Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system

Benjamin H. Shelton

Thesis 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
Christopher Jules White
Paul E. Plassman

May 2, 2013
Blacksburg, Virginia

Keywords: Operating systems, multikernel, high-performance computing, heterogeneous computing, multicore, scalability, message passing

Popcorn Linux: enabling efficient inter-core communication in a Linux-based multikernel operating system

Benjamin H. Shelton

(ABSTRACT)

As manufacturers introduce new machines with more cores, more NUMA-like architectures, and more tightly integrated heterogeneous processors, the traditional abstraction of a monolithic OS running on a SMP system is encountering new challenges. One proposed path forward is the multikernel operating system. Previous efforts have shown promising results both in scalability and in support for heterogeneity. However, one effort's source code is not freely available (FOS), and the other effort is not self-hosting and does not support a majority of existing applications (Barrelfish).

In this thesis, we present Popcorn, a Linux-based multikernel operating system. While Popcorn was a group effort, the boot layer code and the memory partitioning code are the authors work, and we present them in detail here. To our knowledge, we are the first to support multiple instances of the Linux kernel on a 64-bit x86 machine and to support more than 4 kernels running simultaneously.

We demonstrate that existing subsystems within Linux can be leveraged to meet the design goals of a multikernel OS. Taking this approach, we developed a fast inter-kernel network driver and messaging layer. We demonstrate that the network driver can share a 1 Gbit/s link without degraded performance and that in combination with guest kernels, it meets or exceeds the performance of SMP Linux with an event-based web server. We evaluate the messaging layer with microbenchmarks and conclude that it performs well given the limitations of current x86-64 hardware. Finally, we use the messaging layer to provide live process migration between cores.

This work is supported in part by US NSWC under Contract N00178-09-D-3017/0022.

Acknowledgments

Although I am incredibly grateful to all the people who have aided me throughout this endeavor, I would like to thank the following people specifically:

Dr. Binoy Ravindran, for priming my interest and guiding my path, along with Dr. Jules White and Dr. Paul Plassman.

My lovely wife, Sarah Eagle, for putting up with me despite my surliness and frustration.

Dr. Antonio Barbalace, for his invaluable technical input and for being a good friend.

Dr. Alastair Murray, Rob Lyerly, Shawn Furrow, Dave Katz, and the rest of the people on the Popcorn project, for their hard work and undeniable skill.

Kevin Burns, for his Linux wizardry.

Dr. Godmar Back, for making me ask the hard questions.

My welding torch, my soldering iron, and my CW paddles, for keeping me sane.

Contents

List of Figures

List of Tables

List of Acronyms

ACPI	Advanced Configuration and Power Interface
AP	Application Processor
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
BIOS	Basic Input/Output System
BP	Bootstrap Processor
ccNUMA	Cache-coherent, Non-Uniform Memory Access
CPU	Central Processing Unit
FPGA	Field-Programmable Gate Array
GDT	Global Descriptor Table
GPU	Graphics Processing Unit
GRUB	GRand Unified Bootloader, in Linux
HPC	High-Performance Computing (supercomputing)
HT	HyperTransport (from AMD)
IDT	Interrupt Descriptor Table
I/O APIC	Input/Output Advanced Programmable Interrupt Controller
IPI	Inter-Processor Interrupt
IPC	Inter-Process Communication
ISA	Instruction Set Architecture

ISR	Interrupt Service Routine
KVM	Kernel Virtual Machine, in Linux
MPI	Message-Passing Interface
MTU	Maximum Transmission Unit
NPB	NAS Parallel Benchmarks
NUMA	Non-Uniform Memory Access
OS	Operating System
PCI	Peripheral Component Interconnect
PFN	Page Frame Number
QEMU	Quick EMUlator, an multi-platform emulator
QPI	Quick Path Interconnect (from Intel)
RDTSC	Read Timestamp Counter (x86 instruction)
SMP	Symmetric MultiProcessing
SSH	Secure Shell (network terminal)
TFTP	Trivial File Transfer Protocol
TTY	Linux serial terminal (originally from TeleTYpe)
TUN/TAP	Linux network tunnel / network bridge
VM	Virtual Machine

Chapter 1

Introduction

The world of commodity computing has moved firmly into the multicore realm, but the traditional abstraction of a shared-memory machine running a monolithic SMP operating system remains nearly universal. Mainstream operating systems like Linux and Windows use this approach, where the same code runs on all the processors in the system and communication between cores at the OS level occurs implicitly through data structures in shared memory. Scalability optimizations within the Linux kernel have allowed it to provide good performance on today's highly multicore machines [9, 10]. In addition, there have been promising efforts to deal with the legacy baggage of state that is unnecessarily shared across the system by default, instead allowing the OS and applications to cooperate to manage sharing explicitly [8]. Finally, there are strong arguments that cache coherence protocols will be able to scale to even more highly multicore systems than we have today [39].

Nevertheless, there is reason to doubt whether this traditional approach can be further adapted to accommodate the challenges posed by forthcoming hardware and software. There are several significant drawbacks to this approach:

- While this approach has been made to scale quite well to high-core-count hardware, this scalability has come about as a result of a great deal of work. It has been observed that scalability in these OSes follows a cycle: when core count rises to a particular level, the kernel hits a scalability bottleneck; after testing and analysis, the root cause of the bottleneck is found and addressed; and the kernel performs adequately for a while until the next scalability bottleneck is reached [13, 5]. It would be an improvement to have an OS where scalability is dealt with directly in the fundamental design, avoiding these arbitrary issues along the way.

- Traditional operating systems are limited in their ability to leverage heterogeneous hardware.

  At the present time, heterogeneous pieces of hardware are usually treated as accelerators and are not fully integrated with the OS; the OS cannot schedule threads or

processes directly on them, and this must be handled instead by their drivers and by individual applications. This approach has worked well for hardware such as GPUs and FPGAs that is not capable of running an operating system or performing its own system management tasks. However, in the past few years, highly-multicore heterogeneous accelerators such as Intel's Xeon Phi [26] and Tilera's TILE-Gx [16] have been introduced, which do have the ability/need to run an OS. Some of these share an ISA with the general-purpose CPU (e.g. Xeon Phi), and some do not (e.g. Tilera).

In addition, there is reason to believe that future generations of heterogeneous hardware will be more tightly integrated, and that there will be a need for an OS design that can take advantage of this integration. An example of such a platform is ARM's big.LITTLE [49], a single-ISA heterogeneous chip that provides both a high-power core for compute-intensive workloads and a low-power core for less-compute-intensive workloads.

In 2009, researchers at ETH Zurich, in conjunction with Microsoft Research, introduced the idea of a *multikernel operating system*, which they define as one built according to a set of three design principles [5]:

- Make all inter-core communication explicit.
- Make OS structure hardware-neutral.
- View state as replicated instead of shared.

This idea addresses both of the drawbacks discussed above. A multikernel OS treats a multicore, potentially heterogeneous system as though it were a 'distributed system in a box', with shared state kept coherent via explicit message passing. Message-passing allows for a common interface between heterogeneous cores that may not share the same instruction set, and making inter-core communication explicit boosts scalability by eliminating unnecessary sharing of resources between cores. Early efforts in this area have shown promise, but as ground-up designs, they are also limited by their user and developer communities, as detailed in Section 1.1.

In this thesis, we introduce Popcorn, a multikernel operating system based on the Linux kernel. The motivation behind the project is to deliver the benefits of a multikernel OS while still providing the comprehensive environment and strong user and developer communities of Linux.

1.1 Limitations of Past Work

At present time, there are two actively-developed multikernel operating systems: FOS and Barrelfish. Both of these would be classified as research operating systems, which lack the

robust application support, developer community, and installed user base of Linux.

Neither source code nor binaries for FOS are openly available, so it does not constitute a good platform for development and evaluation of multikernel ideas.

While the source code for Barrelfish is available [56], the system presents significant challenges for those wanting to use it to do productive work. For example, to run an OpenMP application on Barrelfish, the application must be compiled on a separate machine, since the OS is not self-hosting. At that point, the OS and the application are loaded onto a TFTP server, the system is booted from the network, and the application to be run is specified through bootloader arguments – due to an issue with ACPI and PCI support on our system, the *fish* command-line shell does not run. Also as a result of this issue, for each subsequent application run, the machine must be hard-rebooted, which takes about five minutes on our 64-core machines.

In addition, our tests on Barrelfish showed that on benchmarks that should be entirely compute-bound, scalability was limited by OS overheads including those of remote thread creation [46]. We see room for improvement in demonstrating a design that will scale to 48 or 64 cores, and hopefully much further.

1.2 Research Contributions

Our contributions include the following:

- We modify the Linux kernel to launch multiple kernel instances anywhere within the physical address space. To our knowledge, we are the first group to do this on a 64-bit machine and to support more than 4 kernel instances.
- We provide a fast and efficient network driver for sharing a hardware network interface between kernel instances.
- We provide an efficient inter-kernel messaging layer and demonstrate its performance.
- We use this messaging layer to provide process migration across kernel instances.
- We release our source code so that others might build upon our efforts.

1.3 Scope of Thesis

The main focus of this thesis is the low-level work needed to bring Popcorn to a usable state, and how this work integrates with the additional design elements of a multikernel OS.

Much of the interesting research work in this space is in the coordination of OS tasks between multiple kernels in order to provide the abstraction of a single system image to applications. The work described in this thesis serves as groundwork for this work to proceed, and these higher-level challenges have been thoroughly considered in the designs detailed here. While this work can be thoroughly evaluated at a low level with regard to throughput, latency, and scalability, and while we can demonstrate individual components of the system (e.g. process migration), this work cannot yet be evaluated in combination with the full system, as the system is not yet complete.

In addition, one of the key challenges moving forward is support for heterogeneity and for hardware that offers hardware message passing. While a port of Popcorn to the Tilera architecture is underway, and while we will briefly mention how we accounted for heterogeneity in our designs, building and testing Popcorn on heterogeneous hardware is outside the scope of this thesis.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

- Chapter 2 provides an overview of the related work in the areas of message passing, communication, and operating system design.
- Chapter 3 describes the basic architecture of Popcorn and the low-level modifications we made to the Linux kernel to enable multiple instances to be booted on the same machine.
- Chapter 4 describes the shared-memory network tunnel used to share a physical network interface between multiple kernels.
- Chapter 5 describes the design and implementation of Popcorn's kernel- and user-space messaging layer.
- Chapter 6 presents a thorough performance evaluation of the Popcorn system.
- Chapter 7 outlines some overall conclusions of this work.
- Chapter 8 offers suggestions for future work.

Chapter 2

Related Work

In this chapter, we will examine existing techniques for inter-core messaging on commodity SMP machines, and we will study how these techniques have been applied in several different multikernel operating systems.

2.1 Background

Current high-end commodity multicore servers, while distributed under the hood, operate using the shared memory programming model, and usually run a monolithic OS like Linux. When a particular core needs a service from the OS, it makes a syscall, switches from user to kernel mode, and the kernel is executed on the same core that requested the service. To support this model, many data structures are shared across all cores by default, since shared state must be visible to all cores.

Studies have found that with a few modifications, the current Linux kernel can scale well on the current generation of multicore machines [9, 10]. These modifications include scalable locks, *per_cpu* variables in Linux, and NUMA support within the kernel. However, there is reason to believe that as the number of cores in a single machine continues to rise, new scalability bottlenecks will be hit, not all of which will be able to be addressed within this conventional approach. In addition, heterogeneous computing has entered the landscape, with new resources like GPUs and FPGAs becoming increasingly widespread, and these new resources do not integrate well into traditional monolithic OSes. These developments have led researchers to explore new design strategies for operating systems for forthcoming multicore machines.

2.2 Messaging and Notification

Messaging has been a well-studied part of computing at both the application and the OS level.

Traditionally, messaging at the application level has been supported through MPI (Message Passing Interface), a standardized API that has been available since the early 1990s [33]. MPI is supported over a wide variety of transports, including shared memory between cores in a multicore machine, TCP/IP networking between nodes in a cluster, and purpose-built high-performance interconnects such as Infiniband in an HPC setting.

In the microkernel approach to OS design, OS services run as processes, and applications access these services through inter-process communication (IPC), so an OS-level IPC layer is necessary, often using messages. Distributed operating systems use messaging to maintain coherent state across many interconnected nodes. More recently, OS-level messaging has been named as one of the fundamental parts of a multikernel OS [5].

From a design perspective, it is important to make a distinction between messaging and notification. Although both are required for a message-passing system to operate, the hardware primitives supporting each are in some cases orthogonal. Messaging refers to the transfer of a block of data from CPU A to CPU B. Notification refers to the mechanism that informs CPU B when a message has arrived, when scheduling is required, or when some other type of waypoint has been reached.

In this section, we discuss both messaging and notification on commodity x86-64 multicore machines. In addition, we discuss additional hardware support that is available in new architectures like Intel's SCC and Tilera's Tile-GX, and we address proposed additions to the x86 architecture to support new OS designs.

2.2.1 Messaging on Commodity Multicore Machines

Commodity x86 multicore machines do not have hardware support for message passing. As a consequence, the most basic way to pass messages between CPUs on these machines is through a shared memory window: the sender copies the message to an address within the window, and the receiver copies the message from the window to a local buffer.

Copy-In / Copy-Out

There is a fundamental source of inefficiency in this approach, which is known as the *copy-in, copy-out* problem. For each message, two memory copies are necessary: one from the sender into the buffer, and one from the receiver out of the buffer. Ideally, only one copy would be required.

An example of how this problem has been addressed is KNEM, an extension for the MPICH2-Nemesis MPI runtime. The basic idea is that the receiver process preallocates a local buffer for the message and then makes a syscall to the kernel with the virtual address of the buffer [37]. The kernel sets up a shared mapping for the buffer between the sender and receiver processes. When the sender sends the message, it is copied directly into the receiver's local buffer rather than through the shared memory window. For large messages, the overhead to enter kernel mode and set up the shared mapping is less than the overhead of the additional memcpy; the authors show significant performance improvements for messages larger than 10 KB [19].

Another way this problem has been addressed is page-flipping, in which the sender writes to a memory page and some mechanism (kernel or hypervisor) adjusts the pagetables so the page becomes present in the receiver's virtual address space. The Xen hypervisor uses this approach to provide fast networking between virtual machines [14].

Cache Optimizations

One of the primary issues affecting the performance of message-passing programs is cache behavior. In the ideal case, a message arrives and is processed by the receiver while it is still warm-cache. In reality, the usual MPI practice is to send the message as soon as it is ready, and if the recipient is not yet ready to process it, the message may be evicted from cache before it is processed, leading to a performance penalty when it is finally read.

In [42], Pellegrini et al present an approach for automated code-level refactoring of MPI code to relocate the send and receive calls to get closer to the ideal case where the delay between message reception and processing is minimized. The authors demonstrate significant performance improvements in real-world MPI applications. These sorts of optimizations should work equally well for MPI programs running under a multikernel OS as they do on a traditional OS, and they merit consideration when writing OS-level messaging code.

2.2.2 Notification on Commodity Machines

In this subsection, we discuss the three major primitives for notification available on commodity x86-64 SMP machines today: polling, inter-processor interrupts, and the MONITOR/MWAIT instructions.

Polling

Polling, or spinning, refers to checking a value in memory repeatedly until some expected action occurs – a counter is incremented, a pointer is moved, or something of the sort.

The advantage of polling is that the application gets the lowest possible latency from the hardware – as soon as the update message from the cache coherence protocol reaches the CPU that is polling, the CPU will fall out of the polling loop and continue execution. In addition, the application gets the greatest possible throughput – if the CPU finishes processing one message and polls for another, and the other message has already arrived, it can immediately begin work rather than waiting for notification. In addition, polling can occur in userspace only without requiring that the kernel be involved, eliminating the overheads that introduces.

The obvious disadvantage of polling is that while the CPU is spinning, it is wasting cycles when it could be doing something else. In some situations, this penalty doesn't matter – in an MPI application in which each core executes a single-threaded process, the core sits idle anyway while waiting for a message, so polling only costs more in terms of power consumption. If the application is well-written and properly load-balanced, wait times will be minimal, so this penalty is low compared to the improved latency. Shared-memory MPI implementations like MPICH2-Nemesis operate in this matter [12].

Inter-Processor Interrupts

Inter-processor interrupts are used on x86-based SMP systems to perform synchronization between processors [27]. They go over the APIC/LAPIC infrastructure between CPUs, which on modern machines is likely to go over the same message-passing infrastructure as the cache coherence protocol. As an example, with Intel's Quick Path Interconnect (QPI), interrupts go over the protocol layer [24].

Within the Linux kernel's SMP implementation, IPIs are used for several reasons:

- Coordinating system management operations such as shutdown and restart.
- Coordinating scheduling – when one CPU schedules, it sends an IPI to other CPUs whose runqueues may have changed.
- TLB shootdown – In a process with multiple threads running on multiple CPUs, when one thread updates a virtual-to-physical memory mapping, its CPU must broadcast an IPI to all the other CPUs running threads of that process to tell them to invalidate the TLB entry for that mapping. In Linux, this code is found in *mm/tlb.c*.The Barrelfish paper introduces an optimized message-passing based method of performing TLB shootdown that scales better than IPIs on large-core-count machines [5].

The main advantage of IPI is that it does not require the remote core to spin, so it can do useful work while waiting.

There are several disadvantages to IPI. First, while sending IPI carries a relatively small overhead, IPI can only be sent from kernel space, so sending an IPI from a user process

would require a syscall into the kernel, incurring the overhead of the mode switch and any resulting cache pollution. Second, receiving IPI carries a significant overhead in transitioning to and from user mode and executing the interrupt handler, plus whatever cache pollution may occur as a result.

Monitor/Mwait

The monitor/mwait instructions were introduced to the Intel x86-64 architecture as part of the SSE3 extensions [27]. The basic idea is to allow a core to be put to sleep until a write to a particular memory address occurs. The *monitor* instruction sets the memory address on which to wait, and the *wait* instruction waits for a write to that address. Under the hood, these instructions work by interacting with the cache coherence protocol; although the exact mechanism is proprietary, it has been hypothesized that when the cache line being monitored moves into the invalidated state, the core is awakened [18].

[3] demonstrates a practical approach to performing notification with monitor/mwait. The authors show better performance with this approach than with a polling-based approach on the same hardware. The performance improvement in this case comes due to the sharing of pipeline stages between each pair of cores in Intel HyperThreading. When one core is spinning, it is consuming resources that the other core could be using; using *monitor/mwait* to put the core to sleep frees up those resources for the other core.

Outside this special case, the main use of *monitor/mwait* is to allow cores that are spinning to enter a sleep state, reducing power consumption. As an example, the Remote Core Locking paper [36] introduces a power-efficient version of their algorithm that uses *monitor/mwait* instead of spinning; the authors claim that this version "introduces a latency overhead of less than 30%" compared to the polling-based version, but consumes less power.

2.2.3 Hardware Extensions

Commodity multicore hardware does not yet support explicit hardware message passing, but many works in the literature reason that this support is likely forthcoming [7].

The Intel Single-Chip Cloud research processor (SCC) provides strong support for hardware message passing [21].

The Tilera Tile64 processor supports hardware message passing in userspace [54]. Tilera provides iLib, a C-based library for programmers to interact with the message passing hardware in a familiar manner similar to sockets.

In [40], the authors of the Barrelfish research OS discuss desired features in future architectures to support OSes moving forward, specifically lightweight inter-core messages and notifications.

2.3 Multikernel OSes and Related Efforts

It is useful to think of the spectrum of operating systems as a gradient from general-purpose monolithic to special-purpose distributed. This idea is illustrated in Figure 2.1.

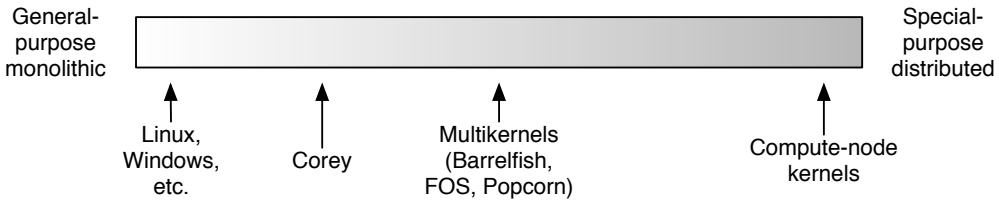

Figure 2.1: Gradient of multicore operating systems

On the far left would be existing operating systems like Linux and Windows that run as a single instance and are expected to be able to run a large variety of workloads.

To the right of these would be operating systems that are not fully distributed at the OS level, but that provide special support for scalability or reliability on multicore machines. Corey, a version of Linux that we will discuss in this section, falls into this category.

In the middle would be systems that are distributed at the OS level on a single machine, but that provide a single system image and as such can support applications based on existing shared-memory programming models. These systems fall under the umbrella of multikernel operating systems and include OSes like Hive, Barrelfish, and FOS, which we will discuss in this section. Note that Popcorn will fall into this category after the work to support a single system image across kernel instances is complete.

On the far right would be fully-distributed compute node kernels that are designed to provide the minimal services necessary to run a particular MPI-based high-performance computing application as fast as possible. As such, these OSes do not provide a single system image. We will discuss these systems briefly, covering only those features that are relevant to Popcorn.

2.3.1 Barrelfish

Barrelfish, a collaboration between ETH Zurich and Microsoft Research, introduced the idea of a multikernel operating system [5]. As described in Chapter 1, the authors define a multikernel OS as one in which inter-core communication is explicit, OS structure is hardware-neutral, and OS state is replicated rather than shared. The authors argue that the motivation behind such an approach is to make the OS more closely match the hardware in current and future multicore machines, which would produce payoffs in performance (including less need for tuning on new hardware) and in support for heterogeneity.

At the same time, the authors propose that such an OS, while being fully distributed under the hood, should still be able to provide much of the programming model that programmers are used to on SMP machines. To demonstrate this, Barrelfish provides an implementation of the OpenMP shared-memory threading library using remote thread creation and shows respectable, if not competitive, performance.

Barrelfish has an innovative approach to messaging on commodity multicore machines. Each message takes up one cache line (64 bytes on the x86-64 architecture) and carries a sequence number in the last few bytes of the cache line. The sender writes the message to a shared memory location, and the receiver polls on the sequence number. When the receiver sees the expected next sequence number, the entire message has arrived, and the receiver falls out of the polling loop.

Barrelfish couples this approach with a hybrid notification method using both IPI and polling. Their strategy is to poll first for some span of time, and then fall back to IPI if no message is received. The authors show mathematically that setting the polling interval to the expected time it would take to service an IPI provides a good starting point for this solution.

On the Intel Single-Chip Cloud research processor (SCC), Barrelfish takes advantage of the messaging and notification primitives that exist in hardware [43].

Barrelfish has proven fertile ground for further multikernel research: further efforts have given it hotpluggable USB support [50] and a Java virtual machine [38].

Work is currently underway to leverage the Drawbridge project [44], which provides Windows 7 as a library OS, to support commodity applications under Barrelfish [4].

2.3.2 Factored Operating System

In 2009, Wentzlaff et al from MIT introduced FOS [53], a factored operating system for commodity multicores. The basic idea behind FOS is to run different OS services on specific cores and have userspace processes send messages to those cores to access these services, rather than running them separately on each core that needs them.

According to [7], the overhead of message passing is roughly equivalent to the overhead of making a syscall into the OS, but FOS still achieves performance gains through improved cache behavior as a result of pinning certain system tasks to certain cores. This is similar to the idea presented in FlexSC [48], in which system calls are performed asynchronously through shared-memory message passing: the approach incurs the cost of messaging but avoids the costs of making a syscall, and gains improved cache behavior through batching of requests.

While Barrelfish messaging channels are allocated at build time for each application using RPC stubs created through a scripting language [5], FOS messaging channels are dynamically allocated at runtime by hashing an application-provided messaging channel ID (for example,

/sys/block-device-server/input). Each messaging channel is protected by a unique 64-bit "capability value" that each authorized process must provide before putting a message into the channel.

FOS provides support for both user messaging (through shared memory) and kernel messaging (through kernel-assisted copying into the remote process's heap). These pathways are used in a hybrid approach; the first messages sent over a given channel use kernel messaging, and if a certain number of messages are sent within a specified amount of time, the channel switches to user messaging.

In addition, the FOS messaging infrastructure supports distributed systems and allows communication with processors on remote nodes that looks the same to userspace processes as communication with processors on the same node.

2.3.3 Corey

Corey, developed at MIT, advances the argument that "applications should control sharing" [8]. The authors observe that shared state within the kernel is a bottleneck to scalability on high-core-count machines, and that Linux developers have had great success in improving scalability by minimizing the data shared between cores. However, they argue that further gains could be made if the OS knew at a fine-grained level whether a particular piece of information (e.g. file descriptor, virtual-to-physical memory mapping) was strictly local to one core or needed to be shared across cores. Corey functions more as a thin monolithic kernel (the MIT team terms this "exokernel") than as a multikernel, but shares with Barrelfish the goal of replicated, not shared, state between cores.

Corey uses three OS mechanisms to accomplish this goal:

- **Address ranges** – Shared-memory applications can create separate address ranges to hold their data structures, each of which can be private (mapped by the local core only) or shared (mapped by multiple cores). The programmer is responsible for indicating whether each data structure is private or shared.
- **Kernel cores** – Certain kernel functions, and their related data, can be dedicated to a single core.
- **Shares** – For operations that look up an identifier (e.g. a file descriptor) and return a pointer to some kernel data structure, applications can manage their own mapping tables, which default to local-only but can be easily expanded to span multiple cores. This approach avoids the unnecessary overhead of having these lookup tables shared globally by default. The programmer is responsible for creating these tables, although Corey provides primitives to make the task easier.

The authors show improved performance on TCP microbenchmarks plus benchmarks based on MapReduce and web server applications.

2.3.4 Hive

Hive uses a multikernel-like approach to provide fault containment on SMP machines [13]. Like Disco and Cellular Disco (discussed below), Hive was built for Stanford's FLASH SMP machine.

The main idea behind Hive is to split up the processors in a machine into groups called *cells*, each of which runs its own independent kernel. (Note that a cell is analogous to a cluster on Popcorn; see Section 3.3 for Popcorn nomenclature.) As in a multikernel, the cells cooperate to provide a single system image to user processes, although this support was not yet complete at the time of publication. In this approach, fault isolation comes from the fact that a fault in one cell is likely to be isolated to that cell and as such will not affect the other cells in the machine. Hive leverages the 'firewall hardware' present in the FLASH SMP machine, which allows a page to be made writable by only a certain subset of CPUs, to achieve memory isolation between cells.

While the main goal of Hive is the graceful handling of faults, the authors also note that Hive's distributed nature provides a "systematic approach to scalability". Like the Barrelfish authors a decade later, they argue that getting a traditional monolithic kernel to scale involves a "trial-and-error process of identifying and fixing bottlenecks", a process that is not necessary with a distributed approach.

Since the FLASH SMP machine was not ready in hardware at the time of publication, Hive was tested in a simulator, and it was able to achieve its fault containment goals when a range of faults were injected. A literature search returned no instances where Hive was benchmarked on physical hardware.

2.3.5 Osprey

Osprey, developed at Alcatel-Lucent Bell Labs, is an OS designed around the multikernel design principles to provide good performance on future multicore machines [45]. Osprey is not a strict multikernel: while the individual kernel-level data structures are partitioned between cores whenever possible, the global kernel space in Osprey is shared across all the cores in the system.

Like Barrelfish, Osprey uses messaging for communication between multiple processes. Each process also communicates with the kernel via two dedicated per-process messaging queues: user-to-kernel (U2K) and kernel-to-user (K2U). When the process enters kernel mode (e.g. through a syscall or interrupt, or when a special 'flush' syscall is performed), the kernel

processes these queues and provides the services requested, coordinating with other kernels if necessary. This approach is similar to Popcorn's multi-monolithic-kernel approach and may inform its future design.

Of special interest is Osprey's comprehensive messaging layer design, which draws from approaches throughout the literature. Whereas messaging in Barrelfish is event-based and does not block, the messaging framework within Osprey provides comprehensive scheduling to support both blocking and non-blocking messaging. This scheduling architecture allows for the use of different messaging channels and notification methods depending on the requirements of each application. Osprey supports notification via polling, IPI, and monitor/mwait; it supports messaging via both many-to-one and exclusive one-to-one queues. It also supports multi-hop messaging, where a message can be relayed by several cores' schedulers to its eventual destination. Finally, it can make optimizations based on the demands of each application. For example, if timeliness is not a concern for a particular application, the messaging layer can batch multiple messages together to reduce the total transmission cost.

Osprey also includes support for real-time applications. Each core's scheduler maintains its own queue of real-time tasks sorted by deadline in order to implement the Earliest Deadline First (EDF) scheduling algorithm. As in Linux, all real-time tasks take priority over all non-real-time tasks.

Osprey has been implemented for both the 32-bit and 64-bit x86 architectures, but the authors have not yet presented performance results.

2.3.6 The Clustered Multikernel

In "The Clustered Multikernel", von Tessin marries the idea of the multikernel with concepts from the formal verification world [51]. According to the author, formal verification is extremely difficult, and the largest formally-verified kernel is seL4, with 8700 lines of C source code. As a result, previous efforts have avoided dealing with concurrency. von Tessin introduces a 'lifting framework' that uses an already-verified microkernel as the basis of a multikernel OS in such a way that the proofs for the microkernel can be reused with 'relatively low effort'. One issue with the approach is the use of a 'big lock' to protect shared state in each cluster; the authors argue that this lock will scale better on modern tightly-coupled multicores than it did with Linux's Big Kernel Lock (BKL), although they show no performance results, and this premise may be somewhat dubious.

2.3.7 Virtualization-Based Approaches

Disco / Cellular Disco

Disco [11] uses virtualization to run multiple copies of IRIX on the same SMP machine. The paper focuses mostly on the design decisions made to support virtualization on Stanford's FLASH SMP machine and to modify the IRIX OS to run under this environment. However, of interest to us in this paper is the authors' finding that running parallel applications under multiple instances of IRIX produced significant performance improvements in applications that place a high load on the OS. Note that since Disco was written, commodity OSes like IRIX and Linux have been refactored to run well on SMP machines, so much of the lock contention and undesired cache behavior that the authors saw in the version of IRIX they tested (IRIX 5.3) would not be seen now. For example, they note that IRIX 5.3 has a single spinlock that protects the memory management data structures and becomes highly contended with multiple CPUs. This lock is similar to the Big Kernel Lock (BKL) in Linux, which was removed in kernel 2.6.37 in 2010.

Cellular Disco [20] builds on Disco, adding resource management and fault tolerance, and further advancing the authors' argument that providing better SMP/NUMA support through virtualization is less difficult, and only slightly less efficient, than doing it at the OS level.

MPI-Nahanni

Nahanni, or *ivshmem* (inter-virtual machine shared memory), is an extension for the Linux kernel virtual machine (KVM) that provides a shared memory window between virtual machines running on the same machine. It does so by providing a virtual PCI device to each virtual machine whose base-address register (BAR) points to the physical address of the shared memory window. After two machines have mapped the window, they can both read and write to it at native speed after the initial cache misses.

MPI-Nahanni builds on Nahanni to run MPI applications on clusters of virtual machines [30]. This gives the user the isolation benefits of VMs and the ability to run on clusters as could be rented from Amazon EC2, but introduces additional costs to support virtualization. Note that MPI-Nahanni is not a full OS – the Linux kernel is not modified, and it provides MPI support through an optimized version of the MPICH2-Nemesis shared-memory MPI runtime.

NoHype

NoHype uses the hardware support for virtualization in modern variants of the x86-64 architecture to run multiple instances of Linux on the same machine without using a hypervisor [31]. The authors cite as motivation the fact that bugs in the hypervisor can be exploited to

gain control of the guest operating systems, so removing the hypervisor removes a potential attack vector. Memory is statically partitioned by using hardware support for Extended Page Tables (EPT) to give each guest the illusion of its own physical address space. Network connectivity is provided by SR-IOV (server resource I/O virtualization), where a single Ethernet card provides each guest OS its own hardware-virtualized PCI device. The authors do not address communication or coordination between guest OSes, since the goal of their work is isolation for purposes of security.

2.3.8 Linux-Based Approaches

In a thorough literature search, we found three projects that have previously booted and run multiple instances of the Linux kernel on the same machine without virtualization. Note that these projects ran only on 32-bit x86 CPUs, all resources were statically partitioned, and none of the projects have released source code. In addition, we found one project that has run Linux and Windows concurrently on the same machine without virtualization.

Twin Linux

The Twin Linux project modified GRUB to boot two independent Linux kernels on a dual-core processor [29]. The kernels are able to communicate with one another through a shared memory region. The authors' primary motivation was heterogeneity at the OS level: they posited that one kernel might be optimized for server-class workloads and another kernel optimized for real-time workloads, and that both kernels could run on the same machine at the same time.

In the implementation shown in the paper, devices were statically partitioned such that one kernel handled the network interface and the other kernel handled the hard disk controller. The authors showed good results compared to SMP Linux when running a combined network and filesystem benchmark; they attributed the improvement in performance to reduced stress on the cache coherence protocol, although they did not show data to support this finding.

An additional weakness of Twin Linux is its approach to memory management. The authors statically mapped the 1 GB of physical RAM in the machine to between the 3 GB and 4 GB mark in the 32-bit virtual address space to allow for shared memory between the two kernels. This approach does not scale to machines with more memory or more cores, and is dangerous because each kernel can easily overwrite the other kernel's data structures, even those that should be private, without having to map them explicitly.

Linux Mint

Linux Mint is a project from Okayama University intended as a higher-performance alternative to virtualization [41]. The motivating goal of Mint is to allow multiple instances of Linux to run on the same machine with statically-partitioned resources, and for each instance of Linux to have performance equivalent to vanilla (unmodified) Linux. The authors explicitly address the idea of a multikernel operating system, identifying as a shortcoming the fact that it would "require users to abandon their existing software assets", something not required with virtualization or with Mint. We disagree with this assertion; there is no reason why a multikernel OS cannot have a layer to provide the hooks an application expects from Linux, requiring recompilation but not source-level modification. In fact, Barrelfish provides such a layer, called *posixcompat* [56].

In order to launch multiple kernels, Mint modified the SMP boot process within the Linux kernel, an approach we borrow for the boot process on Popcorn. Mint also supports partitioning of cores such that a kernel instance can have more than one core, what we term 'clustering' on Popcorn. Like Twin Linux, Mint adjusts the programming of the APIC/LAPIC to forward interrupts from devices to the hardware partition / Linux instance to which they are assigned, an approach we also take in Popcorn. Hardware is statically assigned to each kernel instance, and no virtual devices are provided to facilitate the sharing of physical devices between multiple Linux instances. This approach enables Linux instances to be be restarted or shut down at runtime without affecting the other running kernels.

The authors evaluated Mint on a four-core Intel CPU, and verified that each Linux instance provided performance roughly equivalent to a single kernel executing on a single core in terms of both I/O and CPU performance.

coLinux

coLinux (Cooperative Linux) is a project to allow Linux and Windows to be run alongside each other on the same machine without virtualization [2]. In this approach, the Windows host provides primitives to the Linux guest (memory allocation, networking, video/UI, and debugging), which the Linux guest accesses through a kernel-level driver. While coLinux functions more as a heterogeneous OS, in which Windows and Linux serve different purposes on the same machine, there are lessons that can be learned from how the Windows host virtualizes services for the Linux guest to use.

SHIMOS

SHIMOS [47] runs two Linux kernel instances on a 4-core x86 machine. The intended application is as a higher-performing alternative to virtualization; the authors claim that SHIMOS handles system calls at up to seven times the speed of Xen and compiles the Linux

kernel up to 1.35 times faster than Xen. SHIMOS uses a kernel module similar to *coreboot* to launch a secondary kernel while leaving the primary kernel running; this functionality allows secondary kernels to be booted in any order and restarted at any time.

As in Twin Linux, inter-kernel communication occurs through a dedicated shared memory window in the physical address space. SHIMOS provides memory allocation functions for reserving and releasing blocks of memory from this shared memory window. Like Popcorn, SHIMOS provides a shared-memory virtual network device to allow the secondary kernel to send and receive packets over the physical network interface. As in Popcorn's approach, the packet contents are copied from the Linux `sk_buff` structure on the sender side into a queue in shared memory, and then copied into a new `sk_buff` structure on the receiver side. In a similar manner, SHIMOS also provides a virtual block device.

The main disadvantage of SHIMOS is that there is no underlying messaging layer for handling operating system functions and providing a single system image across kernels; as such, it does not strictly qualify as a multikernel OS.

2.3.9 Compute Node Kernels

Compute-node kernels are a class of operating systems designed to provide very high performance for message-passing-based HPC applications. In particular, these OSes are highly concerned with minimizing noise and jitter. In highly-parallel systems with many thousands of cores, the effect of these factors becomes very important [6]. The design goals of these OSes differ significantly from those of a multikernel – while they run multiple kernels on a single piece of hardware, the goal is to provide bare-bones services at minimal overhead while maintaining strong performance isolation between kernels. That said, the resulting designs can look very similar to a multikernel, at least at a low level, so they are worth examining.

CNK is the lightweight compute-node kernel that runs on the compute nodes of the BlueGene/L supercomputer [1]. It is incredibly lightweight – it runs only one task at a time and does not support scheduling, I/O, or virtual memory. Similarly to how FOS dedicates cores to specific system tasks, CNK uses dedicated nodes to handle specific system services: in this case, I/O. I/O nodes run a separate Linux-based kernel called INK (I/O Node Kernel), and all compute nodes must go through the I/O nodes to perform I/O as CNK does not support it.

Compute Node Linux (CNL) is a compute-node kernel based on Linux that runs on the Cray XT series of supercomputers [52].

ZeptoOS is a similar effort to provide functionality equivalent to CNK on the BlueGene/P supercomputer using a stripped-down version of the Linux kernel [28].

2.4 Summary

While scalable locks and resource management within the Linux kernel have enabled it to scale to the current generation of multicores, increasing core counts and the introduction of heterogeneity have spurred new research into the design of future OSes. The multikernel operating system is one promising approach, along with systems based on fast message passing and new hardware to support them.

Chapter 3

Popcorn Architecture

3.1 Introduction

The existing Linux boot process on x86-64 is wholly intertwined with the history and quirks of the x86 architecture, so major modifications are necessary to support booting multiple kernels on the same machine. In this chapter, we will describe these modifications in detail.

3.2 Background

In this section, we detail architectural features specific to the x86-64 architecture that have a direct impact on Popcorn's boot process and overall design, and we examine how these features are supported in Linux.

Note that our analysis of the Linux kernel in this section and in the following sections is based on the source code to Linux 3.2.14, available for download on kernel.org [32]. Note also that our description of features of the x86 and x86-64 architectures is based on Intel's software developer's manual for these architectures, also available to view or download online [27].

3.2.1 Memory and Page Tables

Support on x86-64

The 32-bit x86 architecture (i386) provides a two-level paging system with a fixed page size of 4 KB that supports a 32-bit virtual and physical address space that can address up to 4 GB of physical memory.

In the Pentium Pro, Intel introduced support for 4 MB huge pages in addition to 4 KB normal pages with Page Size Extension (PSE). Also in the Pentium Pro, Intel introduced support for Physical Address Extension (PAE) to provide support for a 52-bit wide physical address space. With PAE enabled, the processor moves from a two-level paging system to a three-level paging system. PAE allowed 32-bit operating systems to address up to 64 GB of physical memory, though the virtual address space was still limited to 32 bits / 4 GB per process. Note that the huge page size is reduced from 4 MB to 2 MB if PAE is enabled.

The x86-64 architecture introduced long mode, described in more detail in Section 3.2.2, which retained the 52-bit wide physical address space from PAE but added support for a full 64-bit virtual address space for forwards compatibility. In addition, long mode added support for large pages of size 1 GB in addition to 4 KB normal pages and 2 MB huge pages.

To configure paging, the OS creates the initial pagetables and then stores the address of the page directory into a control register (CR3).

To accelerate virtualization, newer iterations of the x86-64 architecture support nested pagetables, called Extended Page Tables (EPT) on Intel and Rapid Virtualization Indexing (RVI) on AMD [17]. Rather than the hypervisor maintaining a set of shadow pagetables to handle translation from guest virtual addresses to host physical addresses, the hardware provides two levels of pagetables: one level to translate from guest virtual addresses to guest physical addresses, and a second level to translate from guest physical addresses to host physical addresses. The drawback in this case is that TLB misses cost twice as much to handle, since there are 8 levels of pagetables to iterate through rather than 4. This issue impacts performance not only on virtualization-based solutions like MPI-Nahanni, but also on NoHype, which is not virtualization in the traditional sense but which utilizes the hardware EPT support.

Support in Linux

The initial entry point to the Linux kernel's C code on the x86-64 architecture is `x86_64_start_kernel()` in `arch/x86/kernel/head64.c`, which performs some basic x86-specific initialization and then calls *start_kernel()* to initialize the platform-independent parts of the kernel.

Upon reaching `x86_64_start_kernel()`, the kernel requires several mappings to be set up in the pagetables:

- Low memory (by default, the first 1 GB of physical memory) must be identity-mapped. Identity mapping refers to setting up the pagetables such that over a particular range of pages, each page's virtual address is the same as its physical address, and vice versa.
- The virtual address `0xffffffff80000000` must be mapped to the 512 MB window containing the kernel. The pagetables for this mapping are created at build time within the kernel; initially, virtual address `0xffffffff80000000` is mapped to physical

address `0x0`, and the mapping is fixed up at boot time based on the physical address where the kernel was actually loaded.

After this point, the kernel becomes responsible for building and managing its own pagetables plus those of all the user processes running under it.

3.2.2 Real, Protected, and Long Modes

When an x86-64 processor is reset and begins executing code, it starts in *real mode*, a mode designed to provide backwards compatibility with the original 8086 processor. In this mode, there is no support for memory paging, and all code runs at a privileged level (there is no mode switching between user mode and kernel mode). Memory is addressed using a segmentation scheme that supports addresses that are 20 bits wide, allowing for 1 MB of memory to be addressed, so all code and data accessed in real mode must be located within the lowest 1 MB of physical RAM.

With the 286 processor, Intel introduced *protected mode*, which supports a 32-bit address space and switching between privileged (kernel) and non-privileged (user) mode. To enter protected mode, the OS sets a bit in a control register (CR0).

With the 386 processor, Intel added support for *paging* to enable the use of virtual memory. In protected mode, paging supports 32-bit virtual addresses mapped to 32-bit physical addresses. To enable paging, the OS must establish a set of page tables mapping virtual to physical addresses, set a control register (CR3) to point to the top-level directory of the page tables, and enable paging by writing to another control register.

The x86-64 architecture introduced *long mode* to support a full 64-bit virtual address space, though the physical address space is limited to 52-bit due to pagetable limitations. To enter long mode, the OS must load a set of 64-bit pagetables and a 64-bit GDT and then set a bit in a control register.

3.2.3 APIC/LAPIC and IPI

Support on x86

In an SMP system, there are several design goals with regards to interrupt handling that must be met:

- Interrupts must be routed properly from the device that generated them to the CPU that is responsible for servicing them.
- Every CPU must be able to send inter-processor interrupts (IPI) to every other CPU.

- The architecture for handling these tasks must be scalable to SMP systems with an increasingly high number of cores.

In the first x86 systems, which were strictly uniprocessor, interrupts were handled using an Intel 8259/8259A programmable interrupt controller (PIC) [22]. To provide unified support for SMP systems, Intel introduced the Advanced Programmable Interrupt Controller (APIC) specification [23]. This approach has been nearly universally adopted, and serves as the basis for interrupt handling on the x86-64 architecture on both Intel and AMD machines.

In this approach, interrupt handling is broken up into two levels. Each CPU has its own local APIC (LAPIC), which handles the interrupts for that particular CPU, including generation and handling of inter-processor interrupts (IPI). For each group of CPUs, there is an I/O APIC, which routes interrupts from devices to the appropriate LAPIC, and which routes IPIs from each source LAPIC to the specified destination APIC. For large-core-count SMP machines, there can be multiple I/O APICs in the same machine.

Support on Linux

Linux enumerates each CPU in the system with a logical CPU ID from 0 to *n-1*, where n is the number of CPUs in the system. However, to send an interrupt to a specific CPU, the I/O APIC must know the physical APIC ID of the destination LAPIC for that CPU, which is not the same as the logical CPU ID. Linux maintains a one-to-one mapping between each CPU's logical CPU ID and its physical LAPIC ID, and the I/O APICs are configured at boot time to follow this mapping.

For systems with 8 cores or fewer, Linux uses the *phys* APIC driver architecture. In this mode, since the logical and physical APIC IDs are the same, the system can perform "shortcut" operations where an IPI is sent to each core specified in a CPU mask in a single hardware operation. On larger SMP systems with more than 8 cores, Linux uses the *physflat* APIC driver architecture instead. In this mode, "shortcut" operations are not available, since the mapping from logical CPU ID to physical APIC ID may be discontinuous. Instead, the "send to mask" operations merely loop through each bit in the mask and perform a separate IPI send operation for each bit that is set.

To overcome this issue, the newest generation of Intel multicore processors supports the x2APIC standard, which provides a hierarchical, cluster-based approach to sending and acknowledging IPIs [25]. However, currently-available AMD hardware, including the hardware we used for developing and testing Popcorn, does not yet support the x2APIC standard.

3.2.4 SMP and Trampolines

In the normal Linux SMP boot process on the x86 architecture, the first CPU to boot is the bootstrap processor (BP). When the system is powered on, the BP begins execution of the BIOS. At some point, control is transferred to the bootloader (GRUB) and then to the Linux kernel. After performing basic initialization, the BP is ready to boot the rest of the CPUs in the system, termed application processors (AP). The BP can boot each AP by setting its initial instruction pointer and sending it an inter-processor interrupt (IPI) to wake it up. However, there is a challenge that needs to be addressed.

In order to enter the kernel code, the AP needs to be in long mode with the kernel's pagetables, global descriptor table (GDT), and interrupt descriptor table (IDT) loaded. However, when an AP is first reset, it starts out in real mode – it can only address the first 1 MB of memory and paging is not yet enabled. To be able to enter the kernel code, the AP needs to be transitioned from real mode to protected mode to long mode, paging needs to be enabled, and some additional initialization needs to be performed. In Linux, these actions take place in the *SMP trampoline*. The setup code for the trampoline is located in `arch/x86/kernel/trampoline.c`, and the assembly code for the trampoline itself is located in `arch/x86/kernel/trampoline_64.S`.

A *trampoline*, in the most general sense, is a piece of code that is used to jump to another piece of code. The idea is that the trampoline is called, does some basic initialization, and then "bounces" to some target piece of code. The SMP trampoline is responsible for transitioning the CPU from real mode to protected and then long mode; setting up the pagetables, interrupt descriptor table (IDT), and global descriptor table (GDT); and then "bouncing" into the kernel code itself.

Figure 3.1 shows a diagram of how this system operates, and more detailed descriptions of each step are as follows:

1. Initially, the trampoline resides within the kernel code, which begins at the 16 MB mark in physical memory by default. However, this is an unworkable location, since each AP starts in real mode and as such can only access the first 1 MB of physical memory. At boot time, the BP reserves a memory window within the first 1 MB of physical memory to hold the trampoline and copies the trampoline into this window.

2. To launch each AP, the BP sets its initial instruction pointer to point to the low-memory trampoline.

3. The BP sends the AP a startup IPI. The AP wakes up and begins executing the trampoline.

4. The AP executes the trampoline and enters the kernel code. At this point, the AP writes to shared memory to indicate that it is alive, executing the idle task, and ready to have tasks scheduled to it.

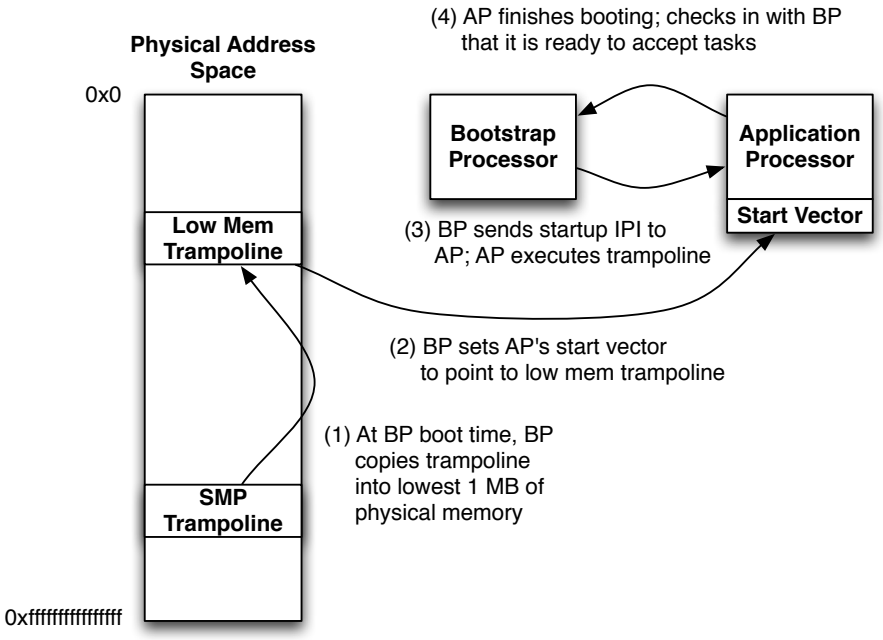

Figure 3.1: Unmodified Linux SMP boot process

3.2.5 Top Half / Bottom Half Interrupt Handling

When servicing interrupts, it is advantageous to minimize the amount of time spent in the interrupt service routine itself, since other system tasks are blocked during this time. In order to accomplish this, Linux provides a top half / bottom half architecture for handling interrupts. The *top half* is the ISR itself, which is executed in kernel mode when an interrupt is asserted, and the *bottom half* is a handler function that runs within the kernel and does the actual work of handling the interrupt. After performing whatever hardware operations are necessary to acknowledge the interrupt (e.g. `ack_APIC_irq()` for IPIs), the top half schedules the bottom half (e.g. with `__raise_softirq_irqoff()` for softirqs), which will run at a time that is more convenient for the kernel.

Linux provides three types of bottom half / top half APIs:

- *softirqs* – these are hardcoded into the kernel, can run on any core, and are used in unmodified Linux for tasks such as handling network devices. In Popcorn, we implement kernel-level messaging using softirqs.
- *tasklets* – these are similar to softirqs, but are more lightweight and can only run on the core that scheduled them.
- *workqueues* – these are similar to softirqs and tasklets, but run in a kernel thread, and thus support operations that can sleep. In Popcorn, we use workqueues to remap

physical memory between kernel instances; since `ioremap_cache()` can sleep, it cannot be called within a softirq or a tasklet.

These APIs are covered in more detail in [15] and [55].

3.3 Popcorn Nomenclature

Definitions on Linux

In unmodified Linux, all the processors in a machine execute a single monolithic *kernel*, which runs on a particular CPU whenever that CPU enters privileged mode, prompted by a syscall or by a device interrupt.

In the x86 SMP boot process on Linux, as discussed in Section 3.2.4, the processor that executes the BIOS and the bootloader and is first to enter the kernel code is termed the *bootstrap processor*, or BP. The remaining processors in the machine are booted via the SMP trampoline and are termed *application processors*, or APs.

Definitions on Popcorn

In Popcorn, all the CPUs run the same kernel code; we use the term *kernel* to refer to a particular version of Linux (e.g. Linux 3.2.14-popcorn).

We retain the Linux definition of the *bootstrap processor* (BSP) as the processor that executes the BIOS and the bootloader. With our hardware, the bootstrap processor is always CPU 0.

We define a *cluster* as a group of CPUs running the same copy of the kernel code. We use the term *kernel instance* to describe the copy of the kernel code running on a particular cluster, along with the associated state. The kernel instance running on the first cluster to boot is termed the *primary kernel instance*, and all subsequent kernel instances are termed *secondary kernel instances*.

Each cluster has a *cluster master*, which is the lowest-numbered CPU in the cluster, and the one that initially boots and enters the kernel code. The remaining CPUs in the cluster are termed *cluster workers*, which are booted by the cluster master using the SMP trampoline and share a kernel image and state with the other members of the cluster. In a cluster with only a single CPU, that CPU is the cluster master, and there are no cluster workers.

The nomenclature makes more sense if we examine it in the context of a feasible system configuration. We present one such configuration in Figure 3.2: an eight-CPU machine running two clustered kernel instances with four CPUs each.

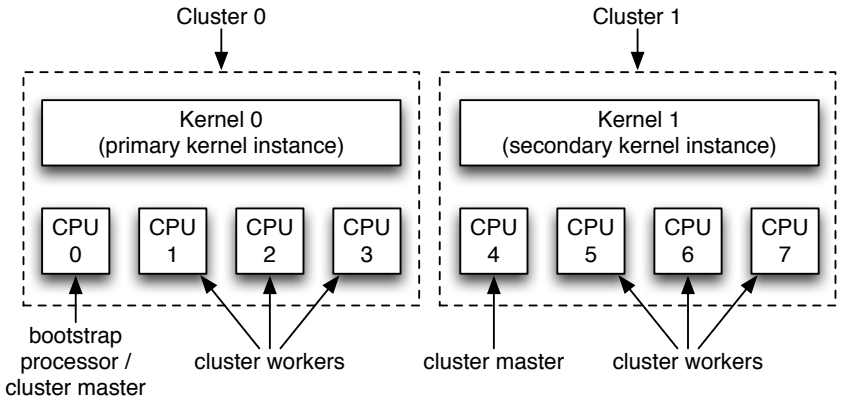

Figure 3.2: Sample system configuration to illustrate terms

In this configuration, Cluster 0 contains CPUs 0-3, and Cluster 1 contains CPUs 4-7. Kernel instance 0, the primary kernel instance, is running on the CPUs in Cluster 0, and kernel instance 1, the only secondary kernel instance in this configuration, is running on the CPUs in Cluster 1. The bootstrap processor is CPU 0, which is also the cluster master of Cluster 0. Cluster 1's cluster master is CPU 4, which is the first CPU in the cluster to be launched.

3.4 Launching Secondary Kernels

In this section, we detail our modifications to the Linux boot process to launch secondary kernels.

3.4.1 Design

We created an additional trampoline in low memory, the Multi-Kernel Boot Secondary Processor (MKBSP) trampoline, to launch the secondary kernel instances. This approach is similar to the one taken by Linux Mint [41], which also modified the SMP trampoline code to boot secondary kernel instances. Based on this design, we are able to boot and run Linux kernels located anywhere within the 64-bit address space.

An alternative approach, one taken by Twin Linux [29], would be to modify the bootloader to launch multiple kernel instances throughout the address space. However, our approach holds several advantages. First, modifications are limited to the kernel itself, whereas in the bootloader-based approach, both the bootloader and the kernel must be modified. Second, our approach allows for secondary kernel instances to be launched at any time after the boot kernel has loaded, whereas with the bootloader-based approach, all the kernels would have to be launched initially, and could not be relaunched dynamically.

3.4.2 Operation

Preparation and Setup

Before booting a secondary kernel instance, several items must be copied into place within the physical address space: the kernel, the ramdisk, and the kernel command line and `boot_params` structure. This is normally done by the BIOS and the bootloader, but must be done by the Popcorn boot code when launching a secondary kernel instance. We currently copy these items into place in physical memory through `/dev/mem`, the Linux device that maps all of physical memory to a file, though a more elegant approach would be to do it in kernel space through additional syscalls, or by fully exploiting the capabilities of *kexec*.

First, the kernel itself must be copied to the correct location. To handle this operation, we have adapted the *kexec* tool, which is already built to handle kernel images. Note that by default, the kernel is compressed, and must decompress itself at boot time with a small stub of code before beginning to run. In Popcorn, we bypass this decompression process. The kernel build process outputs *vmlinux*, an ELF-format binary of the kernel with debug symbols. We *objcopy* the code from this binary into a second ELF-formatted binary, *vmlinux.elf*, in order to strip out the debugging symbols and reduce the size. At this point, kexec is able to read in each segment within *vmlinux.elf* and copy it to the correct location within physical memory.

Second, the initial ramdisk (initrd) must be copied to the correct location, and the `struct boot_params` entry specifying its location to the kernel must be set. We built a simple `copy_ramdisk` program to handle this task.

Finally, the kernel boot arguments for the secondary kernel must be set. Boot arguments that may need to be passed to the secondary kernel are discussed in detail in Section 3.5.1. We built a simple `set_boot_args` program to perform this operation.

Secondary Kernel Boot Process

To launch a secondary kernel, the user passes the `-b` flag to *kexec*, which performs a syscall we created to boot a secondary CPU. The syscall sets the instruction pointer of the CPU to be launched to point to the MKBSP trampoline and then sends a startup IPI to start the CPU executing. At this point, the MKBSP trampoline begins executing with the secondary processor in real mode (16-bit segmented addressing, no paging). The boot process proceeds as shown in Figure 3.3. We highlight a few significant modifications below.

- We do not jump straight into the kernel from `startup_32_bsp`. Although this approach works if the kernel is loaded within the first 4 GB of RAM where it is accessible via a 32-bit jump, it does not work for entering kernels above the 4 GB mark. Hence, we add a portion of the trampoline that executes in 64-bit long mode, `startup_64_bsp`, from which we can make a long jump to anywhere within the full physical address space.

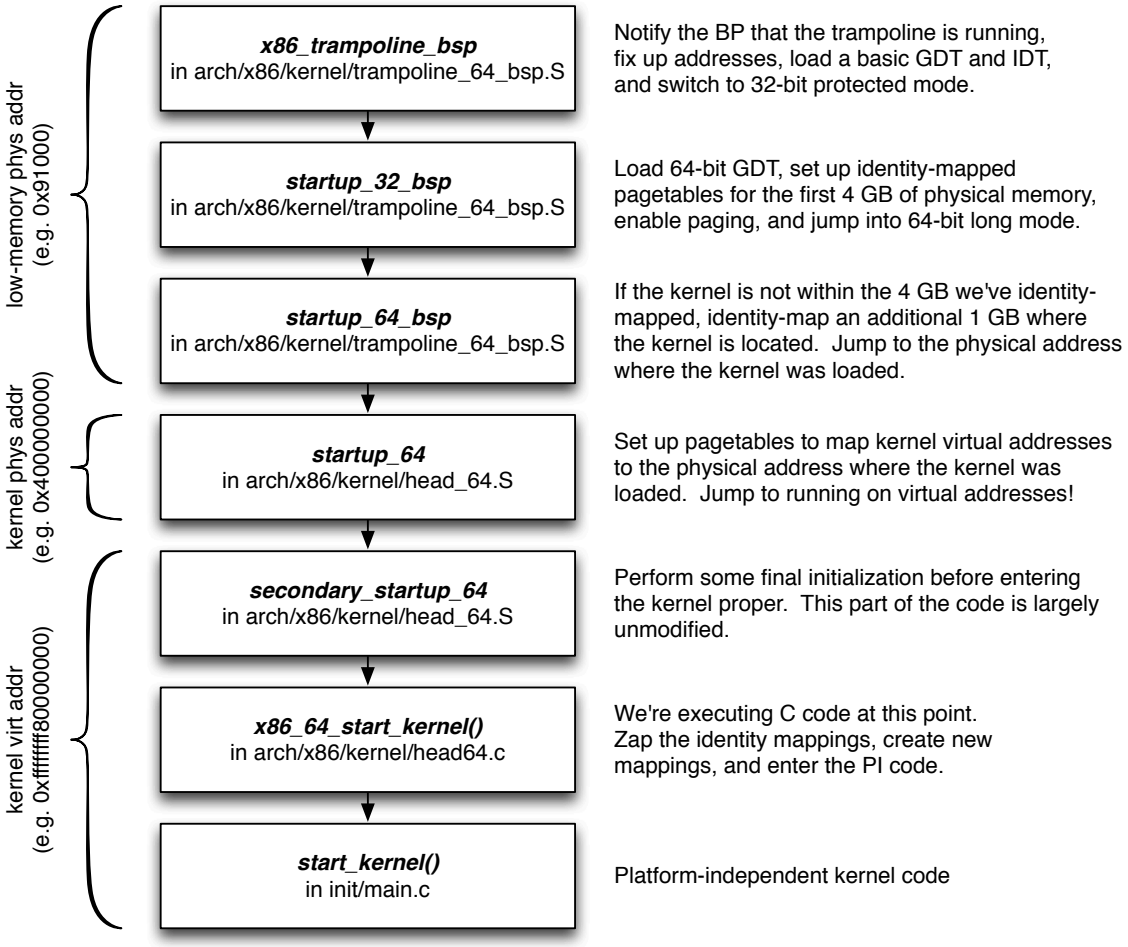

Figure 3.3: Popcorn secondary kernel boot process

- In `startup_64_bsp`, if the kernel is not within the lowest 4 GB of RAM that has been identity-mapped, we must identity-map an additional 1 GB window where the kernel was loaded. To do this, we fill up an 'extra' pagetable with the proper mappings and add it to the page directory.

- In `startup_64`, if the kernel was loaded outside the first 4 GB of physical memory, we need to create the pagetable mappings from kernel virtual address to physical address for the 1 GB window where the kernel was loaded. We do this by populating a spare level-2 pagetable with the appropriate mappings. This spare pagetable was included in the original kernel boot code, but the code to support it was not finished and did not work properly.

3.5 Kernel Modifications

In addition to the modifications to the boot procedure, several modifications are necessary to the kernel proper in order to support booting secondary kernels. In this section, we describe these modifications in detail.

Note that the per-CPU variable work described in section 3.5.4, the PCI device masking work described in Section 3.5.5, and the I/O APIC remapping work described in Section 3.5.6 were performed by Antonio Barbalace and are documented here for completeness; the remainder of the modifications were performed by the thesis author.

3.5.1 Kernel Command-Line Arguments

We added several new kernel command-line arguments to determine the behavior of the primary and secondary kernels:

- `mklinux` – this flag is set when booting as a secondary kernel.
- `present_mask=<list of CPUs>` – this flag is is used to specify which subset of the CPUs in the machine should be booted under a particular kernel instance, as described in Section 3.5.4.
- `pci_dev_flags=vendor0:device0:b,vendor1:device1:b...` – this flag is used to blacklist PCI devices as described in Section 3.5.5.
- `lapic_timer=<value>` – this flag is used to bypass the calibration procedure and pass the local APIC timer scaling value directly to each secondary kernel, as described in Section 3.5.6.

In addition, we make use of the existing `memmap` kernel argument to restrict each kernel to the appropriate partition within physical memory.

3.5.2 Redefining Low/High Memory

At boot time, Linux calculates the highest page frame number that it considers to be in "low RAM", or memory under the 4 GB mark in the physical address space. This is done so that certain data structures and I/O regions can be placed in physical memory that can be addressed by PCI devices whose base-address registers are only 32 bits wide. Since the kernels above the 4 GB mark currently do not need this PCI support, we can change the declaration of the highest low ram PFN to the highest PFN present in the machine.

3.5.3 Support for Ramdisks above the 4 GB Mark

Although the Linux setup code supports loading initial ramdisks from anywhere in the physical address space, the field for the ramdisk's physical address in the `struct boot_params` (the `ramdisk_image` field inside the `struct setup_header`) is only 32 bits wide, so there is no way to specify a ramdisk above the 4 GB mark.

To get around this problem, we added two additional fields to the `struct setup_header`: `ramdisk_shift`, a field containing bits 39-32 of the ramdisk physical address; and `ramdisk_magic`, an 8-bit value that is set to a specified magic number when the `ramdisk_shift` field has been written with a valid value. At boot time, if the magic number is set, the Linux setup code calculates the actual ramdisk physical address as `(ramdisk_shift << 32) + ramdisk_image`.

With these changes, we are able to support ramdisks up to the 1 TB mark in physical memory.

3.5.4 Support for Clustering and Per-CPU Variables

We provide a *present_mask* kernel command-line argument to specify at boot time which CPUs should be owned / booted by a particular kernel instance. Note that on the bootstrap processor (e.g. the processor brought up by the BIOS that initially launches the primary kernel), the *present_mask* must contain CPU 0.

A consistent logical CPU ID space is maintained across kernel instances – for example, 'CPU 2' refers to the same physical processor on all booted kernel instances. In unmodified Linux, when the kernel boots, it is assumed to start on logical CPU 0 regardless of the physical CPU ID of the bootstrap processor. In Popcorn, we adjust this mapping so that the bootstrap processor can be any logical CPU; on the primary kernel, the bootstrap processor is logical CPU 0, but on each secondary kernel, the bootstrap kernel is the *cluster master* CPU, which we define as the CPU indicated by the lowest-order bit set in the `present_mask`. The rest of the CPUs in a cluster are referred to as *cluster workers*.

Throughout the kernel code, Linux makes the assumption that CPU 0 is the bootstrap processor and all other CPUs are application processors, so changes were necessary in many places in the kernel, device drivers included, to support this modification.

Note that on each kernel instance, we reserve the full per_cpu data structures for each physical CPU in the machine (rather than for each CPU assigned to that kernel instance at boot time) in order to provide support for future dynamic remapping of CPUs to kernels.

3.5.5 PCI Device Masking

In our initial version of Popcorn, we statically partition hardware between the boot kernel and each secondary kernel. The boot kernel should not be given access to any hardware that will be reserved for the secondary kernels to use (such as a secondary network card, a GPU, an FPGA, or a serial card), and the secondary kernels should not be given access to any hardware that is owned by the boot kernel (usually, the SATA controller, the USB controller, the primary network device, and the primary graphics device).

To handle this partitioning, we modified the PCI discovery process. Each PCI device in a system is identified in the PCI configuration space by a vendor ID and a device ID. At discovery time, the kernel uses this information for each device in the system to load the appropriate drivers and to call the appropriate functions to initialize each device. For both boot and secondary kernels, we added the `pci_dev_flags` kernel argument to allow a set of blacklisted *(vendor ID, device ID)* pairs to be specified. During the PCI discovery process, if a device is found whose vendor ID and device ID match one of the entries in the blacklist, the kernel skips initialization of the device and continues to the next PCI device found, if any.

3.5.6 APIC Modifications

Device interrupt forwarding

Interrupts from each device need to be forwarded to the CPU that owns the device. At present time, device interrupt forwarding is set up statically at boot, with interrupts from all devices forwarded by default to the primary kernel. We leave it as future work to handle more fine-grained device partitioning or to make device assignment dynamic.

Remapping for Secondary Kernels

During development of the shared memory network driver described in Chapter 4, we encountered an issue where IPIs were sent successfully from CPU 0 to CPU 2, but IPIs from CPU 2 to CPU 0 were not received. We discovered that the CPU IDs used to send IPIs needed logical-to-physical translation on secondary kernels; otherwise, each secondary kernel would assign itself an ID of 0 and map the rest of the APIC IDs accordingly, resulting in an incorrect mapping.

Local APIC Timer

When the Linux kernel boots, it initializes the APIC infrastructure, including performing a calibration procedure involving a timed loop to determine the appropriate scaling factor for the local APIC timers. To time the loop, the kernel instructs the I/O APIC to generate an interrupt at a defined interval and counts the number of loop iterations before the interrupt arrives. In Popcorn, this calibration procedure works reliably on the boot kernel; however, if the secondary kernel attempts to perform this procedure, it hangs waiting for an interrupt that never arrives, since the I/O APIC has already been remapped.

To bypass this calibration procedure on secondary kernels, we created a `lapic_is_bsp()` function that returns true if the currently-executing CPU is the primary kernel's bootstrap processor (e.g. CPU 0) and false otherwise. If the function returns true, we perform I/O APIC and LAPIC initialization; otherwise, we skip the initialization process. In the second case, we provide a `lapic_timer` kernel argument to allow the scaling factor determined by the calibration process on the boot kernel to be passed directly to the secondary kernels.

3.5.7 Send Single IPI

On systems with 8 cores or fewer, the `send_IPI_mask()` function is an efficient method for both broadcast and unicast, since it can send IPI to any subset of the cores in a single step. On systems with more than 8 cores, however, the `send_IPI_mask()` function actually loops through each bit in the mask, checks if it is set, and if so, sends an IPI to the corresponding core. On a 48-core or 64-core machine, the overhead of looping through each bit becomes significant, especially when the IPI is to be sent only to a single CPU.

To achieve a gain in efficiency, we implemented a `send_IPI_single()` function, which sends an IPI to a single CPU only. Microbenchmark results demonstrating the impact of this change are shown in Section 6.2.

3.5.8 Kernel boot_params

During the boot process, the `struct boot_params` structure is copied from low memory into the kernel's virtual address space. However, when launching secondary kernels, it is necessary to access the original `struct boot_params` in order to set kernel parameters that may differ from those passed to the master kernel. To accommodate this need, we cache the physical address of the original `struct boot_params` so that it can be used when needed.

Chapter 4

Shared Memory Network Driver

A large portion of the work in the following chapter was done as the final project for CS 5204, a graduate-level operating systems class taught by Dr. Godmar Back. Dr. Back provided feedback and assistance throughout, particularly in two areas: first, in determining how to evaluate the performance of the network driver; and second, in finding the root cause of a technical problem that led to erratic performance. This work is included in the thesis with permission of the instructor.

4.1 Introduction / Motivations

Providing a fast and reliable networking layer is crucial in a multikernel OS for several reasons:

- The network connection serves as an out-of-band channel for communication between kernel instances. It allows command-line access via SSH and file transfer via SCP. Moreover, we used it to launch MPI processes remotely when launching an application.
- Network connectivity is necessary to support commonly-used Web server applications such as *apache* and *nginx*.
- Networking provides a low-level communications layer on top of which other services can run. Specifically, in a multikernel OS, it might make sense to use, or to build on, an existing network file system, due to shared design goals. As with multikernel file systems, network file systems must provide the abstraction of a single shared file system and maintain coherent state across disparate nodes that communicate only via messages.

4.2 Design Decisions

We decided to use a shared-memory ring buffer to handle communication between kernels. We model our approach after the approach taken by Disruptor [35], an efficient ring buffer library designed for use in low-latency, high-throughput financial applications. With an appropriately-sized ring buffer, it is possible to achieve high concurrency, minimal waiting, and good cache behavior.

We decided to place the shared memory network driver within the kernel, rather than as a userspace process, as a consequence of the way Linux handles networking. The main overhead in this approach is the transition from user mode to kernel mode and back, plus the cache pollution that occurs when the process's working set is evicted from the cache by the kernel code. However, the majority of networking within Linux is already handled within the kernel (e.g. servicing sockets, handling protocols like ICMP), so kernel mode would have to be entered anyway.

Network packets sent between two worker kernels on the same machine must go in two hops: one from the sender to the master kernel, and one from the master kernel to the receiver. This decision was made in order to allow each shared memory ring buffer to have a single producer and a single consumer, while limiting the memory usage to $O(n)$, where n is the number of kernels. In addition, it makes sense given the goal of sharing a hardware interface owned by the primary kernel instance with all of the secondary kernel instances; network communication between the secondary kernel instances comes as a bonus.

4.3 Implementation

4.3.1 Summary of Approach

The implementation of the network driver is based on the Linux network tunnel code in `drivers/net/tun.c`, and is now located in `drivers/net/shmtun.c`. This code provides a tunnel interface where one side of the tunnel is a network device and the other side of the tunnel is a file. Its operation is shown in Figure 4.1. Once created, each tunnel operates in an event-based manner – when a packet arrives at the network device, it is written to a file for a userspace application to read, and when a userspace application writes to the file, a packet is generated and sent out over the network device.

In our approach, we replace the file side of the tunnel with a shared-memory ring buffer, and rather than using events generated by file operations as triggers for network operations, we use IPI-based notification. This approach is shown in Figure 4.2.

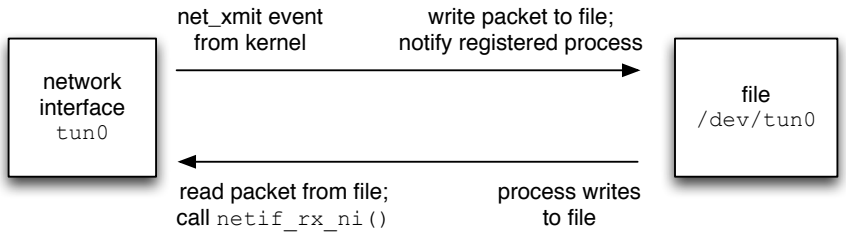

Figure 4.1: Event-based operation of the Linux TUN/TAP network tunnel

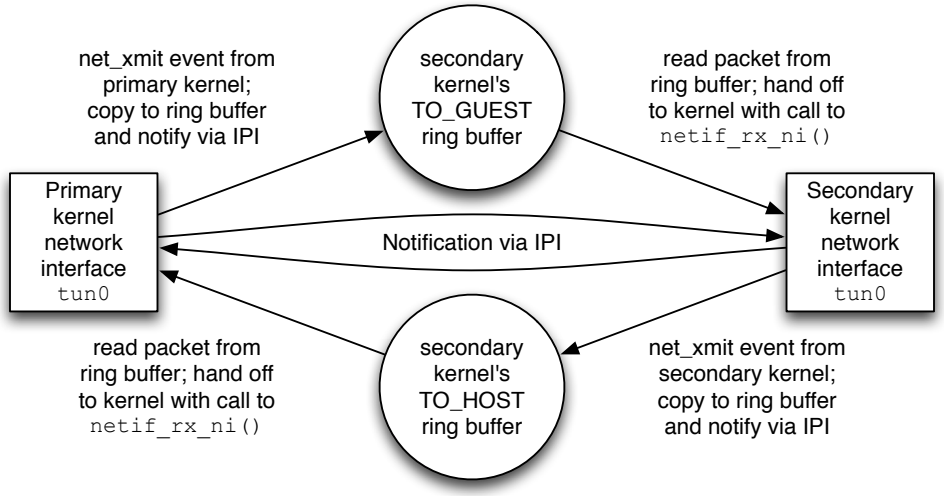

Figure 4.2: Operation of the Popcorn shared-memory network tunnel

4.3.2 Setup and Check-In

Data Structures

A global directory, allocated within the first kernel's physical address space, contains the following:

- The head and tail pointers for each secondary kernel's buffer – these are adjacent to each other in memory for cache purposes, since Kernel 0 must go through them in sequence when pulling out the packets.
- An array of the physical address for each kernel's per-cpu data
- An array of *int_enabled* values, which indicate whether receive interrupts are enabled for a particular kernel.

Each kernel has its own per-cpu data containing two ring buffers – one in the *TO_GUEST* direction and the other in the *TO_HOST* direction.

Check-In Process

Check-in relies on the kernel messaging layer described in Chapter 5.

When the host kernel boots, it kmallocs and initializes the global directory and places its physical address in the `struct boot_params`.

Each secondary kernel must map the global directory, and then kmalloc its own per-cpu buffer and put its physical address in the global directory. In addition, it must notify kernel 0 that it has booted, so kernel 0 can map its per-cpu buffer. This operation is performed using the kernel messaging layer: the secondary kernel sends a message to the primary kernel, the callback function on the primary kernel queues a task in a work queue, and the per-cpu buffer is mapped in a kernel thread. This is required because the `ioremap_cache` function can sleep, so it cannot be run in an interrupt bottom half.

4.3.3 Interrupt Mitigation

For high-bandwidth network interfaces, if one interrupt is generated per packet received, receive interrupts can be generated at a considerable rate. Servicing these interrupts takes away from the cycles that the system has left to handle the incoming packets, to run userspace applications, and to perform other system tasks. However, this process can be optimized; there is no need to send interrupts when the kernel handling the network device is already processing packets.

In our implementation, we leveraged the NAPI network driver infrastructure already present within the kernel. This infrastructure is designed to mitigate the number of interrupts generated by high-speed (1 Gbps or greater) hardware network interfaces. The high-level operation of NAPI can be represented as a simple state machine, the diagram for which is shown in Figure 4.3.

The driver begins in interrupt mode, with polling disabled. When a packet arrives, the network interface generates a receive interrupt. The interrupt handler disables further receive interrupts (usually by writing to a hardware register on the network device) and moves into polling mode. For each subsequent kernel polling cycle, the driver attempts to receive a full budget of packets from the interface (64 packets for most GigE interfaces). If a full budget of packets is received, the driver leaves interrupts disabled and stays in polling mode. Otherwise, if fewer packets are received, the driver turns off polling and re-enables interrupts, moving back to the interrupt state in the diagram. This design provides low-latency notification when packets arrive, but mitigates the flood of interrupts that would otherwise occur when the interface is highly utilized.

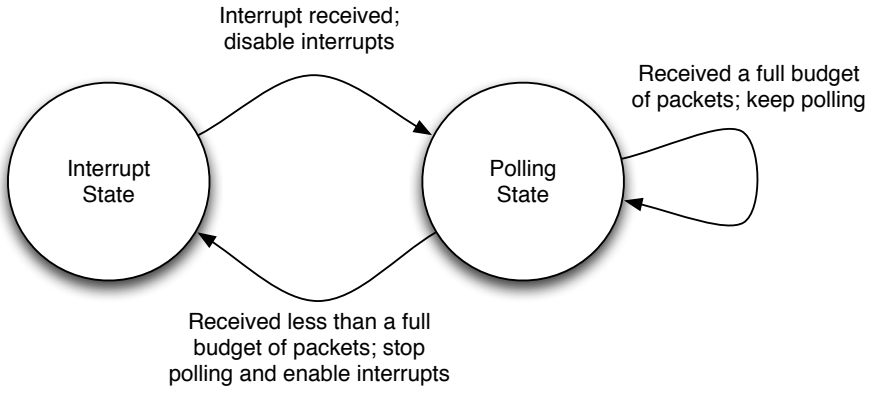

Figure 4.3: State machine for the Linux NAPI driver model

Chapter 5

Messaging Subsystem

5.1 Introduction / Motivation

Per the definition of a multikernel OS, system-wide state at the OS level is replicated, not shared, and this state is kept coherent by passing messages between kernels. This architecture implies the need for a robust kernel messaging layer.

Both Barrelfish and FOS have taken this definition a step further and provided a messaging layer for userspace processes to use as well, utilizing remote procedure calls (RPC) on Barrelfish and per-process-instance endpoints on FOS. The idea is to provide programmers the tools to support user processes on systems without cache coherency or with limited cache coherency, as well as on heterogeneous systems, while still maintaining backwards compatibility for existing applications. We will leave such a messaging layer on Popcorn as future work.

In this chapter, we detail the design, implementation, and evaluation of a messaging subsystem designed around the unique features and constraints inherent in Linux as a multikernel OS.

5.2 Design Principles

The basic design principles for messaging on Popcorn are as follows:

1. Allocate messaging channels between kernels statically, following a many-to-one approach.
2. Provide functionality for efficient multicast and broadcast, and allow multicast groups to be created and destroyed dynamically.

5.3 Unicast Messaging

5.3.1 Overview

Inter-kernel messaging on Popcorn follows a many-to-one approach, with a single receive queue per kernel instance that is created when that kernel instance boots.

By default, a few basic message types (e.g. checkin, testing) are supported. Additional message types can be added by registering callback functions with the messaging subsystem. This modular approach enables the messaging subsystem to be extended at runtime by loading kernel modules.

We intend for inter-kernel messaging to be used primarily for setup and coordination, so the number of messages sent (and the number of interrupts generated) will be moderate. We implemented a rudimentary mechanism for interrupt mitigation in the messaging layer, similar to the one used in the shared-memory network tunnel; evaluation and tuning of this mechanism is outside the scope of this thesis and is left as future work.

5.3.2 Message Handling

FOS suggested that accesses to each receiving buffer could be serialized in a lock-free manner, although they implemented kernel-level messaging using a lock-protected data structure in shared memory instead [7]. We implement such a lock-free approach in Popcorn. Pseudocode for the algorithm is shown in Figure 5.1.

The basic idea is that a kernel wishing to send a message must reserve a ring buffer slot prior to writing into it. The sending kernel gets a *ticket* by performing a fetch-and-add operation on the head of the ring buffer, which increments the head by one and returns the previous value. Upon receiving a ticket, the sending kernel spins until the corresponding ring buffer entry is free (e.g. writing to the slot won't overwrite an entry that has not yet been read). Given the large size of the ring buffer, it is highly unlikely that it will be full, so spinning is unlikely. The sending kernel then writes the message to the buffer, sets the *ready* flag at the end of the message, and signals the receiving kernel via IPI.

Messages are received with a top-half / bottom-half approach using softirqs, similar to the approach taken in the shared-memory network tunnel. Upon receiving an IPI, the receiving kernel notifies the softtirq handler that there is work to be done; aside from this notification, no work is performed within the interrupt handler.

The softirq handler itself is split into two sections.

In the first section, the handler loops through the ring buffer until it is empty, or until some predefined worst-case time has elapsed. To do this, it first spins on the tail of the ring buffer until its *ready* flag is set. The time spent spinning in this case is bounded by the time it

```
initialize:
    head = tail  = 0;

send:
    ticket = fetch_and_add(head, 1);
    spin until tail reaches (ticket - (RB_SIZE - 1));
    insert item into slot at index (ticket % RB_SIZE);
    set slot.ready;

receive:
    if (head == tail):
        return; // buffer is empty
    spin on tail until slot.ready is set;
    memcpy message to local buffer;
    clear slot.ready;
    tail++;
```

Figure 5.1: Lock-free ring buffer operations for inter-kernel messaging

takes to write a message from one CPU to another, which is minimal. (Note that if either the sending or the receiving kernel dies, the other kernel may 'get stuck'.) At this point, the softirq handler kmallocs a struct for the message, copies the message out of the receiving queue and into the struct, and inserts the struct into a linked list representing one of two queues: a high-priority queue and a normal-priority queue. Note that we call `kmalloc()` with the `GFP_ATOMIC` flag so that this operation cannot sleep.

In the second section, the messages placed in the queues in the first section are processed. All messages in the high-priority queue are serviced first, and messages in the normal-priority queue are only processed when the high-priority queue is empty. For each message, a handler function is called to process it depending on its type and the callback functions that have been registered. If a message of unknown type (e.g. without a registered callback function) is received, the message is dropped and an error is logged. If a handler function must perform an operation that sleeps (e.g. remapping physical memory), then it must handle the operation in a work queue that will run in the context of a kernel thread.

5.3.3 Setup and Check-In

The setup process for messaging at boot time is shown in Figure 5.2. At boot time, each kernel instance allocates its own receive window and announces the window's physical address in a globally-accessible array. In this way, each subsequent kernel will be able to map all the previously booted kernels' receive windows at boot time. However, when a new kernel

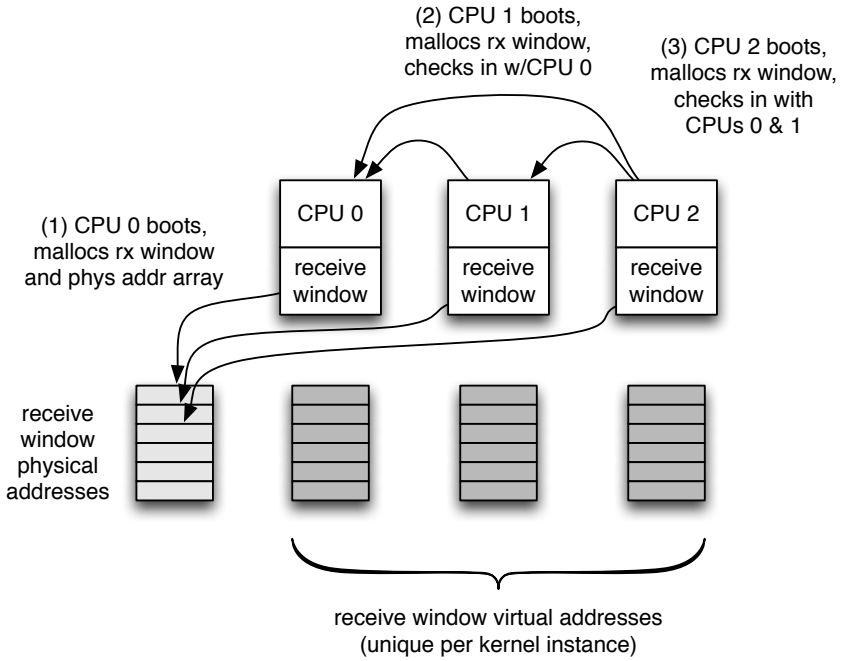

Figure 5.2: Kernel messaging window initialization process

boots, the kernels that are already running must be notified, so that they can map the new kernel's messaging window.

5.3.4 Support for Large Messages

We discovered early on that support for large messages was a desirable feature for the expected applications of this layer. For example, in remote thread and process creation, the path to each library needed by a process is represented as a string whose length can be up to *PATH_MAX*, or 4096 characters. It makes sense to provide support for large messages at the messaging layer rather than relying on a higher-level layer to implement it.

The header of every small message contains four fields to support large messages:

- `is_lg_msg` – set if a small message is part of a large message (1 bit)
- `lg_start` – set if a small message is the first chunk of a large message (1 bit)
- `lg_end` – set if a small message is the last chunk of a large message (1 bit)
- `lg_seqnum` – set to the size in chunks for the first chunk of a large message, and set to the chunk's sequence number for each subsequent chunk (7 bits)

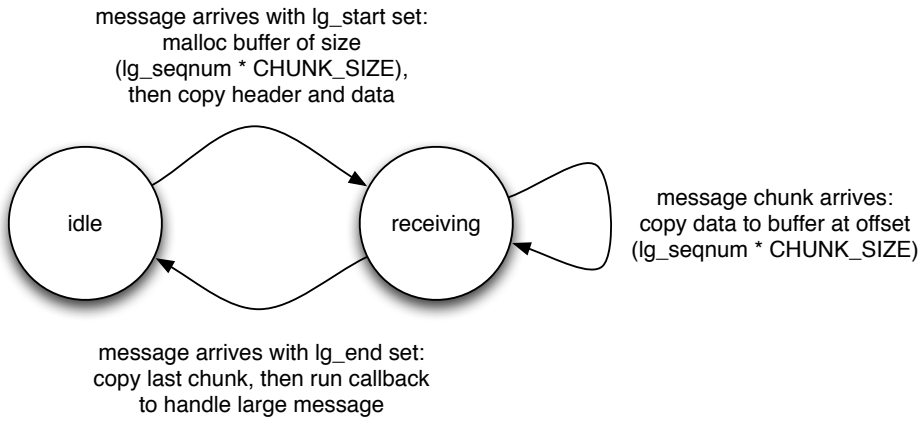

Figure 5.3: State machine for handling large messages

The process used to handle large messages is shown in Figure 5.3.

We perform two optimizations to improve the performance of large messages. First, we reserve a contiguous region in the receiving kernel instance's ring buffer for the large message; rather than getting a ticket for a single slot, we use the fetch-and-add operation to increment the head pointer by the number of slots that will be required. Second, we send only a single IPI per large message. The IPI is sent immediately after the range of slots is reserved; this is guaranteed to be safe because the head pointer has been atomically advanced by the size of the large message, so the receiving kernel instance will poll on its ring buffer until all the chunks of the large message have been received.

5.4 Multicast Messaging

5.4.1 Overview

Whereas unicast messaging uses a many-to-one approach, multicast messaging uses a many-to-many approach, where a message sent by one kernel instance is received by all the other kernel instances in a given multicast group.

The idea behind multicast messaging is that there are many potential use cases where the same message needs to be sent to every kernel in some specified group of kernels. For example, in support for distributed processes and threads, an update to a distributed process' virtual address space needs to be propagated to all of the kernels participating in that process, and the same information (and thus the same messages) needs to be sent to each kernel. There are performance optimizations that can be made if the same message is to be sent to multiple CPUs in the same set.

```
initialize:
    head = tail = 0;
    local_tail = 0 on each CPU;

send:
    ticket = fetch_and_add(head, 1);
    spin until tail reaches (ticket - (RB_SIZE - 1));
    insert item into slot at index (ticket % RB_SIZE);
    slot.read_counter = number of CPUs in mcast group - 1;
    set slot.ready;

receive:
    if (head == local_tail):
        return; // no unread messages present
    spin on local_tail until slot.ready is set;
    memcpy message to local buffer;
    local_tail++;
    if (atomic_dec_and_test(slot.read_counter)):
        atomic_inc(tail);
        clear slot.ready;
```

Figure 5.4: Lock-free ring buffer operations for multicast messaging

In keeping with the functionality that this capability targets, multicast groups on Popcorn are created and closed dynamically at runtime. After the kernel instances are booted initially, there are no active multicast groups in the system (as opposed to unicast messaging, where each kernel instance's unicast messaging channel is opened automatically at boot). As features that use multicast (e.g. distributed processes) are created, the associated multicast groups are created as well, up to some defined system-wide maximum number of groups. Each kernel instance may participate in any or all of these groups.

5.4.2 Message Handling

As with unicast messaging, multicast messaging is done in a lock-free manner. The algorithm used is detailed in Figure 5.4.

5.4.3 Channel Setup and Teardown

We define a *multicast group* as a collection of kernels sharing a multicast receive buffer. At create time, each group receives a globally-defined *multicast group ID*. Each group ID

represents the same multicast group throughout the system, and two mutually-exclusive groups cannot share the same group ID.

At channel creation, the following steps take place:

- A group ID is allocated for the channel, currently the lowest-numbered group ID not in use.
- A multicast window is allocated within the physical address space of the kernel that called `pcn_kmsg_mcast_open()`. From this point forward, this kernel is termed the *group owner*. The group owner may not be removed from the group, and when the group is closed, the group owner is responsible for freeing the group's multicast window.
- The group owner sends a message to the other kernels in the group, telling them to map the group's multicast window.

After this point, all the kernels in the multicast group have mapped the group's multicast window, and all the kernels are set up to poll this window whenever a notification IPI is received. Note that with the current version, the group members and size cannot be changed after the group has been created, and multicast group close has not yet been implemented (see Section 8.1).

5.5 Implementation Challenges

When testing multicast messaging, we found that once out of every several hundred messages, `slot.read_counter` would not be decremented to zero after all the readers had copied out a particular message. Debugging showed that the `atomic_dec_and_test()` operation was not functioning as expected; when `slot.read_counter` was initialized to 2 and atomically decremented twice, the end value would infrequently be 1, not 0 as expected. Further analysis indicated the root cause of this problem.

On uniprocessor machines, and on multiprocessor machines with only one processor active, the Linux built-in atomic operations do not actually lock the bus on x86. On SMP Linux, this is a worthwhile optimization on these machines; since only one CPU is active, there is no need to guard against other CPUs interfering in an atomic operation. The issue arises because in non-clustered mode, each Popcorn kernel instance sees its hardware partition as an n-core machine with only one core active, so the Linux atomic operations do not lock the bus. This behavior is safe in the case of atomic variables that are confined to a single kernel instance's address space, but unsafe in the case of atomic variables in memory regions that are mapped to multiple kernel instances.

To circumvent this issue, we wrote our own versions of the atomic operations needed for multicast that always lock the bus.

Chapter 6

Results

6.1 System Usability

We have verified that we can boot 48 kernels on a 48-core server and 64 cores on a 64-core server. In addition, we have verified that we can run standard Linux binaries under Popcorn on both the boot kernels and the secondary kernels, including kernel compilation (Popcorn is self-hosting).

To demonstrate Popcorn's usability, and to show that it does not incur any unexpected performance penalties on workloads that do not stress the OS heavily, we developed a modified version of the MPICH2-Nemesis MPI runtime to execute under Popcorn, similar to the one provided in MPI-Nahanni [30]. Note that this work was done collaboratively: Alastair Murray and Antonio Barbalace modified the MPICH2-Nemesis runtime and environment to execute under Popcorn, and Shawn Furrow wrote the code to analyze the data and generate the plots. Results for benchmarks from the NPB benchmark suite are shown in a paper we submitted to SYSTOR 2013, which appears as a pre-print on the Popcorn website [46].

6.2 Hardware Costs and Latencies

We ran these benchmarks on *gigi*, a 64-core machine with four 16-core AMD Opteron 6274 processors running at 2200 MHz and 128 GB of RAM (32 GB for each physical processor).

6.2.1 Cache Coherence Latency

Both Barrelfish and FOS rely on messaging by polling on cache lines; per Section 2.2.2, this solution offers the lowest latency and greatest throughput of the messaging channels

available on the x86-64 architecture. It is therefore useful to measure the performance of this channel.

To do this, we ran the cache line ping-pong benchmark provided as an appendix in the FOS messaging thesis [7]. This benchmark sends a cache-line sized message to and from a remote core, an identical approach to the one described and employed by Barrelfish [5]. It uses the x86 `rdtsc` instruction to perform cycle-level timing, and measures and accounts for the overhead of executing this instruction (around 35 cycles). The benchmark was compiled with the `-O2` optimization option in GCC. We pinned the client to core 0 using `taskset`, and we pinned the server to several different cores to determine how the latency varied depending on the relative position of the two nodes. Each test was run with 100,000 trials with a 100K cycle delay between trials. Per the original FOS benchmark code, trials over 10,000 cycles were discarded; they represent cache misses and OS overheads rather than the actual cache line ping-pong latency. The results are shown in Table 6.1.

	avg	median	min	max
same NUMA node (CPU 0 to CPU 2)	425	425	330	9961
same die (CPU 0 to CPU 10)	414	399	114	9889
different die (CPU 0 to CPU 48)	425	429	333	9987

Table 6.1: Single cache line ping-pong latencies (cycles)

6.2.2 IPI Cost and Latency

IPI Cost

We define IPI cost as the number of cycles from when a sender makes a call to send an IPI to when the call to send an IPI returns. To measure this cost, we created a syscall that would take an initial timestamp, send a 'dummy IPI' to a specified CPU, and then take a final timestamp. The interrupt handler on the remote CPU would simply return without doing any work. As in the cache line ping-pong benchmark, we account for the running time of the `rdtsc` instruction when calculating the latency.

We ran 10,000 iterations of this test using both `send_IPI_mask` and `send_IPI_single`. Table 6.2 shows the cost incurred by CPU 0 to send a single IPI to CPU 2.

	avg	median	min	max
send_IPI_mask	194	192	190	1405
send_IPI_single	96	96	93	3127

Table 6.2: Cost to send one IPI (cycles)

The results show that the cost to send an IPI from kernel mode is fairly low, but our addition of `send_IPI_single` decreases the cost by about half.

IPI Latency

The time it takes for an IPI to arrive at its destination is orthogonal to the cost to the sender of sending the IPI, so it is necessary to measure both parameters.

Since the timestamp counters on multiple CPUs are not synchronized, there is no good way to measure one-way IPI latency at the cycle level. Instead, we took a ping-pong approach as with the cache line latency measurements. We wrote a test syscall that performs a specified number of ping-pong IPI tests to a specified CPU and logs the results of each test.

For each test iteration, CPU 0 sets *done* (a CPU-local variable) to false, collects an initial timestamp value, sends a `POPCORN_IPI_BOUNCE` IPI to CPU n, and then spins until *done* is true. When CPU n receives the IPI, its interrupt handler sends a `POPCORN_IPI_CATCH` interrupt back to CPU 0. When CPU 0 receives this second IPI, its interrupt handler collects a final timestamp value and sets *done* to true. As in the previous benchmark, we account for the running time of the `rdtsc` instruction when calculating the latency.

As with the cost-to-send test, we performed 10,000 iterations of each test, and we repeated the test with both `send_IPI_mask` and `send_IPI_single`. Table 6.3 shows the round-trip latency from CPU 0 to CPU 2.

	avg	median	min	max
send_IPI_mask	7507	7750	4128	65672
send_IPI_single	7501	7739	4097	67790

Table 6.3: IPI ping-pong latencies (cycles)

It is interesting to note that this modification only slightly decreases the IPI round-trip time in this case, since for low-numbered destination CPUs, the majority of the superfluous operations occur after the IPI has already gone out.

6.2.3 System Call Latency

We replicated the system call latency test performed in the FlexSC paper [48], in which we implement a syscall that performs a single `rdtsc` instruction and returns the value of the timestamp. By comparing this intermediate timestamp to timestamps taken immediately before and after making the syscall, it is possible to determine the cost in cycles to enter and to exit a syscall. This measurement serves as a useful proxy for the amount of time it takes the system to enter and exit an interrupt handler, specifically the time it takes to switch from user mode to kernel mode and back.

We performed 100,000 iterations of this test. The results are shown in Table 6.4.

	avg	median	min	max
entering syscall	160	160	155	7604
exiting syscall	134	135	133	161

Table 6.4: Cost to enter/exit a syscall (cycles)

6.2.4 Conclusions

We can draw the following conclusions from this data:

- The cost of entering and exiting a syscall handler is non-negligible but reasonable.
- The ping-pong latency for a single cache line through shared memory is smaller than the ping-pong latency for IPI by about one order of magnitude, so it makes sense to send data in this manner preferentially. However, in Popcorn, this is only possible if the receiving core is already in the kernel; otherwise, IPI is always required.
- From kernel mode, the time it takes to send an IPI is very small compared to the time it takes for the IPI to arrive (one to two orders of magnitude smaller). For inter-kernel messaging, it would make sense to send the IPI preemptively to avoid incurring additional latency. However, the Barrelfish approach of spinning for some period of time before sending an IPI makes more sense for userspace messaging, where the additional costs of making a syscall (~ 3 times higher than the cost of sending the IPI itself) must be incurred as well.

6.3 Shared-Memory Network Driver

To measure the performance of Popcorn's shared-memory network driver, we ran a series of benchmarks. As before, we ran these benchmarks on *gigi*, a 64-core machine with four 16-core AMD Opteron 6274 processors running at 2200 MHz and 128 GB of RAM (32 GB for each physical processor). The test setup is shown in Figure 6.1. The test machine for generating traffic in the test-to-guest benchmarks is *roastduck*, a 6-core AMD desktop running Ubuntu 10.04 LTS.

We used the Linux Kernel Virtual Machine (KVM) to provide a performance comparison with Popcorn. For the VM configuration, we ran Ubuntu 10.04 LTS on the host with the default Linux 2.6.32 kernel, along with the provided *kvm* and *kvm-amd* kernel modules and the provided QEMU version 0.12.3. The VM itself ran Ubuntu 10.04 LTS with the

default Linux 2.6.32 kernel. Networking to the VM was provided through a public bridged configuration, which the KVM project lists for the use case where the VM needs to be externally visible on the local network and performance is a concern [34].

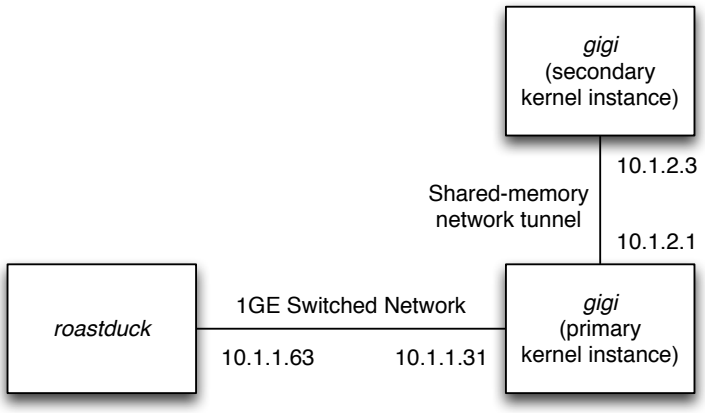

Figure 6.1: *shmtun* driver network setup

6.3.1 TCP Performance

We conducted benchmarks of TCP performance using the *nuttcp* network performance testing tool. The host-to-guest tests involve sending traffic from the primary kernel instance or the host kernel on *gigi* to a secondary kernel instance or virtual machine on *gigi*; the test-to-guest tests involve sending traffic from *roastduck* to a secondary kernel instance or virtual machine on *gigi*. The results are shown in Tables 6.5, 6.6, and 6.7.

Measurement	Popcorn	KVM
Host-guest bandwidth (Mbps)	2243	1127
Host-guest RTT (ms)	0.05	0.46
Host/guest CPU (%)	100/63	10/98
Test-guest bandwidth (Mbps)	909	427
Test-guest RTT (ms)	0.18	1.24
Test/guest CPU (%)	18/7	21/87

Table 6.5: *nuttcp* benchmark results, same NUMA node (to CPU 2)

The data shows that Popcorn's shared-memory network driver is able to deliver the full performance of the 1 Gbit/s link; a quick *nuttcp* test showed nearly identical performance between *roastduck* and *gigi* directly. It is able to do so while incurring minimal load (around 7%) on the secondary kernel instance. (Note that it would be useful to measure the CPU

Measurement	Popcorn	KVM
Host-guest bandwidth (Mbps)	2130	1215
Host-guest RTT (ms)	0.04	0.44
Host/guest CPU (%)	100/63	10/99
Test-guest bandwidth (Mbps)	910	406
Test-guest RTT (ms)	0.15	0.64
Test/guest CPU (%)	18/7	20/91

Table 6.6: *nuttcp* benchmark results, same die (to CPU 10)

Measurement	Popcorn	KVM
Host-guest bandwidth (Mbps)	2082	1236
Host-guest RTT (ms)	0.05	0.37
Host/guest CPU (%)	100/61	10/99
Test-guest bandwidth (Mbps)	910	405
Test-guest RTT (ms)	0.17	0.50
Test/guest CPU (%)	18/7	20/90

Table 6.7: *nuttcp* benchmark results, different die (to CPU 48)

utilization of Popcorn's primary kernel instance, since it is sharing the burden of handling the network traffic.) In addition, Popcorn delivers over 2 Gbit/s between the primary kernel and the secondary kernels; this performance decreases with NUMA distance from the primary kernel, but not by much (less than 10% overall).

While KVM is able to provide greater than 1 Gbit/s throughput between host and guest, it is not able to deliver the full performance of the 1 Gbit/s link; instead, it delivers less than half of it. Looking at the test/guest CPU utilization shows why this is the case: the CPU utilization for the guest VM is nearly 100%, while the utilization for the sender (*roastduck*) remains relatively low. Note that this CPU utilization figure will also show near 100% if overall performance is limited by memory or cache performance rather than raw CPU performance. We hypothesize that the poor performance seen is due to the fact that QEMU must emulate a hardware network interface for the KVM virtual machine, resulting in VM exits and switches from kernel mode to user mode and back, both of which are relatively costly. Testing this hypothesis would require collecting statistics from KVM.

6.3.2 Interrupt Mitigation

As described in Section 4.3.3, a key design goal of the shared-memory network tunnel was to mitigate the number of interrupts generated to reduce the load on the system during periods

of high network traffic.

Like all Linux network drivers, the *shmtun* network driver keeps statistics for the traffic that flows over the interface: packets sent, packets dropped, number of errors encountered, and other similar information. The Linux kernel itself also maintains counters for each type of IPI that occurs; these counters are available in the `/proc` filesystem under `/proc/interrupts`. We added counters for the network tunnel IPIs. With data from both of these sources, we were able to determine how well the receive interrupts were mitigated in a high-traffic situation.

We use the ratio of interrupts generated to packets received as a measure of interrupt mitigation. With a naive approach using only interrupts and not polling (e.g. no NAPI), one interrupt would be generated per packet sent over the interface, so the ratio would be $1 : 1$. A polling-only approach would have a ratio of $1 : \infty$. The results are shown in Table 6.8 for a 10-second test with *nuttcp*.

Setup	Packets Received	Interrupts Generated	Ratio
Host-guest (CPU 2)	1969272	1248196	1 : 1.58
Host-guest (CPU 10)	1888671	1190931	1 : 1.59
Host-guest (CPU 48)	1839414	1094926	1 : 1.68
Test-guest (CPU 2)	785713	114347	1 : 6.87
Test-guest (CPU 10)	785507	109801	1 : 7.15
Test-guest (CPU 48)	785729	84498	1 : 9.30

Table 6.8: Interrupt mitigation results

The results show that interrupt mitigation is indeed effective; the ratio is better than $1 : 1$ in all cases. It is interesting to note that the interrupt mitigation is significantly more effective in the test-to-guest case. Note also that as the NUMA distance between host and guest increases, interrupt mitigation becomes more efficient.

Note that the NAPI *weight* parameter for the network tunnel is set to 64, as is the default for gigabit network interfaces. It would be useful to construct a simple model to predict the effect of the NUMA node, the packet arrival rate, and the value of the *weight* parameter on how well interrupts are mitigated, allowing for more effective tuning and performance optimization. We leave this modeling as future work and address it in more detail in Section 8.4.2.

6.3.3 Web Server Performance

For further evaluation of the performance of the *shmtun* driver, we used it in combination with the *nginx* event-based web server. We chose this web server because it is simple and

fast, and because it is possible to compile all the required libraries into the binary so it can be easily built on one machine and copied to another.

We evaluated its performance relative to two comparable systems, Linux KVM and unmodified SMP Linux. In the Popcorn and Linux KVM tests, the *nginx* web server was run on CPU 2, with 1 GB of memory assigned to the secondary kernel and the guest VM, respectively. In the unmodified SMP Linux test, the kernel was limited to 1 GB of RAM with the *mem=* kernel argument, and the *nginx* web server was pinned to CPU 2 using *taskset*. The tests were repeated for 100 current requests and 1000 concurrent requests; 10,000 requests were performed for each test (higher numbers of requests resulted in hitting the driver reset issue described in Section 8.1). The webpage served is *nginx*'s default welcome page. The results are shown in Figure 6.2.

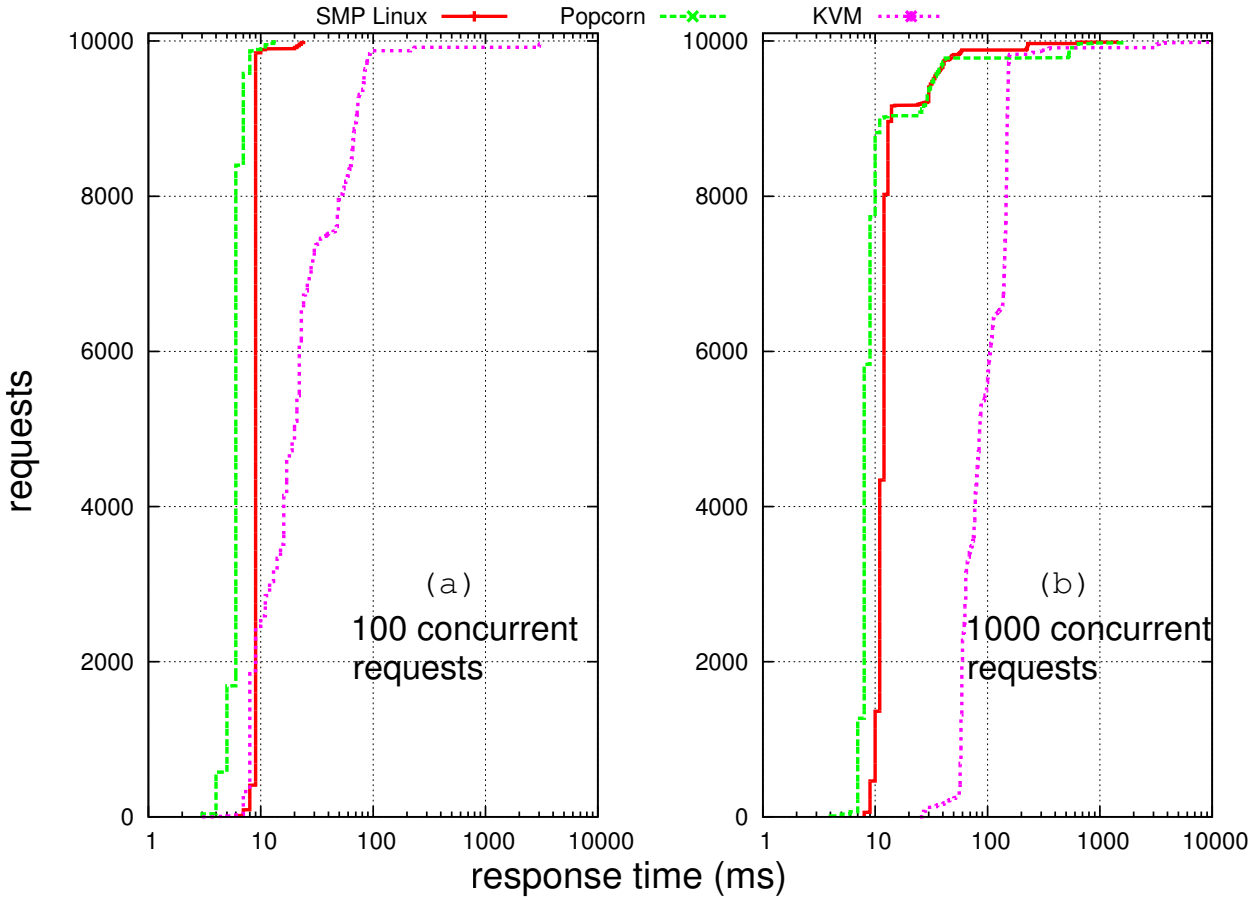

Figure 6.2: ApacheBench results for the *nginx* web server on SMP Linux, Popcorn, and Linux KVM

Looking at the results, it is clear that Linux KVM performs particularly poorly, especially on the higher-concurrency test case. Popcorn and unmodified SMP Linux both perform well, but Popcorn performs slightly better, offering slightly lower response times in the majority

of cases. There are several reasons why this result might occur. First, the primary kernel instance is actually assisting the secondary kernel instance in sending and receiving packets. The primary kernel instance handles the interaction with the hardware network device, while the secondary kernel instance is left with the less-intensive task of handling its side of the shared-memory network tunnel. Second, as a result, there may be improved cache behavior: the code to handle the network interface can stay warm-cache in CPU 0, while the code to handle web server requests can stay warm-cache in CPU 2, and they are less likely to interfere with each other than they would be if run on the same core. A fairer comparison would account for the work performed by the primary kernel instance in the Popcorn results.

6.3.4 Conclusions

The shared-memory network tunnel driver meets its design goals of being able to share a Gigabit Ethernet link with multiple secondary kernel instances with reasonably low CPU utilization and with mitigation of receive interrupts. Although performance is very good, there may be room for further improvement with different approaches; we leave this as future work as described in Section 8.4.

6.4 Kernel Messaging Layer

6.4.1 Small Message Costs and Latencies

The design of OS-level messaging layers is dictated by the design principles and requirements for the systems in which the layers are designed to be used, and it is not always possible to do an apples-to-apples comparison between two designs. One way to verify that a messaging layer is efficient is to take fine-grained measurements of the costs and latencies required to send a message and compare them to the latencies that the hardware offers as measured in Section 6.2.

As before, we used the x86 *rdtsc* instruction to perform these measurements. A test program running in userspace makes a syscall to perform each test iteration; the test is performed in kernel space and the resulting timestamps are returned to userspace to be logged to a file. Messages in this test are sent back-to-back with a 5 ms delay between them. Timestamps taken on the receiving end are encapsulated into a message on the receiving end and sent back to the sender. Note that timestamps taken on different cores are not directly comparable; after several minutes of uptime, we estimated that the timestamp counters of two cores on the same NUMA node were out of sync by about 3000 cycles, although the exact offset is impossible to measure. Note also that although a rudimentary method of interrupt mitigation was enabled in these tests, it did not actually operate; the 5 ms pause between iterations was enough for the state machine to reset itself to the ‘interrupts enabled’ state.

Tables 6.9, 6.10, and 6.11 show the cost in cycles of each step required to send a single cache-line-sized 64-byte message.

measurement	avg	median	min	max	5pct	95pct
insert (*pcn_kmsg_send()* to IPI)	273	275	202	8952	227	342
start of ISR to end of ISR	1252	961	644	1922	913	1750
end of ISR to start of bottom half	325	322	243	3072	311	346
start of BH to removal of msg from ring buffer	840	794	629	2001	776	1038
time from RB to execution of callback function	107	108	74	515	96	121
round-trip time for ping-pong	26329	25478	8338	113892	24920	26331

Table 6.9: Ping-pong message costs and latencies, same NUMA node (CPU 0 to CPU 2)

measurement	avg	median	min	max	5pct	95pct
insert (*pcn_kmsg_send()* to IPI)	482	465	401	6096	418	734
start of ISR to end of ISR	1240	966	648	2204	920	1737
end of ISR to start of bottom half	419	414	245	10437	406	432
start of BH to removal of msg from ring buffer	1008	955	701	11810	938	1269
time from RB to execution of callback function	111	110	83	258	100	124
round-trip time for ping-pong	27898	27089	9286	115532	26495	28062

Table 6.10: Ping-pong message costs and latencies, same die (CPU 0 to CPU 10)

measurement	avg	median	min	max	5pct	95pct
insert (*pcn_kmsg_send()* to IPI)	652	618	546	12289	556	1014
start of ISR to end of ISR	1273	992	686	2001	954	1779
end of ISR to start of bottom half	579	574	312	5048	563	604
start of BH to removal of msg from ring buffer	1287	1338	599	4137	693	1370
time from RB to execution of callback function	100	98	81	183	86	140
round-trip time for ping-pong	29011	28007	10141	115798	27222	28861

Table 6.11: Ping-pong message costs and latencies, different die (CPU 0 to CPU 48)

We can draw the following conclusions from this data:

- As expected, the greatest latency in the process by far is the IPI propagation time.
- Not considering the IPI propagation time, a significant fraction of the time is spent in Linux kernel operations that are compulsory for handling an interrupt. The code from the start of the ISR until the bottom half is scheduled, which is made up of

the `ack_APIC_irq()`, `inc_irq_stat()`, and `irq_enter()` operations, takes over 1200 cycles on average. About 400 cycles elapse from immediately before the bottom half is scheduled to when the bottom half actually begins executing. For future work, it might be advantageous to provide a "fast path" through these pieces of code that is pared down to contain only the operations necessary for this purpose.

- The difference in costs based on the NUMA-ness of our testbed is evident in the data. Note that the cost to insert the message into the ring buffer is an average of 273 cycles on the same node, 482 cycles on the same die, and 652 cycles on a different die. Similar behavior is seen for the time for the receiver to remove the message from the ring buffer. However, for operations that reside in kernel memory that is replicated per CPU, the costs stay relatively constant throughout. For example, the number of cycles that the kernel requires to handle the interrupt within the ISR is an average of 1252 for same-node, 1240 for same-die, and 1273 for different-die.

- The variability for operations involving shared state is significantly greater than the variability for operations involving only local state. For example, the 5th percentile and 95th percentile bounds for the time to schedule and run the bottom half are very narrow. However, the bounds for insertion into / removal from the ring buffer are fairly wide; we hypothesize that this result is due to atomic operations and operations on shared cache lines, whose costs may be dependent upon bus contention and system state. Finally, the bounds for round-trip time are fairly narrow; we hypothesize that this result is due mainly to the IPI latency, which may also be dependent upon bus contention and system state, but which varies relatively little in this test due to the lack of contention caused by the delay between test iterations. Further measurement and analysis would be necessary to pinpoint the causes of this variability.

- Note that we also ran this test without a delay between messages both with and without our rudimentary interrupt mitigation, and we found that in both cases, the mean round-trip time was cut by roughly a factor of two, as was the mean time from start-of-ISR to end-of-ISR. However, further data collection and analysis in this area is outside the scope of this thesis and is left as future work.

6.4.2 Large Message Performance

As described in Section 5.3.4, we provide large message support with optimizations to reduce the number of atomic operations and IPIs required. In this section, we evaluate whether these optimizations work as expected, and whether the resulting performance is good given what the hardware provides.

We measured both cost to send and round-trip time vs. the number of cache-line-sized chunks in the message. We performed 10,000 iterations for each message size, with a 5 ms pause between each iteration. The results of this test are shown in Figures 6.3, 6.4, and 6.5.

The error bars show the 5th and 95th percentiles of the data. Note that at present time, the largest large message size we support is 127 chunks, due to a 7-bit field used for the sequence number in each chunk's header. If larger messages are needed to provide good performance, this field could be expanded or restructured. Note also that although a rudimentary method of interrupt mitigation was enabled in these tests, it did not actually operate; the 5 ms pause between iterations was enough for the state machine to reset itself to the 'interrupts enabled' state.

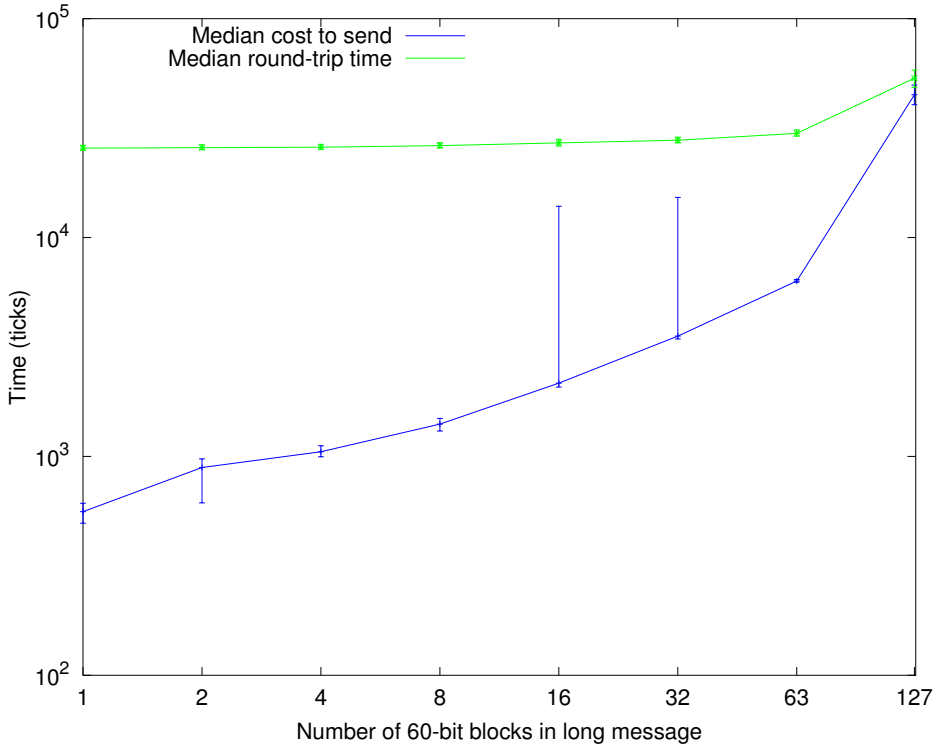

Figure 6.3: Send time and ping-pong time vs. number of 60-byte chunks in large message, same NUMA node (CPU 0 to CPU 2)

We can draw the following conclusions from this data:

- As before, the IPI latency represents the majority of the total latency.
- There is an inflection point above 64 chunks where the time to send increases significantly. We (somewhat arbitrarily) set the ring buffer size to 64 small messages. Since the sender fills up the ring buffer before the IPI reaches the receiver, the sender must block until the receiver receives the IPI, begins to process the ring buffer, and frees up some entries. After this point, transfer between sender and receiver becomes efficient again, as both are effectively performing cache-line polling, as described in Section 6.2.1.

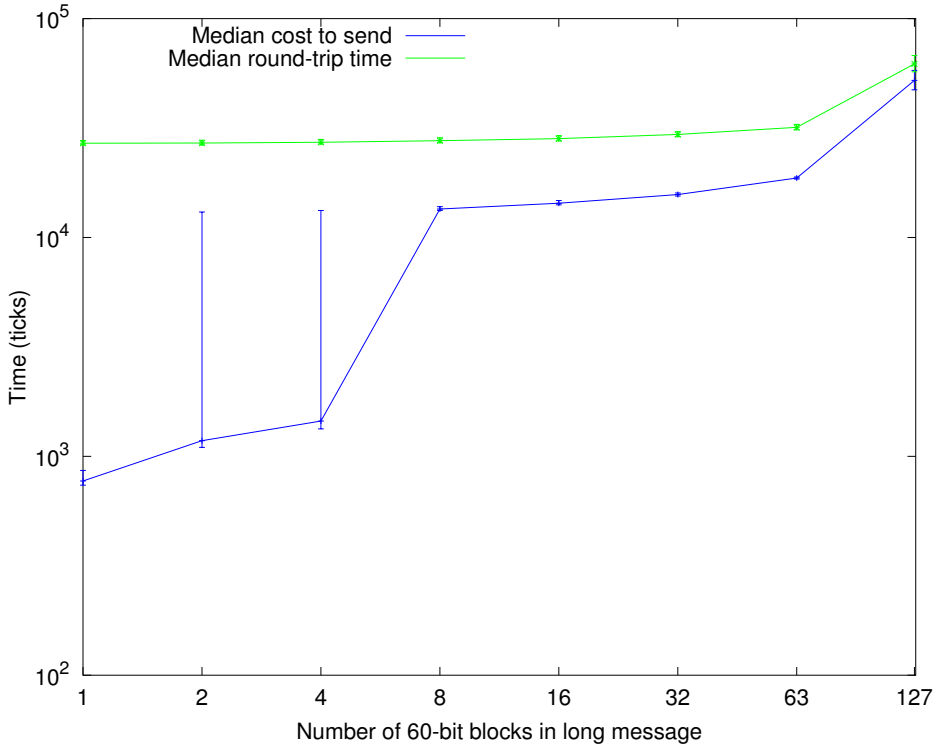

Figure 6.4: Send time and ping-pong time vs. number of 60-byte chunks in large message, same die (CPU 0 to CPU 10)

- This result leads to a rule of thumb for estimating the optimal buffer size to achieve the best performance. The buffer should be large enough that the sender can write into it without exhausting all of the entries until the IPI arrives and the receiver begins clearing out the buffer. In this case, the abrupt reduction in performance when the buffer becomes full for messages above 64 blocks indicates that that the buffer is probably too small. Note that exact tuning will not be possible because the architecture of the ring buffer requires that it contain a power-of-2 number of slots, and that it would be better for performance to overestimate the capacity required.

- We hypothesize that the variability seen in some of the Popcorn cost-to-send results is due to atomic operations and operations on shared cache lines, whose costs may be dependent upon bus contention and system state. As before, further evaluation and analysis would be necessary to determine the exact causes of this behavior.

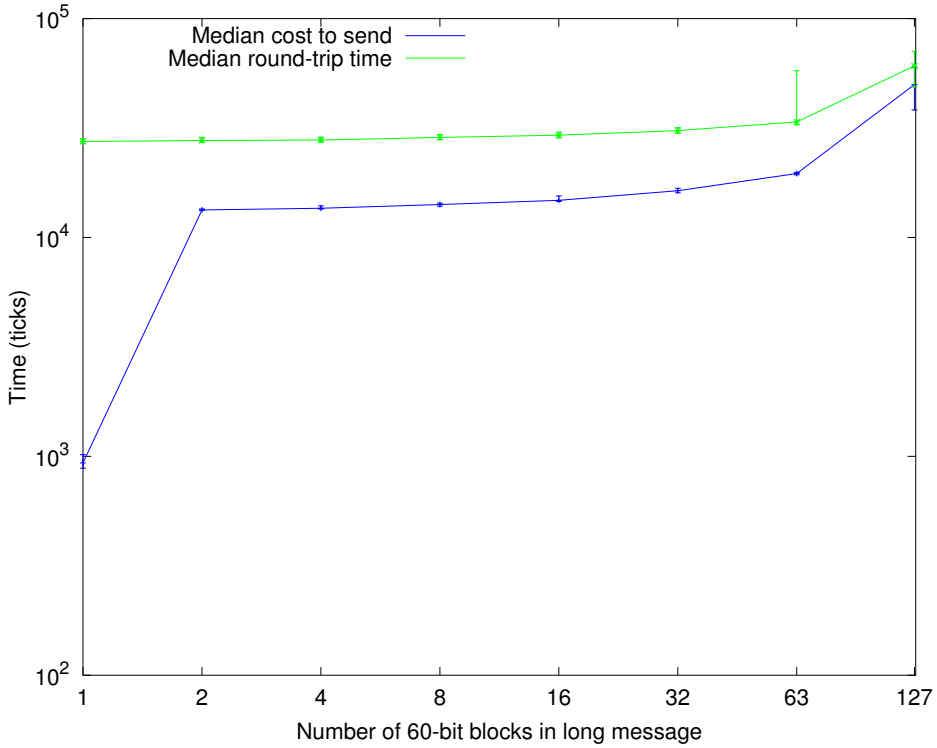

Figure 6.5: Send time and ping-pong time vs. number of 60-byte chunks in large message, different die (CPU 0 to CPU 48)

6.4.3 Multicast Messaging

The main design goal of multicast messaging is to reduce the overhead on the sender when sending the same message to all the members of a group. In the ideal case, the cost to send a message to a group would remain constant as the number of members in the group increases. To test multicast messaging, we set up multicast groups with varying numbers of members and measured the overhead in cycles for both parts of the message send operation: insertion of the message in the multicast window, and notification to the group members to check the multicast window. These tests were performed with a 60-byte message (one slot occupying one cache line), with 10,000 multicast messages sent for each group size. The results are shown in Figure 6.6. The error bars show the 5th and 95th percentiles of the data.

Looking at the plot, the data shows that multicast achieves its design goal for messaging, although not for notification. The cost to put a message into the mcast buffer remains relatively constant, but the cost to send IPIs is the real limiting factor here, increasing roughly linearly with the number of members in the multicast group. As we discuss in Section 8.3, the Intel x2APIC standard may improve the scalability of IPI broadcast/multicast on

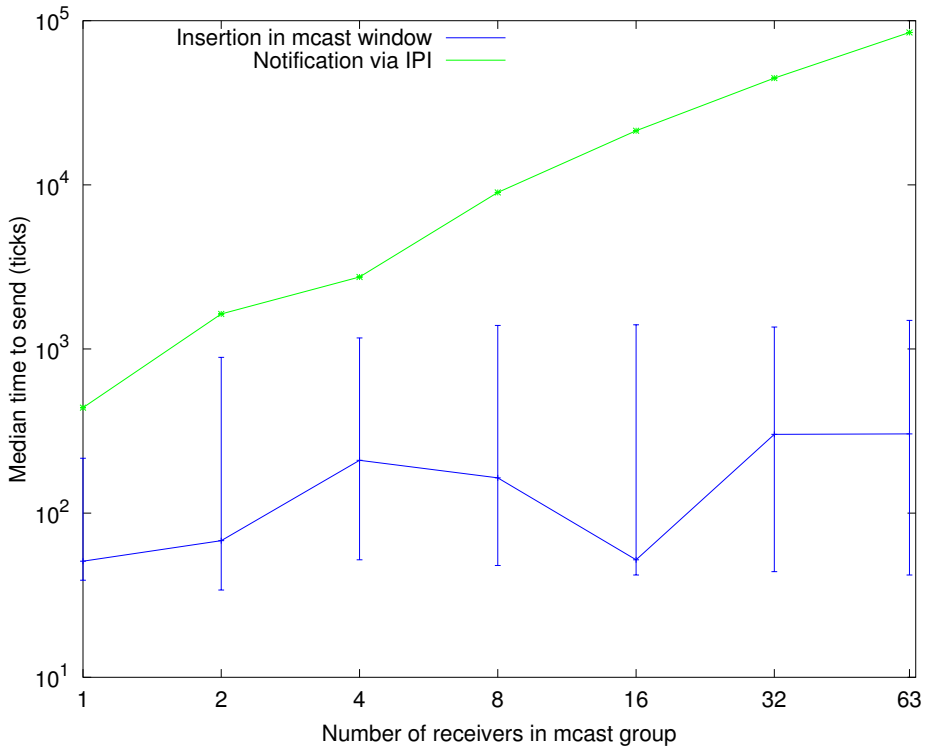

Figure 6.6: Overheads to sender of multicast messaging vs. multicast group size

high-core-count systems, though we lack the hardware to test this hypothesis. Note that as before, the bounds for insertion into the ring buffer are fairly wide; we hypothesize that this variability is due to atomic operations and operations on shared cache lines, whose costs may be dependent upon bus contention and system state.

6.4.4 Comparison to Barrelfish

As the originator of the multikernel OS concept, and as the most-developed multikernel OS with source code available, Barrelfish is a competitor to Popcorn. It is useful to examine both the design and the performance of Barrelfish's messaging layer and make comparisons to what we have developed in Popcorn.

Test Description

The messaging framework in Barrelfish is strictly event-based (as makes sense given a system built entirely on message-passing) and is designed to encourage the programmer to write in this style. The inter-kernel messaging channel only supports one queued message at a time. If a program attempts to send a message when the channel is in use, it receives a "channel full" error code, and is then responsible for registering a "retry send" event callback that the OS will call after the channel is no longer in use.

We wrote a simple ping-pong messaging benchmark for Barrelfish called *xmpl_pingpong*, building off the *xmpl_msg* example program included with Barrelfish. Two instances of *xmpl_pingpong* are launched, one client and one server, on two different kernels running on two different CPUs. The server creates a messaging channel and the client binds to it, and the initial test message is sent in the callback function that runs on the client after the binding process has succeeded. After that, the ping-pong messaging test occurs; in each iteration, the client sends a cache-line-sized message to the server, the server sends back a cache-line-sized message to the client, and the timestamps are logged over the serial port. Note that there is no explicit 5 ms delay between test iterations as in the Popcorn tests; instead, the time it takes to log the results of each test over the serial port serves as an adequate delay. Also note that these tests were run with Barrelfish's round-robin scheduler; per an off-list discussion with the Barrelfish authors, the default scheduler can cause performance issues. Finally, note that this ping-pong messaging benchmark runs in userspace, but due to Barrelfish's microkernel-like architecture, this is the case for all of the OS features, including device drivers and the network stack, so it is still a valid comparison.

As before, we performed 10,000 test iterations for each message size from 1 to 127 cache lines. Also as before, the error bars show the 5th and 95th percentiles of the data.

measurement	avg	median	min	max	5pct	95pct
send function (call to return)	1058	1032	742	2690	895	1377
from send function call to send_cb execution	2254	2195	1438	4128	1855	2850
round-trip time for ping-pong	28654	24343	4312	46374	5291	44122

Table 6.12: Barrelfish ping-pong message costs and latencies, same NUMA node (CPU 0 to CPU 2)

measurement	avg	median	min	max	5pct	95pct
send function (call to return)	1073	1044	753	2113	903	1387
from send function call to send_cb execution	2326	2271	1474	4446	1927	2918
round-trip time for ping-pong	29026	24983	4513	64846	5570	44869

Table 6.13: Barrelfish ping-pong message costs and latencies, different NUMA node (CPU 0 to CPU 10)

Single Message Microbenchmarks

The individual costs and latencies for sending a one-cache-line sized message are shown in Tables 6.12, 6.13, and 6.14.

Note that in all of the tests performed, due to the delay between ping-pong messages, the messaging channel was not contended and an insignificant number of OS-managed retries (fewer than 5 per message size) were encountered.

We can draw the following conclusions from this data:

- The overhead to the sender, both in terms of time to return from the send function and time to enter the send callback function, is relatively independent of NUMA node. The average number of cycles to return from the send function is 1058 on same-node, 1073 on different-node, and 1056 on different-die. The average number of cycles from calling send to entering the send callback function is 2254 on same-node, 2326 on different-node, and 2240 on different-die. The 5th and 95th percentiles of these times

measurement	avg	median	min	max	5pct	95pct
send function (call to return)	1056	1037	748	2003	900	1358
from send function call to send_cb execution	2240	2198	1415	3683	1894	2788
round-trip time for ping-pong	29817	25302	4444	47649	5847	45520

Table 6.14: Barrelfish ping-pong message costs and latencies, different die (CPU 0 to CPU 48)

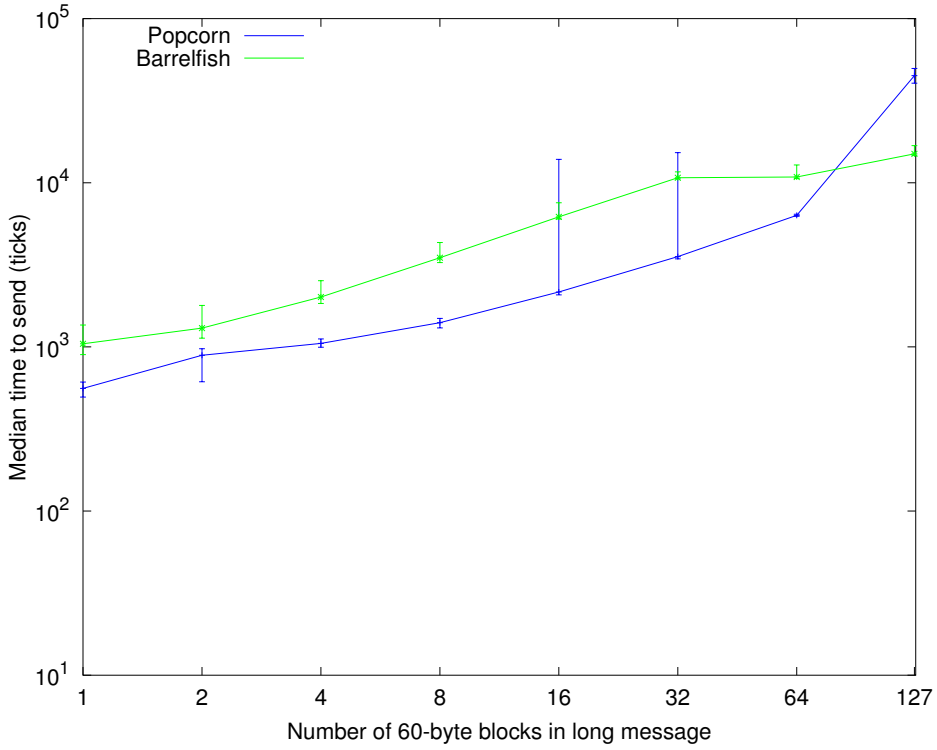

Figure 6.7: Barrelfish vs. Popcorn cost to send comparison, same NUMA node (CPU 0 to CPU 2)

are relatively close, indicating fairly consistent performance. Based on these results, it is likely that the sender in Barrelfish messaging writes messages to a node-local buffer, rather than to a buffer that is local to the receiving core/kernel.

- The minimum round-trip time is just over 4,000 cycles across the board. This is better than we see with Popcorn (7-8K cycles minimum). However, there is significantly more variability in Barrelfish's performance, as we will see in the round-trip time measurements as well.

Batch Message Send Time

Comparisons of the cost to the sender for different batch sizes are shown in Figures 6.7, 6.8, and 6.9.

We can make the following observations about this data:

- The send time on Popcorn is affected much more by the NUMA-ness of the system

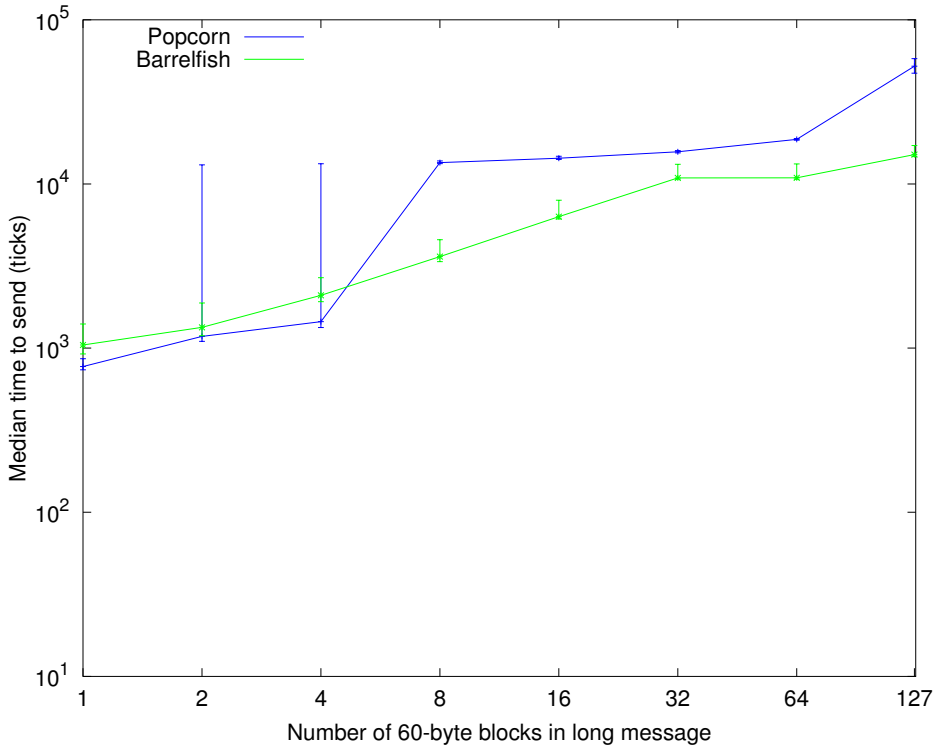

Figure 6.8: Barrelfish vs. Popcorn cost to send comparison, same die (CPU 0 to CPU 10)

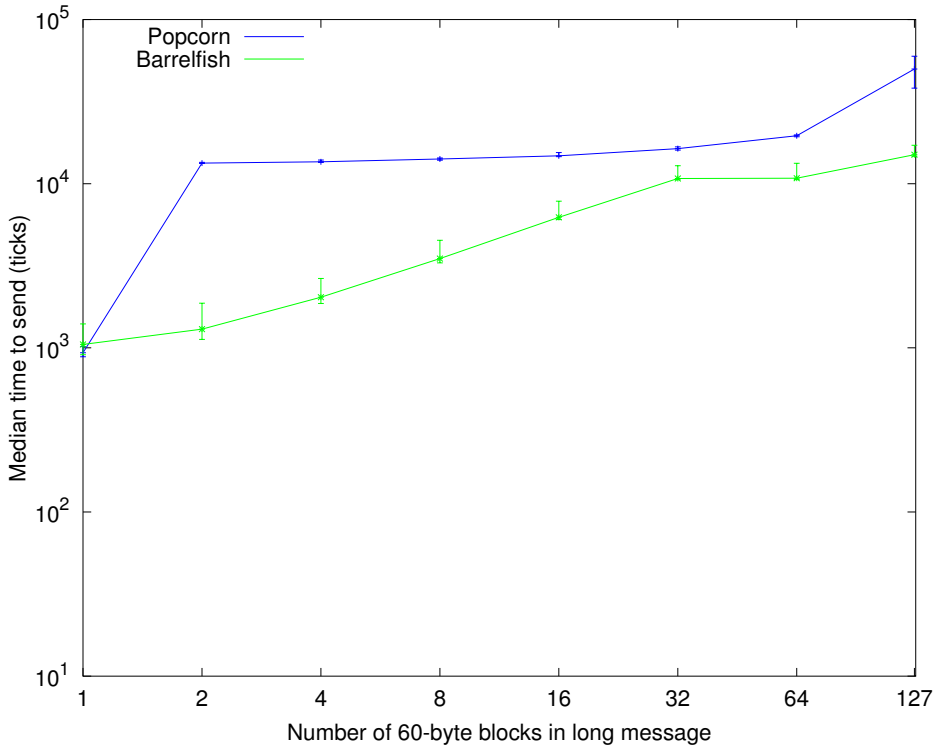

Figure 6.9: Barrelfish vs. Popcorn cost to send comparison, different die (CPU 0 to CPU 48)

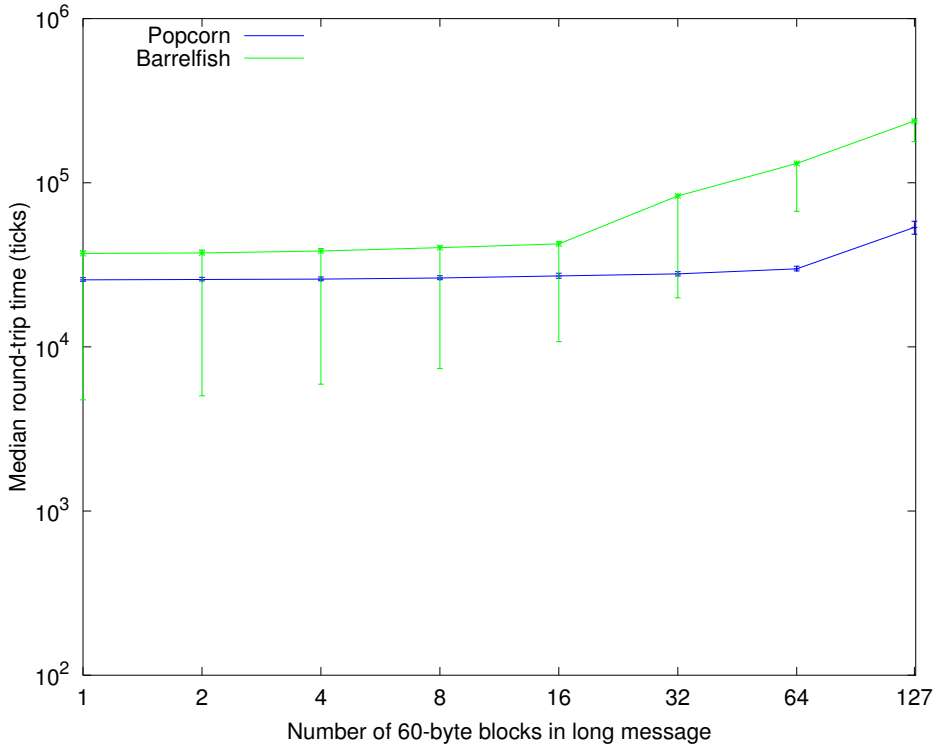

Figure 6.10: Barrelfish vs. Popcorn round-trip time comparison, same NUMA node (CPU 0 to CPU 2)

than Barrelfish is. For same-NUMA-node messaging (Figure 6.7), Popcorn performs better for messages up to size 64 blocks. For same-die messaging (Figure 6.8), Popcorn performs better for messages up to size 4 blocks. For different-die messaging (Figure 6.9), Barrelfish performs better than Popcorn for all message sizes.

- For Barrelfish, the send time flattens out for messages of size 32 and greater.

Batch Message Round-Trip Time

Comparisons of the round-trip time for different batch sizes are shown in Figures 6.10, 6.11, and 6.12.

We can make the following observations about this data:

- Again, Barrelfish shows more variation than Popcorn, which is very consistent. While Barrelfish's best-case performance beats Popcorn at all message sizes below 32 cache lines, Popcorn's median round-trip time beats Barrelfish's median RTT at all message

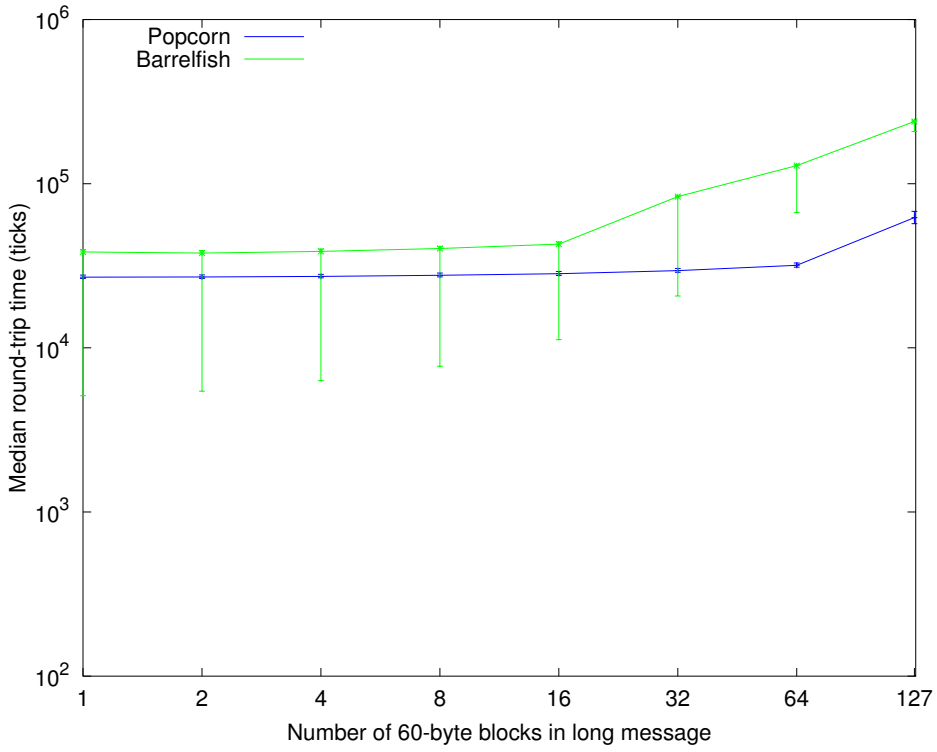

Figure 6.11: Barrelfish vs. Popcorn round-trip time comparison, same die (CPU 0 to CPU 10)

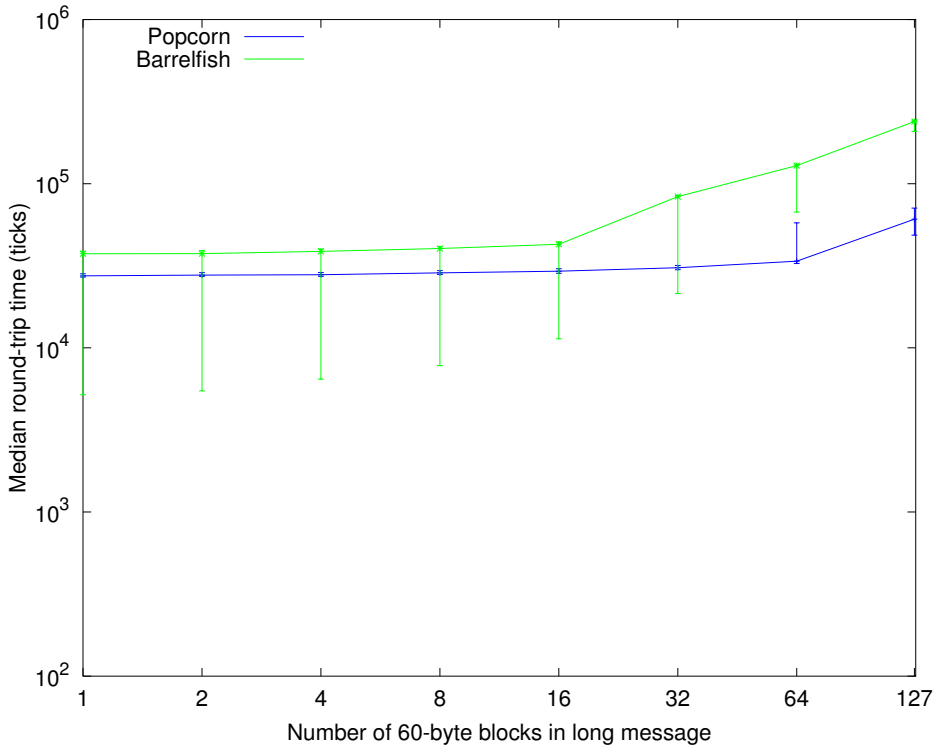

Figure 6.12: Barrelfish vs. Popcorn round-trip time comparison, different die (CPU 0 to CPU 48)

sizes. This result may be due to the fact that Popcorn does not default to polling like Barrelfish does, and thus always provides whatever latency IPI provides, plus some amount of overhead. In contrast, if Barrelfish's initial polling succeeds and it does not have to fall back to IPI-based notification, it can deliver lower latency than IPI. Note that Barrelfish also has additional overhead in translating a binding object to a messaging channel, and perhaps also in transitioning from userspace to kernel space if necessary (e.g. to send IPI).

- For messages of 32 cache lines and greater, the median round-trip time for Barrelfish increases significantly. Coupled with the fact that the median time-to-send flattens out at around the same point, it appears that Barrelfish has made an optimization for messages above a certain size that trades longer round-trip latency for reduced cost to the sender (and potentially lower total cost overall). We leave further measurements in this area, as well as source-level analysis, as future work.

6.5 Process Migration

Note that the work in this section was done collaboratively: David Katz implemented process migration, and Antonio Barbalace tested it and created the plots shown.

As mentioned earlier, much of the interesting work in the multikernel OS space is that which relates to providing a coherent state and a single system image across kernels. This work serves as motivation for the design of our kernel messaging layer. As a test case for this architecture and this messaging layer, a prototype version of process migration was implemented using the kernel messaging layer.

When a process needs to be migrated, the prototype functions as follows:

- First, the process' execution is stopped.
- Next, all the process' memory descriptors (its virtual-to-physical mappings and its memory-area-to-file-area mappings) are marshaled, packed, and sent via the kernel messaging layer to the remote kernel instance to which the process is being migrated.
- Finally, after the remote kernel receives all the messages containing the marshaled and packed descriptors, it reconstructs the process' address space and resumes execution. Note that the process' memory content is not copied between kernel instances; instead, the remote kernels maps in the process' physical address space from the source kernel.

We measured the time required to migrate a process between a kernel instance on CPU 0 and a kernel instance on CPU 2. The results are shown below. Figure 6.13 shows the non-messaging overhead to restart a process on the destination kernel instance after its mappings

have been transferred from the source kernel instance, and Figure 6.14 shows the proportion of the total migration time involved in messaging.

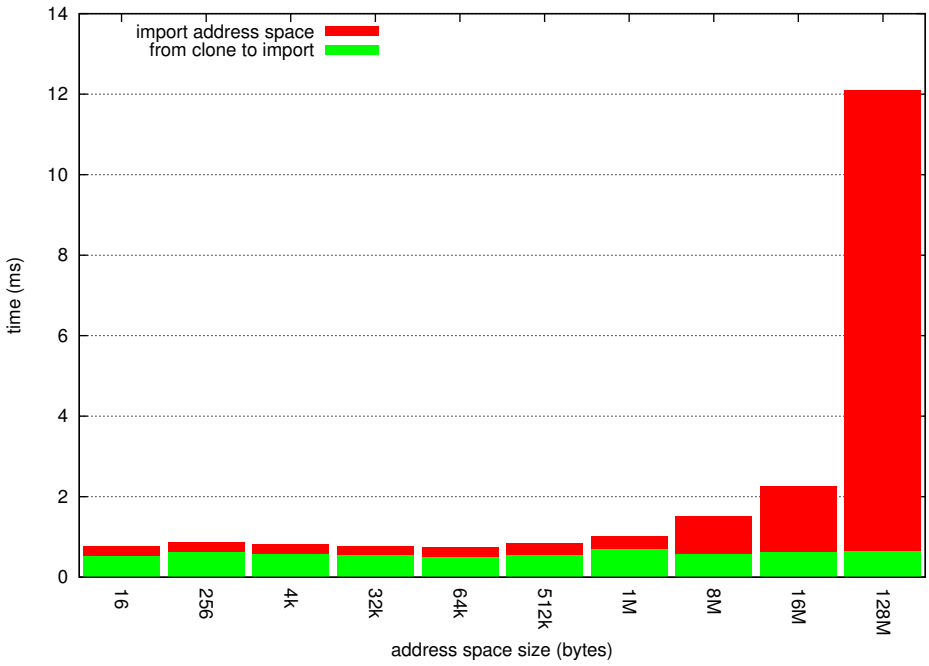

Figure 6.13: Process migration: overhead to restart process after messaging

We can draw the following conclusions from this data:

- In Figure 6.13, we see that the non-messaging overhead in process migration is split into two parts: one from clone to import, which remains relatively constant as the process' address space size increases; and one when the address space is imported, which is made up of some fixed overhead plus some cost that increases roughly linearly with the process' address space size.

- The larger a process' address space, the more memory descriptors that must be sent between kernels for process migration; we see that in the data.

- It takes more than a second to import a 128 MB address space, performance which is clearly far worse than what the hardware is capable of providing.

- It is apparent that at larger address space sizes, the majority of the migration time is spent in messaging. While the performance in this case is not terrible, it suffers for several reasons. First, the kernel messaging layer is designed for passing small control and synchronization messages, not for moving around huge regions of data. Second, the implementation suffers from the copy-in, copy-out problem described in Section 2.2.1. An optimized approach to address both of these issues would be to use the

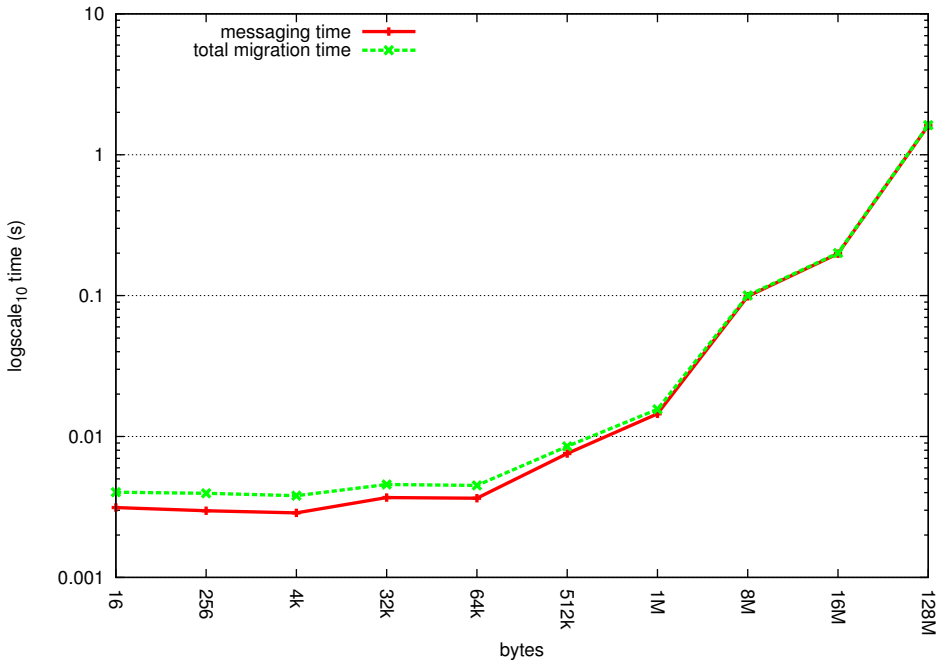

Figure 6.14: Process migration: comparison between messaging time and total migration time

messaging layer to perform coordination between two kernel instances so the source can copy the data directly into the target's physical address space. We believe that such functionality is likely to be required for many such cross-kernel-instance features, and we suggest that it would make sense to provide it within the messaging layer to avoid duplication; we leave this as future work.

Chapter 7

Conclusions

In this thesis, we presented Popcorn, a Linux-based multikernel operating system.

We demonstrated how the existing architecture of the Linux kernel can be repurposed to meet the design goals of a multikernel operating system, demonstrating a shared-memory network tunnel and a unicast and multicast messaging framework built on this design principle. We presented results for process migration between kernels to demonstrate the functionality, performance, and robustness of our solution.

Based on our results, analysis, and overall experience, we can draw several useful conclusions:

The Popcorn system is stable, self-hosting, and useful as a platform for further experiments. We can reliably launch 64 kernel instances on a 64-core machine. After these kernel instances are launched, the user can interact with each of them independently via virtual TTY, via SSH, or over the network. Popcorn is self-hosting; it is possible to compile a Popcorn kernel on any running kernel instance. When running Popcorn within QEMU, the user can use GDB to step through new kernel code.

Linux behaves well in a multikernel context. Specifically, Popcorn relies on two properties of the Linux kernel at a basic level: the *mem=* and *memmap=* kernel arguments, and the support for kernel relocation throughout the physical address space. We found both of these properties to be robust. Even when loading and running kernel instances at high physical addresses, something that is rarely (if ever) tested on x86 machines, we found that the system was exceptionally well-behaved. We saw no instances of kernel instances attempting to access memory outside their assigned range, and the vast majority of the architecture-specific code for x86 (and all of the platform-independent code) was 64-bit clean and required no modification.

Our ring-buffer-based approach works well for sharing a network device and for handling large messages between kernel instances. For data transfer with packets of 1500-byte size (per the Ethernet MTU), the packets are long enough that the time spent in synchronization, notification, and other overheads is reasonable compared to the time spent actually moving data. For large messages, these overheads are also mitigated sufficiently to allow good performance.

There is significant overhead inherent in small messages between kernels; it would make sense to provide a framework for zero-copy transfer of large regions of data between kernels. As shown in Section 6.5, a significant fraction of the process migration time with a naive, single-message-based approach is spent in messaging. We suspect that as Popcorn development proceeds, there will be other features that require transfer of large contiguous regions of data between kernels, and as such, it makes sense to provide this functionality as an OS-level primitive rather than require that the individual features that need it implement it themselves.

In terms of scalability, the cost of IPI-based notification becomes prohibitively high for multicast groups with many members. As shown in Section 6.4.3, while the time required to insert a message into a multicast group's window remains nearly constant as the number of group members increases, the time required to send an IPI to each member of the group increases linearly with the number of group members. This finding concurs with the finding in the Barrelfish paper [5] for why IPI-based TLB shootdown does not scale to many-core systems. However, the tree-based broadcast solution the Barrelfish authors proposed does not fit cleanly into the Linux-as-a-multikernel design concept; it relies on each dispatcher to provide low-latency polling in a way that the Linux kernel is not built to deliver. We believe that this issue would likely be solved by better hardware support for IPI broadcast in high-core-count systems (e.g. Intel x2APIC, as discussed in Section 3.2.3), or by future hardware support for explicit message passing between cores.

7.1 Contributions

To recap, we made the following research contributions in this thesis:

- We detailed the modifications to the Linux kernel that are necessary to boot and run multiple kernel instances anywhere within the physical address space. To our knowledge, we are the first group to accomplish this on a 64-bit machine and to support more than 4 kernel instances.
- We discussed the design and implementation of a shared-memory network tunnel for sharing a hardware network interface between kernels. Through TCP and web server

benchmarks, we established that it performs well and is able to share the full bandwidth of a 1 Gbit/s link.

- We documented the design and implementation of an efficient inter-kernel messaging layer. Through microbenchmarks, we differentiated between the compulsory hardware costs and the implementation overhead, and we showed that our implementation added less than X percent to what the Linux kernel provides. To exercise the messaging layer in a realistic manner, we demonstrated its use in providing process migration across kernel instances.

- We have released our source code so that others might build upon our efforts.

Chapter 8

Future Work

8.1 Open Bugs and Unfinished Features

There are several pieces of work that still need to be addressed:

- There is a bug where the hardware Ethernet link on *gigi* (Intel 82576) will reset itself when passing high amounts of traffic from a test machine to a secondary kernel. When this bug occurs, the *eth0* interface is brought down and cannot be brought back up again. However, the *shmtun* interface from the primary kernel to the secondary kernels stays up and continues to function normally. We are unsure whether this is a bug in Popcorn's network driver or a timeout bug in the Linux driver for the Ethernet interface.
- Although creating and sending to multicast groups is supported and works properly, dynamically adding/removing kernel instances to/from multicast groups does not yet work, nor does closing multicast groups. While not terribly difficult from a technical perspective, these features are intricate in that they require careful design and execution to ensure that the steps required occur in the correct order, and that any messages resident in a multicast group's queue when the group is changed or closed are handled properly.
- Although Popcorn supports clustering, neither the shared-memory network tunnel nor the kernel messaging layer work with it. The first step towards getting these features to work with clustering would be to augment the APIC/LAPIC support to enable each cluster to be addressed uniquely, and to provide APIs that allow these unique cluster IDs to be used in the same manner in which Popcorn currently uses the logical CPU ID.

8.2 Further Evaluation

It would be worthwhile to evaluate the performance of the lock-free queues used in the unicast and multicast messaging implementation under contention. Ideally, this evaluation would consider a simple case (two kernel instances simultaneously sending at maximum capacity to a single kernel instance or to a single multicast group), and also a real-world case as would occur in creating a process that spans multiple kernel instances. Based on the properties of these algorithms, we would expect good performance. However, there may be overheads in terms of synchronization (e.g. locking the bus to perform atomic operations across cores and NUMA nodes) or cache behavior that we have not considered.

It would also be worthwhile to evaluate how the messaging layers of Popcorn and Barrelfish perform under higher contention (e.g. without a delay between sending messages, or with varying amounts of delay). Per [5], it seems that some of the optimizations that Barrelfish has made are to speed up messaging for large numbers of back-to-back messages, and it would be interesting to see how well they succeed at that goal compared to Popcorn's more naive approach.

8.3 OS Work

There is a significant amount of work still needed to support a single system image on top of a distributed OS, much of which is being actively done by another group member, Dave Katz. Specifically, we see non-trivial challenges in the following areas:

- Creating, synchronizing, and terminating threads and processes remotely
- Managing page tables across cores
- Handling libraries – are they shared across kernels, or is there one copy per kernel?
- Handling state that is shared by convention or by standard (e.g. file descriptors)
- Scheduling – who runs on which cores, and how is this made NUMA-aware?

In addition, we do not deal with heterogeneity directly; that work is in progress. A port of Popcorn to Tilera is underway. Additionally, other members of the group are working on ways to compile traditional shared-memory programs to run on heterogeneous architectures, either by converting them to use message passing or by packing binaries for multiple architectures into a single executable.

Finally, there are features on new versions of the x86 architecture that we do not yet support. Specifically, the clustered x2APIC architecture available on the newest Intel Core i7 chips

may ameliorate some of the issues we saw in Section 6.4.3 with regard to multicast messaging scalability. It would be interesting to adapt Popcorn to take advantage of these new features and see if does in fact improve performance.

8.4 Network Tunnel

8.4.1 Alternative Approaches

Although the method for sharing the network interface between kernels outlined in Section 4.2 meets the design goals and makes efficient use of elements that already exist in Linux, it may not be the best possible solution. During the initial design process, several alternative approaches were discussed that may merit evaluation:

- Xen uses a zero-copy, page-flipping based approach for sharing network hardware among virtual machines [14]. To send a packet from the network interface to a VM, the hypervisor writes the packet into a physical page, maps that page into the virtual address space of the VM, and notifies the VM that the packet has arrived.
- Some server-class network cards support the SR-IOV standard, which enables them to appear as a separate PCI device to each one of multiple virtual machines running on the same box. This approach eliminates the overhead of providing a 'virtual switch' in software. The NoHype virtualization solution uses SR-IOV plus IO-MMU to deliver an efficient solution [31].
- The FlexSC/FOS approach would suggest that it might be more efficient cache-wise to run networking as a userspace process on a dedicated core; other processes would pass userspace messages to and from this process to access networking resources.
- Dr. Back proposed that it might be advantageous to pass control of the physical network device between kernels, rather than having the device owned by a single kernel and passing packets to and from that kernel.

8.4.2 Modeling and Optimization

As discussed in Section 6.3.2, it would be useful to create a simple model to predict the effectiveness of interrupt mitigation. Such a model would take as inputs the NUMA nodes of the sender and receiver, the rate and burstiness of the traffic carried over the link, and the value of the NAPI *weight* parameter, and provide as an output the expected ratio of interrupts generated to packets transferred. Formulating such a model would give us a better

conceptual understanding of how each input affects interrupt mitigation, and would enable us to optimize the NAPI parameters for a given link.

In addition, there is likely still room for improvement both in terms of cache performance and in terms of efficiently handling the common case when only one kernel is passing traffic over the interface (e.g. avoiding unnecessary overhead from polling kernel instances that are idle).

8.5 Messaging

It would be interesting to see how our messaging design could be adapted to architectures that support hardware message passing at the user level (e.g. Tilera, Intel SCC).

Bibliography

[1] G. Almasi, R. Bellofatto, J. Brunheroto, C. Caşcaval, J.G. Castanos, P. Crumley, C.C. Erway, D. Lieber, X. Martorell, J.E. Moreira, et al. An overview of the BlueGene/L system software organization. *Parallel processing letters*, 13(04):561–574, 2003.

[2] Dan Aloni. Cooperative Linux. In *Proceedings of the Linux Symposium*, volume 2, pages 23–31, 2004.

[3] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[4] A. Baumann. Non-cache-coherent systems: The Barrelfish experience. Systems for Future Multi-Core Architectures (SFMA'11), 2011.

[5] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

[6] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–12. IEEE, 2006.

[7] A.M. Belay. Message passing in a factored OS. Master's thesis, Massachusetts Institute of Technology, 2011.

[8] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, et al. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 43–57, 2008.

[9] S. Boyd-Wickizer, A.T. Clements, Y. Mao, A. Pesterev, M.F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. 2010.

[10] S. Boyd-Wickizer, M.F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2012.

[11] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[12] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 10–pp. IEEE, 2006.

[13] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 29, pages 12–25. ACM, 1995.

[14] L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. In *Proceedings of the USENIX annual technical conference*, pages 387–390, 2005.

[15] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers*. O'Reilly Media, Incorporated, 2005.

[16] Tilera Corporation. TILE-Gx processor family.

[17] Advanced Micro Devices. AMD-V nested paging. July 2008.

[18] Andy Glew. Monitor-mwait, November 2010.

[19] B. Goglin, S. Moreaud, et al. KNEM: a generic and scalable kernel-assisted intra-node MPI communication framework. *Journal of Parallel and Distributed Computing*, 2012.

[20] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 33, pages 154–169. ACM, 1999.

[21] 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. IEEE, 2011.

[22] Intel. 8259a programmable interrupt controller. December 1988.

[23] Intel. Multiprocessor specification version 1.4. May 1997.

[24] Intel. An introduction to the Intel QuickPath Interconnect. 2009.

[25] Intel. Intel 64 architecture x2APIC specification. March 2010.

[26] Intel. The Intel Xeon Phi coprocessor 5110p: Highly-parallel processing for unparalleled discovery. November 2012.

[27] Intel. Intel 64 and IA-32 architectures software developers manual. August 2012.

[28] K. Iskra, J.W. Romein, K. Yoshii, and P. Beckman. ZOID: I/O-forwarding infrastructure for petascale architectures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 153–162. ACM, 2008.

[29] Adhiraj Joshi, Swapnil Pimpale, Mandar Naik, Swapnil Rathi, and Kiran Pawar. TwinLinux: running independent Linux kernels simultaneously on separate cores of a multicore system. pages 101–108, 2010.

[30] Xiaodi Ke. Interprocess communication mechanisms with Inter-Virtual machine shared memory. Master's thesis, University of Alberta, August 2011.

[31] E. Keller, J. Szefer, J. Rexford, and R.B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 350–361. ACM, 2010.

[32] Kernel.org. Linux version 3.2.14 source code, 2012.

[33] Argonne National Laboratory. The message passing interface (mpi) standard.

[34] Linux-kvm.org. Setting guest network, September 2011.

[35] LMAX. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads, May 2011.

[36] J.P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Work in progress in the Symposium on Operating Systems Principles, SOSP*, volume 11, 2011.

[37] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J.M. Squyres, and J.J. Dongarra. Kernel assisted collective intra-node MPI communication among multi-core and many-core CPUs. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 532–541. IEEE, 2011.

[38] M. Maas and R. McIlroy. A JVM for the Barrelfish operating system.

[39] Milo MK Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78–89, 2012.

[40] J.C. Mogul, A. Baumann, T. Roscoe, and L. Soares. Mind the gap: reconnecting architecture and OS research. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, pages 1–1. USENIX Association, 2011.

[41] Yoshinari Nomura, Ryota Senzaki, Daiki Nakahara, Hiroshi Ushio, Tetsuya Kataoka, and Hideo Taniguchi. Mint: Booting multiple Linux kernels on a multicore processor. pages 555–560. IEEE, October 2011.

[42] S. Pellegrini, T. Hoefler, and T. Fahringer. On the effects of CPU caches on MPI point-to-point communications.

[43] S. Peter, A. Schpbach, D. Menzi, and T. Roscoe. Early experience with the Barrelfish OS and the single-chip cloud computer. In *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium (MARC), Ettlingen, Germany*, 2011.

[44] D.E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G.C. Hunt. Rethinking the library OS from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304. ACM, 2011.

[45] Jan Sacha, Henning Schild, Jeff Napper, Noah Evans, and Sape Mullender. Message passing and scheduling in osprey.

[46] Ben Shelton, Antonio Barbalace, Alastair Murray, and Binoy Ravindran. Towards operating system support for heterogeneous-ISA platforms. In *Proceedings of the 6th International Systems and Storage Conference (submitted)*, 2013.

[47] Taku Shimosawa and Yutaka Ishikawa. Inter-kernel communication between multiple kernels on multicore machines. *IPSJ Online Transactions*, 2:261–279, 2009.

[48] L. Soares and M. Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–8. USENIX Association, 2010.

[49] Ashley Stevens. Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7. 2011.

[50] A. Trivedi, A. Schüpbach, A. Baumann, and T. Roscoe. Hotplug in a multikernel operating system. Master's thesis, ETH Zurich, Systems Group, Department of Computer Science, 2009.

[51] Michael von Tessin. The clustered multikernel: An approach to formal verification of multiprocessor OS kernels. 2012.

[52] D. Wallace. Compute Node Linux: Overview, progress to date, and roadmap. In *Proceedings of the 2007 Cray User Group Annual Technical Conference*, 2007.

[53] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[54] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.C. Miao, J.F. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, 2007.

[55] Matthew Wilcox. Ill do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers.

[56] ETH Zurich. Barrelfish release 2013-03-22, 2013.