 presents Snap Bean: the virtual device developed to support multi-kernel networking system for Popcorn. This device abstracts the read hardware and provides a virtual interface to the Popcorn kernels. This is the cornerstone of the Popcorn Network system.

Chapter 7 provides in-depth description of Angel: opportunistic load balancer for High performance Popcorn network system. It uses a heuristic called SYN-FIN balance to understand system condition and balances the load.

Chapter 8 Concludes and contributes some future directions.

Chapter 2

Related works

In this chapter, previous works related with this thesis are discussed. There are three sections: which discuss work related to filesystem, inter-kernel messaging and advanced network systems.

2.1 File Systems

A lot of research has been done on filesystems for distributed systems. Some of the works are directly applicable to filesystems(e.g. Kerrighed) for multi-kernel OSs. In this section work relevant to the contribution are discussed.

2.1.1 Network File System

First system to be described is Network File System (NFS). NFS is probably the most common and widely deployed distributed filesystem. This was originally developed by Sun micro system. Now it is supported by all major operating systems Linux, UNIX, MacOS, Windows). NFS provides file access from different hosts over network and abstracts the underlying communication from the users. Hosts export files from their disk and other hosts can access those files from the remote. NFS is based on Remote Procedure Call (RPC) developed by sun. RPC is a interprocess communication technique that allows processes to cause a function invocation on a different address space like another process on another system or the same system. Now it is adapted by Linux as well. NFS has 4 versions till date and version 4.00 being the latest. NFS 4.00 incorporates locking and mounting protocols [27] as part of NFS. Moreover, support for strong security, compound operations, client caching features are newly added. Although NFS is a prominent protocol for distributed filesystems it is not sufficient for multi-kernel OS where process and threads migrate to different nodes. It does not have the file descriptor(FD) sharing mechanism between nodes. FDs are essential data structures that keep track of opened files and their offsets in a process/thread.

When a thread/process migrates to another node the opened file descriptors must also migrate with the thread/process, NFS does not provide that feature.

2.1.2 Kerrighed Distributed File System

Kerrighed Distributed File System (KDFS) is a fully symmetric filesystem. KDFS is developed on top of Kernel Distributed Data Manager (KDDM) described in [33]. In this work they describe a consistent data management service for cluster and develop other important services(e.g. filesystem) on top of this system. KDFS is such a service. In KDDM every data block is an object. Objects of same types are kept in the same set. This is defined by the user. After this set is defined it is kept consistent by the KDDM services. Files are also considered as objects.
KDDM stores up to 2 to power 32 objects of 4KB size. Each set is linked with some higher layer services. A shared memory illusion is provided by KDDM. The KDFS is built on two concepts. They are:

- (a) Use of the native filesystem to manage block devices
- (b) Use of KDDM sets to provide unique consistent cluster wide name space [33]. This also provides a global distributed cache.

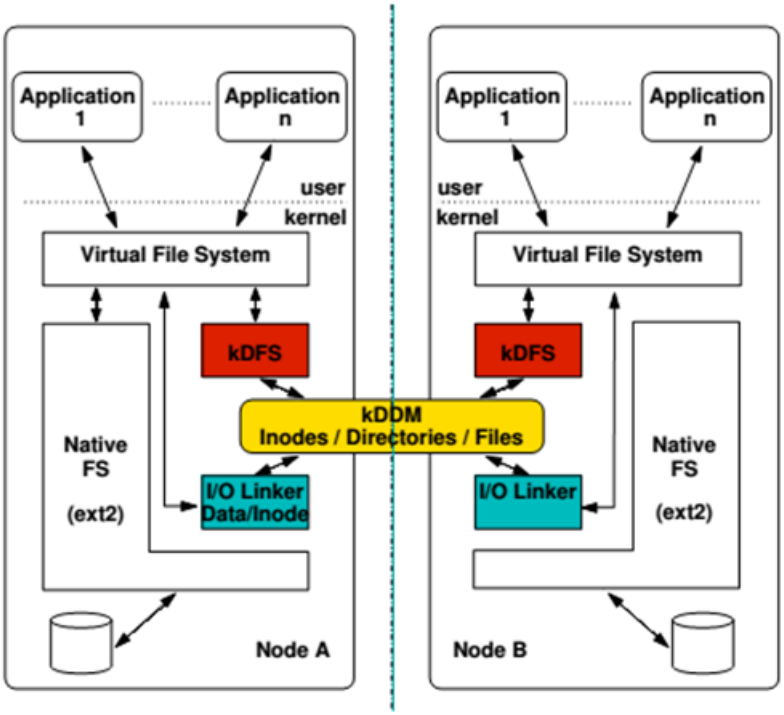

Figure 2.1: Overview of the kDFS system [33]

Figure 2.1 shows how KDFS works through KDDM. It uses the two files into represent each file(.meta,.content). Metadata is stored in .meta file actual data is stored in .contain. This works well for clusters but does not provide support for multi-kernels and process/thread migration.

2.1.3 Hare File System

Hare is a filesystem that provides a POSIX-like interface on multicores without any cache coherence. It provides functionalities for applications on non-cache-coherent cores to share files, directories and file descriptor [21]. This features are not available on traditional network filesystem. Hare proposes a new protocol based on three phase commit to perform directory operations and takes advantage of atomic low latency message passing mechanisms and shared DRAM of non-cache coherent multicores. Hare is implemented on shared DRAM and is not a persistent system. It assumes non-cache coherent system it has to explicitly write-back the cache. To reduce invalidation messages Hare only flush when a file is opened and `flush()`/`close()` is called. This is more relaxed consistency model than UNIX/Linux consistency. Figure 2.2 shows the architecture of Hare.

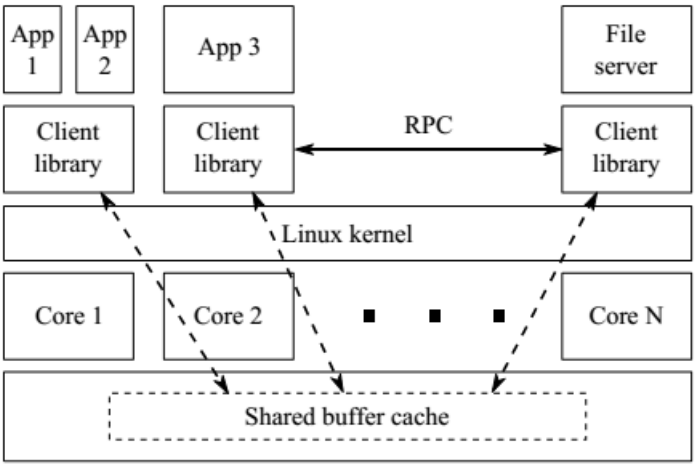

Figure 2.2: Hare design.copied from [21]

It distributes directory operations to multiple file servers using a hash function. Hare, uses three phase commit to perform directory operations. For file descriptor it keeps some state information in servers and some informations in clients. File descriptors can be in two different states local and shared. Although it supports shared file descriptors for processes, it does not support thread migration. It uses a server-client model for shared file descriptors which can become the bottleneck for high core count machines with a lot of threads/processes who share file descriptors.

2.1.4 Barrelfish File System

Barrelfish is a multi-kernel OS from Microsoft Research and ETH ZURICH. Although some work has been done on a new filesystem service for Barrelfish like [48], it uses standard distributed filesystem NFS [21]. Which lacks file descriptor migration capability. The filesystem discussed in [48] is an in memory filesystem based on B+ tree. It provides a faster file seek and creation time, but it does not provide support for file descriptor migration.

2.1.5 Non-cache Coherent File Systems

GPUfs [45] uses the shared DRAM between host and the card. But provides a limited interface compared with POSIX. It cannot support shared file descriptors among host and the card so programs such as make cannot work in this situation. K2 [32] implements a share most operating system based on Linux for multicore mobile platforms with separate domains that have no cache coherence between them [21]. K2 relies on distributed shared memory (DSM) for sharing OS data structures across coherence domains.

To compare NFS [27] with Popcorn filesystem we can see NFS lacks capacity of file descriptor migration. So it is not able to support process/thread migration. Barrelfish filesystem [48] is built on top system memory so it does not practical solution whether Popcorn OS can handle disk access hence is practical solution. Hare [21] implements a relaxed consistency model compared to UNIX standards, Popcorn filesystem is able to provide UNIX file consistency. K2 [32] relies on DSM but Popcorn filesystem does not require DSM.

2.2 Inter-kernel Messaging

In this section, various basic message passing concepts are described, followed by some works on inter-kernel message passing for multi-kernels.

2.2.1 Message Passing Concepts

Messaging is a well explored field in computer systems both in kernel-space and user-space. Message Passing Interface(MPI) has been the cornerstone of user level message passing historically. MPI works on variety of transports, like shared memory, TCP/IP networks, Infinyband interconnect etc.

In case of Microkernel OS designs, the OS runs services, which the applications access through inter-process communication (IPC) [44]. This makes a kernel level IPC a necessity. Distributed operating systems use message passing to keep the coherent state of the cluster. For a multi-kernel

OS messaging at OS level is a fundamental requirement. With out message an efficient message passing system this OS cannot function.

2.2.2 Message Passing in Commodity Multicore Systems

Commodity x86 multicore machines do not have special hardware support for messaging [44]. For this type of systems, it is necessary to use shared memory windows between different CPUs. The sender will copy the message to an address within the window, and send a notification to the receiver which then reads from that address.

There is a fundamental problem in this approach that there has to be a copy from sender to the buffer and a copy from buffer to receiver. This is referred to as copy-in copy-out problem. Many approaches have been devised to reduce these two copies in zero or one copy. Like in KNEM [20] where the receiver side pre-allocates buffers and performs a syscall with the virtual address of the buffer process. Senders copy is directly done on the receiver's buffer. This provides additional performance benefits when there are lot of messages to be sent.

Another concern about message passing is cache behavior. This affects message passing programs to a great extend. Message packet sizes should be well chosen to be aligned with cache line to avoid false sharing. There are also work on auto refactoring of the MPI based to code to optimize cache behavior [38] from the programs. With this work authors showed significant performance benefit on real word MPI applications.

Nowadays, proliferation of OS capable accelerator cards are evident. These cards can run OS themselves in their local memory and communicate with the host using message passing. They can be connected through different buses: most common being PCI Express (PCIe). These cards provide general purpose processing as well as high performance vector processing. Intel Xeon Phi [24] is such an accelerator card. It provides MPI and OpenCL support to the host while it can run applications natively as well. Efficient inter-kernel messaging for such a device is a interesting topic for research.

2.2.3 Different Notification Mechanism

There are different ways for notifying the receiver that a message has arrived. The most common are polling and interrupts.

Polling is a technique where a value in memory is checked repeatedly until some expected action occurs. Polling has the advantage that it provides lowest possible latency from the hardware. As soon as the cache coherence protocol reaches the CPU it comes out of polling loop and message can be delivered. This also provides greatest possible throughput, the CPU processes messages one after another. But this approach has the drawback that it keeps CPU busy all the time for checking one value. This prevents other work which are running in the same CPU to make progress. It also

has an impact on the power consumption of the processor.

IPI are used on SMP machines based on x86 to perform synchronization between two processors [23]. They use APIC/LAPIC infrastructure between CPUs.

The main advantage of IPI is that with this polling is no longer required. So less CPU cycles wasted. But the downside of this is that it has a slightly higher latency than polling method.

For accelerator cards connected through PCIe the data is transferred over the PCIe bus. PCIe 3rd generation provides up to 32Gbps bandwidth [37]. Most accelerator cards provides specific library to support such data transports. For example Intel Xeon Phi comes with its own library for transferring data. Data transfer is done through PCIe aperture window or DMA transfers.

Using all these techniques messaging systems for Multi-kernels have been developed. Barrelfish multi-kernel OS implements a messaging layer based on cache-coherence protocols of x86 based SMPs. COSH [6] by Microsoft Research and ETH Zurich describes coherence oblivious sharing which provides an abstraction over the lower level details of memory transfers in heterogeneous systems. In heterogeneous system different kind of memory configurations are present such as shared cache-coherent, shared non-coherent, non-shared non-coherent. For shared cache-coherent memory COSH does implicit data transfer that is it just sends the physical addresses of the buffer. On the other hand for non-shared non-cache coherent memory it does explicit transfer using DMA.

Popcorn homogeneous [3] version provides a messaging layer based on shared memory windows and IPI notifications. It provides a low latency messaging for homogeneous Popcorn.

2.2.4 Hardware Extensions

Commodity multicore hardware does not have explicit message passing hardware but many works show this support should be available in near future [36]. Intel Single-Chip Cloud research processor (SCC) has message passing primitives [22]. Tilera [49] Tile64 processor also supports message passing instructions. They show very low latency message passing from userspace.

But these are highly restricted and specialized processors and also large size messages may still suffer from high latency.

Popcorn messaging layer does not relay on dedicated hardware messaging feature like [49] or Intel SCC [22]. It is a multi-threaded layer which can be configured to run in different modes unlike the homogeneous Popcorn messaging layer [3]. Popcorn messaging layer does not define different domains like COSH [6]. It uses a hybrid technique to achieve good performance for both small size and large size messages.

2.3 Advanced Networking Systems

Due to proliferation of network based applications (e.g. cloud service,social networking,on-line store etc.) huge workload is created on servers. To cope with this, researches have come up with different approaches to make servers scale. Traditional operating systems, struggle to scale on modern high core count machines equipped with high bandwidth (e.g. 40Gbps) cards, many research efforts are done on scalability of operating system on multicore machines with respect to networking performance. This subsection describes some of the most prominent works.

2.3.1 Affinity Accept

Affinity accept [39] is done at MIT which aims to perform all processing of a single TCP connection on the same core. Traditionally, in Linux kernel incoming network packet processing is done on the CPU which handles the interrupt but the outgoing packet is handled by the CPU which has the relevant user code. As a consequence access to read/write connection state (such as TCP control blocks) often result in cache invalidations [39]. This results in degraded performance. Moreover, Linux implements BSD standard socket which has one accept queue per port. That means for http(port 80) the system has only one accept queue protected by a single lock. The lock creates a lot of contention and prevents scalability of the network stack thus the system as whole. Affinity accept proposes a new system that performs all activities in same core and creates per-core accept queues. In this system interrupt handling workload is spread throughout different cores. It takes advantage of multiple queues present in modern 10 Gbps cards (i.e., Intel 82599) and use hash based filtering (Intel flow director) to distribute traffic to different cores. Therefore, interrupts of the incoming packet is handled by different cores, they also have sever process bound to different cores which use core-local accept queues and locks. In this way, server processes run on the same CPU that handled the incoming packet and send the reply through the same CPU thus localizing the operation. The use of core-local accept queue reduces the contention, enabling far better scalability. They also propose a load balancer that balances the workload if one of the CPUs is overloaded and move the connection flows to other lightly loaded CPUs. This provides better utilization of CPUs in case of unbalanced workload. Although affinity accept reduces the contention on accept queue lock, it still cannot avoid other locks present in the TCP/IP protocol stack. It scales better than Linux but there is opportunity to improve on its performance.

2.3.2 Pika

Pika [7] is a software network stack for multi-kernel OS proposed by MIT CSAIL. Similar to this thesis, it also uses multiple queues in order to distribute connection to different kernels of a multi-kernel setup. Pika splits the network stack into several servers that communicate using message passing of the multi-kernel [7]. Pika proposes a 3-way handshake protocol to establish connection in TCP that maintains the single accept queue of UNIX standard. It also proposes a

load balancer that prefers the allocation of connections to applications with lower service times. Pika uses the same hashing based filtering as Affinity accept to distribute the connection flow. The hash is calculated based on the source port of the incoming packet. Pika system defines three different servers namely Transport Server (TS), Link Server(LS) and Connection Manager(CM). Transport server has the responsibility of TCP/IP flow management that includes encapsulation/decapsulation, Transmission Control Block state management, out-of-order packet management and retransmission [7]. The number of TSes are adjustable according to the system requirements. CM is responsible for connection establishment: when a new connection request comes it speculatively send a offer (ACCEPT-CONNECTION) message to the acceptor application with the lowest service time, and it also notifies LS to send the acceptor a CONNECTION-AVAILABLE message. On the next accept() or select() call the application will claim the connection and it will send to the TS a CLAIM-CONNECTION message which completes the connection [7]. If the application does not claim the connection in time the TS notifies CM that the connection has timed out and CM removes it from its list of acceptors and sends another ACCEPT-CONNECTION message to another acceptor.

Pika is implemented on top of Netmap [42] which provides virtual network cards on top of a single Network Interface Card (NIC). Pika uses Netmap to simulate a multi-kernel setup that has no cache coherent shared memory. It sends messages to keep the acceptor list consistent to all the CMs after a fixed interval of 50us. It uses dedicated processors to poll for messages and do not take them into account while evaluating performance. This can result is significant power wastage for the CPU that is polling for the messages. Moreover, preventing these CPUs to actually do application work.

In Pika acceptors can get connections from remote kernels. Pika associates a huge penalty to remote acceptors to prevent remote connection acceptance as much as possible because in Pika it is not possible to move hardware hash filters so software packet forwarding is required for remote connection acceptance. They do not mention how this situation is handled in their implementation.

Pika proposes a load balancer which chooses the best acceptor for a new connection. It does that by keeping track of the connection service time of each acceptor application and by adding a cost if it is a remote acceptor. Taking this factors into account the balancer decides which acceptor is the best for the new connection.

Pika achieves good throughput compared to Linux. They do not compare their results with any other work. They claim to have reached similar performance to affinity accept [7].

2.3.3 IX

IX is a data plane operating system that provides high I/O performance, while maintaining the key advantage of strong protection offered by existing kernels [9]. It is proposed by researchers from Stanford and EPFL. IX uses hardware virtualization t o separate management and scheduling functions of the kernel (control plane) and network processing (data plane) [9]. For the data plane they develop a native zero copy API and optimizes for both bandwidth and latency by dedicating

hardware threads and networking queues to data plane instances, processing bounded batches of packets to completion, and by eliminating coherence traffic and multicore synchronization [9]. IX also provides synchronization free processing by using receive side scaling (RSS) [13]. It takes the design from middle boxes such as firewalls, software routers, load balancers etc. It tries to break the 4 way trade off between high throughput, low latency, strong protection and resource efficiency [9]. In IX the control plane, responsible for system configuration and coarse-grained resource allocation to application is separated from data plane which runs the network stack and application logic. In commodity operating systems this are inter mingled and this reduces scalability and performance. IX runs data plane kernel and applications on separate level of protection and isolate control plane by leveraging DUNE hardware virtualization [8]. Control plane is a full fledge Linux kernel and data plane is protected library OS on dedicated hardware threads. IX data plane uses batching to reduce data processing cost. IX provides better throughput and latency compared with Linux and user space stacks [26].

2.3.4 Sandstorm

Sandstorm [35] is a userspace network stack that exploits knowledge of web server semantics to improve throughput of current solutions like kernel space stacks. The authors argue that the current network stacks are designed to be as generic as possible in order to support variety of end-node (e.g. user PC), middle-node support (e.g. network gateway). This generality comes at the cost of performance. They revisit the idea of having specialized stacks to enhance performance of certain services like static content web server. They use FreeBSD Netmap Framework [42] which allows mapping of nic buffer rings to userspace to implement their web server. The goal of this work is to introduce a zero copy webserver, avoidance of buffering to mitigate cache footprints, use of batching, processing in user space rather than kernel space etc. The sandstorm server is a combination of three components which strives to achieve the mentioned goals. The components are:

- Netmap I/O library
- libeth a lightweight ethernet library
- libtcpip which is a modified TCP/IP library for sandstrom

Sandstorm uses a webserver that shows better performance compared with nginx but it is not POSIX compatible hence not usable with legacy software.

2.3.5 Network Stack Parallelization

Network stack parallelization is a well studied field in network systems to improve the network performance [52]. Single processor performance is not improving nowadays and focus of develop-

ment has been on SMP based multicores. In order to utilize this parallelism the server applications tend to run multiple thread/process to handle multiple connections simultaneously. The kernels also allow multiple threads to perform networking tasks in kernel space. But the cost of this parallelism is very high as this operations have to be synchronized. The work described in [29] shows that uniprocessor Linux kernels perform 20% better in terms of end to end throughput over 10 Gigabit Ethernet than SMP kernel. This shows there is a significant need to improve the multiprocessor kernels for networking applications [52].

The research on parallelizing the network stack has mainly two forms. message-oriented and connection-oriented. In message-oriented a message is processed by a separate thread, by which messages from the same connection are processed by different threads simultaneously hence improving performance by using parallelism. On the other hand in connection-oriented approach messages from a single connection is handled by a single thread. As a study done in [52] shows that message-oriented and connection-oriented approach both perform better than a non-parallel stack but none of them scale perfectly. They both show reduced performance as more and more connections are added. So a new approach may be required to achieve better scaling.

2.4 Multi-kernel Operating Systems

In this subsection a new operating system for multicore multiprocessors, the Multi-kernel operating system, is discussed. The primary motivation behind a multi-kernel OS is to resolve the scalability issues occurring in monolithic kernels.

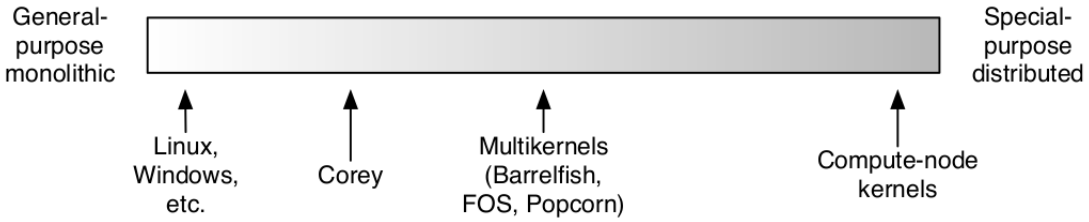

Figure 2.3: Operating system gradient [44]

The generality of OSs can be seen by considering a gradient shown in Figure 2.3, it goes from general purpose to specialized and distributed.

It can be seen that the left is the monolithic kernels like Windows, Linux which run as a single instance and are expected to run variety of different workloads.

Next, to right are operating systems that are not distributed but provide special support for scalability, reliability on multicore machines. Such an operating system is Corey, which is described in later in this section.

In the middle Multikernel OS such as Factored OS (FOS), Popcorn which are distributed OSs in single machine and provide Single System Image(SSI) on top of it. This enables of-the-shelf shared memory applications to run seamlessly. They provide better scalability.

On the furthest right are fully-distributed compute node kernels these are designed to provide the minimal services necessary to run a particular MPI-based high-performance computing application as fast as possible [44].

2.4.1 Barrelfish

Barrelfish project [5] developed by collaboration between Microsoft Research and ETH Zurich. Barrelfish was the first to introduced the concept of multi-kernel OS. The barrelfish multi-kernel operating system dictates explicit inter-core communication, hardware-neutral OS structure, and that OS state is not shared but replicated. The authors argue that motivation behind this design is to match hardware platforms with different capabilities with appropriate kernels. This is aimed at new high-core count multicores and heterogeneous platforms.

Barrelfish, a fully distributed OS under the skin, provides single system image so that programmers can program in shared memory programming model which is used in SMP Linux. It also provides an OpenMP library which show respectable performance.

Barrelfish uses a innovative approach for message passing in commodity multicore machines. The messages take up a cache line and carries a sequence number in the last few bytes of the cache line [44]. The sender writes the message to the shared memory and receiver polls on those bytes. When a new value is observed in these bytes the polling loop is broken. Barrelfish also implements a hybrid messaging notification with both IPI and polling.

Barrelfish takes advantage of Intel Single-Chip Cloud research processor (SCC) which provides explicit messaging and notification primitives [40].

Barrelfish is also providing support for hot pluggable USB [50] and Java Virtual Machine [34].

An effort is on the way to run Windows 7 as library OS [41] on Barrelfish to support commodity applications.

2.4.2 Factored Operating System

FOS is introduced by MIT in 2009 [51]. It is a factored operating sysem for commodity multi-cores [44]. In FOS different OS services are run on specific cores and have userspace processes send messages to those cores to avail their services. It does not run services on each core that needs them.

The overhead of message passing is similar to overhead of making a syscall [36]. But FOS achieves good performance gains because of better cache performance as a consequence of pinning some

OS service to specific cores. It is similar idea to FlexSC [46]. In FlexSC instead of sycall, shared memory messages are used. Although it incurs messaging cost but it achieves better performance by batching system calls. FOS messaging channels are dynamically allocated. Each process define a value to the kernel and channel is allocated by hashing this value.

FOS provides both userspace and kernel space message infrastructure. It also provides support for distributed systems so messages can go to remote cores.

2.4.3 Corey

Corey is an Exokernel developed at MIT. It expands the idea that application should control sharing [10]. The Linux kernel shared state becomes the bottleneck to scalability on high-core-count machines. Linux improves the scalability by reducing shared data between cores but further improvements can be achieved if the kernel explicitly knew what data to share and what data is totally local. Corey uses this to achieve better performance. Corey is not a multi-kernel, however it is similar to Barrelfish in the sense that it also replicates state between cores.

Corey uses three OS mechanisms to accomplish the goals:

- Address ranges - Shared memory applications can are allowed to create separate address space which can be mapped totally locally to a core or shared between cores [44]. The programmer has the responsibility to indicate if a data structure is shared or local.
- Kernel Cores - Certain kernel features are pinned to a single core
- Shares - Some operations that look up identifiers (e.g. file descriptor) and returns a pointer to the data structures these can be global or local. By default these tables are local but can be expanded to be global. Application programmers are responsible for these tables.

Corey shows improved performance on TCP micro benchmarks and benchmarks based on MapReduce [15] and web server applications.

2.4.4 Twin Linux

Twin Linux project boots two independent Linux kernels on a dual core machine [28]. They modify the GRUB bootloader to achieve this. The kernels are able to communicate through shared memory regions. The authors argue that this allows Twin Linux to address heterogeneity in kernels. For example one kernel can be optimized for network workloads and another for real-time workloads.

The devices are statically partitioned such that network card is handled by one kernel and hard disk drive is handled by the other. This shows good performance compared to running filesystem and network system benchmarks together. The authors claim this is due to less stress cache coherence protocol. Although the supporting data to this claim is not present in the work.

2.5 Summary

This section discusses previous work that are relevant with the contribution of this thesis work. It discusses different approaches for scalable filesystems, messaging systems, network system and different multi-kernels. From the discussion it is clear that multi-kernel OS is a very interesting research prospect and can be a solution to many real life problems such as network scalability.

Chapter 3

Popcorn Linux Architecture

3.1 Introduction

Popcorn Linux [3] runs multiple Linux kernels on multiple cores. It boots multiple instances of the Linux kernel on different cores and provides a Single System Image on top of them, thus legacy shared memory applications run without modification. It provides different services to provide SSI illusion to the application. The initial version of Popcorn Linux was limited to homogeneous multicores but currently it has expanded to heterogeneous systems. This section describes the design of Popcorn Linux and the variety of services it provides and also discuss both homogeneous and heterogeneous Popcorn [4].

The goals of the Popcorn project is to provide improved scalability and performance by reducing data sharing between cores and by taking advantage of heterogeneity of modern computing hardware. It also aims to provide Shared Memory programming model to the programmers so that they can develop programs is same manner as they would do for SMP operating systems. And also provide support for large number of SMP applications to take advantage of Multi-kernel OS with minimal or no changes to their source code. Popcorn is a replicated-kernel OS. Although, slightly different from multi-kernels we use the two terms interchangeably in case of Popcorn.

3.2 Basic Architecture of Popcorn

The Basic architecture of Popcorn is illustrated in the following Figure 3.1.

It is exhibited by Figure 3.1 that applications in Popcorn run on top of the SSI and the underlying multi-kernel is abstracted. Popcorn is a replicated multi-kernel as the OS state is replicated in all the kernels [3]. The architecture is generic in both homogeneous as well as heterogeneous setup. It runs a kernel optimized for a particular Instruction Set Architecture (ISA) and uses message

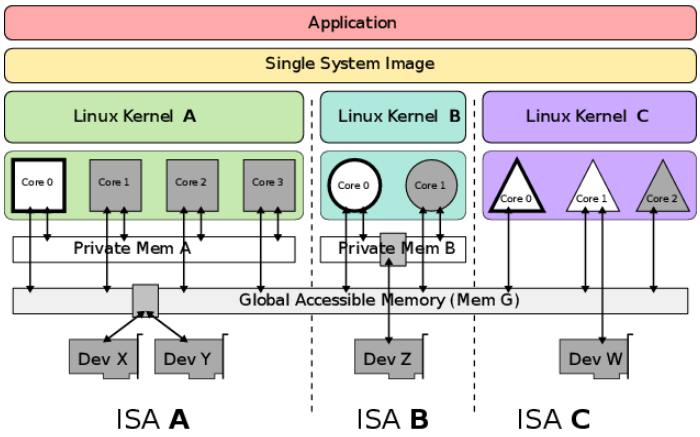

Figure 3.1: Basic architecture of Popcorn Linux

passing in order to keep kernel state consistent. Each kernel has its own private memory, but SSI makes sure that all the memory is seen as Distributed Shared Memory(DSM) by the application. Popcorn kernels do not require shared or cache coherent memory. Although if it is running in such an environment it can take advantage of such a system. In the next sections of the this chapter, design and implementation of various important Popcorn services are discussed.

3.3 Popcorn Services

Popcorn provides a variety of services to support a SSI. The most important are: task migration, mapping based page consistency and address space management protocol, replication based page consistency protocol, POSIX signals and Futex locking mechanism, namespaces etc. [3].

Popcorn is also shipped with a compiler framework to specifically compile code so it can run on heterogeneous platforms.

3.3.1 Process Migration

Processes can migrate between different kernels. Popcorn kernel uses the sched_set_affinity() system call to trap migration requests. When a process wants to migrate to another kernel, its state must be copied on the destination kernel. In order to do that the kernel copies all task state information (e.g. processor registers) and sends a migration message to the destination kernel. The destination kernel launches a new process and implants the state of the migrating process into the new process. The virtual memory area (VMA) of the process must be replicated into the new process and all the pages are also needed on the destination kernel. The page-consistency and VMA management services are responsible for these functions. File descriptors are migrated on demand. All communication happens through Popcorn messaging Layer. When a process migrates to an-

other kernel the process in the original kernel is suspended and is referred as shadow process. It is possible that the process migrate back to the original kernel, at that time this shadow process becomes active again. When a process exits all the active instances and shadow instances are terminated using Popcorn signaling service. Process migration is a key feature in feature in Popcorn. This allows process to migrate to a kernel where it achieves the best performance so that Load Balancing is possible.

3.3.2 Thread Migration

Thread migration is one of the unique features of Popcorn. It allows some threads from a thread group to migrate to different kernel. That is it allows thread groups to be spread across multiple kernels. This presents a lot of challenges to all the services of Popcorn because with process migration all the data structure of the process is on the same kernel so after migration these do not have to be synchronized. But in case of a thread group spread amongst diffent kernels the vma, pages, files everything has to be kept consistent across all kernels of the system. It also stresses the synchronization service to the limit as PTHREAD library (most popular thread library) uses Futex locks to implement many functionalities (e.g., thread_join()). Thread migration provides finer grain control to the operating system to take advantage of multi-kernel approach. It is required to exploit the advantages of heterogeneous systems. Applications consisting of heterogeneous threads can run threads in the architecture that best suits that type of thread. For example Parallel Bzip (pbzip) application has three types of threads some are I/O threads and some are compression, decompression threads. I/O threads run best on the kernel which has control of the storage device where compression/decompression threads run better on vector processors. With thread migration it is possible to harness this heterogeneity. Thread migration is not supported by other Multi-kernel OSs.

3.3.3 Address Space Migration

Popcorn implements address space migration to support thread/process migration. The address space information is stored in sparse data structures. In Popcorn the kernel data structures are not shared but replicated per-kernel [30]. Popcorn implements partial replication of address space based on a protocol to keep the data structure consistent. The address space can expand, shrink and modified based on memory requirements of the program [30]. For process migration this consistency is not required as process executes on a single kernel. For distributed thread groups the partial replication protocol is very important as threads are spread around different kernels. It is not required that the address space information is exactly same in all the kernels as different threads in a thread group may not require the information about the entire address space. Popcorn takes advantage of this fact to reduce number of messages exchanged to keep a consistent state.

3.3.4 Page Consistency Protocol

In Popcorn thread/process migrates without their used pages. Page migration is performed on-demand to reduce the number of messages and reduce thread/process migration time. Two different mechanisms are implemented in Popcorn, the first is mapping based and second is replication based.

Mapping Based Page Consistency

This mechanism allows Popcorn to take advantage of cache coherent shared memory system. Popcorn partitions the system RAM when it boots. The memory is private to each kernel. So when a thread or process migrates to another kernel and access any data it is generates a page-fault. Popcorn hooks are inserted inside Linux page-fault handling mechanism. They check if it is a remote process/task, if it is then it sends a message to the original kernel from where it has migrated for the page. In mapping based mechanism the original kernel sends the physical memory address of the page to the requesting kernel. When the requesting kernel gets the reply it will insert this page into the VMA of the process. The remote kernel re-maps these physical memory address into virtual address using remap_pfn_range() and adds this the proper VMA in the process address space. Upon completion the protocol returns page_handled signal to the process/thread. Now if this is a remote thread/process and the original kernel cannot find the page that indicates its a genuine page-fault and it is handled locally.

This mechanism does not work when shared-cache coherent memory is not present. For that Popcorn implements replication based page-consistency protocol which is described next.

Replication Based Page Consistency

This mechanism is used in absence of shared-cache coherent memory. This is extremely important for heterogeneous systems where memory is non-shared and non cache-coherent. Popcorn implements a protocol to support such system. This is described in [43]. It is based on the MESI protocol. It works like the mapped version except the fact instead of sending the mapping it sends the page data itself in a message. The transfer of page messages are expensive so many optimizations are devised to reduce such messages. As Popcorn provides Unix memory consistency guarantee, it is difficult to reduce the number of messages. An optimized version of this work is implemented in [4] for heterogeneous Popcorn. Further optimization of this work is done in [2] which tries to device a mechanism through which pages are mapped with PCIe aperture and messages are send to invalidate cache. This is a complementary work to [4]. It covers the case where there is shared memory but no cache-coherence. Replication can also show performance in shared-cache coherent setup. This stems from the fact that in a Non Uniform Memory Access (NUMA) system the memory accessed from different nodes can have a performance impact because of higher access times hence replicating the read-only pages can improve performance.

Popcorn has a flexible page consistency protocol which can take advantage of all kind of memory configurations. This allows Popcorn to work with high-core count multicore or heterogeneous systems like Xeon-Xeon Phi servers or combination of both.

3.3.5 Popcorn Namespace Service

Namespaces are introduced in SMP Linux to create sub environments, as a lightweight virtualization at OS Level [3]. In Popcorn namespaces are used to provide a single environment across all the kernels. Applications launched inside Popcorn namespace can use its services. Namespaces are introduced in Linux kernel 3.8 so it was backported and modified for Popcorn kernel (based on Linux 3.2). After kernels bootup, they join via messaging layer and PID, IPC, CPU namespaces are all made available through the namespace.

3.3.6 POSIX Signals

Popcorn provides SSI and a unified System view. Signals can be sent to any process in the Namespace from any kernel. This signal must find the correct process and deliver the signal. Any signal send to a process has to go to the primary kernel (the first kernel booted in the system) and then follows the chains of migrations to reach the current active instance of the process. Signals are common way of communication between processes.

3.3.7 Inter-Thread Synchronization

The glibc library relies heavily on futex. Futex was ported to Popcorn to support applications that use glibc. We extend futex to Popcorn by adopting a client/server model. The server kernel, which is the kernel that the locks physical page belongs to, maintains the futex queue. This enforces serialization [30]. The secondary kernels are clients and send request to the primary kernel for wait and wake operations. The primary kernel is responsible for waking up the first task in the queue via message. When the thread receives the message it becomes the owner of the lock and wakes up.

3.4 Homogeneous and Heterogeneous Popcorn

Popcorn supports both homogeneous and heterogeneous multi processor systems. It has been deployed on high core count multiprocessor and overlapping heterogeneous setup with Xeon-Xeon Phi. It is able to run OpenMP, PTHREAD and other shared memory application on both setups. Although the underlying implementations are different but the higher level designs are same on

both versions. To support heterogeneity Popcorn compiler project [4] provides a tool which allows executable to run across heterogeneous processor islands.

Chapter 4

File System

Popcorn provides a uniform view of the filesystem to all the kernels. It also has to provide support for process and thread migration. This has two main challenges. First, is to make sure every kernel has the same (root, bin, home, proc, user etc.) directories and Second is when processes and threads migrate to different kernels, their opened file descriptors must migrate properly and kept consistent. The first challenge is handled using Network File System (NFS). And for the second one a messaging based consistency protocol is developed and implemented on top of Popcorn messaging subsystem. The high level overview of the system is demonstrated in Figure 4.1

4.1 Booting with Identical Root Directory

In order to boot with the same root as the primary kernel, Popcorn internal network device in combination with NFS is used. The necessity of NFS stems from the fact that Popcorn secondary kernels do not have direct access to the storage of the system hence the Hard Disk Drive (HDD) is invisible to secondary kernel. Popcorn secondary kernels are booted using a modified version of (boot loader) which boots the kernel on one core with a minimal root filesystem. Popcorn inter-kernel network device is then loaded and it can be used to communicate with other kernels. In order to have the same filesystem on all the kernels, the primary kernels root filesystem is exposed using NFS. The secondary kernel will mount and switch root to the primary kernels root after booting. In order to achieve that we create a directory (/exports) in primary kernels filesystem. The required directories inside e.g. bin,sbin,usr,home are created and the corresponding directories from system / directory are mounted. For example /bin is mounted on /exports/bin with access privileges to allow secondary kernels to read/write in the directory. When boot process of a secondary kernel is complete, the Popcorn network tunnel is established then the init script mounts the exposed root system from the primary into directory named new_root. Then it unmounts the proc filesystem and remounts it on /new_root/proc/. Then it performs switch root on new_root directory which provides the secondary with identical root filesystem of the primary. The proc filesystem is also

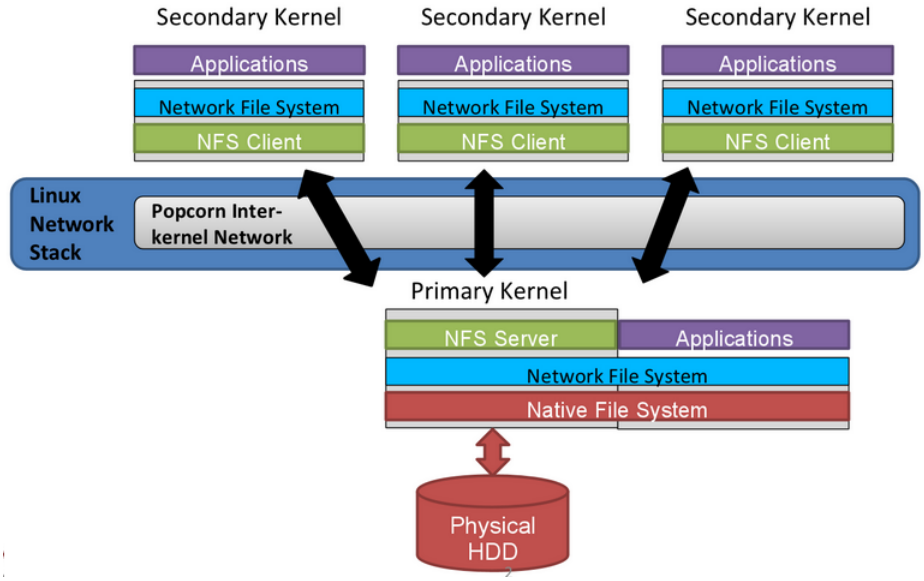

Figure 4.1: Basic architecture of Popcorn Linux File system

part of Popcorn Single System Image (SSI) and provides identical view to all kernels. NFS version 4.0 is used in this implementation as it does not require additional user space daemons like port-mapper on NFS 3.0 which makes implementation simpler. NFS provides different levels of access rights hence mounting is done with full access rights with no_root_squash option, which allows user accounts from secondary to have full access. With this implementation identical filesystem to all kernels, is achieved.

4.2 File Descriptor Migration

File descriptor migration completes Popcorn live thread/process migration.

4.2.1 File Descriptors in Linux

In Linux every object is considered as a file. For example devices are modeled as a file and is uniformly accessed using filesystem interface (open, read, write, close etc.). When a process/thread open a file for accessing the object the kernel creates a structure called file descriptor and provides a number to the process/thread which it uses to access the object. Each process / thread group has a file descriptor table. Threads of same process or thread group share the same file descriptor table. The file descriptor is a index to this table. Each entry in this table contains information about the file opened. When a thread/process migrates to a remote kernel the file descriptor must also be migrated to ensure successful execution. This requires two functionalities. First, a mechanism which

allows a file descriptors to migrate and Second, a protocol that keeps file descriptors consistent in the whole system.

4.2.2 File Descriptor Migration Mechanism

The migration mechanism is developed on top of Popcorn messaging subsystem and process/thread migration mechanism. File descriptors (FD) are migrated on demand. When a thread/process migrates to a remote kernel its current working directory has to be the same as the one on the original kernel. The current directory information is migrated during thread/process migration. When an access to a file previously opened in the original kernel is performed, the descriptor for that file is not found in the remote kernel as the remote thread/process task structure does not have this FD. In order to access the file the remote kernel queries the original kernel for this FD. The original kernel sends the state information of the FD to the remote kernel (file path, access rights, open flags, current offset etc). The file path is identical in all the kernels (this is implemented by the NFS) and is opened again in remote kernel and state information is implanted in the descriptor. The implementation details of the mechanism is described in a later section.

4.2.3 File Descriptor consistency Protocol

File descriptor information must be kept consistent in all the kernels. There is difference in requirement in case of process and threads. Each process has its own file descriptor tables in their task structure, hence it is sufficient to migrate the information once every migration. This information is not shared amongst any other processes so it is not required to keep this consistent in all the kernels. But in case of threads it is not sufficient to keep information local as FD tables are shared by all the threads of a thread group. In Popcorn, a thread group can be spread amongst multiple kernels. To maintain correctness the information in FDs must be kept consistent in every kernel where the particular thread groups member thread is present. The most important and most changeable information is the file offset value. This value indicates the position of read/write pointer in the files, that is how bytes have been read/written. This value changes with every access (read, write, lseek) and must be reflected in all the kernels that have member threads of the thread group to ensure correctness. To manage this a owner based protocol is developed. Each file descriptor can be in one of the states (JUST_OPEND,OWNER_MIGRATING, OWNER_MIGRATION_DONE, UPDATE_NEDDED). FDs also have two different modes Exclusive, Non-Exclusive. The kernel where this FD was first created is called the host kernel of the FD. In Popcorn for every thread group an additional thread called main thread is created. This thread performs all the multi-kernel related work such as memory management, synchronization etc. An additional functionality is added to the main thread that handles file related tasks. The main thread of host kernel is responsible for handling all tasks related to the FD. When a FD is opened it is in JUST_OPEND state and the thread that opened it assigned to be the owner. Upon first access to the file using the FD mode changes to Exclusive. Now if any other thread from the same thread group uses the same FD, the

mode is changed to shared. If a thread migrates to a remote kernel, update messages must be sent to the host kernel to keep the FD data consistent. There are few cases to consider here. If the FD is in Exclusive mode it does not send any update message to host kernel. If it is in shared mode it must send updates to the host kernel. If a thread migrated with an Exclusive FD but later some other thread wanted to use it, then host kernel sends message to the thread and Exclusive mode is broken and updates about the access would then be sent to the host kernel. The following scenario illustrate how file descriptor migration works.

In this scenario we consider the case where the process/thread is migrated.

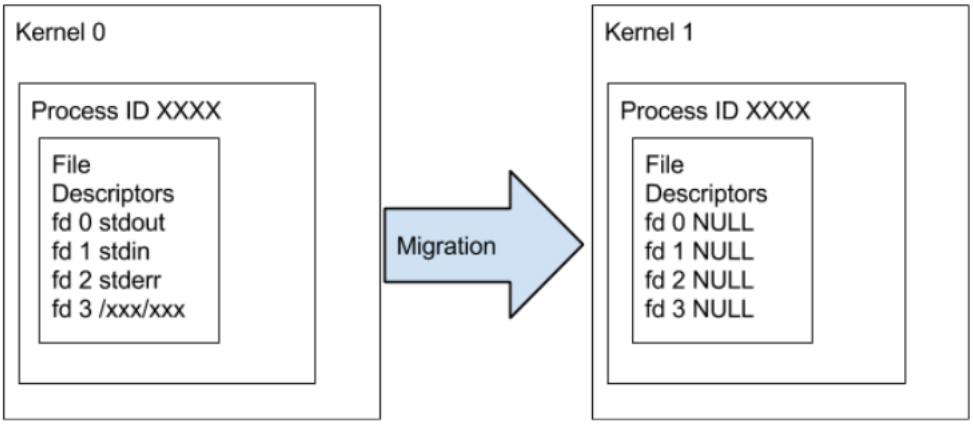

Figure 4.2: Thread migrating with out file descriptor

Now if the migrated process / thread wants to use a file descriptor for example fd 3. It will not find it (demonstrated by Figure 4.2) in its file descriptor table. Then it will send a query to its owner node to see if it has information about it. If it has the owner node sends filepath, mode, flags and current file offset. It is shown in Figure 4.3.

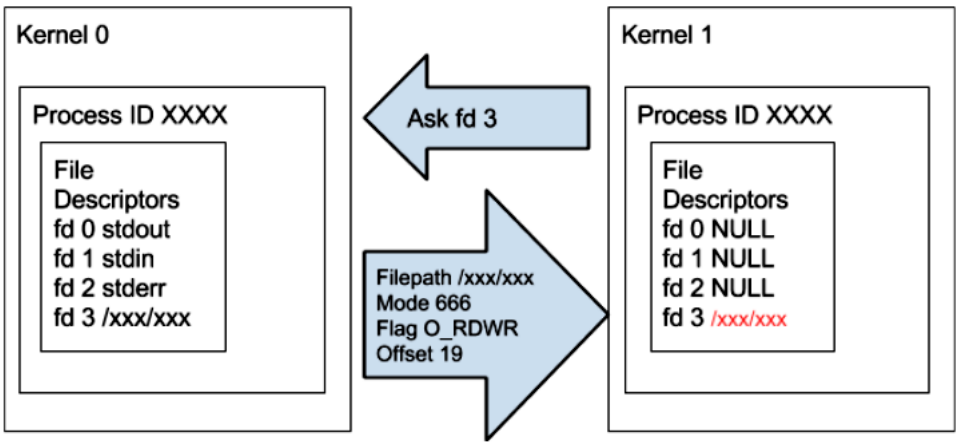

Figure 4.3: Intreaction between origin kernel and remote kernel

The protocol also provides four functionalities namely Remote Open, Get Remote Offset Info, Set Remote Offset Info and Remote Close. The description of the functionalities are given below.

Remote Open

When a file is opened in a process / thread the Linux kernel checks the FD tables to find out the next free FD index. In Linux FD table is maintained using a bitmask where a bit is set when a descriptor is in use. A look up for unused bit is performed and set up to 1 to indicate that this is in use and initialization of that particular entry is done. Now the file open mode and flags are tested for proper access permissions and file is opened using the Linux Virtual File System(VFS). The offset value is set to 0 and file path and other information is stored in the FD table entry. The FD table must be consistent in all the kernels. Now in case a thread migrates to a remote kernel and attempts to open a file a remote open procedure must be performed. The remote open procedure is initiated by remote thread by calling open() syscall. In open() Popcorn filesystem hooks are inserted to trap the call. In the function the kernel checks if this thread is a remote thread or not. If it is a remote thread a remote_open_request message is send to the host kernel where the thread group was first created. The main thread on that kernel is woken up and it looks for free a FD and sets it up to indicate that it is in use. The host kernel sends a remote_open_reply with the FD index. It also stores which kernel and which thread opened this file. After receiving the reply the file is opened in the remote kernel. Now if for any reason the opening has failed a remote_open_failed message is sent to the host kernel where the FD is freed for later use. Figure 4.4 shows the interaction between the remote kernel and the host kernel.

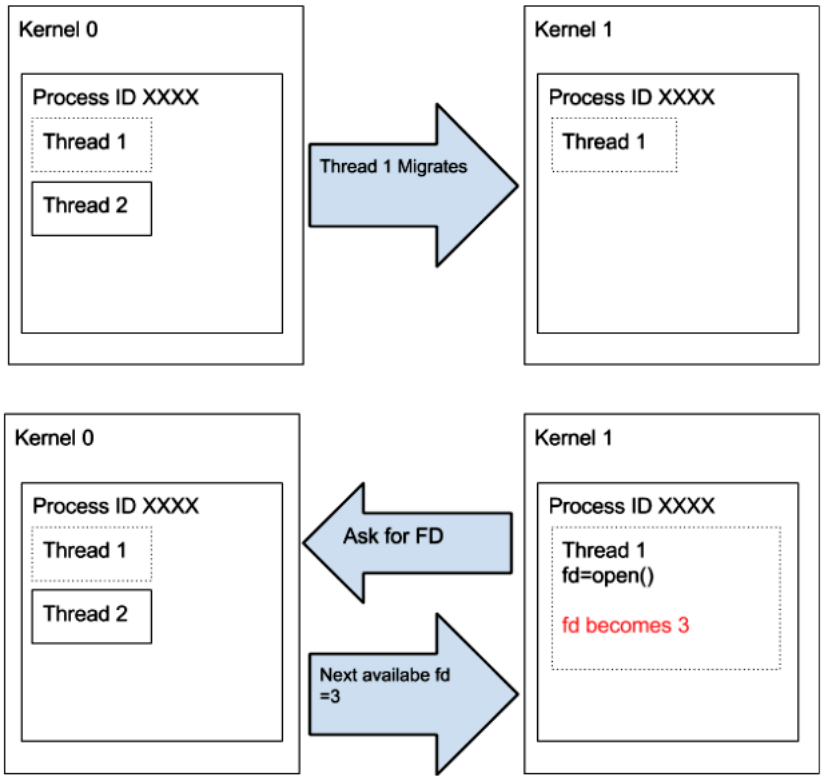

Figure 4.4: Remote file open operation

Get Remote Offset Info

When a thread group is distributed before accessing a file with a file descriptor it must get the latest file offset info. It checks the mode of the FD. If it is in Exclusive mode and it is the owner of the FD then it just gets the file descriptor locally. Now if it is in Exclusive mode and the thread is not the owner of that FD it must ask the owner node of the FD for update. The owner node then changes the mode to Non-Exclusive and asks the owner thread to send updates of every access. Upon receiving the update the offset value is sent to the asking thread. If the FD is in Non-Exclusive mode the thread must ask the owner node for offset update before it accesses the file. This is not required for a process migration as process are not distributed amongst multiple kernels.

Set Remote Offset Info

When a thread group is distributed and a change is made by a thread (as a read, write, lseek) in the offset value then this value must be sent to the owner node. In this case also if the FD is in Exclusive mode and the thread is owner thread then no update message is sent. But if it is Non-Exclusive mode or this is not the owner thread then the update message must be sent to owner

node. Non-Exclusive mode is very expensive as many messages will be exchanged to keep the value of the offset updated. But most applications do not share FDs with in threads to avoid various synchronization issues. If threads are using shared FDs it is upto the application to synchronize the access rather than the kernel. So in general case Exclusive mode is used and number of messages are low.

Remote Close

Any thread can close a FD and it will not be accessible by any other thread.So when a thread would like to close an FD such operation must be coordinated with the owner node. Any further query from any other thread will get a negative response indicating that FD is no longer valid. The owner node also sends a notification to the thread group's home kernel to indicate that this FD is now closed and free to re-use. The FD table is updated accordingly.

Now two scenarios are considered one shows a Non-Exclusive mode case and another shows a Exclusive mode case.

Scenarios

Now two scenarios are considered one shows a Non-Exclusive mode in Listing 4.1 case and another shows a Exclusive mode case Listing 4.2 using pseudo code.

Listing 4.1: Non-Exclusive mode access

```
fd=0;
Thread1()
{
    while(10 reads){
        read(fd,buffer,size);
    }
}
Thread2()
{
    while(10 reads){
      read(fd,buffer,size);
    }
}
mainThread()
{
    fd = open(path/file_name,O_RDONLY,666)
    spawn Thread1
    spawn Thread2
}
```

In this case the `mainThread()` is the owner of the FD owner node is kernel 0 and Thread1 and Thread2 are using that FD. If we suppose that Thread1 is migrated to kernel 1 it must get the offset information from owner node kernel 0. When kernel 0 receives a get_offset_info message it searches for the owner thread group and retrieves the information and sends a message back. Now if Thread2 is still on the owner node (kernel 0) it does not send any messages but just updates the offset locally. If Thread2 had migrated to another kernel, say kernel 2 then messages from both kernel 1 and kernel 2 are received and processed. This results in a lot of messages so Non-Exclusive mode is expensive.

Now we consider the scenario where Exclusive mode is used.

Listing 4.2: Exclusive mode access

```
Thread1()
{
    int fd = open(path/file_name,O_RDONLY,666)
    while(10 reads){
        read(fd,buffer,size);
    }
}
Thread2()
{
    int fd = open(path/file_name,O_RDONLY,666)
    while(10 reads){
      read(fd,buffer,size);
    }
}
mainThread()
{
    spawn Thread1
    spawn Thread2
}
```

In this scenario each thread is using its own FDs to access the same file. This allows Exclusive mode access. In Exclusive mode access no messages are exchanged as they are private to the thread. Exclusive mode has very low overhead and is a very common way of programming multi-threaded programs (e.g. Pbzip, iozone). So for the common case filesystem has very low overhead.

Cascade Migration Support

In Popcorn it is possible for threads to migrate multiple times which is referred as cascaded migration. Cascaded migration presents some challenges to the kernel. When an owner thread migrates from the owner node the owner node keeps record of which kernel it has migrated to. Now if the owner thread migrates again a message must be sent to the owner node to let it know which kernel

it has migrated to. The current file offset value is send back to the owner node and the state of the FD is set to OWNER_MIGRATING. In this scenario if a query is made for that FD the owner node replies with the value last received from the owner thread. When the owner thread migration is completed and an access to FD is performed it sends a query to owner node and gets the current offset.Upon receiving of the message the owner node sets state OWNER_MIGRATION_COMPLETE. Owner Node tells the owner thread whether the Exclusive mode is still valid or not. Threads can migrate back to previous kernel as well which is referred to as back migration. For this case the FDs owned by the thread are set to UPDATE_NEEDED state to indicate that this FD is present here in this kernel but updates are required.

4.3 Extension of Linux Virtual File System Layer

In Linux every file is access through its Virtual File System(VFS) Layer. In this layer the generic filesystem (FS) functionalities are implemented and call back functions are exposed so that proper function are registered to access the files. Popcorn FS inserts hooks in the VFS layer functions to provide support for thread and process migration.

4.3.1 Remote Operations Implementation

The four remote functionalities are implemented with use of Popcorn inter-kernel message passing layer. A file called remote_file.c is added inside kernel directory of the Linux source tree. The file consists of around 800 lines of code. For handling remote open requests an additional table is added to the task structure. It is a pointer to remote_file_table structures which is allocated if any thread of a thread group is migrated. One thread group has only one such data structure. When a file is opened from remote, a query for the next available FD number is made to the home kernel of the thread group. Instead of adding an entry to the actual FD table an entry is made to the remote FD table. This is done so that the main thread does not have to access the FD table which is protected by lock and can be under heavy contention. The remote FD table is only accessed by the main thread hence a lock is not required. The VFS layer function is do_sys_open() in open.c where files are opened. To support remote open 200 lines of code are added in this file. the most important function is remote_thread_open(). Another file read_write.c provides functionalities for do_sys_read() do_sys_write(). In these functions the offset of the file is read and modified. In the case of Popcorn distributed thread group it may be required to read the offset from remote kernels and also propagate the updates to remote kernels. To facilitate this functionality 80 lines of code are added in read_write.c. modification on do_sys_(read, write, vread, vwrite, lseek) is done in accordance with Popcorn thread migration requirements by adding hooks. These hooks are only activated when the thread group is distributed so non-distributed local threads/processes are not affected. The Popcorn FS implementation encompasses lot of modification and addition of new functionalities in VFS and other layers of the Linux filesystem. A total of around 1200 lines of code are added to the kernel source for this implementation.

4.4 Results

The primary focus of this work is to enable the thread/ process migration rather than performance. The correctness of the protocol is vetted by running microbenchmarks as well as applications such as pbzip and iozone. Microbenchmarks are used to validate successful operation of remote functionalities such as remote open/close, read, write, lseek. Threads with cascaded migration are also tested and found to be correctly functioning. To test process migration iozone filesystem benchmark is used. It migrates from one kernel to another and performs filesystem intensive workloads. All the test completed successfully. To test distributed thread groups Parallel Bzip(pbzip) application is used. In pbzip there are heterogeneous threads some are reading the data some are compressing and some are writing back to the disk. The threads were spread across different kernels and compression of 1 GB binary file was performed. After compression the compressed file was decompressed and compared with the original one and no difference was found. This work is also ported on Linux kernel 3.12 which is the kernel for Popcorn Arm-x86 heterogeneous platform.

Chapter 5

Messaging Layer

The messaging layer is the most critical part of Popcorn. All inter-kernel communication is performed through the messaging layer. It has a critical effect on Popcorn's performance as every other service is built on top of it. In this thesis an overlapping heterogeneous system is described. A low latency messaging layer is developed to facilitate Popcorn services for this system. Figure 5.1 shows the Popcorn's software building block including the messaging layer.

5.1 Intel Xeon-Xeon Phi Heterogeneous System

Intel Xeon Phi is an accelerator card for x86 platforms. It is a a Many core processor used to accelerate highly parallel programs. The Xeon Phi has 57 cores 4 hardware threads each, which provide 228 CPU to process. It has 6GB of memory and 1.1 GHz clock speed. It has vector arithmetic instructions which are not supported in x86 instruction sets. It is in-order processing so it does not have some synchronization instructions. Due to these differences, a program compiled for x86 cannot run on Xeon Phi and vice-versa.

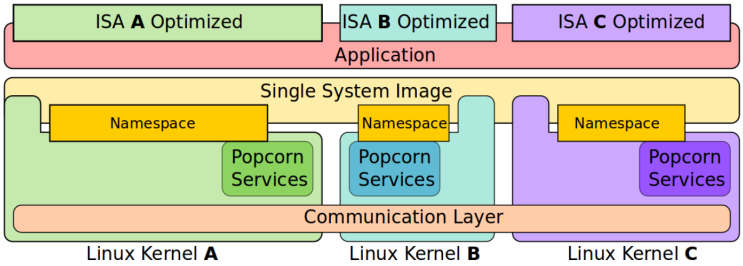

Figure 5.1: Messaging layer (or communication layer) stitching the kernels together to work as a single system

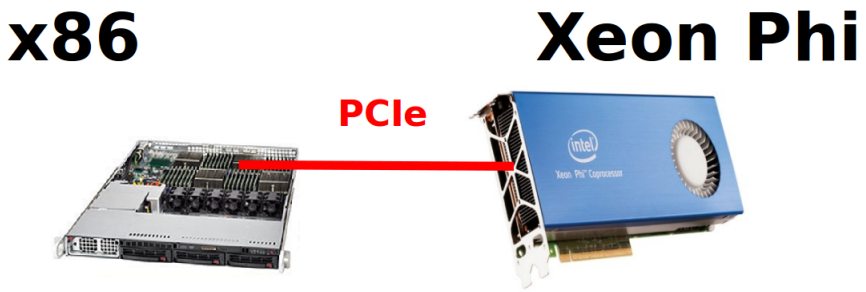

Figure 5.2: Hardware setup in the system

The Xeon Phi is connected with the host through PCIe bus. Upon boot it loads a full fledged kernel with its own memory. The Single System Image is provided by the Popcorn kernel using the messaging framework.

5.1.1 The System Setup

The system is based on an Intel Xeon processor, therefore x86 ISA, 64bit, and a Xeon Phi accelerator connected through PCIe x16 generation 3 bus 4GB PCI memory mapping is required for correct operation. Figure 5.2 demonstrates the setup.

The host and the device are not cache coherent although their memory can be accessed using PCIe window mapping. Intel also provides a library called the Intel Symmetric Communications Interface, (SCIF) [25], which is part of Intel's Many core Platform Software Stack (MPSS). SCIF was ported to Kernel 3.2 and modified to serve Popcorn's messaging layer requirements.

5.1.2 Symmetric Communications Interface

Intel (SCIF) provides a mechanism for inter-node communication. A node in SCIF context is another Xeon Phi or a Intel Xeon processor. The underlying low level details of data transfer is abstracted by using SCIF. More detailed description about SCIF and how it is used in Popcorn messaging layer is provided in the following sections.

5.1.3 SCIF Concepts

This section describes basic SCIF concepts, functionalities and their usages.

SCIF Nodes

A SCIF node is a physical endpoint in the SCIF network. The host and Many Integrated Core Architecture (MIC) devices are SCIF nodes. From the SCIF point of view, all host processors (CPUs) under a single OS are considered a single SCIF (host) node [25].

SCIF Ports

A SCIF port is a logical destination on a SCIF node [25]. It is similar to a TCP or UDP port in networking context. The SCIF port and node id are unique in SCIF network which is analogous to a complete TCP/IP address (IP address and Port number). Just like TCP/IP there are some well known ports that are used by other MPSS services (e.g., Power Management) which opens and communicates between the host and device.

SCIF Connection APIs

SCIF programming is similar to socket programming. It provides interfaces that are analogous to Linux TCP/IP socket programing. In SCIF a port is accessed through entity called Endpoint. An endpoint can be listening, i.e., waiting for a connection request from another endpoint, or connected, i.e., able to communicate with a remote connected endpoint [25]. It provides the following functions in the kernel space to create the connection.

- scif_epd_t scif_open(void)
- int scif_bind(scif_epd_t epd, uint16_t pn)
- int scif_listen(scif_epd_t epd, int backlog)
- int scif_accept (scif_epd_t epd, struct scif_portID* peer, scif_epd_t* newepd, int flags)
- int scif_close (scif_epd_t epd)

The process for establishing a connection is similar to socket programming: A process calls scif_open () to create a new endpoint; scif_open () returns an endpoint descriptor that is used to refer to the endpoint in subsequent SCIF function calls. The endpoint is then bound to a port on the local node using scif_bind (). An endpoint which has been opened and bound to a port is made a listening endpoint by calling scif_listen (). To create a connection, a process opens an endpoint and binds it to a local port, and then requests a connection by calling scif_connect (), specifying the port identifier of some listening endpoint, usually on a remote node. A process on the remote node may accept a pending or subsequent connection request by calling scif_accept (). scif_accept () can conditionally return immediately if there is no connection request pending, or block until a connection request is received [25].

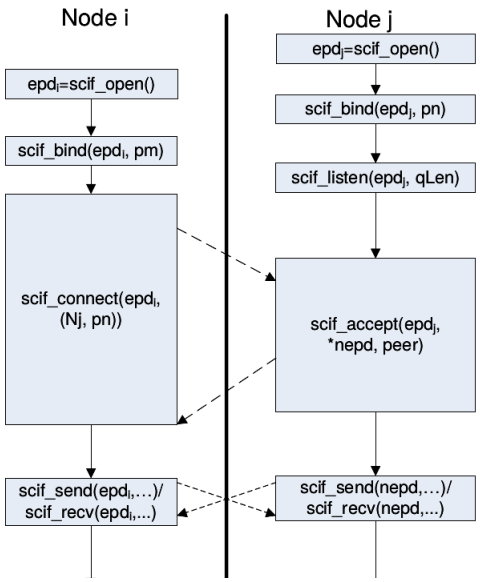

Figure 5.3: SCIF connection establishment [25]

When the connection request is accepted, a new connected endpoint is created, bound to the same port as the listening endpoint. The requesting endpoint and the new endpoint are now connected endpoints that form the connection. The listening endpoint is unchanged by this process. Multiple connections may be established to a port bound to a listening endpoint. Figure 5.3 illustrates the connection process. In this example, a process on node *i* calls scif_open(), which returns endpoint descriptor edp_i. It then calls scif_bind() to bind the new endpoint to local port pm, and then calls scif_connect() requesting a connection to port pn on node j. Meanwhile, a process on node j calls scif_open(), getting back endpoint descriptor epd j , binds the new endpoint associated with epd j to local port pn, and calls scif_listen() to mark the endpoint as a listening endpoint. Finally, it calls scif_accept() to accept a connection request. In servicing the connection request, scif_accept() creates a new endpoint, with endpoint descriptor nepd , which is the endpoint to which edp_i is connected. The endpoints associated with edp_i and nepd are now connected endpoints and may proceed to communicate with each other. The listening endpoint associated with epd j remains a listening endpoint and may accept an arbitrary number of connection requests [25].

SCIF Messaging functions

After a connection is established messages can be transmitted and received using SCIF functions between the two ends.

- int scif_send(scif_epd_t epd,void* msg,int len,int flags)
- int scif_recv(scif_epd_t epd,void* msg,int len,int flags)

Messages can be upto 4GB long. However, Intel recommends to use this layer for short messages. And use Remote Memory Access (RMA) for larger data as send queues are relatively short and requires multiple interrupts to send a large message in multiple turns. Both scif_send() and scif_receive() has blocking and non-blocking calls. The version that is best suited is used for

SCIF Memory Registration functions

This function allows a remote endpoint to access the local endpoint's memory directly. In order to use Remote Memory Access based on Direct Memory Access engine the memory must be registered with the local endpoint.

- off_t scif_register(scif_epd_t epd, void* addr, size_t len, off_t offset, int prot_flags, int map_flags)
- int scif_unregister(scif_epd_t epd, off_t offset, size_t len)

Offset returned by the register function is used to access a particular memory location. This functions are extensively used in Popcorn messaging layer implementation.

SCIF Memory Mapping functions

This functions allow physical pages to be mapped to some virtual address.

- int scif_pin_pages(void* addr, size_t len, int prot_flags, int map_flags, scif_pinned_pages_t* pages)
- int scif_unpin_pages(scif_pinned_pages_t pinned_pages)
- off_t scif_register_pinned_pages(scif_epd_t epd, scif_pinned_pages_t pinned_pages, off_t offset, int map_flags)

scif_pin_pages() pins the set of physical pages which back a range of virtual address space, and returns a handle which may subsequently be used in calls to scif_register_pinned_pages () to create windows which represent the set of pinned pages. The handle is freed by scif_unpin_pages() [25].

SCIF Remote Memory Access functions

The SCIF RMA functions provide functionalities to access the remote end point memory using DMA engine.

- int scif_readfrom(scif_epd_t epd, off_t loffset, size_t len, off_t roffset, int rma_flags)

- nt scif_writeto(scif_epd_t epd, off_t loffset, size_t len, off_t roffset, int rma_flags)

The scif_readfrom () and scif_writeto () functions perform DMA or CPU based read and write operations, respectively, between physical memory of the local and remote nodes of the specified endpoint and its peer. The physical memory is represented by specified ranges in the local and remote registered address spaces of a local endpoint and its peer remote endpoint. Specifying these registered address ranges establishes a correspondence between local and remote physical pages for the duration of the RMA operation. The `rma_flags` parameter controls whether the transfer is DMA or CPU based [25]. In Popcorn messaging implementation this two functions are used to implement DMA transfer.

5.2 Popcorn Messaging Layer

In general messaging layer works as an event driven system where messages act as an event. When a kernel receives a message, it triggers a registered messaging handler.. From the message header a particular call back function or handler is invoked. Messages are self-explanatory and they contain sufficient information such that the requested operation can be performed. Handler functions from each service (e.g., page consistency, single system image, thread migration etc.) must be registered at service activation. The layer is designed to be highly pluggable and easy to use. It exposes functions in accordance with the previous homogeneous messaging layer. It only requires few functions to be rewritten and it can work with different underlying communication mechanism. For example, the version discussed in this thesis is developed on top of SCIF but can be easily modified to be used with TCP/IP networks. The lower level implementation details are abstracted from the upper level services.

- int pcn_connection_staus()
- int pcn_kmsg_register_callback(enum pcn_kmsg_type type, pcn_kmsg_cbftn callback)
- int pcn_kmsg_unregister_callback(enum pcn_kmsg_type type)
- int pcn_kmsg_send(unsigned int dest_CPU, struct pcn_kmsg_message *msg)
- int pcn_kmsg_send_long(unsigned int dest_CPU, struct pcn_kmsg_long_message *lmsg, unsigned int payload_size)
- void pcn_kmsg_free_msg(void *msg)

pcn_connection_staus() function is used to query whether the connection establishments are completed in the system. This functionality is required for some services that start during kernel boot and need to know if the messaging layer is activated or not.

pcn_kmsg_register_callback() is used to register event callbacks. An event identifier and a callback function is taken as parameter and stored in a globally accessible data structure.

pcn_kmsg_unregister_callback() is used when an event is no longer valid and its callback function is removed from the event data structure.

pcn_kmsg_send() is the send function which is used to send small messages. In the homogeneous messaging layer it was a requirement to have separate send messages for less than 64 Byte and greater than 64 Byte messages because of implementation constraints. But in this implementation this function acts as a wrapper to pcn_kmsg_send_long() pcn_kmsg_send_long() is primary function that sends messages of various sizes. Messages up to 8 KB can be send at one time. The maximum message size is 6 KB which is a page replication message. Messaging layer also provides a Bulk message option where up to 400 pages can be sent at once. The feature is used in Page consistency protocol optimization with prefetching.

void pcn_kmsg_free_msg() is used to free memory allocated to messages after sending.

This set of functionalities are sufficient to support all the Popcorn services. implementation of SSI, page replication, page synchronization protocols. The remaining sections describe different versions of the messaging layer, their design and implementation.

5.3 Single Channel Popcorn Messaging layer

This initial version of Popcorn messaging layer was implemented as a single channel system on top on SCIF. SCIF library is designed as a per process communication library, hence it is not suitable for inter-kernel messaging layer. In SCIF, before every message sent a connection has to be created to the intended remote node, after connection creation the message can be sent. The connection is closed when the process ends. Popcorn kernels communicate with each other and it is not in a process context hence it makes connection establishment on every message send which is very expensive. In order to cope with this issue several changes in SCIF code are made and a mechanism is devised to keep connections alive and reuse the same connection each time when sending inter-kernel messages. The following sections provide detailed description of the design and implementation of Popcorn single Channel messaging layer.

5.3.1 Design and Implementation of Single Channel Messaging Layer

The design goals are to avoid connection creation on every message sent and to provide low latency. In order to avoid connection creation every time, a mechanism is devised. Userspace process have process space boundaries for this reason a connection opened in one process is not visible to other processes. This limitation is not present in the kernel space. Any connection opened in kernel space can be accessed from all kernel threads.

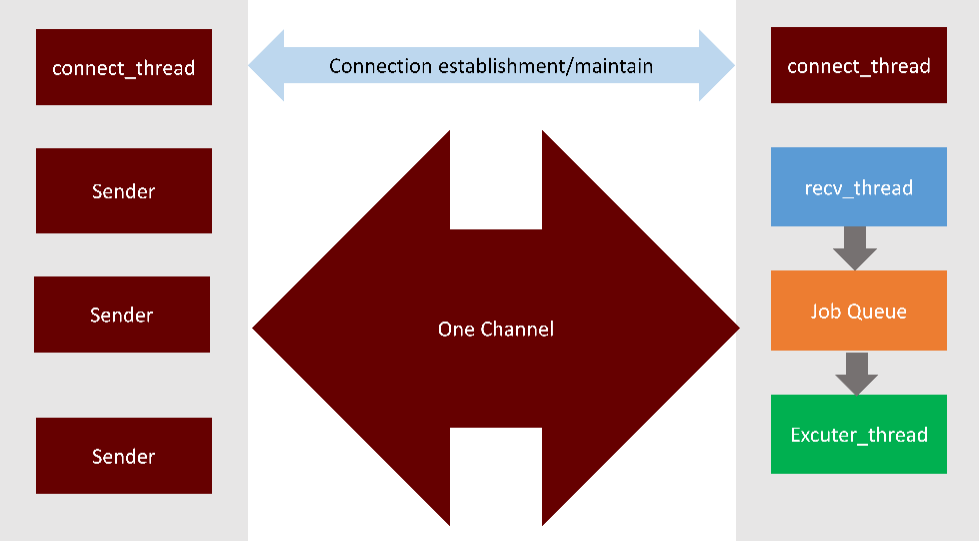

Figure 5.4: Design of Single Channel Layer

In this system three threads namely (connection_thread, receive_thread, executor_thread) are used to create maintain and handle incoming messages.

connection_thread creates connections between kernels and maintains them. On the host side it creates a SCIF socket and binds to a port and waits for the device to boot. When the device boots up the connecting thread connects to the port opened by the Host and connection creation is done. The connection_thread then keeps the connection alive. If this thread is terminated the established connection is destroyed. after connection is established the messages can be sent and received. recv_thread has the responsibility to receive messages. It waits for messages to arrive on a SCIF socket. After receiving the message it puts the message in a job queue. The receiving thread is very critical for Popcorn performance, so to keep low latency, the execution of the message handler is separated from the receiving thread. In the homogeneous version of Popcorn messaging layer handlers are executed in softirq context which creates issues if the handler function is long or has to sleep for some lock. With this split design this problem can be avoided. The design is shown in Figure 5.4.

To implement this system, SCIF library functions are used extensively. SCIF provides two ways to transfer data across host and device. First, is using scif_send()/scif_rcv() which is PIO based access. And DMA functions scif_writeto() and scif_readfrom() which can be used with DMA engine. During testing it was found that for certain message size certain method is faster. And this is asymmetric between Xeon to Xeon Phi and Xeon Phi to Xeon communication. DMA based transfer has an additional cost of DMA setup hence it is only beneficial to use this on a certain message size. In order to take advantage of both PIO mode and DMA mode transfer a hybrid system implemented. PIO mode is used for small message sizes and for larger sizes are transferred through DMA. The following figure shows performance of scif_send() scif_writeto()

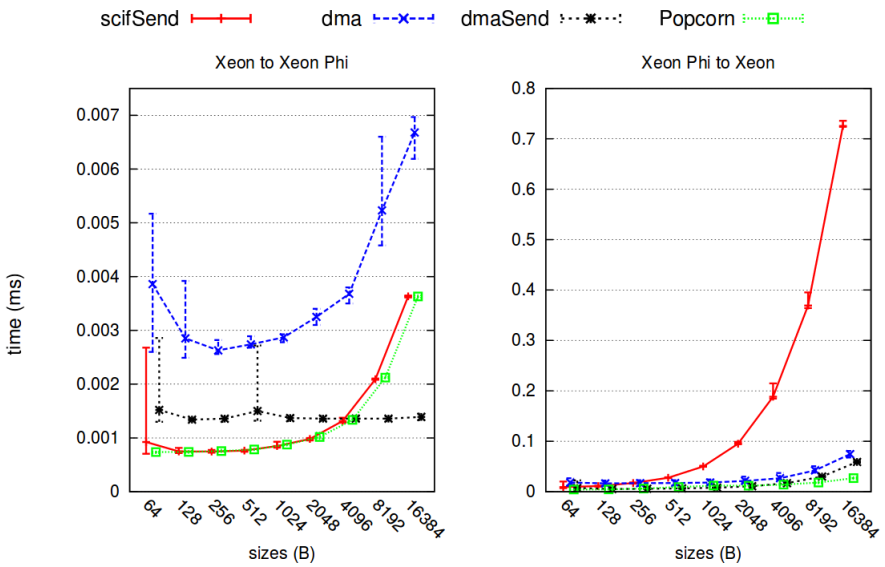

Figure 5.5: Latency of the message passing

and pcn_kmsg_send_long() [4] .

Figure 5.5 demonstrates that Popcorn messaging layer takes advantage of both techniques and provides fastest transfer speeds. One notable thing is that on Xeon side messaging never switches from PIO mode as largest message size is 8 KB on the on the hand Xeon Phi side it switches around 512 KB message size. This asymmetry is likely a reflection of difference in clock frequency and core topology between the two processors.

In order to perform DMA transfer using SCIF we need to have DMA buffers registered on the remote endpoint and the registered offset must be known as described in section 5.1. During connection setup all these information is retrieved by the connection_thread. This information is used to perform DMA transfer. All DMA buffers are preallocated and registered in boot time. 256 buffers of 8 KB size, are allocated at boot time. After each DMA transfer, a small message is sent to the receiver to notify it about the message and it also contains the DMA buffer number. The data is then copied to a new buffer and delivered to the upper layer.

Performance of this single channel layer suffers from scalability issues when number of sending threads increase. As Xeon has 8 cores and Xeon Phi has 228 cores, hence many threads can simultaneously send messages. In that case single channel messaging layer becomes the bottleneck and PCIe bandwidth is not fully utilized. In the next section multiple channel Messaging layer is described.

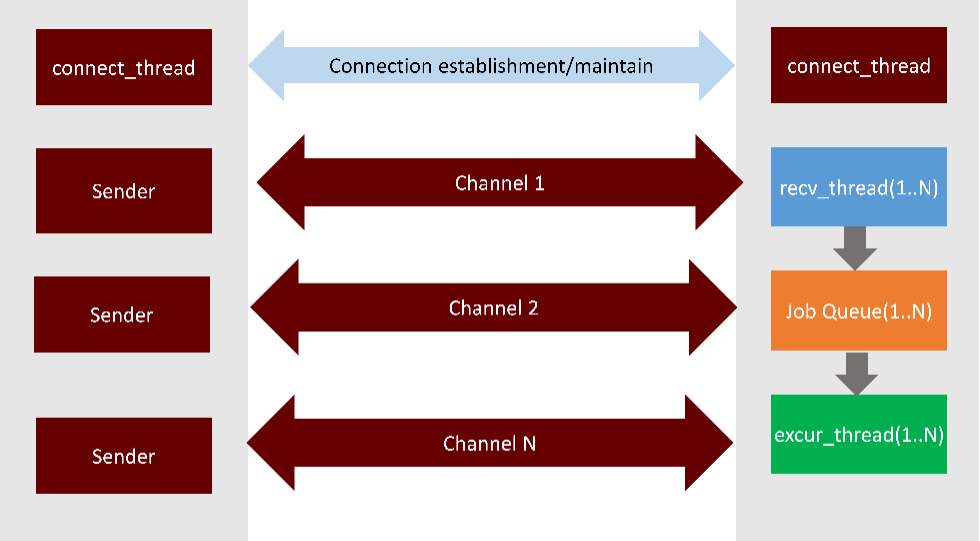

Figure 5.6: Design of Multi-channel Messaging Layer

5.4 Multi-Channel Messaging Layer

To avoid bottleneck situation and to better utilize the PCIe bandwidth a multiple channel messaging layer in developed on top of the version described in the previous section. Xeon Phi has 8 hardware DMA channels that can work in parallel. The multiple channel messaging layer can be thought of parallel versions of the single channel version.

As shown in the Figure 5.6 the system has N number of recv_threads,executor_threads and rcv_queues. This reduces the contention by N times. To determine which channel should be used to send a message a ticket mechanism is used. Ticket number is divided by number of channels and the remainder is used as the channel number. This provides a good spread of the messages among different channels.

The optimal value of N is experimentally determined to be 8. The following Figure 5.7 demonstrates application performance comparison of varying number of Xeon cores and channels.

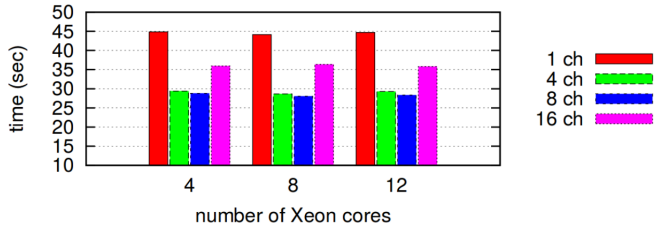

Figure 5.7: Performance of Messaging Layer on varying cores and N

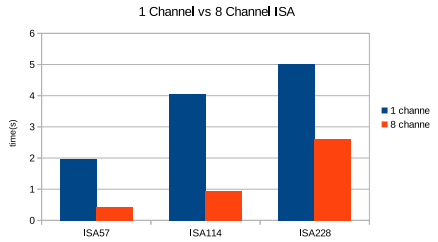

Figure 5.8: Performance of ISA with varying number of threads

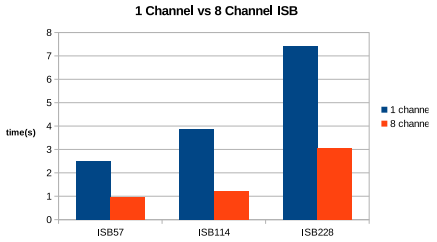

Figure 5.9: Performance of ISB with varying number threads

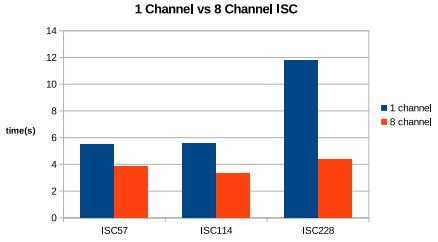

Figure 5.10: Performance of ISC with varying number threads

It can be seen that 8 channels provide the best performance in terms of application execution time over any number of cores. For all further experiments number of channel are fixed on 8. To demonstrate the performance gain through multiple channels a benchmark Integer Sort (IS) from NAS Parallel Benchmark Suite (NPB) is used as it is memory intensive and exchanges a lot of messages in between host and device.

From Figures 5.8, 5.9, 5.10 it can be seen multiple channel provides much better performance over single channel version. Execution time goes down as low as 4.5 times. As we see in IS class A (IS.A) with 57 threads on multi-channel average execution time is 0.44 second and on one channel is 1.92 seconds. It can be seen multi-channel always provides significant performance gains.

The multi-channel design is a non-buffered design. For this reason the sending threads must wait one each channel's lock to send messages which is not necessary on most of the cases. This unnecessary wait causes performance loss and threads keep waiting till the message is send. To avoid this situation a Buffered messaging layer is designed and implemented. This is described in the next section.

5.5 Buffered Messaging Layer

The initial messaging layer was implemented without any buffering. Sending threads wait on the channel lock to send messages. The lock is held until the message is send through the channel. This incurs huge contention on the messaging channels. All senders spin while one is sending and CPU cycles are wasted. Non-buffered design cannot scale when there is a large number of threads. The buffered version provides a solution to this problem.

As demonstrated by Figure 5.11 the design incorporates sending threads. Each channel has an associated send thread which is responsible for sending the message through the channel. They also have a send buffer where the threads store the messages to be sent, therefore do not wait on the channel lock.

Another addition in the messaging system is introduction of lazy delete using a clean up thread. In

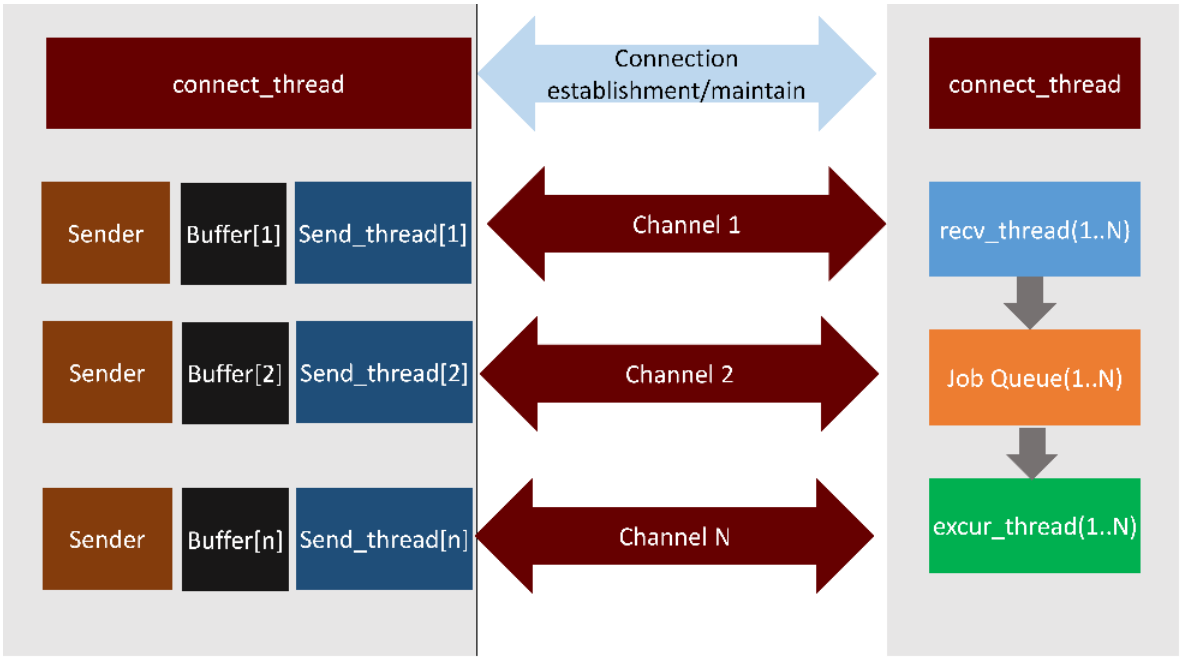

Figure 5.11: Design of Buffered Messaging Layer

the initial implementation sending threads create and send messages and delete them themselves after sending. In the buffered messaging layer the delete is logical, it only sets a bit in the packet header which indicates that it can be deleted. The send thread, after sending each message checks this bit to see if it can be deleted, if the bit is not set then it queues the message to a delete list. The Clean Up thread searches the delete queue and check each listed message to see whether its delete bit is set or not. If the delete bit is set then the memory is physically freed, otherwise it stays in the list.

Figures 5.12, 5.13, and 5.14 show performance benefit of buffered messaging layer. It can be seen that buffered messaging layer provides over 50% performance benefit over non-buffered version

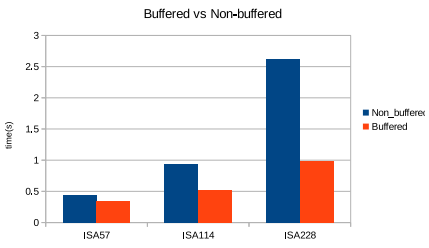

Figure 5.12: Performance of IS.A with varying number of threads

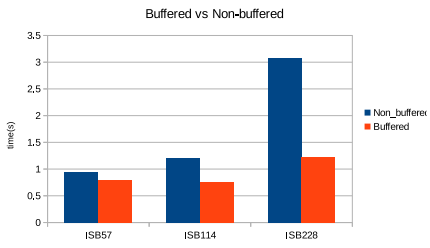

Figure 5.13: Performance of IS.B with varying number threads

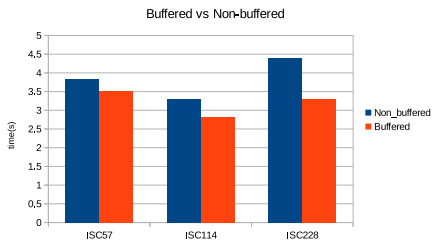

Figure 5.14: Performance of IS.C with varying number threads

on IS benchmark.

5.6 Results

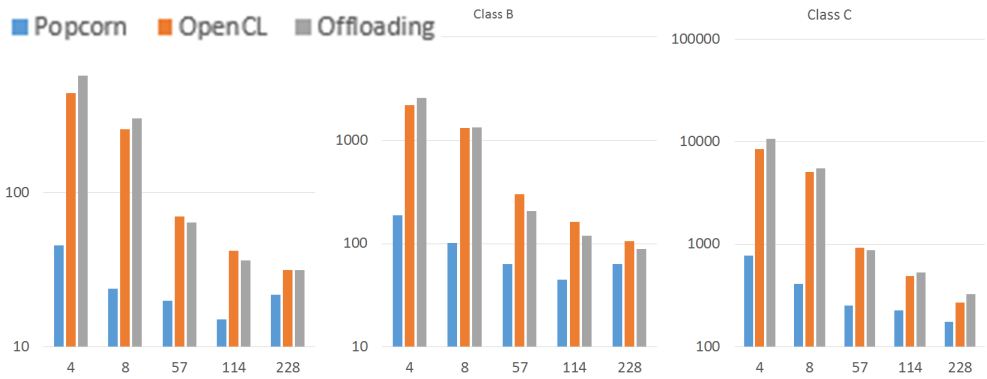

Figure 5.15: Popcorn compared with other mechanisms

In this section we demonstrate how Popcorn performs against other techniques of utilizing heterogeneity, namely OpenCL, Intel Offloading model. The Figure 5.15 shows Scalar Penta-diagonal solver (SP) from NPB benchmarks. It can be seen that Popcorn performs better than the competitors.. Although this gain not only due to fast messaging layer but also efficiency of other components. The complete evaluation of messaging layer is included in [4]

Chapter 6

NetPopcorn: A Highly Scalable Software Network Stack

This chapter describes an innovative software network stack for Popcorn, called NetPopcorn. It is a software network stack that takes advantage of Popcorn's replicated operating system data structures to achieve much higher scalability than vanilla Linux.

In this chapter we overview the motivation that drives the development of NetPopcorn. The next two chapters describes two key components of NetPopcorn, Snap Bean and Angel.

Ethernet is by far the most common network technology adopted today and it is becoming increasingly faster. Commodity Ethernet adapter cards for the home/office market have 1Gb/s speed, while server-class machines for the data center are usually equipped with 10Gb/s Ethernet cards. Commodity Ethernet adapter cards at the line rate of 40Gb/s and 56Gb/s are increasingly available from different manufacturers, while several manufacturers are already shipping 100Gb/s adapters. This market trend of faster node-to-node interconnectivity poses the problem of how incoming data streams can be efficiently handled by the CPUs of a single computer. Today, a single CPU core is barely able to handle a sustained 1Gb/s of Ethernet traffic. For higher bandwidths, multicore processors came to the rescue – a parallel software network stack enables 10Gb/s and 40Gb/s Ethernet traffic to be handled effectively. However, the software network stack implemented in traditional operating systems, such as Linux, does not scale when the network traffic goes up and/or the number of available cores in the platform increases. Therefore, there is an immediate need for improved system software that enables efficient processing of tomorrow's higher speed Ethernet data streams.

Why traditional operating systems design cannot fulfill the future networking demands? One of the major problems is scalability. Even if the number of cores per chip is increasing (meaning there is more computational power for more demanding data streams), system software tends to be plagued by scalability issues due to the way shared state access is managed among CPU cores. Core access to shared state usually relies on cache coherency, which has been shown to scale poorly [5, 51].

In the following we present our initial evaluation of the Linux operating system's network stack scalability. Considering the wide adoption of Linux in the research community as a vehicle to discover new techniques, we believe that the same problems plague other operating systems with similar design (e.g. Solaris, AIX, etc). A known problem of the Linux kernel is its poor scalability of TCP/IP *accept* [39, 7], i.e., the number of accepted TCP/IP connections per unit of time, which is limited by the system software. This problem is important because it affects an entire class of applications that provide Internet services on a single IP address and port, including web servers and key/value stores. These are applications at the base of the vast majority of todays Internet services. We deployed an experimental setup to study the scalability bottlenecks in the Linux kernel 3.x based on 10Gb/s Ethernet technology. The experiment consists of a burst (10min) of short-lived TCP/IP connection requests on a single webserver, each of which requested a small 64 byte web page. The setup consists of one server machine with two Intel 12-core processors, 2-way hyper threaded (48 total CPU threads), and two client machines each equipped with 4 AMD 12-core processors (48 total cores per machine). The machines were equipped with Intel 82599 Ethernet adapter cards and connected via a Cisco Nexus 5010 Ethernet switch. On the server we ran the multi-process Apache webserver, which spawns 400 processes by default. All processes wait on the same IP address and port for incoming connections. Each client machine is running one instance of the Apache Bench [17] test application configured with 1000000 total requests and concurrency level set to 200. The concurrency level sets the number of requests generated in parallel. Note that any higher concurrency level saturates the load on the processors but not the 10Gb/s Ethernet bandwidth, demonstrating a scalability problem on the software transmission path.

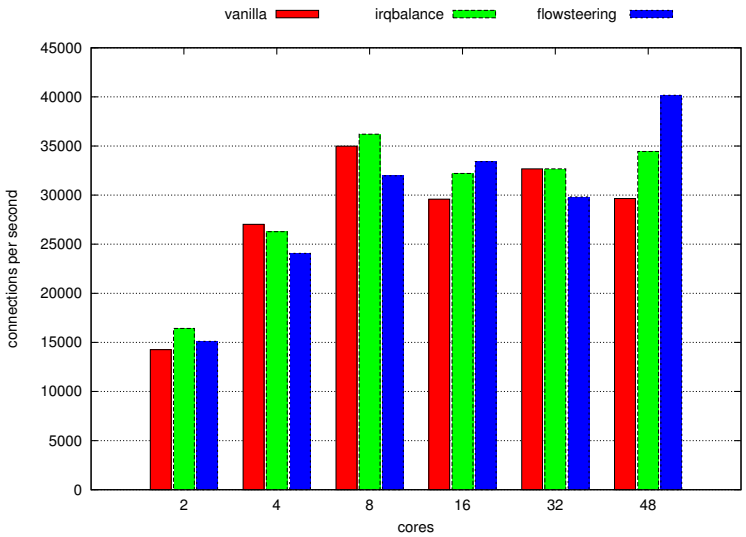

Figure 6.1: Connections accepted per second

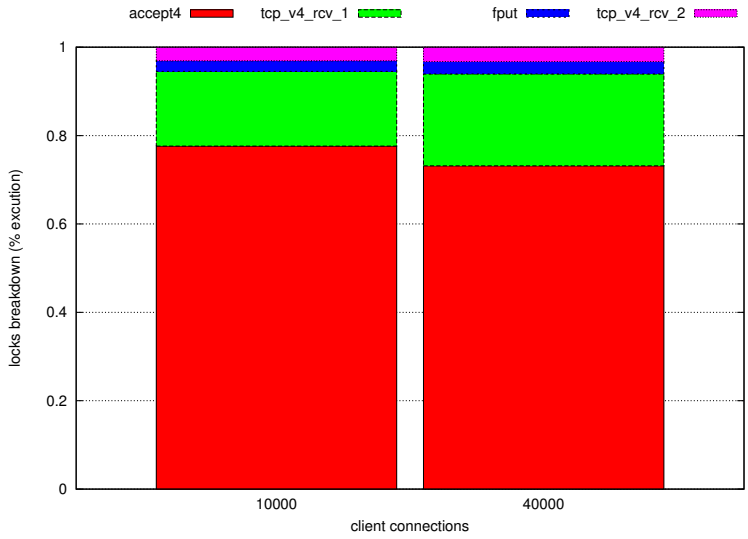

Figure 6.2: Break down of locks in apache server instance

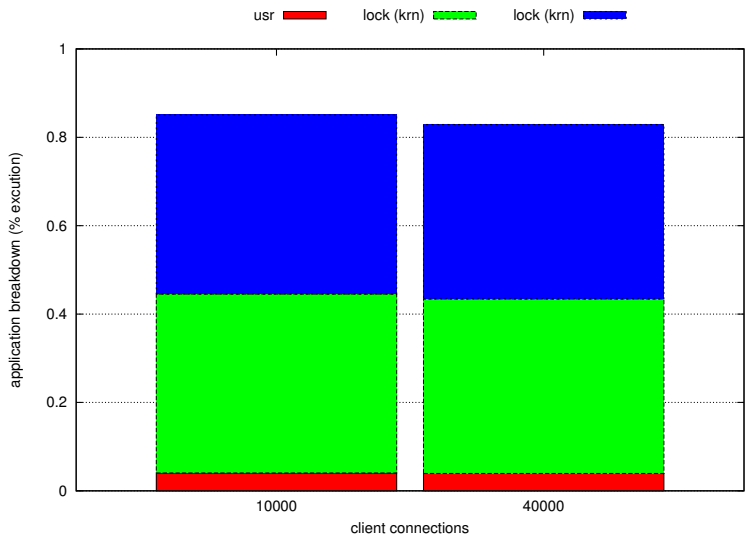

Figure 6.3: Breakdown of applications

Figure 6.1 shows the number of accepted connections per second by varying the number of processor cores enabled on the server machine (all other cores are turned off). The Figure 6.1 shows that after 8 cores the number of accepted connections stagnates, and even drops slightly for higher numbers of cores. The test was repeated with vanilla Linux (vanilla), with dynamic interrupt rerouting (irqbalance) in order to balance the number of packets processed per core, and with software receive flow steering [1] (flow steering) which has been shown to provide performance benefits over the previous methods. However, there is no net improvement when adopting any optimization

on top of vanilla Linux, and the full network interface bandwidth is never achieved during the experiments. The current Linux network stack suffers from poor scalability on the accept path.

Now another question raises: why the number of accepted connections does not scale up? It is due to the operating system. In fact, when more than 8 cores are enabled the server machine starts to become not fully loaded, meaning it should be able to serve more connections. However, server processes are waiting in the kernel spinning on shared data structures instead of serving incoming connection requests. This is captured by Figure 6.2 and Figure 6.3 which report a breakdown of the execution of all Apache processes in the system when varying the number of generated connections. Figure 6.2 shows that Apache spends up to only 4% of its execution time in user space executing application code. Up to 65% of the execution time is spent in the kernel spinning on various locks (25% of those on the accept queue), and only roughly 30% of the execution time is spent executing kernel code, i.e., everything in the kernel except synchronization. Figure 6.2 breaks down the cost for accept_krn() from Figure 6.3 and shows the four main sources of overhead on accept().

The tcp_v4_rcv_2 and tcp_v4_rcv_1 costs are at the TPC/IP level of the network stack while fput and accept4 costs are at the software socket interface level.

As found in [39, 7] the issue in Linux's TCP/IP accept path is due to a single shared queue and hash table data structures protected by a single (coarse grain) lock. Motivated by previous work on scalability and by these results, one might think that replacing single lock data structures with fine grain locking data structures can solve the issue. Figure 6.2 shows that one can expect a possible reduction of circa 25% on total application execution time when removing the coarse grain lock on the accept data structure. However, replacing the data structure, considering it is even feasible, will still leave the application spinning on locks in the kernel for another 40% of its execution time, leaving room for improvements outside the software network stack. Now that we have seen the existing scalability problems in monolithic kernels which stems from their design itself. Monolithic kernels rely heavily on share data structures across multiple cores, which come under contention due to locks. This compelled us to explore the avenue of replicated-kernel (special form of multi-kernel) OS such as Popcorn as a solution to this problem, hence we embodied NetPopcorn.

Chapter 7

Snap Bean: A Replicated-Kernel Device Driver Model For Multiqueue Devices

In a replicated-kernel each device is assigned to only a single kernel instance at the time. Therefore, a single device can only be accessed from a single kernel Such kernel proxies the access to the device to all other kernel instances. In [3] it is shown that proxying is efficient, however the kernel that implements the proxy can be overloaded by the messaging that is necessary to software forwarding the network packets to each of the other kernels. In order to mitigate this overhead we designed Snap Bean Device Driver, which does not require software forwarding of network packets. Snap Bean Device Driver does not substitute a Linux device driver but can be snapped on the original device driver to enable it to run in a replicated-kernel OS with few modifications.

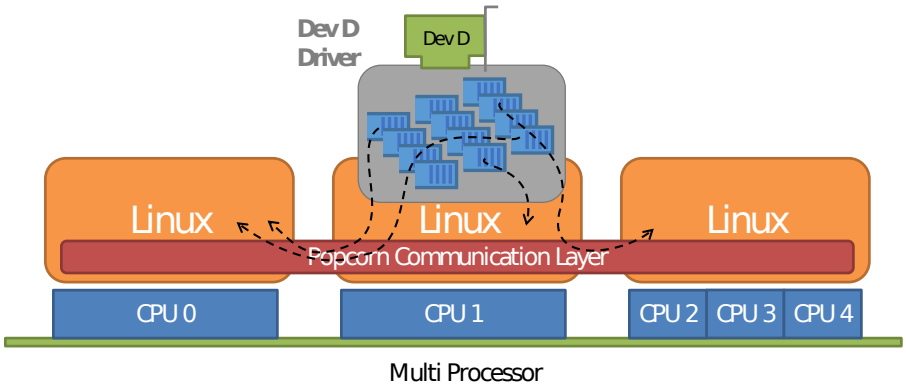

Figure 7.1: Popcorn proxy network device

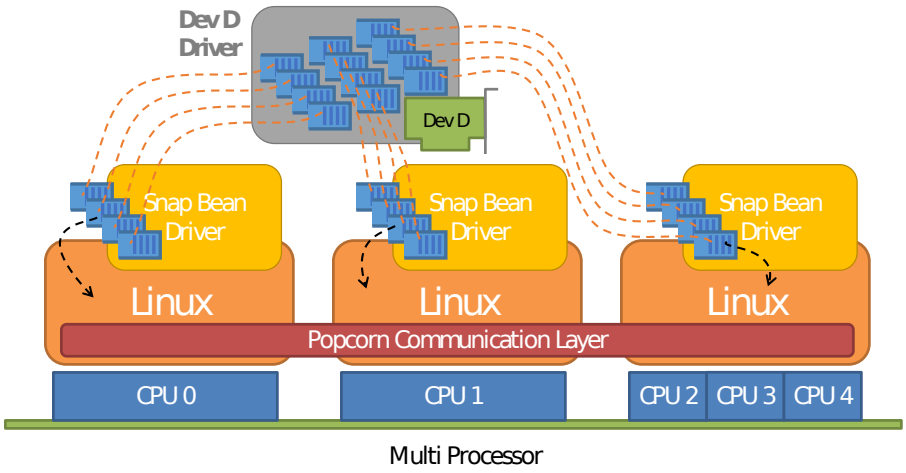

Figure 7.2: Snap Bean network device

7.1 Snap Bean: Design

The Snap Bean is designed to work with all modern multi-queue network devices (e.g. Intel 82599ES [14]). When a multi-queue device (or SRIOV) is available, the hardware provides an opportunity to reduce the software messaging otherwise necessary within a Popcorn proxy driver. Snap Bean is aware of the multiple queues, and exposes the handling of these queues to different kernel instances.

Each queue in a multi-queue device is simply a ring buffer of incoming packets (for the Rx side) or of packets that have to be transmitted (for Tx side). Today a network device integrates 16 to 4096 queues (or even more!). Those queue are queue pairs: Tx and Rx have different ring buffers. Every queue is also programmed for interrupt delivering for each receiving or transmitting event. Queues, are equally distributed between CPUs in a system. Therefore, a CPU that is transmitting a packet will use the queues associated to itself to send that packet. However, when a packet arrives, to which queue will it be delivered? Modern network cards have a filtering pipeline through which every incoming packet goes through that will decide, by for example the calculation of a hash, the queue to which it will be delivered. Multiple filtering capability are implemented in hardware.

The Snap Bean device driver can be plugged on each network device driver and allow the sharing of the hardware queue among multiple kernel instances in a multiple kernel OS, see Figure 7.2. Differently from the proxy driver in Figure 7.1 packets are received from the hardware from each kernel (black arrow). In Figure 7.2 the orange dotted line indicates that the ring is memory mapped in that kernel, and managed by that kernel.

The Snap Bean driver is compatible with the Linux network interface. Moreover, it uses the Popcorn communication layer to interact with the other snap Bean drivers on the other kernel instances for initialization and eventual load balancing (see next Section). More functionalities, to handle network setup are also implemented by this driver. At initialization time the driver register a net-

work device, and it emulates ARP, based on the data maintained by the already loaded kernels. Moreover, all the other information for ARP tables and routing are also migrated automatically at this level.

Even if queues are statically associated to each kernel which packets are delivered to each queue is decided by the card based on a hash function that is user-defined usually based on a 5 values tuple, as source address and port, destination address and port and protocol [14]. For the purpose of this work we define a flow group, a group of ports with identical upper 12 bits to be a filter rule that makes packet with those incoming source ports to be delivered to a particular queue. A flow group can be assigned to a single queue at a time. At initialization a static and homogeneous assignment of flow groups to queues (thus kernel) is created. At initialization each kernel has an equal number of flow groups.

7.2 Snap Bean: Implementation

To implement Snap Bean we faced many challenges. The most important are:

- Transmit and receive from the secondary kernels directly.
- Assign kernel local buffers for sending and receiving network packets
- Routing tx/rx interrupts to appropriate kernels for handling.
- Provide a load balancing framework.

We have implemented Snap Bean on Intel ixgbe driver shipped with Linux 3.2 kernel. Our target hardware is Intel 82599ES 10 Gigabit Ethernet card.

7.2.1 Tx/Rx from Secondary kernels

By secondary kernel we mean a kernel that does not own the device. Only one kernel owns the device and initializes the PCI space for the device. To enable Tx/Rx from a secondary kernel without engaging the primary kernel we had to provide a way to access the device registers. To achieve this we need to find the PCI address space mapping from the primary kernel and implant the mapping in the secondary kernel.

In order to do that Snap Bean driver sends the primary kernel a remote_eth_dev_info_request. It contains information about the secondary kernel. The primary kernel sends back vital information for accessing the device registers back to the secondary kernel. It contains base address of the device, the address space length. When this response is received at the secondary kernel, Snap Bean driver remaps this space into the secondary kernel's address space. Intel ixgbe driver keep all

information related to device in a data structure called ixgbe_device. Snap Bean makes kernel-local copies of this data structure to reduce contention. All information about each Tx/Rx ring is kept in some other data structures Snap Bean remapped them in memory. Each kernel only accesses the data structure for its own tx/rx queue. With these steps the secondary kernel is able to access the registers on the device without reinitializing it and without primary kernel's involvement. Intel 82599ES has a separate register set for each queue. There are 64 sets of identical sets of registers for 64 queues. Secondary kernels only access registers of their own queue so there in no contention on the data going in and out. The physical address of all the queues are separate from each other and accessed through PCIe commands so corruptions does not occur.

Transmit (Tx) operation

The Intel 82599ES transmission mechanism is handled by using DMA and few control registers. In short, when a new skb (kernel data structure that holds network packets) is to be transmitted, it has to be registered in a ring descriptor. These descriptors are data structures used by the driver to keep track of which skb is to be send through the hardware. These are pre-allocated and their physical address is written in hardware register. In Intel's term it is the tail pointer. So, to transmit a skb we register its physical address in one of the descriptors and write the descriptor to the tail pointer. The hardware DMA will transfer the data to the card where it will eventually be sent. Snap Bean registers as a regular network card to the secondary kernel. Thus, when a packet has to be sent, the TCP/IP stack, after adding the required headers sends it to the Snap Bean driver to be sent. The Snap Bean does the transmission step described in Intel 82599 data sheet [14] to send the data using the tx queue assigned to that kernel. It does not need to send any message to primary kernel. As it only access the registers assigned to the queue managed by this kernel there is no contention. It uses the copied PCI space mapping to access the registers.

The used skb buffers are cleaned by the interrupt handler. Snap Bean interrupt cleans the skbs after they are sent.

Receive (Rx) Operation

When a new packet arrives at a particular queue an interrupt is generated to the CPU. Intel ixgbe driver uses NAPI("New API") [19] to receive packet. In this mechanism one each occurrence of interrupt the kernel will collect more than one packet. This number is assigned when the driver is loaded. This lowers the interrupt latency and provide better throughput. Snap Bean also uses this technique to handle interrupts. How interrupts are redirected to separate kernels is discussed in the next section.

Like Tx, also Rx is handled by the DMA. Therefore the DMA engine must know where to send the data. In order to do this Snap Bean allocated 512 preallocated skbs and setup the DMA to transfer data in those buffers. This buffers are local to each kernel this reduces cache contention while improving locality. In Intel 82599ES has registers which maintain this buffer informations.

Figure 7.3: MSIx address format

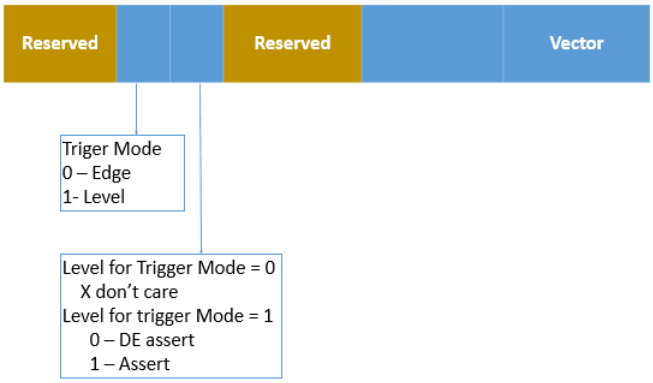

Figure 7.4: MSIx Data Format

Like Tx buffer descriptors the Rx buffers are also held in a descriptor, and the physical address of this buffer is written in a register to allow DMA engine to know where to store the data. The DMA engine changes a status in the buffer descriptor to indicate this buffer has new data.

Like Tx, Rx buffers are cleaned and refilled in the interrupt handler.

7.2.2 Interrupt Routing

Intel 82599 uses Message Signaled Interrupt (MSIx). MSIx helps in many ways as it does not require an Interrupt pin to raise an interrupt. For example each of 64 queues of Intel 82599ES has its own interrupts. In legacy interrupt system it would require 64 pins and which is not feasible in PCIe. MSIx allows for multiple interrupts in the devices. With MSIx every interrupt is basically a PCI read to a special memory location owned by a specific core.

These MSIx interrupts are assigned when the device is initialized. Each MSIx capable device must have registers which hold MSIx information. MSIx interrupt send a message when an event happen and the processor is notified of the interrupt.

The MSIx message has two parts, address part and data part.

As we can see from Figure 7.3 in the address part has a field called the destination ID . This is the CPU id of for which this message will be delivered to. In the data part we can see from Figure 7.4 there is vector which is used by the kernel to know which interrupt number has happened and which Interrupt service routine must be called.

For Popcorn the primary kernel assigns all the MSIx entries at device initialization time, secondary kernels do not reinitialize the device. All the interrupts are configured to be handled by the primary kernel and CPU 0. Now, when a secondary kernel boots up the Snap Bean Driver has to redirect the interrupt of the queue which will be handled by that CPU to that kernel. For this we developed a mechanism in which the driver allocates a free interrupt in the secondary kernel and looks for its vector in that kernel. Then it sends that vector to the primary kernel along with the kernel ID. Upon receiving the kernel id the primary kernel looks up the the queue that is assigned to that kernel, then it finds out the interrupt number of that queue. It then looks for the data structure that has the information about the APIC id of the CPU running the secondary kernel. After this is found the primary kernel modifies the Destination ID field in the Address part of the MSIx message and changes the Vector field in the data part with the vector it received from the secondary kernel.

Now any interrupt generated on queue will be redirected to the secondary kernel. On the secondary kernel Snap Bean registers the interrupt as its interrupt and uses the NAPI to register the handler. It follows the step described in the Intel 82599 data sheet to properly handle interrupts.

7.2.3 Framework for Load Balancing

Snap Bean provides all the framework required to support Angel: The load balancer. Snap Bean provides functionalities to change the hardware setup to move connection flows. Intel 825959ES uses hashing on 5 tuples of the network packet (protocol,source IP, Destination IP, source Port, destination Port) to redirect flows to the proper queues. Snap Bean provides functions to move the flows from one kernel to the other. We will discuss this more in next chapter.

7.3 Results

We have tested Snap Bean to find out weather it adds any overhead. We tested the driver with a micro benchmark to test the ping-pong latency. First we tested with the primary kernel with Intel ixgbe driver and later tested Snap Bean from the secondary kernel.

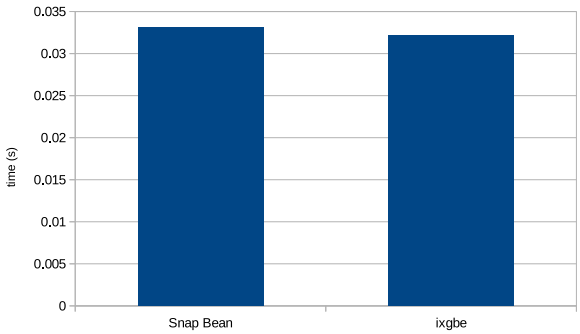

Figure 7.5: Snap Bean vs ixgbe simple mirco benchmark

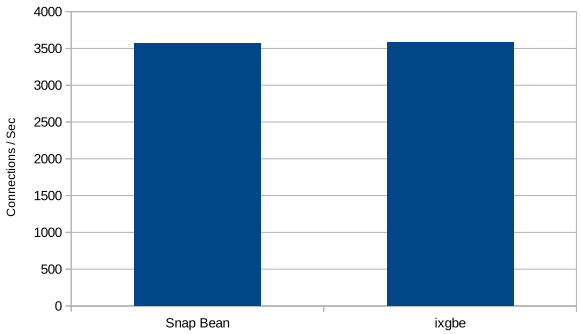

Figure 7.6: Snap Bean vs ixgbe apache benchmark

The results are shown in the Figure 7.5. In the test the we use a simple client and server where the client sends a small request to the server and it replies with the time on the server. It can be seen the driver performance is almost identical to ixgbe. We further test this by using apache benchmark and run one instance which generated 100000 connection requests for a 4 Byte file and compare the results. Figure 7.6 demonstrates, Snap Bean add minimal overhead (less than 1%) on top of the ixgbe driver.

Chapter 8

Angel: An Opportunistic Load Balancer

We observed that the work-sharing algorithm designed in the context of distributed scheduling can be applied in the context of multi-queue NICs. Differently from a scheduling queue a receive ring buffer is not filled in by the software but by the hardware. Moreover, while in distributed scheduling a single task can be moved between queues, here a virtual group of connections, a flow group, should be moved all together. We have designed Angel: a load balancer which distributes flow groups when there is an imbalance in the the network workload. Although a static partition of the flow groups works well when the incoming packets are homogeneously distributed among the flow groups; a non-homogeneous distribution, or a distribution that targets a narrow subset of queues creates a performance degradation of the network service running at the application level. However, the other kernel instances sit idle while they can be used to keep the expected level of service. In order to mitigate this issue we propose to dynamically reassign the flow groups to different kernels. The load balancer performs this dynamic reassignment. To move a flow group we are faced with a challenge that occurs from the fact that we are in a multiple kernel environment, where we have separate network stacks for each kernel. Because a flow group is composed by multiple active connections, the naïve idea of a load balancer is to move the connections to another kernel. However, migrating active connections in a replicated-kernel OS can be very expensive; all the kernel state and the network stack state should migrate, moreover, with live connections, possibly different packets should be moved by software (via the messaging layer) until the hardware changes the queue destination of the packets. To avoid this problem we came up with an heuristic that exploits the design of TCP/IP and avoids messaging.

8.1 Connections Statistics

We keep information about the flow group usages. Every time a connection is created or closed we increment or decrement a reference counter in its flow group. We move a flow when this value is zero which indicates that there are no connections through this flow group. At this point we are

ready to migrate a flow group.

Additionally, for each kernel we keep the total active number of connections. A high-number of active connections means that the kernel is overloaded, therefore past a threshold that kernel will start to send out zeroed flow groups starting a load redistribution activity. Other then an history of active connection we are also keeping the CPU load usage. A kernel thread periodically updates these values that are used in the distributed re-balancing process. Such thread maintains a moving average of these values, it represent a compact representation of its history to be communicated to the other kernels

8.2 Load Redistribution

When Angel detects that a kernel exceeds the maximum connection pressure threshold it broadcast a message to all the other kernel instances. Each kernel that receives that message will answers either with a message that says that it cannot accept any flow group because it is already overloaded, or with a message that summarize its load status in terms of CPU load and flow group usage. After receiving the responses from the peer kernels Angel sorts the values coming from the non overloaded kernels by descending CPU load and re-assigns flow groups starting from the highest ranked. As many flow groups as possible are moved for each redistribution act (a minimum threshold has been set). A kernel that receives a flow group offer confirms with an acknowledge message to the offering kernel. The load balancer on the offering kernel than changes the flow group destination queue. Note that only the owner of a flow group interact with the hardware to move the flow group. When the minimum amount of flow groups was not moved, Angel retries until it can move them. However, a maximum threshold exists.

8.3 Flow Group Migration

Each flow group is associated to a single ring buffer at any instant, therefore to a single kernel only this is enforced by the hardware design. Due to the fact that any kernel can eventually manage any flow group, every Snap Bean driver instance maintains a table with the state of each flow group. Each flow group can be in three states: ACTIVE, ROUTED, or TRANSITING. In a static configuration, in which Angel is not moving any flow group between kernels, the kernel flow groups are either in ACTIVE state, or in ROUTED state. The former are the ones which packets will be delivered from the NIC to the specific kernel, while the latter are the ones which packets will be delivered by the NIC to other kernels than the specific one. Thus, only the packets that hash to ACTIVE flow groups are delivered to the software and will instantiate connections. However, when Angel moves flow groups between kernels, to avoid situations in which we migrate an already opened connection, we switch the flow group state from ACTIVE to TRANSITING when the number of connections switches to zero. When the flow group state is TRANSITING

any packet received that belong to such group is discarded by the software. However, at lest for TCP/IP, this is not a problem, the source will issue a new connection after a timeout. After the NIC has been reprogrammed to deliver the packets to another queue, the first packet of the flow group will trigger the new kernel to communicate the previous kernel to switch the state of that flow group to ROUTED. While this kernel will switch the flow group state to ACTIVE.

8.4 Implementation

Angel is implemented as a kernel thread. It is launched when the Snap Bean driver is loaded in the kernel. Angel utilizes the functionalities provided by the Snap Bean driver to load balance. A more detailed description of Angel is provided in the next sections.

8.4.1 Initialization

The 82599 has 64 Tx/Rx queues and each one is associated with one Popcorn Linux kernel instance. In order to have distribution of connections, we use Intel Flow director filters. There is also Intel Receive Side Scaling (RSS) technique which distributes packets to different queues but it only works for 16 queues. The Flow director filter has 32k filter entries at maximum. This filter considers 5 tuples (Source address, Source port, Destination IP address, Destination port Address and protocol). The filter shares the on device memory of 82599 with receive buffers. That means if we use all 32k filters, the hardware receive buffer memory is reduced which can result in performance degradation. In order to minimize the number of entries in the filter we statically partition the connections based on their incoming source port. Flow filters creates hash based of the 5 tuples and uses that to match incoming packets. We set it up to only consider the highest 12 bits of the source port of the incoming packet. So we have 4096 entires with each representing 16 ports,we call these port groups. Port groups are assigned to kernels in homogeneous fashion. We use the formula (port mod total_kernel_no) and assign that port to the kernel with that ID.

As defined in the previous Chapter, we call this value a flow group. Because each of this entries represents 16 ports. So for that reason if one of these entries are reassigned 16 source ports are reassigned.

8.4.2 Re-routing of flows

Initially the flow-groups are equally distributed and the fact that source ports are quite random, gives a very good distribution to the different kernels. However, can be situations where some particular ports can get more than others and some kernels can get overloaded. In this situation Angel will re assign some flows to others. It needs Snap Bean functionalities to rewrite the flow director filter to reassign the flows to other kernels.

In an event of system overload Angel wakes up and It sends a load_balance_info requests. The other kernels may or may not agree to take other kernels load depending on their own load. If the receiving kernel is overloaded itself it sends negative values back to the requesting kernel. However, if it is not overloaded it will spend its CPU usage history and SYN-FIN balance history to the requesting kernel. The load balancer sorts the replies and sends redirects flows to the willing kernels with lower SYN-FIN balance and CPU usages.

8.4.3 SYN-FIN Heuristic

We use a heuristic we call the SYN-FIN balance. This is basically the difference of the number of syn packets the system has seen and number of fin packets the system has seen. The first step of TCP connection establishment is sending of a SYN packet from the client to the server. When we receive a SYN packet we increment the SYN count. On the other hand TCP connection tear down phase the server will send a packet with the FIN bit set(FIN/FINACK), we increment the FIN count. Their difference gives us an indication of how many active connections are there is the system. When the server receives a new SYN packet it checks to see if the SYN-FIN balance has reached a certain threshold or not. If it has a load balance request is posted which wakes up Angel.

8.5 Important Statistics

The load balancer is implemented as a kernel thread which waits on load balance requests and wakes up when it gets one. We have another thread that keeps statistics of the system. It keeps the statistics of SYN-FIN balance and CPU usages in a moving weighted average. These statistics are required for Angel to load balance the system. The statistics thread uses weighted_cpuload() to find out CPU usages. It keeps taking sample at every 200 ms. This value can be configured by user. These statistics are sent when a load_balance_info request is made.

8.6 Experiments

In this section we describe many experiments we did with NetPopcorn and show the results.

8.6.1 Hardware

The experiments have been conducted on a setup with 3 client machines and 1 server interconnected by a dedicated 10Gb CISCO Nexus 5010 Ethernet switch, therefore there is no external traffic that interfere with the experiments. Each machine is equipped with a 10Gb Intel 82599ES Ethernet card. The client machines mount an AMD FX 8350 8 cores processor at 4GHz and 12GB

of RAM each. The server machine is a dual socket Intel Xeon E5-2695v2 with 96GB of RAM. Each Intel Xeon is a 12 cores processor, 2-way hyper-threaded, with 30MB of cache. During our initial evaluations (see Section ??) we attested that 3 client machines are sufficient to generate enough traffic to push Linux to its limits within the experimental hardware used. Differently from our initial evaluation here we disable hyper-threading, therefore the presented experiments are limited to 24 cores.

8.6.2 Software

We compared NetPopcorn with vanilla Linux, and with Linux Affinity Accept [39] (we will refer to it simply as affinity-accept, or affinity). NetPopcorn is based on Linux 3.2.14, vanilla Linux is version 3.2.14, and affinity-accept is based on Linux 2.6.24. We quantified the differences in running the experiments on vanilla Linux version 3.2.14 and 2.6.24 being inferior to the 2%, therefore we don't report Linux 2.6.24 numbers. Moreover, we didn't port affinity-accept to Linux 3.2.14, but we had to fix it to make it properly support epoll() patches are available at our project website. NetPopcorn is built on top of Popcorn Linux for multicore platforms [3, 30]. NetPopcorn adds only around 2k lines to the source base. The split device drivers is about 1.7k lines including Angel; the modifications to the 82599 Linux device drivers amounts to 350 lines; and the interrupt reprogramming code is around 100 lines. The source code is available on the our website along with any modified test application and benchmark. NetPopcorn has been configured with one core and 4GB of RAM per kernel.

The experiments are done using webserver (Apache and Lighhttp).

Web Serving

We extensively compare NetPopcorn and competitors using two different web serving applications: Apache [16] and lighttpd [31]. Each of these applications stresses a different level of the operating system software due to their different design. Apache starts multiple processes all awaiting for incoming connections on the same socket using the accept() system call. Lighttpd starts multiple processes as well, but each of them is using the epoll() system call on the same listening socket. accept() sits it a lower layer in the system software than epoll, moreover the first was introduced before than the latter in operating system. For every configuration we launched 10 instances of Apache or Lighttpd per-core. We evaluated Apache and Lighttpd with the Apachebench [17] test application distributed with the Apache webserver. The webserver is started on the server machine first, then the client machines coordinate and each starts 10 instances of Apachebench in parallel. Each Apachebench instance is configured with concurrency level set to 1000, 100000 connections, and fetches a static file of 4B (that contains a random generated number as in [7]). A minimal page size has been used in order to stress the operating system and study its scalability in the number of connections that can be handled at any time. The benchmark has been modified to generate connections within a specific subset of TCP/IP source port numbers to stress Angel.

8.7 Evaluation

In this section we show that a replicated-kernel OS design with our network design outperforms the usual SMP network stack, while emulating the POSIX semantic. We analyze the scalability in terms of requests per second, failed connections and request latency, for Apache and Lighttpd. We quantify the same metric in NetPopcorn with and without load balancing. In a setup with an homogeneous distribution of requests 8.7.1 with asymmetric distribution (Section 9.3) and targeted distribution 8.9. When not explicit, each experiments has been run 10 times. We report averaged results. Experimental results reports different numbers of CPUs (or cores). In the case of Affinity-accept and Linux we used the Linux's `maxcpus` boot time argument to force the OS to use only a subset of the available CPUs, for NetPopcorn, we load that specific number of kernels (one core per kernel).

8.7.1 Scalability

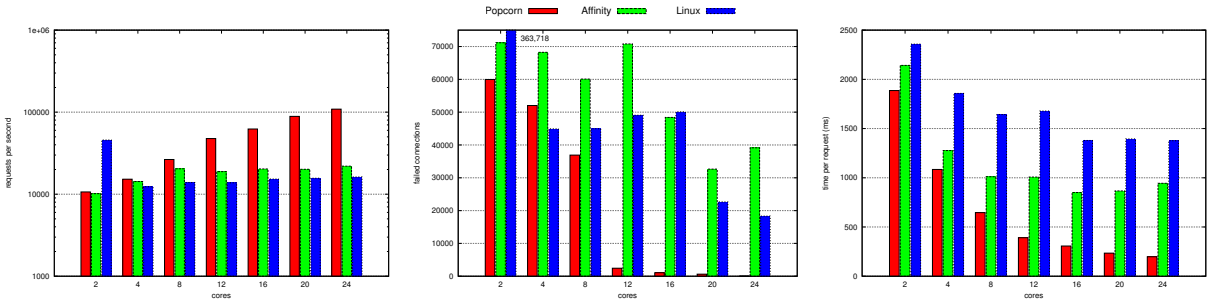

Figure 8.1: Apache webserver, number of requests per second.

Figure 8.2: Apache webserver, number of failed connections.

Figure 8.3: Apache webserver, time to serve a request.

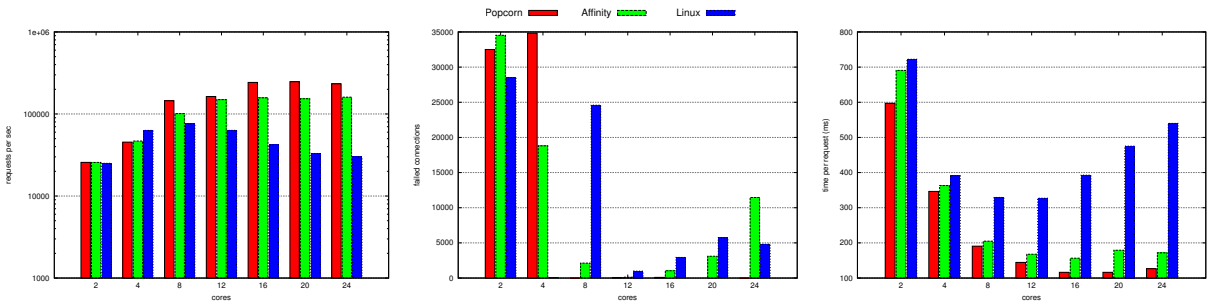

Figure 8.4: Lighttpd webserver, number of requests per second.

Figure 8.5: Lighttpd webserver, number of failed connections.

Figure 8.6: Lighttpd webserver, time to serve a request.

Apachebench has been used to exercise Apache and Lighttpd. Affinity-accept and Linux have been started normally. For this set of experiments NetPopcorn has been started with an equal distribution of flow groups to kernels and load balancer enabled, although we will show we compare

NetPopcorn with and without load balancer through the end of this section. Figures 8.1, 8.2, 8.3, report respectively Apache's average number of requests per second, number of failed connections per run, and time per request. Figures 8.4, 8.5, 8.6 report the same values but for Lighttpd. All these numbers has been collected under stable state (i.e., after a warm up transient).

Comparing Apache's and Lighttpd's graphs the main take away is that NetPopcorn is able to serve always more requests per seconds than Linux and Affinity accept (despite in two low-CPU count cases); Affinity accept outperforms Linux but it is doing much better on epoll() than on accept() (more details on this below). Linux network stack behaves better when concurrency is handled in the event poll layer; with epoll(), Linux can handle up to 90k connections per second on 8 CPUs, however adding additional CPUs doesnt give any benefit, in fact the number of serviced connections is dropping for Lighttpd, and is staying still for Apache. In the Apache experiment Linux has a peak number of served connections (46k) on 2 CPUs; thus lead one to think that is the best performing system when only 2 CPUs are used. This is misleading, in fact, Linux accepts an high number of connections, but many of them are timing out, see Figure 8.2 and Figure 8.5 (because of this unusual behavior we re-run the test 100 times confirming the pathological behavior). Affinity-accept can accept roughly 30% more connections than Linux when more than 8 cores are used, however it is affected by the same stagnating effect as Linux itself the number of accepted connections per second does not go up. With Lighttpd the benefits of Affinity-accept compared to Linux are more evident: after 8 CPUs the number of connections per second drops down for Linux while is keeping to scale up for Affinity-accept. Again, this highlights the benefits of epoll() interface compared to the usage of accept() decoupling the network stack from scheduling makes applications perform better. This behavior is less evident in NetPopcorn, which keeps scaling in number of serviced requests per second when running Lighttpd as well as Apache, even if the maximum number of serviced connections is circa double in the case of Lighttpd (109k vs 248k). NetPopcorn serves up to 8 and 7 times more connections than Linux and 2 and 5 times more connections than Affinity-accept, respectively for Lighttpd and Affinity-accept, both cases for 24 CPUs.

NetPopcorn drastically reduces the number of failed connection from 8 CPUs up, despite of the test application. With Lighttpd the number of failed connections drops down in the order of the tens, while with Apache in the order of hundreds. Affinity-accept shows a greater number of failed connections than Linux when using Apache, however this is not always true when switching to Lighttpd. Both Linux and Affinity-accept have between 2 and 3 order of magnitude more failed connections than NetPopcorn (for 8 or more CPUs).

Figure 8.6 report the average total time per request from the client side (Apachebench) that includes any sort of software overhead (therefore the actual value is less important than the trend itself). NetPopcorn is the quickest at any core count showing that a replicated-kernel OS provides benefits already at low core-counts (on 2 CPUs up to 23% faster than Linux and 16% faster than Affinity-accept, for Apache and Lighttpd respectively). Similar consideration drawn for the number of the served connections per second can be drawn about the time per request. Linux scales only up to 8 CPUs, on Lighttpd it anti-scales after 8 CPUs. Affinity-accept doesn't provide much improvements after 12 CPUs showing a stagnating performance. Instead, NetPopcorn continues to scale up to 24

CPUs.

8.7.2 Overhead of Angel

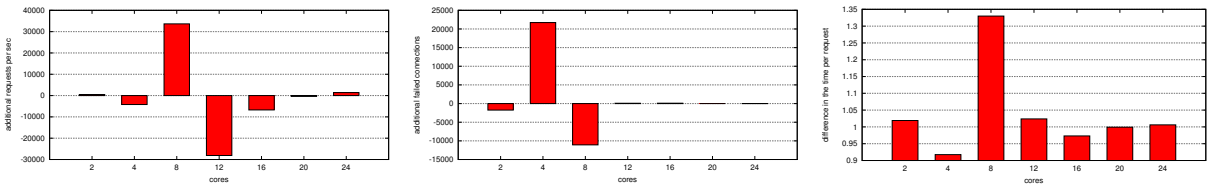

Figure 8.7: Lighttpd webserver, additional requests per second served due to Angel.

Figure 8.8: Lighttpd webserver, additional failed connections due to the Angel.

Figure 8.9: Lighttpd webserver, with and without Angel, time per request comparison.

In order to highlight the benefits and overheads of having an active load balancer that migrates flow groups between kernel instances we run the Lighttpd experiments without Angel. Figures 8.7, 8.8, 8.9 respectively show the additional request per second that can be served with Angel (a negative value means that Angel reduces the performance), the additional failed connections with Angel (a negative value means that the load balancer improves the performance), and the ratio of time per request with and without Angel. These graphs show that the for an homogeneous distribution of requests NetPopcorn benefit from Angel especially when the number of cores is 8, increasing the number of connection serviced of 32k and reducing the number of failed connections of 11k. For higher number of CPUs Angel strives to improve the performance of the applications without a too negative effect.

8.8 Asymmetry Unbalance

In this section we studied the situation in which the packet distribution is still homogeneous but all the flow groups are mapped to a single kernel. This experiment exercise the capability of the Angel to quickly re-balance the flow groups among different kernel instances. Because neither Linux not Affinity-accept can be easily tuned to re-create a similar scenario we compare the version of NetPopcorn with Angel (Balanced) with the one without Angel (Unbalanced). Results show the great benefit of the designed Angel.

Figures 8.10, 8.11, 8.12 depicts the distribution of the total time per request with and without Angel for different CPU counts. The profile for the unbalanced case doesn't change varying the number of CPUs flow groups don't move among kernels therefore making it useful to have more CPUs. With the Angel increasing the available CPUs, the number of connections serviced very quick (in

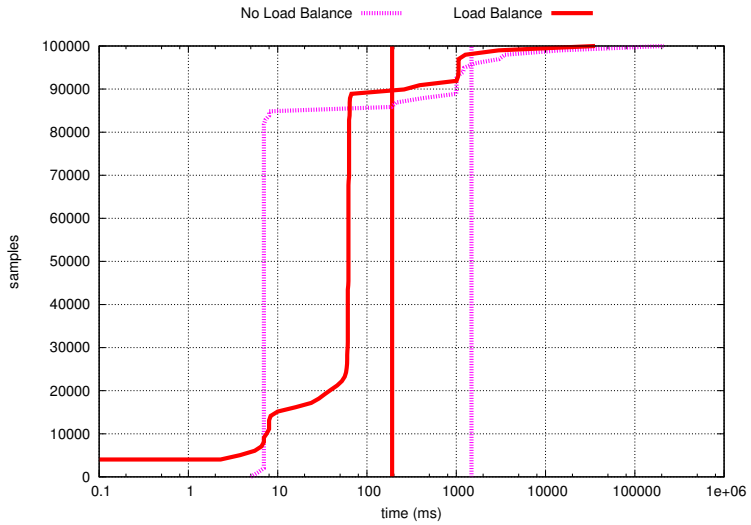

Figure 8.10: Lighttpd webserver, response time profile in the case all the flow groups are on one kernel without Angel (Unbalanced), and with Angel on 8 Cores

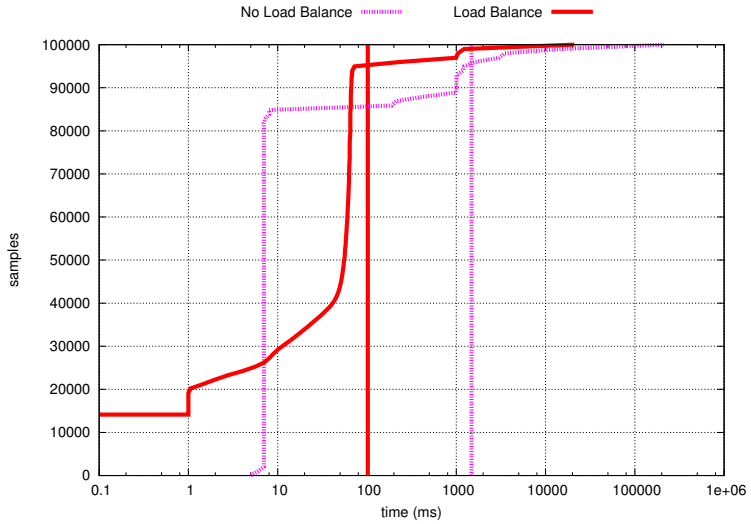

Figure 8.11: Lighttpd webserver, response time profile in the case all the flow groups are on one kernel without Angel (Unbalanced), and with Angel on 16 Cores

the range of 0.1ms) increased (up to 30k with 24 CPUs). Moreover, the tail latency also reduces of an order of magnitude.

Figure 8.15 shows the average values of the profile distributions: the total time for a single request decreases for both cases. However, the unbalanced case is 6.8 times slower than the balanced one. This is reflected in Figure 8.13 that graphs for different number of cores the average number of serviceable requests per second the load balancer provides an order of magnitude improvement in the number of serviced connections. The same applies to the number of connections, Figure 8.14

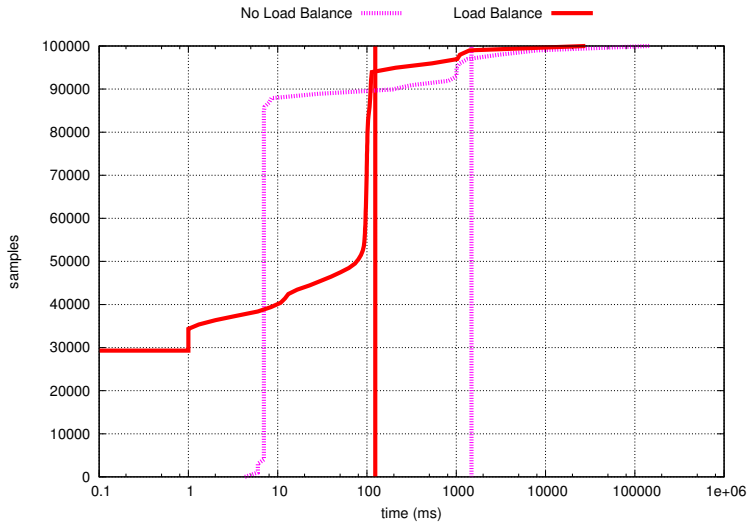

Figure 8.12: Lighttpd webserver, response time profile in the case all the flow groups are on one kernel without Angel (Unbalanced), and with Angel on 24 Cores

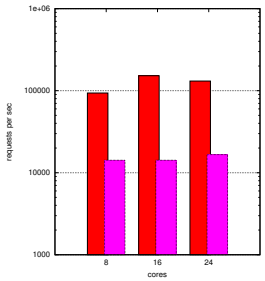

Figure 8.13: Asymmetric assignment. Popcorn with/without Angel. Lighttpd requests per second.

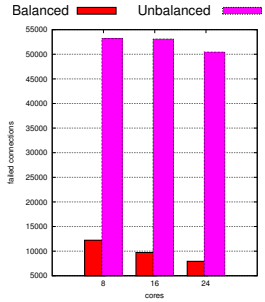

Figure 8.14: Asymmetric assignment. Popcorn with/without Angel. Lighttpd failed connections.

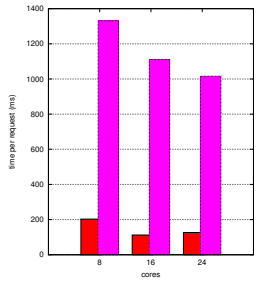

Figure 8.15: Asymmetric assignment. Popcorn with/without Angel. Lighttpd time per request.

shows this trend.

The last metric of evaluation is the time Angel requires to converge, i.e., balance the load (in terms of flow groups) among the kernels. Because an equal distribution of flow groups per kernel doesn't guarantee the system is balanced, we used the total execution time of a burst of requests as the metric. Figure 8.16 shows the worst case request time for the balanced and unbalanced case for 8, 16 and 24 CPUs. Increasing the number of CPUs the total execution time shrinks (from 22s to 16s to 14s) due to the load balancer, compared to an execution of c.a. 160s (an improvement of up to 11.4 times).

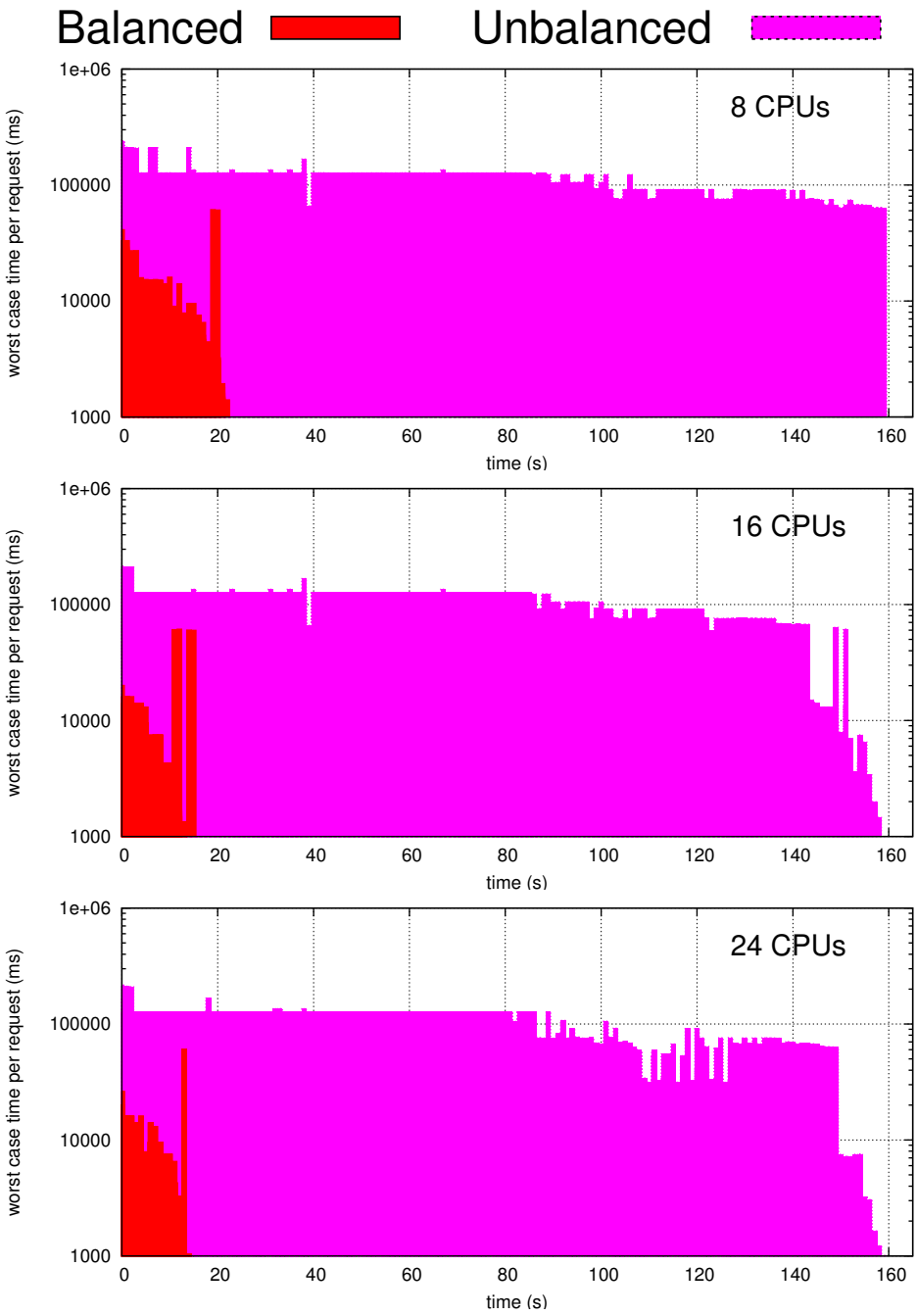

Figure 8.16: Execution trace of Lighttpd webserver on Popcorn with and without load balancer for 8, 16, and 24 CPUs. All flow groups are on a single kernel and the requests are homogeneously distributed.

8.9 Attack Balance

In this section we evaluate how good is the load balancer to react to a situation in which flow groups are equally assigned to kernel instances but connection requests are directed to only one

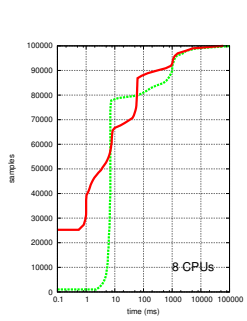

Figure 8.17: Lighttpd web-server, time per request profiles attacking one kernel with 8 total kernels

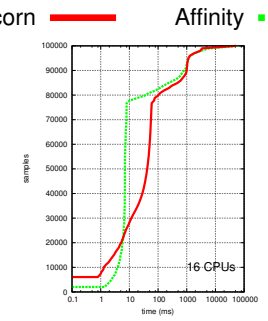

Figure 8.18: Lighttpd web-server, time per request profiles for profiles attacking one kernel with 16 total kernels

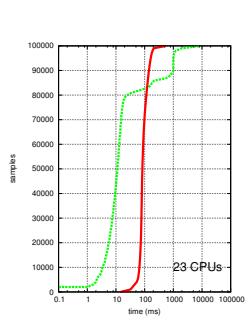

Figure 8.19: Lighttpd web-server, time per request profiles attacking one kernel with 23 total kernels.

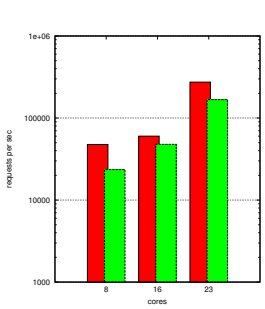

Figure 8.20: Popcorn vs Affinity. Requests per second.

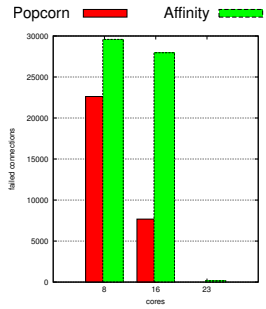

Figure 8.21: Popcorn vs Affinity. Failed connections

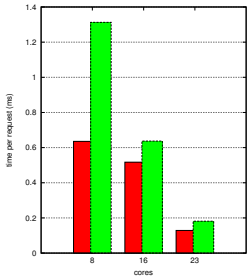

Figure 8.22: Popcorn vs Affinity. Time per request

kernel respect to an initial flow group assignment. (Note that we modified Apachebench for this experiment.) We compare NetPopcorn to Affinity-accept, for which we were also able to recreate an exactly similar experimental setup. Similar to Section 8.8 the experiment consists of a burst of connections. Unfortunately, the methodology used to run the experiments completely freeze Affinity-accept for the case of 24 CPUs, therefore we reported, for both competitors Figures 8.17, 8.18, 8.19 to show the time per request profile varying the number of CPUs for NetPopcorn and Affinity-accept. The fact that NetPopcorn doesn't rely on shared data structure allows improved tail latency increasing the number of available CPUs. NetPopcorn worst-case response time is more than an order of magnitude faster than Affinity-accept for 23 CPUs. Affinity-accept graph shows that while increasing the number of cores the number of connections, that have a very short total time for the request increases this is inverse in NetPopcorn whose value decreases. This is due to the aggressive load balancing strategy implemented that keep retrying redistributed the work, and this requires more work increasing the number of available CPUs but without affecting performance.

The time per request profiles are averaged and the results are reported in Figure 8.22 . Interestingly, increasing the number of cores the value of time per request difference between NetPopcorn and

Affinity-accept become more similar. However, NetPopcorn responds faster than Affinity-accept, up to 2 times faster for 8 cores.

Figure 8.20 and Figure 8.21 show the number of requests per second served and the number of failed connections for this experiment for NetPopcorn and Affinity-accept. NetPopcorn always handles more connections and is subject to less number of failed connections confirming the value of a replicated-kernel design versus an SMP for this class of applications. The number of failed connections in Figure 8.21 drops down to 0 for NetPopcorn and 164 to Affinity-accept: both solutions are able to re-balance the load in case of unbalanced burst (as a security attack). However, NetPopcorn has the capability of handling more request, up to 63% more than Affinity-accept (23 CPUs).

As in the previous Section we are also reporting the worst case total time per request execution traces in Figure 8.23 varying the number of available CPUs. Those reveals as the worst case total time per request is more bounded in NetPopcorn but also the total time to execute the benchmark is half with NetPopcorn compared to Affinity-accept.

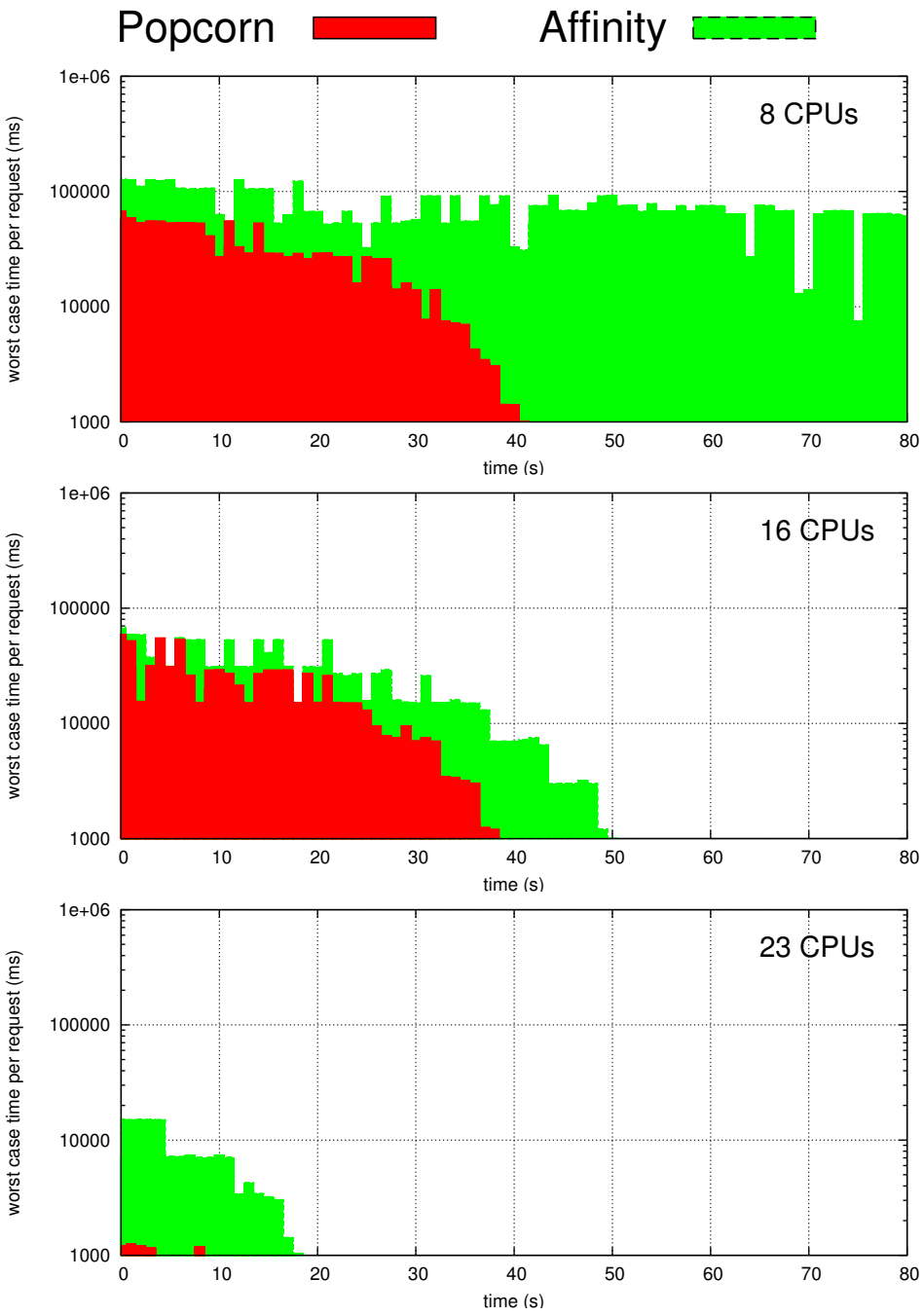

Figure 8.23: Execution trace of Lighttpd webserver comparing Popcorn and Affinity on 8, 16, and 23 CPUs. All flow groups are homogeneously distributed but requests are initially directed to one kernel only.

Chapter 9

Concluding Remarks And Future work

9.1 Conclusion

With this thesis's first contribution filesystem, we enabled Popcorn to take another step towards SSI, along with other services the filesystem is a key component of Popcorn. The next contribution is messaging layer for heterogeneous Popcorn, this allowed Popcorn to expand to heterogeneous Platforms. The messaging layer is the heart of Popcorn that stitches all the kernel instances together and makes it into one system. We show incremental performance improvement from different updates in design and implementation of the messaging layer. From the results it can be seen from single channel to buffer-multi channel the messaging layer reduces execution time by 75%. We Also show that with this messaging layer Popcorn is faster than other methods like OpenCL,Intel Offloading model etc.

The next, two contributions utilize Popcorn to find up to 8x performance over Linux and 5x performance over Affinity accept which considered the best work on network scalability on Linux till date. We show through rigorous testing and multiple metrics that NetPopcorn performs better than other state of the art work. Even under attack situation Angel can keep the system functional. The metrics used to show the superiority of NetPopcorn are the most desired from a server.

9.1.1 Contributions

To summarize, the major contributions of this work are:

- We developed a scalable file descriptor migration protocol for Popcorn Linux. We modified the Linux VFS Layer functions to allow Popcorn services to allow four important functionalities. Namely remote_open , get_file_offset, set_file_offset and remote_close. Together, these are sufficient to support POSIX file interface on Popcorn. The Popcorn file system design based only on messaging without any shared memory requirement. This work allows

threads to migrate seamlessly to different processor islands. We have successfully ran real world applications such as iozone and pbzip to prove its functionality.

- We developed a low latency, pluggable kernel level messaging layer and improved its performance iteratively. We modified Intel SCIF for our requirements and developed a hybrid messaging layer that switches transfer mode according to the message size. We introduced multi-threaded design to utilize the parallelism available in Intel Xeon Phi accelerator. We compare Popcorn's performance against other state of the art Like OpenCL, Intel LEO. With the latest messaging layer Popcorn out performs the competition. The performance improvements shown in section 5.6 is done while keeping other Popcorn services same and only changing the messaging Layer. So it clearly demonstrates the improvements found by the messaging layer designs. The concept has been extended to Arm-x86 platform in recent times.

- Snap Bean device driver is the cornerstone of NetPopcorn. It is a high performance low overhead virtual device exposing multiple queues in PCI network card to different kernels. We developed this driver considering Intel 825959ES 10 Gbps card. The driver acts as a real device to the secondary kernels and performs Tx/Rx operations by accessing remapped device registers, making communication to the primary kernel unnecessary. Results shown is section 7.3 shows that Snap Bean performance is similar to ixgbe performance.

- The last contribution is Angel the load balancer for NetPocorn. This allows the system to be stable under unbalanced workloads. NetPopcorn works on a heuristic we refer as SYN-FIN balance to detect high number of connection requests in the system. It then requests Angel to load balance with other kernels. As shown in section 8.7 Angel has very small overhead but is extremely useful and provides very good performance in a unbalanced situation. It has been tested with synthetic data specifically attacking one kernel, where it successfully load balanced to provide respectable performance. It is also tested with an extreme case where all flow groups are assigned to one kernel while all the other kernels are not receiving any packet. Even on this situation it balanced the load and achieved much higher performance.

9.2 Future directions

For the filesystem work it would be the next logical step to design a full fledge filesystem and renounce the use of NFS. Popcorn does not support direct disk access from the secondary kernels, so a caching based filesystem can be designed, which will outperform NFS. In Figure 9.1 we can see a possible design. As we can see from the figure this new design is based on file cache technique. The cache allows less massages across the Popcorn messaging layer. This cache based file system will act like a normal filesystem but it will be a distributed file system underneath. Applications can use the filesystem like any other filesystem as all the lower level mechanisms are abstracted.

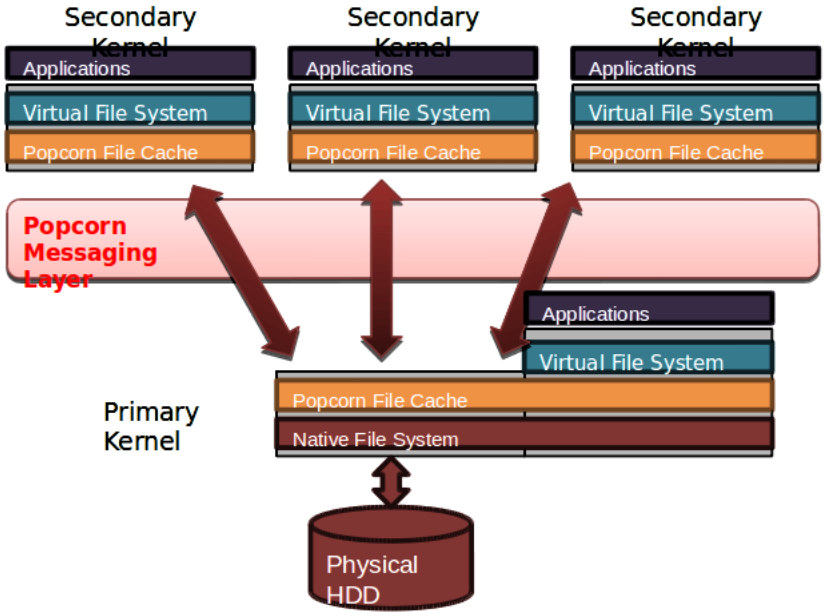

Figure 9.1: Possible Design with filesystem

On messaging layer effort we can go deeper and try to send without SCIF library. In that library there are some redundant code which are not needed for kernel messaging layer. Further exploration can be done with asymmetric number of messaging channels for two sides.

On the NetPopcorn we have parameters that have huge impact in load balancing actions. Like the how many flows must be moved when the balancer wakes up, another parameter controls what percent of flows can be distributed by one kernel, Auto-adjustment of these parameters is an interesting work for the future.

Closer integration with the application can provide better scheduling decisions. The server applications with better service statistics can be given priority when choosing the application to handle new connections. This can provide better response times if come server threads are slower than others.

Bibliography

[1] Scaling in the Linux Networking Stack, 2015.

[2] Saif Ansary. The case for resurrecting distributed virtual shared memory. In *ACM Student Research Competition, 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[3] Antonio Barbalace, Alastair Murray, Rob Lyerly, and Binoy Ravindran. Popcorn: a replicated-kernel os based on linux. In *Proceedings of The 4th Workshop on Systems for Future Multicore Architectures*, 2014.

[4] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. Popcorn: Bridging the programmability gap in heterogeneous-isa platforms. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 29:1–29:16, New York, NY, USA, 2015. ACM.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[6] Andrew Baumann, Chris Hawblitzel, Kornilios Kourtis, Tim Harris, and Timothy Roscoe. Cosh: Clear os data sharing in an incoherent world. In *Proceedings of the 2014 International Conference on Timely Results in Operating Systems*, TRIOS'14, pages 3–3, Berkeley, CA, USA, 2014. USENIX Association.

[7] Nathan Z. Beckmann, Charles Gruenwald III, Christopher R. Johnson, Harshad Kasture, Filippo Sironi, Anant Agarwal, M. Frans Kaashoek, and Nickolai Zeldovich. Pika: A network service for multikernel operating systems. *MIT CSAIL Technical Report*.

[8] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[9] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[10] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[11] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[12] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 12–25, New York, NY, USA, 1995. ACM.

[13] Microsoft Corp. Receive Side Scaling, 2014.

[14] Intel Corporation. Intel 82599 10 gbe controller datasheet, 2015.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[16] Apache Software Foundation. Apache http server benchmarking tool, 2015.

[17] Apache Software Foundation. Apache http server benchmarking tool, 2015.

[18] Linux Foundation. Ext4 filesystem, 2009.

[19] Linux Foundation. Napi(new api:extension to the device driver packet processing framework), 2009.

[20] Brice Goglin and StéPhanie Moreaud. Knem: A generic and scalable kernel-assisted intra-node mpi communication framework. *J. Parallel Distrib. Comput.*, 73(2):176–188, February 2013.

[21] Charles Gruenwald, III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. Hare: A file system for non-cache-coherent multicores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 30:1–30:16, New York, NY, USA, 2015. ACM.

[22] P. Gschwandtner, T. Fahringer, and R. Prodan. Performance analysis and benchmarking of the intel scc. In *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, pages 139–149, Sept 2011.

[23] Intel Corporation. Intel 64 and IA-32 Architectures Software Developers Manual platforms, 2011.

[24] Intel Corporation. Xeon Phi product family, 2013.

[25] Intel Corporation. Intel Xeon Phi Coprocessor System Software Developers Guide, 2014.

[26] Intel Corporation. Intel xeon phi coprocessor system software developers guide, 2014.

[27] ITEF. RFC NFS4, 2015.

[28] Adhiraj Joshi, Swapnil Pimpale, Mandar Naik, and Swapnil Rathi. Twin-linux: Running independent linux kernels simultaneously on separate cores of a multicore system, 2010.

[29] Sanjiv Kapil, Harlan McGhan, and Jesse Lawrendra. A chip multithreaded processor for network-facing workloads. *IEEE Micro*, 24(2):20–30, March 2004.

[30] David Katz, Antonio Barbalace, Saif ansary, Akashay Ravichandran, and Binoy Ravindran. Thread migration in a replicated-kernel os. In *Proceedings of the 35th International Conference on Distributed Computing Systems (ICDCS XXXV)*, 2015.

[31] Lighttpd Fly Light. Lighttpd webserver, 2015.

[32] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300, New York, NY, USA, 2014. ACM.

[33] Adrien Lbre, Renaud Lottiaux, Erich Focht, and Christine Morin. Reducing kernel development complexity in distributed environments. In Emilio Luque, Toms Margalef, and Domingo Bentez, editors, *Euro-Par 2008 Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 576–586. Springer Berlin Heidelberg, 2008.

[34] Martin Maas. A JVM for the Barrelfish Operating System.

[35] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 9:1–9:7, New York, NY, USA, 2013. ACM.

[36] Adam Matthew. Message passing in a factored os, master thesis, 2011.

[37] PCI-SIG. PCI Express Base Specification Revision 3.0 , 2010.

[38] S. Pellegrini, T. Hoefler, and T. Fahringer. On the effects of cpu caches on mpi point-to-point communications. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 495–503, Sept 2012.

[39] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, New York, NY, USA, 2012. ACM.

[40] Simon Peter, Adrian Schpbach, Dominik Menzi, and Timothy Roscoe. Early experience with the barrelfish os and the single-chip cloud computer. In Diana Ghringer, Michael Hbner, and Jrgen Becker, editors, *MARC Symposium*, pages 35–39. KIT Scientific Publishing, Karlsruhe, 2011.

[41] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 291–304, New York, NY, USA, 2011. ACM.

[42] Luigi Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[43] Marina Sadini, Antonio Barbalace, Binoy Ravindran, and Francesco Quaglia. A page coherency protocol for popcorn replicated-kernel operating system. 2013.

[44] Benjamin H Shelton. Popcorn linux: enabling efficient inter-core communication in a linux-based multikernel operating system, 2013.

[45] Mark Silberstein, Bryan Ford, and Emmett Witchel. Gpufs: The case for operating system services on gpus. *Commun. ACM*, 57(12):68–79, November 2014.

[46] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[47] Ashley Stevens. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. Technical report, 2011.

[48] Manuel Stocker. Towards a file-system service for the barrelfish os, 2012.

[49] Tilera Corporation. TILEncore platforms, 2015.

[50] Animesh Trivedi. Hotplug in a multikernel operating system, 2009.

[51] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, April 2009.

[52] Paul Willmann, Scott Rixner, and Alan L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 8–8, Berkeley, CA, USA, 2006. USENIX Association.