DISS.ETH NO. 20664

Resource Management in a Multicore Operating System

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by
SIMON PETER
Diplom-Informatiker, Carl-von-Ossietzky Universität Oldenburg
born October 13, 1981
citizen of Federal Republic of Germany

accepted on the recommendation of

Prof. Timothy Roscoe, examiner
Prof. Steven Hand, co-examiner
Prof. Gustavo Alonso, co-examiner
Prof. Markus Püschel, co-examiner

2012

Abstract

Trends in hardware design have led to processors with many cores on a single die, which present the opportunity for commodity computing to become increasingly parallel. These *multicore* architectures bring with them complex memory and cache hierarchies and processor interconnects. While the execution of batch parallel applications has been researched in the context of high-performance computing (HPC), commodity hardware is evolving at a faster pace than specialized supercomputers and applications are interactive, requiring fast system response times and the ability to react to ad-hoc workload changes. Leveraging and managing the existing potential for parallelization thus presents a difficult challenge for the development of both commodity operating systems and application programs, which have to keep up with hardware developments and present nimble solutions.

This dissertation presents the design and implementation of operating system mechanisms to support the execution of a dynamic mix of interactive and parallel applications on commodity multicore computers. The main goals are to provide a system that is scalable with an increasing number of processor cores, is agile with a changing hardware architecture, and provides interactive response time to the user when running a mix of parallel, interactive applications.

I describe a new operating system architecture, the *Multikernel*, and report about a concrete implementation of it, called *Barrelfish*. The Multikernel takes a novel view on the underlying hardware architecture: as a network of autonomous, heterogeneous processors. The Barrelfish operating system is structured as a distributed system of servers to facilitate easy restructuring of OS services according to the hardware architecture. Applying techniques from the field of distributed systems, Barrelfish demonstrates that the operating system can scale and perform equally well as manually tuned operating systems, like Linux, and in some cases better, while remaining agile with a range of different multicore systems.

I present the design and implementation of the inter-process communication system and process scheduler within Barrelfish and show how it can be made scalable and agile by applying the Multikernel design principles. Finally, I apply the gang scheduling technique from HPC and show how it can be made nimble to support interactive response times via a novel technique called *phase-locked scheduling* to support scheduling a dynamic mix of parallel, interactive applications.

Zusammenfassung

Hardwaredesign Trends haben zu Prozessoren mit vielen Kernen auf einem einzelnen Chip geführt, welches der elektronischen Datenverarbeitung ermöglicht, paralleler zu werden. Diese *multicore* Architekturen ziehen komplexe Speicher und Cache Hierarchien, sowie Prozessorverbindungen mit sich. Während die Ausführung von parallelen Stapelverarbeitungsanwendungen im Kontext von Hochleistungsrechnern gut erforscht ist, entwickelt sich Massenhardware schneller als spezialisierte Supercomputer und die Anwendungen sind interaktiv, was schnelle Systemantwortzeiten und die Fähigkeit, schnell auf ad-hoc Änderungen der Arbeitslast zu reagieren, voraussetzt. Die Ausnutzung und Verwaltung des existierenden Parallelisierungspotenzials stellt daher ein schwieriges Problem für die Entwicklung sowohl von Massenbetriebssystemen, als auch Anwendungsprogrammen dar, die mit der Hardwareentwicklung mithalten und flinke Lösungen anbieten müssen.

Diese Dissertation stellt das Design und die Implementierung von Betriebssystemmechanismen vor, welche die Ausführung einer dynamischen Mischung von interaktiven und parallelen Anwendungen auf multicore Massencomputern unterstützt. Die Hauptziele sind ein System bereit zu stellen, welches mit einer steigenden Anzahl an Prozessorkernen skaliert, Agilität gegenüber einer sich verändernden Hardwarearchitektur bietet und dem Anwender interaktive Antwortzeiten bereitstellt, während es eine Mischung aus parallelen und interaktiven Anwendungen ausführt.

Ich beschreibe eine neue Betriebssystemarchitektur, den *Multikernel*, und berichte über eine konkrete Implementierung derselben, genannt *Barrelfish*. Der Multikernel nimmt eine neue Sicht auf die darunterliegende Hardwarearchitektur: Als Netzwerk autonomer, heterogener Prozessoren. Das Barrelfish Betriebssystem ist strukturiert als verteiltes System von Servern, welches ein leichtes Restrukturieren von Betriebssystemservices anhand der Hardwarearchitektur ermöglicht. Dabei wendet Barrelfish Techniken aus dem Bereich der verteilten Systeme an und demonstriert, daß das Betriebssytem ebenso gut skalieren und funktionieren kann, wie manuell auf die Hardware abgestimmte Betriebssysteme, wie zum Beispiel Linux, und in einigen Fällen sogar besser, während es agil gegenüeber einer Anzahl verschiedener Multicoresysteme bleibt.

Ich stelle das Design und die Implementierung des Interprozesskommunikationssystems und des Prozess-Schedulers vor und zeige wie

beide Systeme skalierbar und agil gemacht werden können, indem die Designprinzipien des Multikernels angewandt werden. Schließlich wende ich die Gang-Scheduling Technik aus dem Feld des Hochleistungsrechnens an und zeige, wie es durch eine neue Technik, genannt *phase-locked scheduling*, flink gemacht werden kann, so dass es interaktive Antwortzeiten beim Scheduling einer dynamischen Mischung aus parallelen und interaktiven Anwendungen unterstützt.

Acknowledgements

I would like to thank my supervisor Timothy Roscoe for all his help, patience, support and advice, and the entire Barrelfish team, at ETH and Microsoft, for all the insightful discussions, around the whiteboard, in corridors, and elsewhere, in particular Paul Barham, Andrew Baumann, Pierre-Evariste Dagand, Jana Giceva, Tim Harris, Rebecca Isaacs, Ross McIlroy, Adrian Schüpbach, and Akhilesh Singhania.

Also, I am heavily indebted to the authors of the work that this dissertation builds upon, namely the Barrelfish team, interns, students, and collaborators. Without them, this dissertation would not exist. In addition, I would like to acknowledge my examination committee for the detailed feedback on numerous drafts of this document.

I would like to thank the researchers, staff, and interns at the Microsoft Research labs in Cambridge and Silicon Valley for the incredibly fun and rewarding time.

Finally, I would like to thank my parents for their patience and support through all the years, as well as Akhilesh Singhania, for being a very good and supporting friend in good and in tough times. Thanks for making all of this so much more worthwhile!

Zurich, August 2012.

Contents

Chapter 1

Introduction

This dissertation presents the design and implementation of an operating system and its mechanisms to support the execution of a dynamic mix of interactive and parallel applications on commodity multicore computers.

Our goal is to provide a system that

1. is *scalable* with an increasing number of processor cores,
2. is *agile* with a changing hardware architecture, that is, capable of adapting to the increasing diversity of hardware platforms and exploiting specific hardware features when available without sacrificing a clear structure of the system in the interest of performance, and
3. provides *interactive* response time to the user when running a *mix* of *parallel* and *interactive* applications.

During the research for this dissertation, a prototype of this operating system was created. I use this prototype to argue the following thesis:

> An operating system can achieve the goals of scalability and agility on a multicore computer by making all inter-core com-

munication explicit, not assuming any specific hardware features, such as shared memory, cache coherence, or a specific memory interconnect topology, and by viewing state as replicated instead of shared.

Furthermore, it can support interactive application response times on a commodity multicore computer by employing inter-process communication and scheduling components that multiplex hardware efficiently in time and space, react quickly to ad-hoc workload changes, reason online about the underlying hardware architecture, reason online about each application, and allow resource negotiation between applications and the operating system.

The set of applications that benefit from these ideas is very diverse and ranges from traditional throughput oriented task-parallel workloads, such as web-servers and databases, to emerging *recognition, mining, and synthesis* (RMS) desktop workloads [Dub05, LV03].

1.1 Motivation

As hardware manufacturers reach the processing speed boundaries of what is physically possible using traditional hardware architecture, computing becomes increasingly parallel in order to keep up with the demands for more processing power. On the hardware side, this is achieved by integrating a growing number of processor cores into processor dies. Machines with multiple cores on a die are called *multicore* machines. Leveraging and managing the existing hardware potential for parallelization presents a difficult challenge for the development of software systems, as it brings the issue of scalability with it. In this case, *scalability* is the ability of software to yield execution speedup proportional to the number of available processing cores.

In particular, operating systems are an important part of the overall software stack. Many applications invoke operating system services to help carry out operations that must scale. Thus, it is important that the services presented by the operating system are themselves scalable in order not to prevent scalability of the rest of the software stack.

A fundamental problem of any operating system is the provision and multiplexing of hardware resources to user-space applications. Resource management primitives can be invoked frequently by commodity applications. For example, several file descriptor allocations and deallocations are typically carried out by network applications when handling each new client connection and has been shown to become a scalability bottleneck when invoked frequently [BWCC+08].

There are many other published scalability shortcomings pertaining to operating system resource management in multicore systems. For example, mis-scheduling of parallel applications [PSB+10], contention on spin-then-block locks [JSAM10], and lock granularity of CPU dispatch [BBD+09]. Many of these are symptoms of a more general problem of classical shared-memory operating system architecture, which has been shown by many research publications of the field (covered in Section 2.2) to be suboptimal in light of hardware performance possibilities. At a high-level, this is due to three factors:

1. **Lock contention.** Shared-memory data structures are typically protected by some means of mutual exclusion, such as locks. Increasing the number of execution contexts trying to access the same data structure results in contention, which slows down overall progress and becomes a scalability bottleneck.

2. **Cache coherence.** Main memory access latencies have become so slow in comparison to processor speeds that adequate performance can only be achieved if special cache memories are employed that are closer to processor cores. This results in an architecture where caches are core-local and data shared by multiple cores has to be kept coherent among their caches. This is typically achieved by a hardware-implemented cache-coherence protocol that communicates among caches by passing messages on a special high-speed memory interconnect.

   I will show later in this dissertation that naive usage of hardware cache coherence can become a scalability bottleneck. This is due to the mere nature of sharing data across a network of individual cache memories. If cores have to go to remote cache memory frequently in order to fetch or update a datum, this results in higher memory access latency.

It is thus important that data is placed local to the core operating on it and not moved to other caches often.

3. **Abstraction.** Classical operating system application interfaces, such as POSIX, abstract away many of the properties of multicore hardware. Instead, the illusion of a multiprocessor with a single shared memory is upheld. This abstraction causes overheads, which, due to cache coherence, starts already at the hardware level.

Looking ahead, as more and more cores are added, it is not clear whether hardware cache coherence itself can scale indefinitely [BWCC+08, BPS+09]. Hardware designers already consider alternative, non-cache-coherent architectures. For example, both the Intel Single-Chip Cloud Computer (SCC) [HDH+10] and Microsoft's Beehive processor [Tha10] have been designed without hardware cache-coherence.

At the same time, multicore hardware is becoming more diverse [SPB+08]. For example, Intel's QuickPath [Int08] and AMD's HyperTransport [Hyp04] cache coherence interconnects present two very different ways to structure the hardware architecture to route cache coherence messages: one is a ring, the other a mesh. In addition, hardware is evolving fast: different cache hierarchies, non-uniform memory architectures, and specialized accelerator cores, such as graphics accelerators, are being integrated into multicore platforms. All of these innovations have happened in the past decade. I am going to investigate them and their impact on operating systems more closely in Section 2.1.1.

Complex commodity operating systems are faced with an increasingly diverse range of hardware and modifying them to support this diversity is a difficult task. For example, removing the Windows dispatcher lock to improve dispatch scalability was described as a "heroic" effort that involved the modification of 6000 lines of code in 58 files of the Windows source code to cope with feature interactions [Rus09].

Given their fundamental importance and the amount of problems they face, it makes sense to investigate the very techniques and mechanisms that are employed in operating systems for commodity multicore computers.

1.2 Contribution

In this dissertation, I make three principal contributions:

1. I describe an operating system architecture, termed the *Multikernel*, that is scalable and agile. This is achieved via a novel view on the underlying hardware architecture: as a network of autonomous processors. This view supports a programming paradigm requiring a very clear separation of data that completely avoids sharing and its associated problems of locking and cache-coherence and replaces it with explicit message-passing-based communication among cores. Operating system services are structured as distributed systems placed on top of these networks and are malleable to the underlying hardware characteristics, such as communication latency and connectivity among processors.

2. I describe the components necessary to support the execution of a dynamic mix of interactive and parallel applications within a Multikernel-based operating system, namely mechanisms for low-latency communication between mutually distrusting processes, both on the same and among separate processor cores, and a process scheduler that facilitates the scheduling of a mix of parallel and interactive applications via an interface that allows more insight into these applications' scheduling requirements.

3. I describe and evaluate a novel scheduling mechanism called *phase-locked gang scheduling*. This mechanism builds on the previously described components to allow for interactive system response when scheduling applications that are both *parallel and interactive*.

Via a series of benchmarks, I demonstrate that Barrelfish can scale and perform equally well as manually tuned operating systems, while remaining agile with a range of different cache-coherent and non-cache-coherent multicore systems. Also, I demonstrate response times below 5 seconds for a hypothetical future parallel interactive network monitoring application. Response times of these short durations provide almost immediate feedback to users and enable a class of compute-intensive problems to be treated by

interactive applications using parallel algorithms. This provides users with new, interactive ways to investigate and treat this class of problems.

1.3 Dissertation Structure

This dissertation is structured such that background literature, and related work relevant to a topic are placed within the chapters that treat the topic, instead of providing a single overarching chapter for related work. I have motivated the thesis at a high level within this chapter. I am going to iterate the motivation in a topical fashion in the following chapters, providing more detail.

The Multikernel structure, as well as relevant features of its implementation, Barrelfish, are described in Chapter 2. In particular, the inter-domain message passing subsystem, process structure, and protection model are described. The Multikernel operating system architecture is structured so as to support scalable and agile implementations of operating system services designed for present and future multicore hardware architectures.

In Chapter 3, I describe the design and implementation of the scheduling subsystem of Barrelfish that facilitates the scheduling of a mix of parallel and interactive applications in a commodity multicore computing scenario, by allowing applications to specify detailed scheduling requirements and by scheduling via a novel mechanism termed *phase-locked gang scheduling*.

In Chapter 4, I evaluate the scalability and agility of the scheduler implementation. I also evaluate how well phase-locked gang scheduling compares to traditional gang scheduling solutions in providing low overhead and short response time to parallel applications.

Finally, Chapter 5 concludes the dissertation, by summarizing and providing an outlook to future research.

1.4 Related Publications

The work presented in this dissertation constitutes a part of the Barrelfish operating system research project and as such is depending on, as well as supporting, the work of others. Some of the work this dissertation depends upon has been published and is mentioned here for reference:

[Dag09] Pierre-Evariste Dagand. Language Support for Reliable Operating Systems. Master's thesis, ENS Cachan-Bretagne – University of Rennes, France, June 2009.

[DBR09] Pierre-Evariste Dagand, Andrew Baumann, and Timothy Roscoe. Filet-o-Fish: practical and dependable domain-specific languages for OS development. In *Proceedings of the 5th Workshop on Programming Languages and Operating Systems*, October 2009.

[Raz10] Kaveh Razavi. Barrelfish Networking Architecture. Distributed Systems Lab, ETH Zurich, 2010.

[Men11] Dominik Menzi. Support for heterogeneous cores for Barrelfish. Master's thesis, ETH Zurich, July 2011.

[Ger12] Simon Gerber. Virtual Memory in a Multikernel. Master's thesis, ETH Zurich, May 2012.

[Nev12] Mark Nevill. Capabilities subsystem. Master's thesis, ETH Zurich, 2012.

[Zel12] Gerd Zellweger. Unifying Synchronization and Events in a Multicore Operating System. Master's thesis, ETH Zurich, March 2012.

Some of the work presented in this dissertation has previously been published in various venues. The relevant publications are:

[SPB+08] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.

[BPS+09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[PSB+10] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd Workshop on Hot Topics in Parallelism*, June 2010.

[PSMR11] Simon Peter, Adrian Schüpbach, Dominik Menzi, Timothy Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium*, July 2011.

[BPR+11] Andrew Baumann, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. Barrelfish Specification, ETH Zurich, July 2011.

[PRB10] Simon Peter, Timothy Roscoe, and Andrew Baumann. Barrelfish on the Intel Single-Chip Cloud Computer. Barrelfish Technical Note 005, ETH Zurich, September 2010.

Chapter 2

The Multikernel

This chapter describes the Multikernel architecture, as well as one concrete implementation, Barrelfish. Being agile with diverse multicore system architectures, while providing a model that allows seamless operating system scalability are the main design goals.

The Multikernel is based on message passing. Nimble inter-process communication primitives, especially among cores, are a requirement for the efficient, scalable execution of applications on top of a Multikernel operating system implementation. This chapter describes and evaluates the design and implementation of these primitives in Barrelfish. The Barrelfish implementation represents one point in the Multikernel design space and is certainly not the only way to build a Multikernel.

This chapter is structured as follows:

Section 2.1 motivates the Multikernel. Hardware diversity, processor interconnect latency, the performance of message passing versus shared memory, and cache coherence protocol scalability are discussed.

Section 2.2 provides background on past multiprocessor and message-passing-based computer architectures and the operating systems thereon.

Section 2.3 describes the hardware platforms used to evaluate Barrelfish throughout this dissertation.

Section 2.4 describes the Multikernel architecture, its design principles, and the resulting implications for concrete implementations.

Section 2.5 describes the actual choices made in the implementation, Barrelfish, and explains which choices are derived directly from the model described here and which are in place due to other reasons.

Section 2.6 describes and evaluates the design and implementation of the inter-domain communication primitives, both on the same and across distinct cores.

Section 2.7 surveys related work relevant to the Multikernel design and message passing primitives.

Section 2.8 summarizes this chapter.

2.1 Motivation

In the desktop and server space—the typical domain of general purpose operating systems—uniprocessors and multiprocessors consisting of just a few processors were the norm for a long time and scalability or platform diversity were easy to solve at the operating system level.

Commodity computers with multicore processors are pervasive today and it is likely that future core counts are going to increase even further [Bor07], as Moore's Law continues to hold and transistor counts keep increasing. Multicore architectures are a recent development and research into multicore operating systems is being carried out at the time of this writing, with several approaches to improve operating system scalability in general, and process scheduling in particular. I am going to cover these approaches in Section 2.2.

Being able to scale with this increasing number of cores is an important software problem today and operating systems are not spared from it. As an

important layer in the software stack, operating systems can become scalability bottlenecks, even if application software is able to scale.

Furthermore, multicore architectures are diverse, with different, non-uniform interconnect topologies, cache hierarchies and processor features [SPB+08]. This is a main difference to the cache-coherent, non-uniform memory (cc-NUMA) and symmetric multiprocessing (SMP) architectures used in high performance computing. Multicore operating systems have to be built to be agile with the different architectures in existence and cannot be tailored to one particular architecture that may not exist anymore in a few years' time.

In this section, I argue that computers increasingly resemble networked systems with very similar properties, such as topology and node heterogeneity and latency, and software should be programmed in a similar manner to software running on such networked systems. In the following, I give several examples.

2.1.1 Hardware Diversity

Multicore architectures are constantly evolving. This brings diversity in manufactured systems with it and operating systems have to be able to adapt, instead of being designed for one particular hardware configuration. A particular example of operating system tailoring to a specific hardware architecture are reader-writer locks [DS09], which are employed in the Solaris operating system on the Sun Niagara processor [KAO05]. Cores on the Niagara processor share a banked L2 cache and thus a lock using concurrent writes to the same cache line to track the presence of readers is a highly efficient technique, as the line remains in the cache. On other multicore systems that do not share L2 caches, this is highly inefficient, as cache lines have to be moved constantly between the readers' caches, incurring a high latency penalty.

Production commodity operating systems, such as Linux, Windows, and Solaris, used to optimize for the most common architecture on the market and are becoming less efficient as hardware becomes more diverse. This can be seen by the amount of work that is currently being done to optimize operating systems to new multicore architectures, both within research, as well as industry.

For example, the Linux scalability effort [LSE] is a community-based effort to improve Linux scalability on multicore systems with large processor counts. Algorithms for scalable synchronization in shared-memory multiprocessors have been developed [MCS91] and read-copy-update mechanisms were introduced to improve multiprocessor scalability [MW08], but had to be revised to improve real-time responsiveness [GMTW08]. Several scalability improvements have been worked into the Windows operating system, for example, the removal of scheduler lock acquisition within code handling timer expiration [Rus08] and receive-side scaling technology to enable better scalability of network packet processing [Mic].

In some cases, modifying the operating system can be prohibitively difficult. For example, improvements to the Windows 7 scheduler architecture to aid it become more scalable on modern hardware modified 6000 lines of code scattered across the operating system and have been described as "heroic" [Rus09]. Similar effort was put into updating the Linux read-copy-update implementation [MW08], as several other features had to be updated that tightly depended on the old implementation.

But also within a single machine, hardware can be diverse. For example, performance-asymmetric architectures that share the same instruction set architecture (ISA), but have processors with different performance characteristics [SNL+03, IKKM07], are widely proposed to provide a better power-performance ratio. These architectures allow parallel programs to execute on a fleet of low-power, low-performance cores, while sequential programs execute on a few high-performance cores. Other architectures provide specialized ISAs to optimize certain functionality [HBK06] and many accelerator cores are increasingly programmable, such as graphics and network processors.

2.1.2 Interconnect Latency Matters

Hardware memory interconnect architectures, such as HyperTransport [CH07] and QuickPath [Int08], have replaced shared bus toplogies with a message passing infrastructure. For example, the memory topology presented in Figure 2.1 for a commodity PC server shows that caches are connected by point

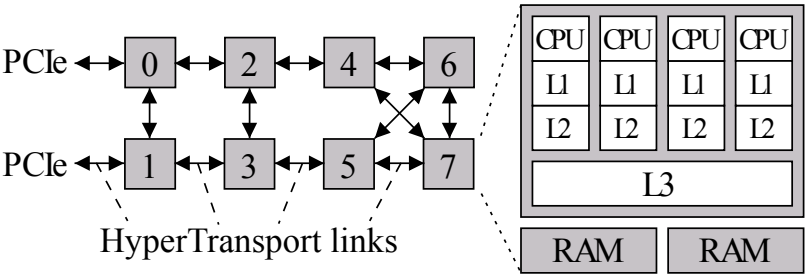


Figure 2.1: Node layout of an 8×4-core AMD system.

Access	cycles	normalized to L1	per-hop cost
L1 cache	2	1	-
L2 cache	15	7.5	-
L3 cache	75	37.5	-
Other L1/L2	130	65	-
1-hop cache	190	95	60
2-hop cache	260	130	70

Table 2.1: Latency of cache access for the 8×4-core AMD machine in Figure 2.1.

to point links. On these links, messages are passed to communicate the state of cache lines among caches of different processors.

The problems these topologies exert are very similar to those observed in networked systems. Link latency and utilization, especially over multiple hops of the interconnect, are playing an important role in program efficiency and scalability. In the future, these problems will increasingly impact cores on a chip, as these become more complex. For example, the AMD Magny Cours architecture uses HyperTransport links between two sets of six cores on the twelve-core chip [Car10]. Other multicore architectures, such as Larrabee [SCS+08, KPP06] and Tilera [WGH+07, VHR+07], use grid and torus-based interconnect topologies among cores with complex communication trade-offs.

This complexity can be demonstrated with a simple cache access micro-

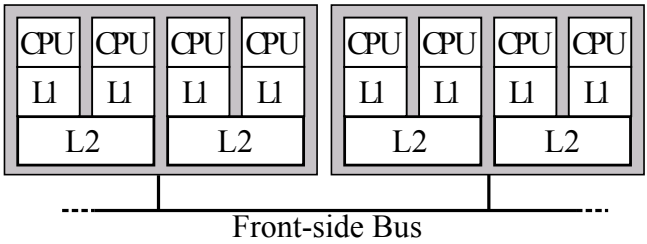


Figure 2.2: Node layout of a 2×4-core Intel system.

Access	cycles	normalized to L1	per-hop cost
L1 cache	3	1	-
L2 cache	14	4.6	-
Other L1/L2	80	26.6	-
1-hop cache	128	42.6	48

Table 2.2: Latency of cache access for the 2×4-core Intel machine in Figure 2.2.

benchmark, whose results are shown in Table 2.1 for the 8×4-core AMD machine presented in Figure 2.1. The benchmark shows cache line access latency for all caches in the memory hierarchy of the machine. The results are comparable to those presented by Boyd-Wickizer *et al.* for a 16-core AMD machine [BWCC+08]. As can be seen, remote cache line access can be up to 130 times slower than local L1 access and still up to 17 times slower than local L2 access.

To show that these problems are prevalent also on other architectures, I conduct the same benchmark on a 2×4-core Intel machine, depicted in Figure 2.2, and show the results in Table 2.2. As can be seen, even on this smaller, bus-oriented multicore machine with only 8 cores, remote cache line access can be up to 42.6 times slower than local L1 access and still up to 9 times slower than local L2 access. Non-coherent message-passing-based architectures, like the SCC, explicitly specify remote MPB access latencies per hop. The trend shown by these benchmarks is for these problems to exacerbate as multicore machines become larger.

Commodity operating systems use shared memory data structures that abstract away from the cache interconnect topology and designers do not reason about topology from first principles. Instead, performance problems show up later and their causes have to be found by complicated debugging. While debugging, developers have to pay great attention to shared data structure and algorithm layout, such that typical problems, like false and superfluous sharing and cache line thrashing can be found. To eliminate these problems, developers have to apply hardware-specific modifications to these data structures and algorithms, customizing the code for a particular hardware architecture.

On the other hand, in a distributed systems context, we can describe cache operations as the cache performing a remote procedure call (RPC) to a remote cache to fetch a cache line. Designing the operating system using these concepts allows us to explicitly reason about coherent cache access, such as the location of data and how long it will take for it to travel to the local cache.

2.1.3 Remote Procedure Call vs. Shared Memory

Lauer and Needham have argued that there is no semantic difference between shared memory data structures and use of message passing in an operating system. In fact, they argue that they are duals and neither model is inherently preferable [LN78]. However, choosing one over the other can be a performance trade off, depending on the hardware architecture that the system is built upon. For example, one system architecture might provide primitives for queuing messages and thus a message-passing based operating system might be easier to implement and be more performing. Another system might support fast mutual exclusion for shared-memory data access and thus a shared memory operating system might perform better.

An experiment[1] can show this trade off on contemporary commodity hardware. The experiment compares the cost of updating a data structure using shared memory with that of message passing on a 4×4-core AMD system. Figure 2.3 shows update latency against number of cores operating on the data structure for updates of various sizes.

[1] This experiment was conducted by Paul Barham.

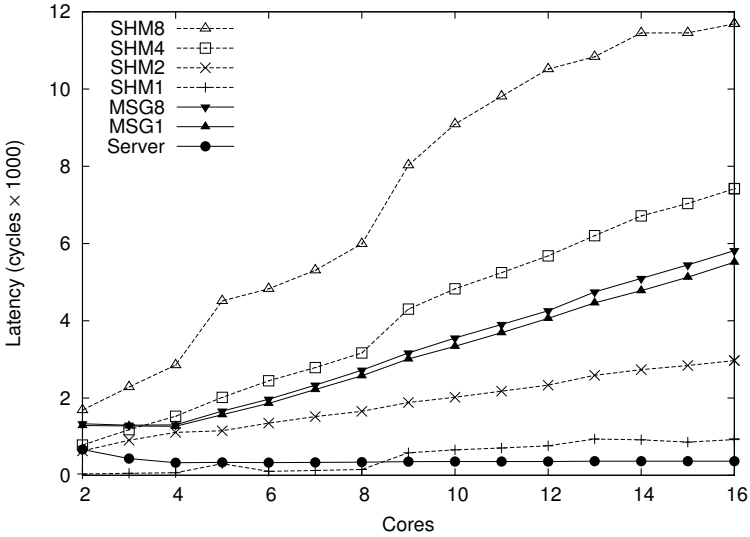


Figure 2.3: Comparison of the cost of updating shared state using shared memory and message passing on a 4×4-core AMD system.

In the shared memory case, each core directly updates the same small set of memory locations (without locking) and the cache-coherence mechanism migrates data among caches as necessary. The curves labeled *SHM1–8* show the latency per operation (in cycles) for updates that directly modify 1, 2, 4 and 8 shared cache lines, respectively. The costs grow approximately linearly with the number of cores and the number of modified cache lines. Although a single core can perform the update operation in under 30 cycles, when 16 cores are modifying the same data it takes almost 12,000 extra cycles to perform each update. All of these extra cycles are spent with the core stalled on cache misses and therefore unable to do useful work while waiting for an update to occur.

In the case of message passing, client cores issue a lightweight remote procedure call [BALL91], which we assume fits in a 64-byte cache line, to a single server core that performs the update on their behalf. The curves labeled *MSG1* and *MSG8*, show the cost of this *synchronous* RPC to the

dedicated server core. As expected, the cost varies little with the number of modified cache lines since they remain in the server's local cache. Because each request is likely to experience some queuing delay at the server proportional to the number of clients, the *elapsed* time per operation grows linearly with the number of client cores. Despite this, for updates of four or more cache lines, the RPC latency is lower than shared memory access (*SHM4* vs. *MSG8*). Furthermore, with an asynchronous or pipelined RPC implementation, the client processors can avoid stalling on cache misses and are free to perform other operations.

The final curve, labeled *Server*, shows time spent performing each update operation as measured at the server end of the RPC channel. Since it excludes queuing delay, this cost is largely independent of the number of cores (and in fact decreases initially once there are enough outstanding client requests to keep the server 100% utilized). The cache efficiency of the single server is such that it can perform twice as many updates per second as all 16 shared-memory cores combined. The per-operation cost is dominated by message send and receive, and since these costs are symmetric at the client end, we infer that the difference between the *Server* and *MSGn* lines represents the additional cycles that would be available to the client for useful work if it were to use asynchronous messaging.

This example shows scalability issues for cache-coherent shared memory on even a small number of cores. Although current operating systems have point-solutions for this problem, which work on specific platforms or software systems, the inherent lack of scalability of the shared memory model, combined with the rate of innovation we see in hardware, will create increasingly intractable software engineering problems for operating system kernels.

Figure 2.4 shows the result of repeating the experiment on the 2×4-core Intel machine. In this case, we are plotting latency versus the number of contended cache lines. We execute the experiment once for 2 and once for all 8 cores of the architecture, both for message passing to a single server core and for shared-memory. Again, we see that when contending for a growing number of cache lines among a number of cores, remote procedure calls increasingly deliver better performance than a shared memory approach. When all 8 cores contend, the effect is almost immediate. When only 2 cores contend, we need to access at least 12 cache lines concurrently

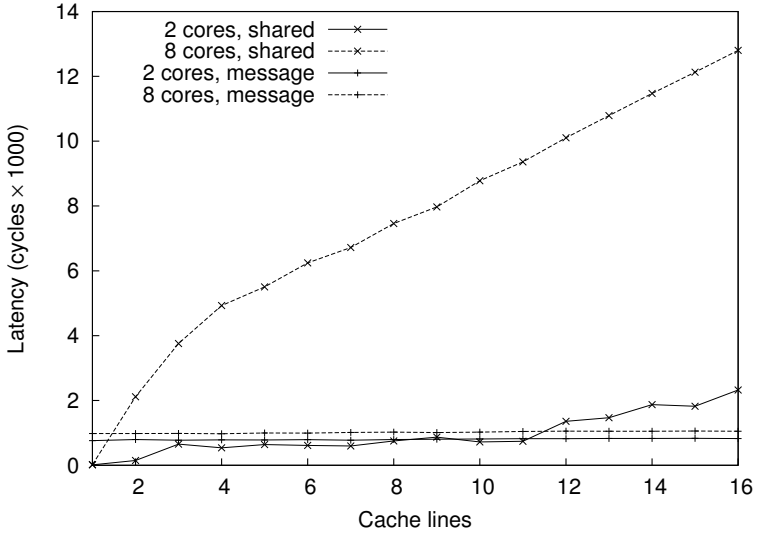


Figure 2.4: Comparison of the cost of updating shared state using shared memory and message passing on the 2×4-core Intel system.

before observing the effect. This confirms that the problem persists on other architectures.

2.1.4 Cache Coherence Protocol Scalability

As more cores are integrated into multicore architectures, hardware cache-coherence protocols are becoming increasingly expensive. It is not clear to how many cores these protocols will be able to scale, and so it is possible that future multicore systems might not be fully cache-coherent [WA09, MVdWF08, Bor07]. Mattson *et al.* argue that cache coherence protocols might not be able to scale beyond 80 cores [MVdWF08]. Even if interconnects remain fully coherent for a while, software can leverage substantial performance gains by bypassing the cache-coherence protocol [WGH+07].

Finally, accelerator cores, like graphics and network controllers, are not

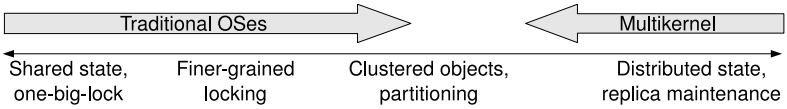


Figure 2.5: Spectrum of sharing and locking disciplines.

cache-coherent with the rest of the system and research non-coherent multicore processors, such as the Intel Single-Chip Cloud Computer [HDH+10], have already been built to investigate the use of non-coherent shared memory for the entire architecture.

2.1.5 Discussion

While it is unclear whether cache coherence is going away, the number of cores in a system is going to increase and will bring more change to interconnect topologies. An operating system that is based on message passing is a better fit for these architectures, as it can be deployed on non-coherent systems without change and can exploit the underlying hardware message passing facilities better.

The fast pace of hardware changes is a problem for commodity operating systems, which are not designed with a changing hardware architecture in mind and struggle to keep up. Furthermore, the increasing hardware diversity will make it prohibitively expensive for developers to adapt their optimizations to these new platforms. A distributed system model allows us to efficiently re-structure the operating system to the changing underlying hardware without the need to rewrite vast portions of it.

Figure 2.5 shows the design spectrum between completely shared approaches, over fine-grained locking, to fully distributed models. Traditionally, operating systems originated on the left hand side, as they were typically designed with a uni-processor in mind and then gradually evolved to multiprocessors with ever higher processor counts, such that today's commodity operating systems are situated towards the center, where frequently accessed data structures are partitioned across cores. Research operating systems proposed clustered object mechanisms that improve data locality of these par-

titions even further [GKAS99]. The approach presented in this dissertation originates at the right hand side: We design the operating system and reason about it as a distributed system, where all state is replicated by default and consistency is maintained by distributed agreement protocols. In those cases where it is appropriate, we employ sharing of data structures to optimize the model.

2.2 Background

Machines that use message passing among cores have existed for a long time. The Auspex [Bli96] and IBM System/360 implemented partially shared memory and used messages to communicate between non-shared regions. Furthermore, message passing has been employed on large supercomputers, such as the Cray T3 and IBM Blue Gene, to enable them to scale beyond the limits of cache coherence protocols. Unsurprisingly, their operating systems resembled distributed systems. The Multikernel model has similar goals with these approaches, but additionally has to be agile with a variety of multicore architectures.

Using message passing instead of shared memory has been explicitly investigated in the past on the BBN Butterfly Plus multiprocessor architecture [CDL+93]. This work found a bias towards message passing for the latency of kernel operations and is early evidence that message passing can make sense in a multicore operating system.

Similarly, the DEC Firefly multiprocessor workstation [TS87], is designed as a network of MicroVAX 78032 processors communicating via remote procedure calls that are implemented by the use of message passing. The Taos operating system has been designed in the fashion of a distributed system of multiple services sequestered into separate address spaces that communicate via RPC.

Several extensions have been made to existing commodity operating systems, like Linux, OS X, Windows, BSD, and Solaris, over the past decade to help them scale with an ever increasing number of cores. Recent examples include modifications to the Linux kernel to better support a “shared-nothing” application structure [BWCM+10], as well as the previously men-

tioned efforts to reduce locks and their granularity in the Windows kernel [Rus09]. Many of these developments are summarized and put into historical context by Cantrill and Bonwick [CB08]. Research comparing the scalability of Linux, BSD, and Solaris on multicore architectures finds that none of these operating systems ultimately scales better than another [CCSW10, BWCM+10].

Similarly, the L4 Microkernel has been extended for scalability on multicore machines [Uhl07]. The main problem in a Microkernel is that of lock granularity for kernel operations that are naturally more fine-grain than those of monolithic kernels. The extension introduces a mechanism to allow programs to explicitly specify expected parallelism and uses an adaptive locking mechanism to switch between coarse and fine-grained locking, according to the specification.

Microkernel architectures, such as Mach [BGJ+92], employ message-based communication among processes, but do so to achieve protection and isolation. Multiprocessors were either absent from consideration or the Microkernel itself remained a shared-memory, multi-threaded system and does not manage cores independently. While some message-based coordination of user-space services that execute on distinct cores might exist in Mach, independent management of individual cores was not an explicit design criterion for this operating system.

Disco and Cellular Disco [BDGR97, GTHR99] were based on the premise that large multiprocessors can be better programmed as distributed systems, an argument complementary to the one made in this dissertation. Similarly, the Hive operating system [CRD+95] used a distributed system of independent kernels, called *cells*, to improve system scalability and contain software and hardware faults to the cells in which they occur. While in the Hive operating system shared memory was still required to provide performance competitive with other multiprocessor operating systems available at the time, these systems can still be seen as further evidence that the shared-memory model is not a complete solution for large-scale multiprocessors, even at the operating system level.

2.3 Evaluation Platforms

Barrelfish has been implemented for a range of contemporary multicore architectures and machines, among them the x86 line of Intel processors (IA-32 and Intel64), the Intel Single-Chip Cloud Computer, several ARMv5 and ARMv7 based architectures, and Microsoft's Beehive processor. Throughout this dissertation, I demonstrate that the ideas are practical by evaluating different parts of the Barrelfish prototype on a number of machines derived from these architectures. I focus my evaluation by comparing against the Linux operating system. Where appropriate, I elaborate on the performance of other commodity operating systems, such as Windows and Solaris.

My choice of machines is comprised of Intel and AMD models of the Intel64 architecture, which are prevalent in the commodity computing space. Some of these machines were introduced in Section 2.1. I also include the Intel Single-Chip Cloud Computer research architecture as an example of what a commodity computer could look like in the future. I am describing the machines once in detail here and will refer to them later in the document.

2.3.1 x86-based Systems

The *2×4-core Intel system* has an Intel s5000XVN motherboard [Int10] with 2 quad-core 2.66GHz Xeon X5355 processors and a single external memory controller. Each processor package contains 2 dies, each with 2 cores and a shared 4MB L2 cache per die. Both processors are connected to the memory controller by a shared front-side bus, however the memory controller implements a snoop filter to reduce coherence traffic crossing the bus.

The *2×2-core AMD system* has a Tyan Thunder n6650W board [TYA06b] with 2 dual-core 2.8GHz AMD Opteron 2220 processors, each with a local memory controller and connected by 2 HyperTransport links. Each core has its own 1MB L2 cache.

The *4×4-core AMD system* has a Supermicro H8QM3-2 board [Sup09] with 4 quad-core 2.5GHz AMD Opteron 8380 processors connected in a square topology by four HyperTransport links. Each core has a private 512kB L2 cache, and each processor has a 6MB L3 cache shared by all 4 cores.

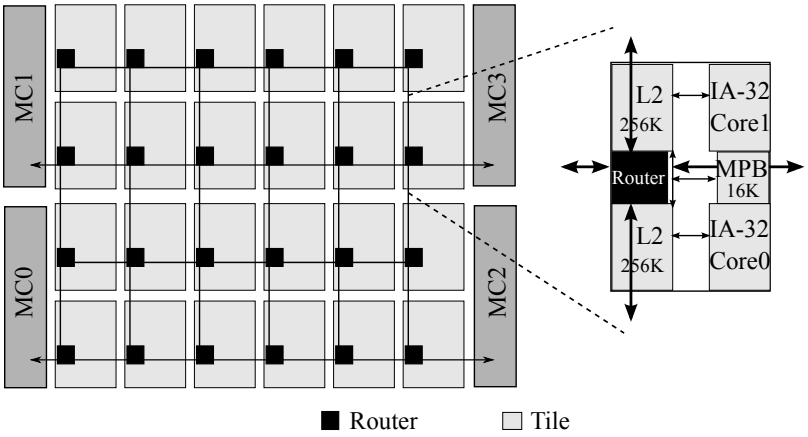


Figure 2.6: SCC architecture functional diagram.

The *8×4-core AMD system* has a Tyan Thunder S4985 board [TYA06a] with M4985 quad CPU daughter card and 8 quad-core 2GHz AMD Opteron 8350 processors. I present the interconnect in Figure 2.1. Each core has a private 512kB L2 cache, and each processor has a 2MB L3 cache shared by all 4 cores.

All x86-based systems are equipped with an Intel e1000 Ethernet network interface card that delivers network throughput of up to 1Gb/s.

2.3.2 Intel Single-Chip Cloud Computer

The Intel Single-Chip Cloud Computer [HDH+10] is a non-cache-coherent architecture, based on IA-32 Pentium P54C cores, clocked at 533MHz. It contains 48 of these cores, grouped into tiles of two cores each and linked by a two-dimensional, high-speed mesh network. Figure 2.6 shows a functional diagram of the SCC architecture.

The SCC uses a message passing architecture. Messages are exchanged on the mesh network via routers. A router exists on each tile and is shared by both cores. At the left and right edges of the mesh, two connections

route messages to one of four memory controllers (MCs). Each core has 256KB of *non*-coherent L2 cache, and in addition each pair shares a 16KB on-tile message-passing buffer (MPB). MPBs are memory-mapped regions of SRAM that support fast non-coherent reads and writes of MPB data at cache line granularity[2]. To support cache-line operations, each core has a write-combine buffer that can be used to write to SRAM and DRAM alike. In order to support mutual exclusion of writers to the same region of SRAM, a test-and-set register is provided per core.

A cache invalidate opcode named `CL1INVMB` has been added to the architecture to allow software to control cache operations and invalidate stale cache lines when they should be read fresh from either SRAM or DRAM. This opcode executes within 2 processing cycles of the architecture.

2.4 The Multikernel Architecture

In order to achieve the goals of scalability and agility, a number of design principles is followed that was not previously tried in this combination in designing an operating system for commodity multicore computers. This section presents the Multikernel architecture and discusses these three main design principles. They are:

1. Make all inter-core communication explicit.
2. Make operating system structure hardware-neutral, that is, the architecture does not assume any specific hardware features, such as shared memory, cache coherence, or a specific memory interconnect topology. Such hardware features can be used as optimizations in implementations of the model.
3. View state as replicated instead of shared.

The first design principle is realized by using message passing instead of shared memory for inter-core communication. This has the benefit that it

[2]The size of a cache line on the SCC is 32 bytes.

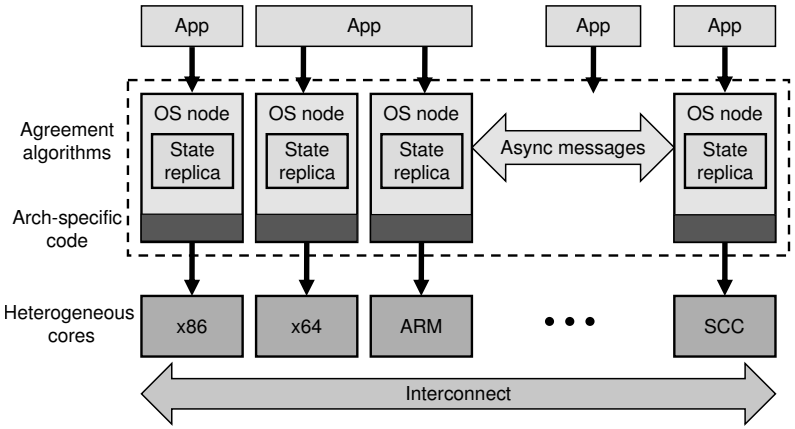


Figure 2.7: The Multikernel model. Several potentially heterogeneous cores are connected via a hardware memory interconnect. On top of this run per-core OS nodes that use agreement protocols and asynchronous message passing to keep replicated state consistent within the system. On top of this run applications that can utilize several OS nodes concurrently for parallel processing.

does not assume cache-coherent shared memory and is thus a natural fit for non-cache-coherent architectures. Furthermore, this architecture does not share data among processor cores by default and thus encourages a scalable implementation.

The second design principle is realized by using a component architecture that allows OS services to be structured as distributed systems placed on *networks* of cores. The structure of these distributed systems can be freely defined at OS boot-time according to the underlying hardware characteristics, such as communication latency and connectivity among cores.

The third design principle is realized by applying replication techniques and agreement protocols from the field of distributed systems, which I will survey in Section 2.4.3. These techniques allow OS services to replicate state only among those cores that need access and to optimize replication strategies for different network configurations.

In summary, this results in structuring the operating system as a distributed system of servers that run on a network of distinct cores and communicate via message passing, instead of using shared memory. Figure 2.7 shows an overview of the architecture. The above principles allow the operating system to benefit from the distributed systems approach to gain improved performance and scalability, natural support for hardware heterogeneity, greater modularity, and the ability to reuse algorithms developed for distributed systems.

2.4.1 Explicit Inter-core Communication

We design the operating system such that all inter-core communication happens explicitly via message passing. There will be no sharing of data structures, except for those needed to efficiently implement message passing primitives, such as message channels.

As we have seen in the results shown in Figure 2.3, messaging can improve performance as the number of cache lines modified grows. We anticipate that many data structures exist within an operating system that require modification of multiple cache lines to finish an operation on the data structure.

Note that disallowing inter-core sharing within the operating system does not preclude *applications* sharing memory among cores, only that the operating system design itself does not rely on it. I will present the support for shared memory applications, which will likely remain important due to a typically simpler structure that is easier to program for, in Section 2.5.5.

Message passing makes all inter-core communication explicit and forces the operating system developer to think hard about what state he communicates with other cores, as well as how many of them and which. This enforces a core-local policy by default that can be selectively broken through if the developer realizes that cross-core communication is really needed. In contrast, classical operating systems typically employ point solutions to optimize the scalability of hot-path data structures, like those discussed in Section 2.2. These are difficult to evolve as multicore hardware changes and ignore feature interactions that might require updates to large portions of the operating system implementation, especially when multiple data structures need to be updated concurrently.

Explicit communication allows the operating system to employ optimizations from the field of networked systems, such as pipelining and batching. Pipelining allows having multiple outstanding requests at once that can then be handled asynchronously by a service, typically improving throughput. Batching allows sending a number of requests within one message or processing a number of messages together and is another technique to improve throughput. I am going to demonstrate this experimentally in Section 2.6, which investigates message passing performance. These optimizations are also going to help us gain superior performance of application-level benchmarks conducted in Section 2.6.4, versus shared-memory operating system approaches.

Furthermore, message passing is a natural way to handle cores with heterogeneous ISAs and those that are not cache-coherent, or even share memory, with the rest of the system, as it makes no special assumptions about either. Shared memory data structures are naturally assumed to be in cache-coherent memory and shared code naturally has to execute on the same ISA. For example, it is possible to execute a single Barrelfish OS instance on a machine consisting of an x86-based host PC and an SCC peripheral [PSMR11], while no memory is shared among either platform. This is made possible by communicating solely via message passing channels.

Message passing allows communication to be asynchronous. This is not the case for the hardware cache-coherence protocol, which stalls the processor until the required cache operation is completed, as was demonstrated in Section 2.1.3. Asynchronous communication allows cores to do other useful work or halt to save power, while waiting for the reply to a particular request. An example is remote cache invalidation: completing the invalidation is not usually required for correctness and could be done asynchronously, instead of waiting for the operation to finish.

Finally, using explicit communication makes systems more amenable for analysis (by humans or automatically). Explicit communication forces developers to use well-defined interfaces and is thus naturally more modular. As a consequence, explicit communication eases software evolution and refinement [FAH+06], as well as robustness to faults [HBG+06]. Indeed, there is a substantial theoretical foundation for reasoning about the high-level structure and performance of a system with explicit communication among concurrent tasks, ranging from process calculi such as communi-

cating sequential processes [Hoa] and the π-calculus [Mil99], to the use of queuing theory to analyze the performance of complex networks [Jai91]. The treatment of analysis techniques is outside the scope of this dissertation, but is an important avenue for future work and we will return to it in Section 5.1.

2.4.2 Hardware-neutral OS Structure

We design the operating system such that it is not tailored to a specific hardware structure. Instead, we employ a multi-server design that we can restructure according to the underlying hardware configuration. This way, we minimize the reliance on implementation that needs to be written for a particular architecture. Only the message transport mechanisms and device drivers still have to be tailored to the architecture.

This design minimizes the amount of changes that have to be done to the implementation in order to adapt to a new hardware architecture and eliminates feature interactions due to hardware-specific data structure designs and the mechanisms that handle them.

As this dissertation will show in Section 2.6, the performance of an inter-domain communication mechanism is strongly dependent upon hardware-specific optimizations. The separation of low-level messaging and high-level algorithms that is a natural consequence of the message passing approach allows both of these to evolve without impacting each other, resulting in a cleaner design.

These hardware-neutral design features make the operating system agile with the changing hardware landscape. For example, I will show in Section 4.1.4 how Barrelfish supports the same application performance as manually tuned operating systems on different multicore platforms.

A final advantage is that both the protocol implementation and message transport can be selected at run-time. For example, different transports may be used to cores and devices on IO links, or the implementation may be fitted to the observed workload by adjusting queue lengths or polling frequency.

2.4.3 State Replication

For some operating system state, it is necessary that it be accessed by multiple processors. Traditionally, this is done via shared data structures protected by locks. In a Multikernel, explicit message-passing among cores that share no memory leads to a model where global operating system state is replicated instead of shared.

Replication is a well-known technique for improving the scalability of operating systems [GKAS99, ADK+07], but is generally employed as an optimization to an otherwise shared-memory kernel design. In a Multikernel, all potentially shared state is replicated and accessed or updated using message passing. Depending on consistency requirements, state updates can be long-running operations to keep all replicas coherent and depend strongly on the workload. The scheduler presented in Chapter 3 is one example operating system service that employs replication to facilitate scalability.

Replicating data structures can improve system scalability by reducing load on the system interconnect, contention for memory, and overhead for synchronization. Also, bringing data nearer to the cores that process it will result in lower access latencies and reduce processing times. Replication is required to support processing domains that do not share memory and is inherent in the idea of specializing data structures for particular core designs. Making replication of state a design requirement for the Multikernel makes it easier to preserve operating system structure and algorithms as underlying hardware evolves.

Replication is also useful to support changes to the set of available, running cores, as we can apply standard mechanisms from the distributed systems literature. This can be used, for example, upon a CPU hot-plug or shut down in order to save power.

What we gain in performance through replication comes at the cost of replica maintenance. This means higher latency for some operations, depending on the workload, and means an increased burden for the developers who must understand the data consistency requirements, but allows them to precisely control the consistency policy. For example, a TLB flush can be achieved by a single multicast, as it has no requirements on the ordering of flush operations over cores. Other operations might require more elaborate agreement

protocols.

Several replication techniques and consistency schemes exist in the literature and I briefly discuss here their applicability to the Multikernel:

1. *Transactional replication* is used for transactional data, such as found within database management systems and is typically described along with a strict set of restrictions, called the *ACID properties* [HR83], that define a valid transaction. The predominant consistency scheme for transactional replication is *one-copy serializability* [Kem09]. File systems most closely resemble transactional systems and thus transactional replication is an adequate fit. However, in many cases, the full ACID properties are too restrictive for adequate performance and are often relaxed, for example in replication for the Network File System version 4 [Pet06].

2. *State machine replication* is used predominantly within distributed systems and relies on the assumption that these systems can be modeled as state machines and reliable broadcast of state transitions to replicas is possible. Maintaining consistency is modeled as a *distributed consensus* problem and algorithms, such as Paxos [PSL80], exist to achieve it. State machine replication is used in distributed services, such as Google's Chubby lock service[Bur06]. Mutual exclusion is an integral part of an operating system and thus this replication model is useful in implementing it in a Multikernel.

3. *Virtual synchrony* [BJ87] is a replication technique used in *group communication systems*, such as Transis [DM96]. Maintaining group membership, as well as efficient, but reliable multicast of operations carried out within a group, especially with regard to message order, are the main focus of consistency schemes in this area.

   Within a Multikernel, virtual synchrony can be used for services that involve event notification, such as publish-subscribe. In such services, applications can subscribe to messages of a given topic and an event callback is invoked whenever a new message of that topic is published to the group.

   Examples of such systems within a Multikernel include the name and coordination service, Octopus [Zel12], as well as file system buffer

caches. Octopus supports creation of named information records that can be subscribed to by applications. Update events are delivered to subscribers whenever a record is updated. This service is useful, for example, in the creation of the terminal subsystem, which is used by multiple applications at the same time. Information about terminal line discipline is stored in an Octopus record and can be updated by applications with access to the terminal. Upon an update, all applications using the same terminal are informed to use the new line discipline.

File system buffer caches can use group communication to cooperatively cache buffers among cores. Per-core lists of cached blocks can be kept and group communication can be used to avoid fetching blocks from cores that are distant or overloaded.

Other consistency schemes, such as *eventual consistency* [Vog09], relax the time constraints on when replicas have to be consistent and can be applied to any of the replication techniques presented above. As within distributed systems, these schemes can be important within a Multikernel to improve the performance of replicated operations at the cost of system consistency.

2.5 System Structure

A drawback of an operating system model relying solely on message passing is that certain hardware performance optimizations may not be applicable, such as making use of a shared cache between cores. Messages might still be transported with lower latency on cores that share a cache, but laying out shared data structures in shared cache memory requires a shared-memory model.

To explore to what extent the Multikernel architecture can be applied in practice, Barrelfish, a prototype Multikernel operating system, was developed. This section describes the structure of the Barrelfish implementation of the Multikernel model, as well as the implementation of vital system components.

Specifically, the goals for Barrelfish are that it:

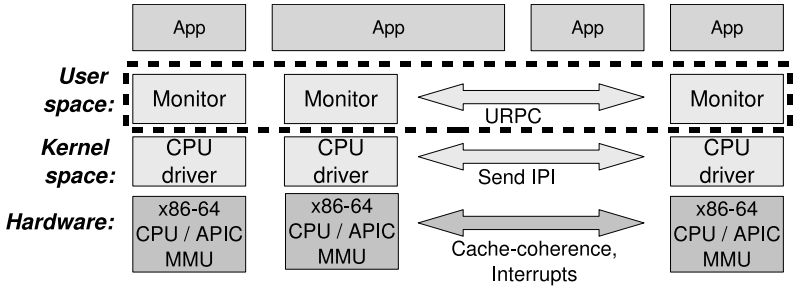


Figure 2.8: Barrelfish structure on the x86-64 hardware architecture. User-space monitor processes, which implement OS services, communicate via user-mode remote-procedure calls. Inter-processor interrupts might be employed for notification of message arrival at destination cores and are handled by privileged CPU drivers. Finally, both messages and interrupts are delivered on the cache-coherent interconnect between cores. On top of this run the applications, which utilize OS services.

- gives comparable performance to existing commodity operating systems on current multicore hardware;
- demonstrates evidence of scalability to large numbers of cores, particularly under workloads that stress global operating system data structures;
- can be re-targeted to different hardware, without refactoring;
- can exploit the message-passing abstraction to achieve good performance by pipelining and batching messages;
- can exploit the modularity of the operating system to place operating system functionality according to the hardware topology or load.

In Barrelfish, we place independent low-level operating system instances on each core and have them communicate via messages. The low-level instances consist of a privileged-mode CPU driver and a user-mode monitor process, as shown in Figure 2.8. CPU drivers are structured such that they execute completely core-locally. All inter-core messaging necessary for low-level operating system service operation is carried out by the monitors.

CPU drivers might only be involved in delivering inter-processor interrupts as a means of notifying remote cores. As a distributed system, monitors and CPU drivers represent the functionality of a typical Microkernel: Scheduling, inter-domain communication, and low-level resource management are all implemented within these two components.

The rest of Barrelfish consists of device drivers and other system services (such as network stacks, memory allocators, etc.), which run in user-level processes. Device interrupts are routed in hardware to the appropriate core, demultiplexed by that core's CPU driver, and delivered to the driver process as a local message.

2.5.1 CPU Drivers

CPU drivers in Barrelfish are responsible for managing the per-core low-level hardware features, such as context switching of processor register state, system calls, traps, device interrupts and core-local timers. CPU drivers run in privileged mode and it is in our interest to make them as simple as possible, such that the risk of bugs impacting arbitrary system state is reduced, akin to the design of Microkernels. Thus, we do not incorporate any high-level functionality into the CPU driver, in particular dynamic allocation of memory. As such, it is possible for us to design the CPU driver in a completely event-driven, non-preemptable, single-threaded manner, using only a single, pre-allocated stack that is not retained upon a return to user mode. OS state is managed by manipulating capabilities, which user-mode passes to the CPU driver upon a system call. Capabilities are explained in Section 2.5.3. This makes it easier to write and debug CPU drivers, compared to classical monolithic kernels, where all services, including device drivers are executing in privileged mode and state of multiple threads has to be kept simultaneously. In Barrelfish, we disable interrupts upon entry to the CPU driver and carry out only one driver operation at a time, before leaving the CPU driver with interrupts enabled.

The CPU driver is small. Including platform-specific code for all seven supported architectures, it is comprised of only 22,133 lines of code, out of which 2,981 lines are in Assembly. To compare, the x86-64 specific code has a volume of 5,455 lines of code. Resident CPU driver code has a size

of 185kB and data has a size of 4.7MB. This allows us to replicate CPU driver code and data to core-local memory on a NUMA system, which can improve performance as the frequently used CPU driver is always loaded from memory close to the executing core.

As with an Exokernel [EKO95], a CPU driver abstracts very little from the underlying hardware. Instead, it performs dispatch and fast local messaging between processes on the core. It also delivers hardware interrupts to user-space drivers, and core-locally schedules user-space processes. The CPU driver is invoked via standard system call instructions with a cost comparable to Linux on the same hardware.

As with Microkernels, CPU drivers are heavily specialized for the hardware architecture they execute on, including data structure layout and system call calling conventions. This improves execution time of its low-level services, which are invoked frequently.

In contrast to Microkernels, CPU drivers execute entirely local to a core and do not share data or exchange messages with other cores. Thus, inter-processor communication is not offered as a service by CPU drivers. Also, CPU drivers are not responsible for scheduling application threads.

The CPU driver interface to user-space consists of only three system calls:

1. A call to invoke a capability, which can contain a parameter to choose a capability type-specific command,
2. a call to yield the CPU either to a specific or any other runnable process, and
3. a call to conduct local message passing (LMP), which is described and evaluated in Section 2.6.

Only the capabilities that describe CPU driver objects can be invoked with the first system call. These objects manage the CPU-local interrupt dispatch table, create and destroy processes on the CPU, map and unmap pages and page tables, send inter-processor interrupts to other cores, and manipulate a domain's capability address space.

2.5.2 Process Structure

The Multikernel model leads to a different process structure than in a typical monolithic multiprocessor operating system. A process in Barrelfish is represented by a collection of *dispatcher* objects, one on each core on which it might execute, instead of a single shared process control block.

Within Barrelfish, we also refer to such a collection of dispatchers as a *domain* and throughout this dissertation, the terms domain and process will be used interchangeably. Strictly speaking, a domain is more general than a process, as it describes any collection of co-operating dispatchers. For example, a domain does not necessarily need to share a common virtual memory address space. Within the work treated by this dissertation, however, there was no need to go beyond using domains in an identical fashion to processes.

Dispatchers on a core are scheduled by the local CPU driver, which invokes an up-call interface that is provided by each dispatcher to resume execution. This mechanism is similar to the one used in Psyche [MSLM91] and to scheduler activations [ABLL91], and contrasts with the Unix model of simply resuming execution. Above this up-call interface, a dispatcher typically runs a core-local user-level thread scheduler. The upcall interface allows information to be passed from the operating system to the application and I am going to describe it in more detail in Section 3.5.2.

A dispatcher consists of code executing at user-level and a data structure located in memory, split into two regions. One region is only accessible from the CPU driver, the other region is shared in a read and write accessible fashion between dispatcher and CPU driver. Protection is realized by mapping the two regions on pages with different access rights (recall that the CPU driver executes in privileged mode). The fields of the structure that are relevant for scheduling are listed in Table 2.3 and described below. Beyond these fields, the user may define and use their own data structures. For example a stack for the dispatcher code to execute on, thread management structures, and so forth.

Barrelfish's process structure is a direct result of the distributed nature of the system and serves to improve scalability and agility. Dispatchers are core-local entities that hold all of their own code and data on memory close to

Table 2.3: Fields in the CPU driver-defined part of the dispatcher control structure. The R/W column designates read-only or read-write access rights from the user-level dispatcher.

Field name	Type	R/W	Short description
disabled	word	R/W	If non-zero, the CPU driver will not up-call the dispatcher, except to deliver a trap.
haswork	pointer	R	If non-zero, the CPU driver will consider this dispatcher eligible to run.
crit_pc_low	pointer	R	Address of first instruction in dispatcher's critical code section.
crit_pc_high	pointer	R	Address immediately after last instruction of dispatcher's critical code section.
entry points	5 pointers	R	Functions at which the dispatcher code may be invoked, described below.
enabled_save_area	word array	W	Area for the CPU driver to save register state when enabled
disabled_save_area	word array	R/W	Area for CPU driver to save register state when disabled
trap_save_area	word array	W	Area for the CPU driver to save register state when a trap or a page-fault occurs

the local core. Dispatchers can be placed arbitrarily on cores, based on the underlying hardware topology. Domains spanning a heterogeneous set of cores are supported, as dispatchers can be specially compiled for the cores' ISA and communicate via message passing.

Also, the process structure is a result of the vertical structure of the operating system. The vertical structure aids in application agility, by giving applications more control over resource allocation policies. Arbitrary implementations of such policies can be realized in a library OS, according to application structure. The assumption being that the application has better knowledge about how its algorithms map to the underlying hardware topology. A number of data structures and mechanisms result from this vertical process structure, namely *disabled* and *enabled* modes, *register save areas*, and *dispatcher entry points*. I am describing them in the following subsections.

Disabled and Enabled Modes

A dispatcher can operate in one of two modes: enabled or disabled. We use these modes to facilitate user-level thread scheduling, described in Section 2.5.5. Enabled mode is the normal operating mode when running a program's thread. Disabled mode is used to execute the user-level thread scheduling code. This allows applications to implement their own thread scheduling policies, according to application structure.

The modes control where the CPU driver saves application register state and how it resumes a dispatcher. When the CPU driver resumes a dispatcher that was last running while disabled, it restores its machine state and resumes execution at the saved instruction, rather than up-calling it at an entry point and requiring the dispatcher to restore machine state itself.

A dispatcher is considered disabled by the CPU driver if either of the following conditions is true:

- The disabled word is non-zero.
- The program counter is within the virtual address range specified by the crit_pc_low and crit_pc_high fields.

The range specified by the crit_pc_low and crit_pc_high fields is required to allow a dispatcher to restore a thread context and transition to enabled mode in one atomic operation. Within the critical region, the dispatcher sets the enabled flag before it resumes the state and continues execution of a thread. If these two operations were not atomic, a dispatcher could be preempted by the CPU driver while restoring the thread context and the restore operation would not be resumed by the CPU driver, as the dispatcher is already enabled.

Register Save Areas

The dispatcher structure contains enough space for three full copies of the machine register state. The CPU driver stores the register state to one of these areas upon the following conditions:

1. The trap_save_area is used whenever the program maintained by the dispatcher or the dispatcher itself takes a trap.
2. The disabled_save_area is used whenever the dispatcher is disabled, and
3. the enabled_save_area is used in all other cases.

Figure 2.9 shows important domain execution states and into which register save area state is saved or restored from, upon a state transition (Trap and PageFault states have been omitted for brevity). The starting state for a domain is "notrunning" and depicted with a bold outline in the figure.

Register save areas were also used in the Nemesis [LMB+96] and K42 [IBM02] operating systems. In contrast to Nemesis, Barrelfish domains do not contain an array of save slots and cannot direct the CPU driver to store to specific save slots. It is thus more similar to the model used in K42.

Dispatcher Entry Points

Unless restoring it from a disabled context, the CPU driver always enters a dispatcher at one of five entry points. Whenever the CPU driver invokes a

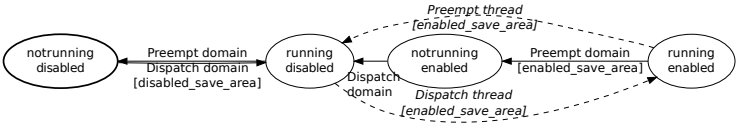


Figure 2.9: Dispatcher state save areas are used when transitioning among domain execution states. Arrows from right to left involve saving state into the labeled area. Arrows from left to right involve restoring state from the labeled area. Regular text and lines denote state changes by the CPU driver. Dashed lines and italic text denote state changes by a user-space dispatcher. The state save area used is denoted in square brackets on the transition label. The bold node denotes the starting state. No state is modified upon a transition from the notrunning disabled state to the running disabled state.

dispatcher at any of its entry points, it sets the disabled bit. Upon entering, one machine register always points to the dispatcher structure. The value of all other registers depends on the entry point at which the dispatcher is invoked, and is described here.

The entry points are:

Run A dispatcher is entered at this entry point when it was previously preempted. The last time it ran it was either enabled or yielded the CPU. Other than the register that holds a pointer to the dispatcher itself, all other registers are undefined. The dispatcher's last machine state is stored in the enabled_save_area.

PageFault A dispatcher is entered at this entry point when it receives a page fault while enabled. On entry, the argument registers contain information about the cause of the fault. Volatile registers are saved in the enabled_save_area. All other registers contain the user state at the time of the fault.

PageFault_Disabled A dispatcher is entered at this entry point when it receives a page fault while disabled. On entry, the argument registers contain information about the cause of the fault. Volatile registers are

saved in the trap_save_area . All other registers contain the user state at the time of the fault.

Trap A dispatcher is entered at this entry point when it is running and it raises an exception (for example, illegal instruction, divide by zero, breakpoint, etc.). The machine state at the time of the trap is saved in the trap_save_area , and the argument registers convey information about the cause of the trap.

LMP A dispatcher is entered at this entry point when an LMP message is delivered to it. This can happen only when it was not previously running, and was enabled. On entry, four registers are delivered containing the message payload. One register contains a pointer to the receiving endpoint data structure, and another contains the dispatcher pointer.

The diagram in Figure 2.10 shows the states a *dispatcher* can be in and how it transitions among states. The exceptional states Trap and PageFault have been omitted for brevity. We can see that preemption always returns the dispatcher to the notrunning state. When not running, a dispatcher can be entered by the CPU driver either by being scheduled (via the `schedule()` function) or if an LMP is received (via the `idc_local()` function). The dispatcher can then decide to dispatch a thread (via its `resume()` function) and transitions to the running state. When a system call is made (via the `syscall()` function), the dispatcher stays in the running state.

Dispatcher entry points are a core mechanism to make explicit to user-space the occurrence of certain events, such as a dispatch or a pagefault. This is necessary to achieve the goal of multiplexing hardware efficiently in time and space, by allowing user-space applications full control over the scheduling of threads within their timeslice.

While dispatch of a dispatcher is made explicit via the Run entry point, de-scheduling is made explicit by the CPU driver scheduler incrementing a per-dispatcher de-allocation count. This allows a language runtime to determine whether a given dispatcher is running at any instant and, for example, allows active threads to *steal* work items from a preempted thread via a lock-free queue, as done in work-stealing systems, such as the Cilk multithreaded runtime system [FLR98]. In this case, dispatcher entry points are

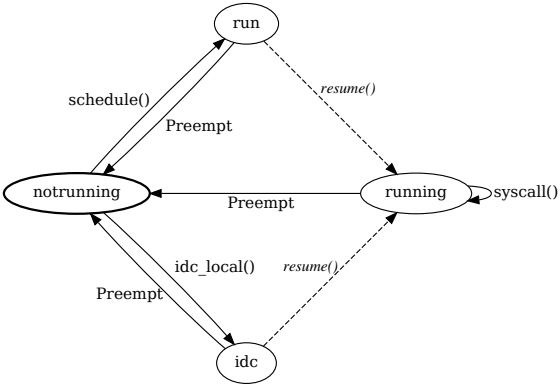


Figure 2.10: Typical dispatcher states. Trap and PageFault states omitted for brevity. Regular text and lines denote state changes by the CPU driver. Dashed lines and italic text denote state changes by user-space. The starting state is in bold.

used in achieving the goal of reacting quickly to ad-hoc workload changes. In a similar fashion, dispatcher entry points also help achieve the goal of allowing resource negotiation between applications and the operating system, by providing a path for the operating system to communicate core allocation changes to applications, by communicating when a time slice starts and when it ends.

2.5.3 Protection Model

We utilize capabilities [Lev84] for protection and access control. In particular, we build upon the seL4 capability model [DEE06], which we have ported to Barrelfish. I briefly describe the model here.

Capabilities are a protection mechanism where access is granted to a particular resource if the access requesting party holds the corresponding capability referencing the resource. Capabilities themselves cannot be forged and are protected by the privileged CPU driver, which is the sole entity allowed to create new capabilities.

In Barrelfish, capabilities are stored in a special capability address space,

of which one exists per dispatcher. Processes refer to capabilities by their address and capabilities are stored in a special CPU driver-protected and managed page table data structure.

In order to incur an action to be performed on an object, the corresponding capability needs to be *invoked* by the incurring process. Invocation is realized by a system call in Barrelfish to enforce protection. To ensure integrity, a type system enforces compatible capability invocations. For each capability type, a distinct set of invocations exists. For example, a page table capability possesses invocations to add and delete entries to and from the table. Compatible page table entry capabilities are page frame and other page table capabilities, depending on the allowed nesting of page tables supported by the hardware architecture.

Capability types exist for raw memory, page tables, page frames, dispatchers, as well as capability page tables. Page tables and frames make up the virtual address space of a domain, while capability page tables define the capability address space. All of these can be allocated by means of casting, or *retyping*, of a raw memory capability to the desired type. Once a raw memory capability has been retyped, the type system ensures that no further retyping of the same raw memory capability is possible. The Barrelfish specification [BPR+11] contains an overview of all capability types and their relationships.

The CPU driver offers invocations on capability page tables to support copying and revoking of capabilities within the same address space. It is also possible to send a capability as part of a message among domains. For messages on the local core, this is carried out by the CPU driver. When sending capabilities on messages across cores, the monitors are involved: The core-local monitor will serialize the capability and send it to the destination core's monitor. That monitor will deserialize the message and send it via a local message to the destination dispatcher.

The model cleanly decentralizes resource allocation in the interest of scalability. However, it also complicates the implementation of resource allocation, which now has to happen entirely in user-space and it is no more efficient than per-processor service instances implemented via other means. In fact, all cores must still keep their local capability lists consistent to avoid situations such as user-level acquiring a mapping to a protected object, like

a page-table, by leveraging an inconsistent capability table view.

2.5.4 Monitors

Monitors are responsible for managing the potentially replicated global operating system state and include most of the core functionality usually found in the kernel of a traditional operating system, such as a global scheduling policy. The monitors are user-space processes and therefore schedulable. Hence they are well suited to the split-phase, message-oriented inter-core communication of the Multikernel model, in particular handling queues of messages, and long-running remote operations.

As will be described in Section 2.6, monitors are responsible for interprocess communication setup, and for waking up blocked local processes in response to messages from other cores if this is supported by the underlying messaging protocol. A monitor can also idle the core itself (to save power) when no other processes on the core are runnable, by invoking special CPU driver functionality. Core sleep is performed either by waiting for an interprocessor interrupt (IPI) or, when supported by the hardware architecture, the use of appropriate monitor instructions to wait for incoming messages from other cores, such as MONITOR and MWAIT on the x86 architecture.

In the current implementation, monitors communicate within a fully connected network, requiring n^2 connections, with n being the number of monitors in the system. A more space-efficient way to communicate could be realized by a multi-hop routing layer. This layer could also facilitate communication to programmable peripherals, which typically require a proxy connection and cannot be directly communicated with, as a direct connection from host PC processors to peripheral processors is typically not available. Such a layer was demonstrated to be feasible by Alexander Grest [Gre11].

The CPU driver is heavily specialized for the core architecture, whereas the monitor has very little hardware-specific functionality. Only code handling processor boot-up and special message channel signaling protocols are architecture-specific. The downside of the monitor/CPU driver separation is that invocations from processes to the operating system are now mostly local RPC calls (and hence two context switches) rather than system calls,

adding a constant overhead on current hardware of several thousand cycles. Executing the monitor in privileged mode would remove this penalty, at the cost of a more complex privileged code base.

2.5.5 Threads and Shared Address Spaces

Threads in Barrelfish are implemented entirely in user-space, by a particular dispatcher implementation. The dispatch up-call from the CPU driver scheduler facilitates scheduling threads at user level. Most current programs use shared memory data structures and multi-threading for parallelism. Thus, the threads package in the default Barrelfish library OS provides an API similar to POSIX threads, which provides support for implementing the traditional model of threads sharing a single address space across multiple dispatchers (and hence cores) by coordinating runtime libraries on each dispatcher. This coordination affects three operating system components: virtual address spaces, capabilities, and thread management, and is an example of how traditional operating system functionality can be provided over a Multikernel.

We call a process that shares the virtual address space across multiple cores a *spanned domain*. Standard synchronization primitives, such as mutexes, condition variables, semaphores and spin-locks are provided and a round-robin thread scheduling scheme is implemented. I anticipate that language run-times and parallel programming libraries will take advantage of the ability to replace the default threads package with a custom one.

In the default Barrelfish library OS, thread migration and inter-core communication necessary to implement synchronization mechanisms is carried out via messages passed by the normal inter-core message passing facilities between dispatchers involved in a threaded operation. For example, when a thread on one dispatcher releases a mutex upon which a thread of another dispatcher is blocked, a *wake up* message is sent to that dispatcher to resume execution of that thread, so it can acquire the mutex.

When sharing a virtual memory address space, capabilities also need to be shared as they protect access to the shared pages and other associated CPU driver objects. The Barrelfish user-space library operating system is responsible for keeping the capability space consistent across cores. At the

moment, only an initial replication of all capabilities is accomplished and subsequent memory allocation is forced to be core-local and non-sharable. Mark Nevill is currently investigating more appropriate capability replication strategies [Nev12]. These will presumably vary among capability types. Also, an appropriate virtual memory subsystem that builds upon capability replication is currently being designed by Simon Gerber [Ger12].

Implementing threads in user-space has the advantage of light-weight context switch among threads, by not requiring a crossing to privileged mode, and helps in achieving the goal of multiplexing hardware efficiently in time and space. Also, it allows a user-space application full control over the scheduling of its threads, a means to achieve the goal of reacting quickly to ad-hoc workload changes.

Implementing threads in user-space has the disadvantage of a more complicated context switch. We can see this especially when lazily switching context of the FPU: FPU state has to be saved and restored at various points during context switch, both in privileged mode and in user mode. While this happens, we have to be careful not to touch the FPU while a lazy context switch is being carried out. For example, in the steady state of the system, the CPU driver is responsible for saving FPU state of a preempted thread and the dispatcher is responsible for restoring the FPU state of the next runnable thread. This makes sense, as it is the dispatcher that knows which thread to dispatch next. However, FPU state may not change while a dispatcher is manipulating it. That means the CPU driver has to transparently save and restore FPU state for a dispatcher while it switches FPU context. To simplify the implementation in Barrelfish, the CPU driver thus eagerly saves and restores FPU state for a disabled dispatcher and lazily saves FPU state for an enabled dispatcher.

Another disadvantage is that user-level threads require an asynchronous system call interface. We support this in Barrelfish, but it complicates the implementation of system services, by making each call split-phase.

2.5.6 Knowledge and Policy Engine

Barrelfish employs a user-mode service called the *system knowledge base* (SKB) [SPB+08], which collects and maintains information about the un-

derlying hardware topology in a subset of first-order logic. The current implementation of the SKB is based on the ECL^{i}PS^{e} constraint programming system [AW07], which has been ported to Barrelfish by Adrian Schüpbach and is described in his dissertation [Sch12].

The SKB is populated with information from the ACPI subsystem, PCI bus probes, and CPUID data. It is also updated from online measurements, such as UMP communication latency and bandwidth between all core pairs of the system. Finally, a priori information that cannot be discovered or measured (such as the interconnect topology of various system boards, and quirks that correct known flaws in discovered information, such as ACPI tables) is statically inserted into the SKB.

This information can be queried both by the operating system and application software in order to lay out threads optimally on the non-uniform system architecture and to choose appropriate inter-core synchronization algorithms, for example replication strategies. This aids operating system and application agility. The SKB is used by and has an interface to the scheduler, which I describe in Chapter 3.

2.6 Inter-domain Communication

In this section, I describe the implementation of the inter-domain communication mechanisms, both on the same, as well as across distinct cores. All inter-domain communication is structured into three layers:

- *Interconnect drivers* realize hardware message passing primitives, such as messaging via cache-coherent shared memory among cores or a directed, light-weight context switch on the same core on x86.
- *Notification drivers* implement low-level receiver notification mechanisms upon message arrival, such as inter-processor interrupts on x86. This avoids the need for polling for incoming messages.
- A marshaling *state machine* is responsible for providing a unified API between users of the message passing facility that deal with high-level

abstract message types and the low-level interconnect and notification driver implementations.

The marshaling state machine is automatically synthesized from an abstract description of valid message types and parameters for a particular protocol. A domain-specific language called Flounder and corresponding compiler have been developed for this purpose [Bau10], but will not be described in further detail in this dissertation.

Communication channels are established and torn down via interconnect driver-specific means implemented in the monitor. The following subsections describe what is required for each particular interconnect driver.

We start this section with an overview of how naming is realized in Barrelfish, so that channels among domains can be established via a process called *binding*.

2.6.1 Naming and Binding

Before a channel can be established among two distinct domains, we have to provide a means for the two domains to identify each other. In Barrelfish, this is done by giving them names. Furthermore, to discriminate among different services a domain might be offering, names are given not to domains, but to services they *export*.

Exporting of a service is done by making it known to other domains. The canonical way to do this in Barrelfish is to register the service with the *name service*, which has a fixed identifier that is known by every domain. In the current implementation of Barrelfish, the name service is a centralized service that executes on the bootstrap processor of the system.

The name service resolves a human-readable character string containing the name of a service to an *interface reference* (IREF). An IREF is a unique identifier specifying a particular service of a particular dispatcher on a particular core. New IREFs are allocated via requests to the core-local monitor. Every domain has a pre-existing, dedicated channel to the local monitor for this purpose. Monitors thus know all IREFs for their local cores and can

forward channel *bind* requests to a dispatcher of the appropriate domain, based on this information. If an IREF does not specify a service on the local core, the local monitor will forward the bind request to the appropriate core, based on the core's identifier stored in the IREF. IREFs are of appropriate size to store the complete information necessary to route a bind request.

The destination domain can respond to the bind request via its monitor channel to send an endpoint object back to the requesting domain, which will be forwarded by the monitors. The nature of this endpoint object is interconnect specific and described in the following subsections.

2.6.2 Same-core Communication

The CPU driver implements a lightweight, asynchronous same-core interprocess communication facility, which delivers a fixed-size message to a process and if necessary unblocks it. More complex communication channels are built over this using shared memory. As an optimization for latency-sensitive operations, we also provide an alternative, synchronous operation akin to LRPC [BALL90] or to L4 IPC [Lie95], not further described here.

To setup a channel using this primitive, which we call *local message passing* (LMP), the corresponding LMP interconnect driver will request an endpoint capability of the destination domain via the monitors as part of the bind process. The endpoint capability identifies the endpoint of a new channel to a particular destination dispatcher, chosen by the destination domain. To make the channel bi-directional, the initiator of the bind will send an endpoint capability in return. Once both endpoint capabilities are delivered, they can be invoked directly by the respective domains to send messages on the LMP channel.

Message payload is delivered directly via a directed context switch, which keeps all payload in processor registers. Only in the case of the destination dispatcher not being ready to receive is the payload stored by the CPU driver in a pre-allocated buffer in the dispatcher data structure of the receiving dispatcher that is identified by the endpoint capability. In this case, message receipt is signaled via a special flag in that memory range, which has to be polled by the receiving dispatcher.

System	cycles (σ)	ns
2×4-core Intel	845 (32)	318
2×2-core AMD	757 (19)	270
4×4-core AMD	1463 (21)	585
8×4-core AMD	1549 (20)	774

Table 2.4: One-way LMP latency on the 2×2-core AMD system.

Table 2.4 shows the one-way (program to program) performance of LMP on the 2×2-core AMD system. To compare, L4 performs a raw IPC in about 420 cycles on the same system. Since the Barrelfish figures also include a scheduler activation, user-level message dispatching code, and a pass through the thread scheduler, we can consider this performance to be acceptable.

2.6.3 Inter-core Communication

Inter-core interconnect drivers among the systems evaluated in this dissertation vary by architecture. I describe the interconnect and notification drivers for the x86 and SCC architectures in the following subsections.

The channel binding mechanism is identical among the inter-core communication mechanisms described in this section: a region of shared memory is allocated by the destination domain upon a bind request. This shared region is identified by a page frame capability, which is sent back on a bind reply message via the monitors. When the page frame is delivered, both domains split the memory region into two halves, each one containing a distinct messaging endpoint.

x86 User-mode Message Passing

On the x86 platform, Barrelfish uses a variant of user-level RPC (URPC) [BALL91] between cores, which we call user-mode message passing (UMP): a region of shared memory is used as a channel to transfer cache-line sized

messages point-to-point between a writer and a reader core. On the x86-based evaluation machines, the size of a cache line is 64 bytes. In contrast to URPC, UMP is not a procedure call, but instead an asynchronous, split-phase communication mechanism.

Inter-core messaging performance is critical for Multikernel performance and scalability, and the implementation is carefully tailored to the cache-coherence protocol to minimize the number of interconnect messages used to send a message. For a HyperTransport-based system, the shared region of memory contains two circular buffers of cache line-sized messages, each representing a uni-directional endpoint. Writer and reader core maintain their respective buffer position for each endpoint locally.

For each message, the sender writes sequentially into a free cache line in the buffer, while the receiver polls on the last word of that line, thus ensuring that in the (unlikely) case that it polls the line during the sender's write, it does not see a partial message. In the common case, this causes two round trips across the interconnect: one when the sender starts writing to invalidate the line in the receiver's cache, and one for the receiver to fetch the line from the sender's cache. The technique also performs well between cores with a shared cache and on systems using Intel's QuickPath interconnect.

Both parties need to know how many messages are in the buffer. This is communicated by acknowledging the receipt of messages via special acknowledgement messages that are sent on the other endpoint of a channel. The acknowledgement message simply contains the number of messages received thus far and messages are implicitly numbered sequentially. This way, any sequence of messages may be acknowledged as a batch, but earlier messages may not be omitted from being acknowledged. This is possible as the transport is lossless. As long as a message has not been acknowledged, it is assumed to be still in the buffer and may not be overwritten.

As an optimization, pipelined UMP message throughput can be improved at the expense of single-message latency through the use of cache prefetching instructions. This can be selected at channel setup time for workloads likely to benefit from it.

Receiving UMP messages is done by polling memory. Polling is cheap because the line is in the cache until invalidated. However, it is unreasonable

System (Clock speed)	Cache	Latency cycles (σ)	Latency ns	Throughput msgs/kcycle
2×4-core Intel (2.66GHz)	shared	180 (34)	68	11.97
	non-shared	570 (50)	214	3.78
2×2-core AMD (2.8GHz)	same die	450 (25)	161	3.42
	one-hop	532 (26)	190	3.19
4×4-core AMD (2.5GHz)	shared	448 (12)	179	3.57
	one-hop	545 (11)	218	3.53
	two-hop	558 (11)	223	3.51
8×4-core AMD (2GHz)	shared	538 (8)	269	2.77
	one-hop	613 (6)	307	2.79
	two-hop	618 (7)	309	2.75

Table 2.5: UMP performance.

	Latency cycles	Throughput msgs/kcycle	Cache lines used Icache	Cache lines used Dcache
UMP	450	3.42	9	8
L4 IPC	424	2.36	25	13

Table 2.6: Messaging costs on 2×2-core AMD.

to spin forever, as this wastes too many CPU cycles that can be utilized by other runnable dispatchers. Instead, a dispatcher awaiting messages on UMP channels will poll those channels for a short period before blocking and sending a request to its local monitor to be notified when messages arrive. At present, dispatchers poll incoming channels for a predetermined amount of time before blocking. This can be improved by adaptive strategies similar to those used in deciding how long to spin on a shared-memory spinlock [KLMO91].

Table 2.5 shows the UMP single-message latency and sustained pipelined throughput (with a queue length of 16 messages) on the x86 evaluation machines; hop counts for AMD refer to the number of HyperTransport hops between sender and receiver cores.

Table 2.6 compares the overhead of the UMP implementation with L4's

IPC on the 2×2-core AMD system[3]. We see that inter-core messages are cheaper than intra-core context switches in direct cost, and also have less cache impact and do not incur a TLB flush. They can also be pipelined to trade off latency for throughput.

SCC Message Passing

This section describes what is needed to implement message passing for the rather different SCC architecture. The SCC does not have coherent caches, but a hardware-accelerated message passing infrastructure based on fast SRAM. I refer the reader back to Section 2.3.2 for an overview of the architecture. We will find that relatively few modifications are required and that they are confined to a small part of the system, namely the interconnect and notification drivers.

The inter-core interconnect driver for the SCC is based on the cache-coherent user-level message passing (UMP) driver used on x86 systems. With some modifications this mechanism reliably transports cache-line sized messages via non-coherent shared memory on the SCC.

The modifications are:

- The size of a message, according to the hardware cache-line size, is 32 bytes.
- Memory for both the send and receive message channel is allocated in the core's region of shared RAM that is initiating a connection to another core. Additional message-passing experiments, not shown here, have shown this to be more performant than allocating the memory close to the receiver core.
- Memory for the message channel is mapped as cacheable, message passing buffer type, enabling the use of the write-combining buffer for faster memory writes.

[3] L4 IPC figures were measured using L4Ka::Pistachio of 2009-02-25.

While UMP was originally optimized for cache-coherent transports, these optimizations do not hurt performance on the SCC and these modifications suffice to use the interconnect driver on the SCC.

The polling approach used by UMP on x86 to detect incoming messages is inappropriate on the SCC, since each poll of a message-passing channel requires a `CL1INVMB` operation followed by a load from DDR3 memory, which is costly. Consequently, an additional SCC notification driver augments the UMP driver with notification support.

The notification driver uses the on-tile message-passing buffer memory (MPB) for efficiently communicating the identifiers of active UMP message channels, and an inter-core interrupt as the notification mechanism. It is implemented within the CPU driver on each core, as follows:

One ring-buffer of IDs of those channels containing unread payload is held statically in the receiver's MPB. The buffer is written only by senders and read only by the receiver. However, there are two read-shared cache lines before the buffer, holding the current write and current read position, respectively.

A buffer entry spans a cache-line (32 bytes). Currently, only a 16-bit channel ID is written to that cache-line, limiting the number of distinct notifiable channels to 65,536. The rest of the space is unused. Using a cache-line per ID allows a sender to write new channel IDs to the buffer without having to read the cache line for already existing IDs first, which was found to be too costly.

A special notification capability is used on the sender side, holding the core ID and channel ID of the receiver core of the notification. When invoked, the sender's CPU driver

1. acquires the test-and-set lock for the receiving core,
2. reads the current write position from the receiver's MPB,
3. writes the channel ID into the next slot in the receiver's MPB,
4. updates the current write position in the receiver's MPB, and

5. reads the receiver's interrupt status.

6. If no inter-core interrupt is pending, it writes the status word with the interrupt bit set to trigger a remote interrupt, and

7. clears the test-and-set lock for the receiving core.

On the receiver, a core-local message passing endpoint capability is registered in a special notification table inside the CPU driver. This table maps channel IDs to local endpoints that will be signaled on notification. When the receiver core is interrupted, it looks up all pending channel IDs present in the MPB ring-buffer, and dispatches an empty message on each registered endpoint in the table. If no endpoint is registered, the notification is ignored. In user-space, the notification driver triggers the thread used to wait on incoming messages for the corresponding message channel, which can then read the message.

In case of the receiver ring buffer being full when the sender tries to write a new channel ID, the sender aborts the process and returns with an error code to the sending user-space application, indicating a failed notification. The user-space application will try to notify the receiver again at a later point in time (currently unimplemented as this case was not reached during any benchmarks). Rolling back the entire message send operation is not easily possible, as the receiver might have been actively polling for a message and might already be reading it, without seeing the notification first.

Allocation of new channel IDs is managed by the monitor of the receiving core as part of the bind process. The CPU driver does not allocate channel IDs.

Discussion of SCC Message Passing Design

At first sight, it may seem odd to use main memory (rather than the on-tile MPB) for passing message payloads, and to require a trap to the CPU driver to send a message notification. This design is motivated by the need to support many message channels in Barrelfish and, furthermore, more than one isolated application running on a core. The SCC's message-passing

functionality does not appear to have been designed with this use-case in mind [PRB10].

Two further design alternatives were considered: *notification trees* and *payload in MPB*. The former was implemented and benchmarked, but it turned out to have worse performance than the ring buffer implementation presented above. I am going to describe it briefly here, for reference.

Notification trees use the same notification scheme as ring buffers, but employ a bitmap of channel IDs, represented as a two-level tree in the receiver's MPB. One bit for every distinct channel that can be notified. Tree nodes are of the size of one cache-line (256 bits). The tree's layout in the ring buffer is such that the root node occupies the first cache-line. All other nodes are leaves and are stored in left-to-right order after the root node. There are 255 leaves which contain a bit for each notifiable channel, yielding 65,280 notifiable channels. A bit set in the root node indicates that the corresponding leave contains set bits and should be considered when the tree is traversed. In this scheme, sending a notification can never fail: A bit can either be set or is already set, in which case no further notifications need to be sent for the respective channel ID.

Sending a notification for a given channel ID in this design requires the sender to

1. acquire the test-and-set lock for the receiving core,
2. read the root node from the receiver's MPB,
3. set the bit for the corresponding leaf node of the channel ID in the root node,
4. write the root node to the receiver's MPB,
5. read the leaf node from the receiver's MPB,
6. set the bit for the corresponding channel ID in the leaf node,
7. write the leaf node to the receiver's MPB, and
8. read the receiver's interrupt status.

9. If no inter-core interrupt is pending, write the status word with the interrupt bit set to trigger a remote interrupt, and

10. clear the test-and-set lock for the receiving core.

This mechanism requires two full cache-line reads and two full cache-line writes from/to the receiver's MPB and two bit operations as opposed to only two 32-bit reads and two full cache-line writes in the ring buffer scheme. We originally proposed notification trees, assuming the cost to access remote MPBs would be an order of magnitude lower, as well as cheaper bit operations on full cache-lines. After implementing and benchmarking this scheme, it turned out not to be the case. I assume the slow bit operations to be due to the size of the operands. Holding a full cache-line would require 8 integer registers on the Pentium. With only 7 present, we always have to go to memory in order to execute a bit-scan. An expensive operation, especially when the cache does not allocate on a write miss.

The final design devised is *payload in MPB*. In this scheme, instead of notifying the receiver of a message in shared RAM, the message payload itself is written to the MPB, obviating the need for shared RAM. The drawback of this scheme is that it requires complicated book-keeping, as messages are variable size and might not fit into the rather small 8KB receive buffer. It also complicates managing quality of service, as multiple applications now compete for the same receiver MPB. This forbids receiving applications the use of the payload inside the MPB directly, as the MPB has to be freed up as quickly as possible to allow other competing applications to make progress. This requires copying the message out of the MPB into private RAM of the receiver, which makes this scheme more expensive than the two previous ones, especially as SCC caches do not allocate on a write miss.

We conclude, firstly, that relatively few modifications are required to implement operating system message passing primitives for the rather different SCC architecture. Also, we conclude that care has to be taken to implement the right primitives for the hardware at hand in order to keep adequate messaging performance, based on the hardware's features and capabilities.

	Barrelfish	Linux
Throughput (Mbit/s)	2154	1823
Dcache misses per packet	21	77
source → sink HT traffic* per packet	467	657
sink → source HT traffic* per packet	188	550
source → sink HT link utilization	8%	11%
sink → source HT link utilization	3%	9%

* HyperTransport traffic is measured in 32-bit dwords.

Table 2.7: IP loopback performance on 2×2-core AMD.

2.6.4 Application-level Messaging Performance

We evaluate higher-level messaging performance by carrying out a messaging experiment on the IP loopback service provided by Barrelfish. Applications often use IP loopback to communicate on the same host and thus this benchmark stress-tests an important communication facility, involving the messaging, buffering and networking subsystems of the operating system. IP loopback does not involve actual networking hardware and thus exposes the raw performance of the operating system primitives.

Shared-memory commodity operating systems, such as Linux, use in-kernel network stacks with packet queues in shared data structures. On Barrelfish, IP loopback involves a point-to-point connection between two user-space domains that each contain their own IP stack. This design has the benefit that it does not require kernel-crossings or synchronization of shared data structures. By executing a packet generator on one core and a sink on a different core, we can compare the overhead induced by an in-kernel shared-memory IP stack compared to a message-passing approach.

Our experiment runs on the 2×2-core AMD system and consists of a UDP packet generator on one core sending packets of fixed 1000-byte payload to a sink that receives, reads, and discards the packets on another core on a different socket. We measure application-level UDP throughput at the sink, and also use hardware performance counters to measure cache misses and utilization of the HyperTransport interconnect. We also compare with Linux, pinning both source and sink applications to individual cores, while

the network stack executes in the kernel.

Table 2.7 shows that Barrelfish achieves higher throughput, fewer cache misses, and lower interconnect utilization, particularly in the reverse direction from sink to source. This occurs because sending packets as UMP messages avoids any cache-coherent shared-memory other than the UMP channel and packet payload; conversely, Linux causes more cache-coherence traffic for shared-memory synchronization. Barrelfish also benefits by avoiding kernel crossings.

While it would be easy to take the Barrelfish loopback setup and execute it in Linux, this does not work easily for a setup with an actual network card and driver involved, which will have to execute in privileged mode in Linux and thus require kernel crossings and shared memory data structures.

Web-server and Relational Database

A more realistic scenario involves a web-server serving both static and dynamic web content from a relational database.

First, we serve a 4.1kB static web page off the 2×2-core AMD machine to a set of clients, and measure the throughput of successful client requests using httperf [MT98] from a cluster of 17 Linux clients.

On Barrelfish, we run separate processes for the web server (which uses the lwIP stack [lwI]), e1000 driver, and a timer driver (for TCP timeouts). Each runs on a different core and communicates over UMP, allowing us to experiment with placement of processes on cores. The best performance was achieved with the e1000 driver on core 2, the web server on core 3 (both cores on the same physical processor), and other system services (including the timer) on core 0.

For comparison, we also run lighttpd [lig] 1.4.23 over Linux 2.6.26 on the same hardware; we tuned lighttpd by disabling all extension modules and logging, increasing the maximum number of connections and file descriptors to 1,500,000, using the Linux `epoll` event handler mechanism, and enabling hardware check-summing, scatter gather and TCP segmentation offload on the network interface.

Test-case	Request throughput	Data throughput
Barrelfish-static	18697 requests/s	640 Mbit/s
Linux-static	8924 requests/s	316 Mbit/s
Barrelfish-dynamic	3417 requests/s	17.1 Mbit/s

Table 2.8: Static and dynamic web server performance on Barrelfish versus static performance of lighttpd on Linux.

Table 2.8 presents the results of this experiment in the Barrelfish-static and Linux-static rows. The Barrelfish e1000 driver does not yet support the offload features, but is also substantially simpler. The performance gain is mainly due to avoiding kernel-user crossings by running entirely in user space and communicating over UMP.

Finally, we use the same load pattern to execute web-based SELECT queries modified from the TPC-W benchmark suite on a SQLite [SQL] database running on the remaining core of the machine, connected to the web server via UMP. Table 2.8 shows the result of this experiment in the Barrelfish-dynamic row. In this configuration we are bottlenecked at the SQLite server core.

2.6.5 Summary

One should not draw quantitative conclusions from the application-level benchmarks, as the systems involved are very different. Furthermore, enormous engineering and research efforts have been put into optimizing the Linux and Windows operating systems for current hardware. On the other hand, Barrelfish is inevitably more lightweight, as it is less complete. Instead, the benchmark results should be read as indication that Barrelfish performs *reasonably* on contemporary hardware, satisfying the goal of giving comparable performance to existing commodity operating systems from this section.

For the message-passing microbenchmarks, I make stronger claims. Barrelfish can scale well with core count for these operations and can easily

adapt to use more efficient communication primitives if the hardware provides these, as in the case of the SCC, satisfying the goals of scalability and agility. Finally, I also demonstrate the benefits of pipelining and batching of messages without requiring changes to the OS code performing these operations.

Bringing up a new operating system from scratch is a substantial task and limits the extent to which one can fully evaluate the Multikernel architecture. In particular, the evaluation presented in this dissertation does not address complex application workloads or higher-level operating system services, such as a storage system.

2.7 Related Work

Most work on operating system scalability for multiprocessors to date has focused on performance optimizations that reduce sharing. Tornado and K42 [GKAS99, ADK+07] introduced clustered objects, which optimize shared data through the use of partitioning and replication. However, these systems do not employ message passing to keep replicas coherent and shared memory remains the norm.

Similarly, Corey [BWCC+08] advocates reducing sharing within the operating system by allowing applications to specify sharing requirements for operating system data, effectively relaxing the consistency of specific objects. As in K42, however, the base case for communication is shared memory. In the Multikernel model, the operating system is constructed as a shared-nothing distributed system, which may locally share data (transparently to applications) as an optimization and no specific assumptions are made about the application interface.

The Nemesis [LMB+96] operating system, designed to ensure quality-of-service for multimedia applications, also used an approach around fine-grained OS services and message passing, but on uniprocessors and also does not take a variety of different hardware architectures into account. Similarly, the Tessellation operating system [LKB+09] explores spatial partitioning of applications across cores for isolation and real-time quality-of-service purposes. Real-time quality-of-service is not considered in this the-

sis, while applications that are both parallel and interactive are not considered in Tessellation.

More closely related to the Multikernel approach is the fos system [WA09] which targets scalability through space-sharing of resources. The main argument of the fos system is that splitting operating system resources across cores, and even across machines, provides better scalability in multicore and cloud computing scenarios. Only little has been published about the performance and structure of the fos system.

Finally, Cerberus [SCC+11] presents an approach to scale UNIX-like operating systems to many cores in a backward-compatible way that mitigates contention on shared data structures within operating system kernels by clustering multiple commodity operating systems on top of a virtual machine monitor, while maintaining the familiar shared memory interface to applications.

These systems focus on scaling, rather than addressing the diversity challenges that also accompany the trend toward multicore processors. An exception is Helios [NHM+09], which extends the Singularity operating system to support heterogeneous multiprocessors and provides transparent point-to-point inter-process communication using a specifically designed channel abstraction [FAH+06], but does not tackle the problem of scheduling parallel or interactive applications.

Prior work on "distributed operating systems" [TvR85], such as Amoeba [vRT92] and Plan 9 [PPD+95], aimed to build a uniform operating system from a collection of independent computers linked by a network. There are obvious parallels with the Multikernel approach, which seeks to build an operating system from a collection of cores communicating over memory links within a machine, but also important differences: firstly, a Multikernel may exploit reliable in-order message delivery to substantially simplify its communication and improve inter-core communication performance. Secondly, the latencies of intra-machine links are lower (and less variable) than between machines and require different optimization techniques. Finally, much prior work sought to handle partial failures (i.e. of individual machines) in a fault-tolerant manner. This dissertation treats the complete system as a failure unit and is thus able to exploit performance optimizations that are not possible when also considering fault-tolerance, such as eliminat-

ing timeouts on message delivery and being able to send using an arbitrarily large message buffer, without needing additional messages to acknowledge a smaller window of messages that might otherwise be lost.

Several alternative user-space inter-core message passing implementations exist. For example, the FastForward cache-optimized concurrent lock-free queue is optimized for pipeline parallelism on commodity multicore architectures [GMV08]. FastForward is able to enqueue or dequeue pointer-sized data elements in 28.5 to 31 nanoseconds on an AMD Opteron 2218 based system, clocked at 2.66 GHz. The one-way latency of a cache-line sized message on that machine would thus amount to 456 nanoseconds, approximately 2.8 times slower than Barrelfish on the fastest performing AMD-based system considered in this dissertation. However, FastForward provides a proof of correctness, which was not done for Barrelfish's message passing facilities. Similarly, the Kilim [SM08] message-passing framework for Java provides fast and safe message passing among threads of a single protection domain. Unfortunately, nothing precise could be found about Kilim's performance.

2.8 Summary

An enormous engineering effort has been made in optimizing present day commodity operating systems, such as Linux and Windows, for current hardware, and it would be wrong to use the large-scale benchmark results for comparison between these operating systems and Barrelfish. Instead they serve as indication that Barrelfish performs reasonably well on contemporary hardware, indicated by our first goal in Section 2.5.

Since the Barrelfish user environment includes standard C and math libraries, virtual memory management, and subsets of the POSIX threads and file IO APIs, porting applications is mostly straightforward. However, this evaluation does not address complex application workloads, or higher-level operating system services such as a storage system. Moreover, I have not evaluated the system's scalability beyond currently-available commodity hardware, or its ability to integrate heterogeneous cores.

Computer hardware is changing faster than system software, and in particular operating systems. Current operating system structure is tuned for a coherent shared memory architecture with a limited number of homogeneous processors, and is poorly suited to efficiently manage the diversity and scale of future hardware architectures.

Since multicore machines increasingly resemble complex networked systems, I have proposed and described the Multikernel architecture as a way forward. I view the operating system as, first and foremost, a distributed system which may be amenable to local shared-memory optimizations, rather than a centralized system which must somehow be scaled to the network-like environment of a modern or future machine. By basing the operating system design on replicated data, message-based communication among cores, and split-phase operations, one can apply a wealth of experience and knowledge from distributed systems and networking to the challenges posed by hardware trends.

Barrelfish, an initial, relatively unoptimized implementation of the Multikernel, already demonstrates many of the benefits, while delivering performance on today's hardware competitive with existing, mature, shared-memory kernels.

Chapter 3

Scheduling in a Multikernel

This chapter describes the design principles and implementation of the Barrelfish process scheduler architecture. Scalable and agile process scheduling that is able to respond at interactive timescales is a requirement for the execution of a dynamic mix of interactive, parallel applications on a multicore operating system.

Finally, I describe how to apply a particular idea from the HPC literature, namely that of *gang scheduling* within the Barrelfish scheduler architecture to schedule a mix of interactive, parallel applications on commodity machines. I call the resulting technique *phase-locked gang scheduling*, which reduces the amount of inter-core communication required to synchronize per-CPU schedules.

This chapter is structured as follows:

Section 3.1 motivates the Barrelfish scheduler design. System diversity, performance when scheduling multiple parallel applications concurrently, and interactive workloads are discussed.

Section 3.2 surveys scheduling background. Covered topics are parallel scheduling in high-performance computing, scheduling with information

from applications, traditional scheduling in commodity operating systems, as well as recent studies in multicore scheduling.

Section 3.3 motivates why gang scheduling is an important technique to enable parallel interactive applications, by putting the properties of these applications in context with previous HPC applications that benefitted from gang scheduling and showing how an example such application can achieve better response time over best-effort scheduling in the Linux operating system.

Section 3.4 describes the design principles of the Barrelfish scheduler architecture. Time-multiplexing of cores, scheduling at multiple timescales, reasoning online about the hardware, reasoning online about each application, and communication between OS and applications are discussed as the main principles.

Section 3.5 presents five concepts used within the scheduler to achieve the design principles presented in Section 3.4. Deterministic per-core scheduling, scheduler manifests, dispatcher groups, scheduler activations, and phase-locked scheduling are discussed.

Section 3.6 presents the implementation of the scheduler within the Barrelfish operating system.

Section 3.7 surveys related work relevant to the scheduling ideas presented in this chapter.

Section 3.8 summarizes this chapter.

3.1 Motivation

We start by observing that the trend towards multicore systems means that commodity hardware will become highly parallel. Based on past trends, we can assume that mainstream applications will increasingly exploit this hardware to increase productivity. I characterize three key programming models that make use of hardware parallelism:

1. **Concurrency.** The program executes several concurrent tasks that each achieve a part of the overall goal *independently*. For example, an audio player application typically executes one thread to drive the user interface and another thread to play back and decode a music stream. These threads can execute on independent cores.

2. **Parallelism.** The program partitions the overall task into many smaller ones and executes them *in parallel* to achieve a common goal. For example, a 3D graphics drawing program partitions the rendition of an image into a mosaic of smaller images that each core can then process.

3. **Asynchrony.** The program executes a long-running operation as part of a task, but has more work to do. It can execute the long-running operation on one core and execute another task on another, while waiting for the first core to return the result. For example, a web server can process one user's request on one core and already start the next request on another core, while waiting for the operation on the first core to complete.

This dissertation is concerned with *parallel programming*. Within this area, a new class of "recognition, mining, synthesis" (RMS) workloads is anticipated by many to yield attractive applications on commodity platforms [Dub05, LV03, SATG+07, ABC+06]. These workloads have the property that they are both computationally intensive, as well as easily parallelizable. They thus can benefit tremendously from parallel speed-up, while having the potential to run within an end-user scenario.

In the past, highly-parallel machines were generally the domain of high-performance computing (HPC). Applications had long run times, and generally either had a specialized machine to themselves or ran in static partitions thereof; the OS was often more of a nuisance than an essential part of the system. A diversity of hardware architectures, dynamically changing workloads, or interactivity are not the norm in high-performance computing. In contrast, commodity computing has dealt with a variety of asynchronous and concurrent programming models, but parallel computing was not the norm.

Recently, with the rise of RMS workloads and parallel run-time systems, such as OpenMP [Ope08], Intel's Threading Building Blocks [Rei07], and

Apple's Grand Central Dispatch [App09], general-purpose machines increasingly need to handle a dynamic mix of parallel programs with interactive or soft real-time response requirements. This implies that the scheduling machinery will be invoked more frequently than in HPC, as we have to deal with ad-hoc workload changes induced by user input and be able to produce a new schedule within a reasonable time span on the order of milliseconds to be responsive to the user.

Furthermore, the scheduler has to be scalable with an increasing number of cores, as well as agile with the frequently changing hardware architecture. This is in line with the Barrelfish design principles as presented in Chapter 2.

In the following subsections, I am going to address the aspects of agility with a diversity of systems, scheduling multiple parallel applications, and interactive workloads in more detail and point out how current approaches fail in delivering all three of them together.

3.1.1 System Diversity

Parallel application performance is often highly sensitive to hardware characteristics, such as cache structure [ZJS10]. As systems become more diverse, manual tuning for a given machine is no longer an option as it becomes too much effort.

The HPC community has long used *autotuners* like ATLAS [WPD01] to effectively specialize code for a specific hardware platform. However, their usefulness in a general-purpose scenario is limited to the subset of applications amenable to offline analysis, which are dominated by scientific computing workloads [PSJT08].

Dynamic adaptation at run-time still is an option and is in fact made easier by run-time programming models like OpenMP [Ope08], Grand Central Dispatch [App09], ConcRT [Mic10] and MPI [Mes09], which make the communication and synchronization aspects of a program explicit. Such runtimes can improve performance, for example, by dynamically choosing the best thread count for a parallel code section, or accounting for hardware characteristics like whether two cores are on the same package.

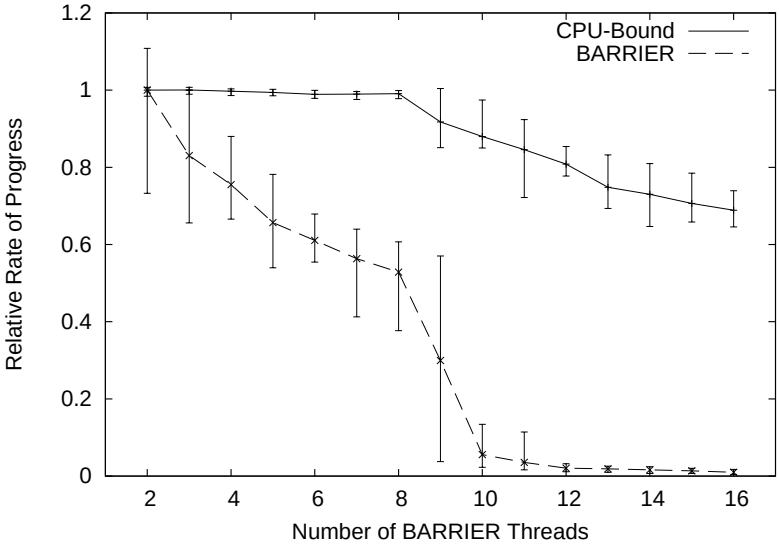


Figure 3.1: Relative progress for 2 concurrent OpenMP jobs on the 4×4-core AMD system. Mean, minimum, and maximum observed over 20 runs of 15 seconds each are shown.

While applying such heuristics within the run-time can work well for batch-oriented HPC jobs on a small range of machines, it may not work across all architectures, and may not be suited for more complex interactive, multi-phase applications which mix parallel and sequential components. They may not even reflect all program requirements, such as a need for some threads to be scheduled simultaneously [Ous82, FR92].

3.1.2 Multiple Applications

HPC workloads have generally enjoyed exclusive use of hardware, or a static partition thereof. However, in a general-purpose system multiple simultaneous parallel programs can interfere significantly.

I show this by example. The main workload for this example (and through-

out the chapter) are two synthetic parallel programs that use the OpenMP [Ope08] runtime system. OpenMP allows programs to control parallelism and synchronization via *directives*. These directives can, among other things, start a parallel region of a program, or synchronize among several threads. I use the Intel OpenMP run-time that ships with the Intel C++ compiler version 11.1 and execute the programs on the Linux operating system version 2.6.32. I found this combination of run-time and operating system to be the most performant. I also examined the GNU OpenMP run-time that ships with the GNU Compiler Collection on the Linux operating system and the Sun OpenMP run-time that ships with the Sun Studio Compiler on the Solaris operating system, but neither of these combinations outperformed the combination of Intel run-time on the Linux operating system.

The first OpenMP program is synchronization-intensive and called *BARRIER*. BARRIER executes a small amount of computation (incrementing a variable in an array) and then executes the OpenMP BARRIER directive to synchronize with all threads in the program. It does so in a tight parallel loop, using the PARALLEL directive, such that it synchronizes with all threads on each loop iteration. The other program simply executes the same computation in a tight loop, without the BARRIER directive and is called *CPU-Bound*.

The Intel OpenMP library uses spin-then-block primitives to implement the barrier operation, where the run-times will spin on a shared variable for a while before relinquishing the CPU to the operating system to make room for another runnable application. This is a reasonable technique for any parallel program executing in a time-shared environment [KLMO91] and dramatically reduces the time taken to execute a barrier operation when the synchronizing threads are executing all at the same time.

Figure 3.1 shows these two OpenMP applications interfering on the 4×4-core AMD system. I measure the progress of each program by the number of loop iterations executed over 15 seconds. I vary the total number of threads demanded by the BARRIER program between 2 and 16 and use a different value for each run of the experiment. The CPU-bound application always uses 8 threads in this example.

When there are fewer *total* threads than cores, BARRIER's progress depends both on the number of threads and how the OS places them on cores.

This is due to two factors:

1. a barrier among threads on the same package costs approximately half as much in terms of execution time as one among different packages. This is due to the inter-core communication latency, which, on the 4×4-core AMD machine, was measured to be 75 cycles between two cores within the same package and 160 cycles between two cores on different packages.

2. The overall cost to execute a barrier rises with the number of threads involved in the barrier operation. This is simply due to the fact that more threads need to be communicated with.

Linux' scheduler is oblivious to the fact that inter-core synchronization is cheaper on the same package and will randomly place BARRIER's threads on cores each time the experiment is executed. This accounts for the high performance variance for BARRIER with low thread counts, even though enough cores are available to schedule all threads simultaneously.

When the applications contend for cores, their performance degrades unequally: CPU-bound slows down linearly, but BARRIER's progress rate collapses. This is due to the fact that preemption of any thread by the CPU bound program can cause synchronization within the BARRIER program to take orders of magnitude longer. This, in turn, is due to the CPU bound program executing for an entire time slice (it does not block) before control is given back to the BARRIER program.

The UNIX adaptive feedback scheduling mechanism that is also employed in Linux and which is supposed to boost the priority of blocking applications, does not provide the desired benefit in this case, as the BARRIER application also spins to execute the barrier operation. There is no simple solution around this problem. For example, we cannot modify the amount of time spent spinning, since we would like to retain the property of being able to execute barriers quickly in the uncontended case and there is no simple way for an application to learn about or influence contention for cores.

This simple study shows the pitfalls of scheduling a mix of workloads on multicore systems. Smart runtimes such as McRT [SATG+07] were not de-

signed to solve this problem, since it is one of lack of coordination *between* runtimes.

3.1.3 Interactive Workloads

Desktop and other interactive workloads impose real-time requirements on scheduling not usually present in HPC settings. Applications often fall into one of three categories:

- Firstly, there are long-running, background applications which are *elastic*: they are not sensitive to their precise resource allocation, but can make effective use of additional resources if available. One example is a background process indexing a photo library and using vision processing to identify common subjects. This is a common example of an RMS workload.
- In contrast, some background applications are *quality-of-service sensitive*. For example, managing the display using a hybrid CPU-GPU system should take precedence over an elastic application using GPU-like cores. While these applications require some quality of service, their requirements change rarely.
- Thirdly, we must handle bursty, interactive, latency-sensitive applications such as a web browser, which may consume little CPU time while idle but must receive resources promptly when it receives user input, leading to ad-hoc changes in resource requirements.

Moreover, multicore programs *internally* may be more diverse than most traditional HPC applications. The various parts of a complex application may be parallelized differently – a parallel web browser [JLM+09] might use data-parallel lexing to parse a page, while using optimistic concurrency to speculatively parallelize script execution.

Traditional work on scheduling has not emphasized these kinds of workloads, where the resource demands may vary greatly over timescales, which are much shorter ($\approx$ 10ms) than found in traditional HPC scenarios.

3.2 Background

This section introduces the scheduling concepts used in this chapter and provides an overview of past work in this area. Parallel scheduling in high-performance computing and traditional commodity OS scheduling are covered here, as well as using application information to manage resources in HPC and grid computing, and recent studies in multicore scheduling. At the very end of this section, I conduct a short study of blocking cost versus busy waiting cost on synchronization primitives to highlight the problems faced in scheduling parallel applications.

3.2.1 Parallel Scheduling in High-Performance Computing

Work on parallel scheduling in high-performance computing is mainly concerned with batch and gang scheduling of long-running, non-interactive programs. Nevertheless, the HPC community is concerned with multicore architectures, as they present a viable and cheap alternative to Supercomputers. In this section, I survey briefly the parallel scheduling work in HPC.

Originally introduced as *coscheduling* by Ousterhout [Ous82], the class of techniques that I collectively refer to as *gang scheduling* includes several variants [FR92, AD01, FFPF05, FFFP03]. All are based on the observation that a parallel job will achieve maximum overall progress if its component serial tasks execute simultaneously, and therefore aim to achieve this through explicit control of the global scheduling behavior. Compared to uncoordinated local scheduling on each processor, gang scheduling is particularly beneficial for programs that synchronize or communicate frequently, because it reduces the overhead of synchronization in two ways.

First, gang scheduling ensures that all tasks in a gang run at the same time and at the same pace. In parallel programs that use the single program, multiple data (SPMD) model, work is typically partitioned equally among threads. This allows threads to reach synchronization points at the same time.

Second, it is typically cheaper for a running task to resume from a period of *busy waiting*, in which it remains active without releasing the processor by *spinning*, than from *blocking* and voluntarily relinquishing control to the local scheduler, allowing another task to execute while it waits. I am going to examine this trade-off more closely in Section 3.2.5. However, if the task spins for a long time, it wastes processor cycles that could be used by other applications. Gang scheduling enables busy-waiting synchronization by reducing average wait times, and thus the time wasted in spinning.

The claimed benefits of gang scheduling include better control over resource allocation, and more predictable application performance by providing guaranteed scheduling of a job across all processor resources without interference from other applications. It achieves this without relying on other applications to altruistically block while waiting. Gang scheduling has also been proposed to enable new classes of applications using tightly-coupled fine-grained synchronization by eliminating blocking overheads [FR92].

Gang scheduling also has drawbacks. Typical implementations for HPC systems rely on centralized coordination for each gang dispatch, leading to problems such as higher context switching cost and limited scalability, compared to local scheduling policies. In the HPC space, these are mitigated through the use of long time slices (on the order of 100 milliseconds or more), or hardware support for fast dispatch synchronization [FPF+02]. For example, the STORM resource manager is able to sustain gang scheduling on a supercomputer with special hardware support at a time quantum of 2 milliseconds [FPF+02] and is able to scale up to at least 64 nodes. The authors claim the possible support of interactive applications, but do not present an evaluation.

Other downsides of gang scheduling include underutilization of processors due to fragmentation when tasks in a gang block or the scheduler cannot fit gangs to all available cores in a time slice. The problem of *backfilling* these processors with other tasks is complex and much researched [Ous82, FFPF05, CML01, WF03].

The first thorough examination of the tradeoffs of gang scheduling performance benefits was conducted by Feitelson et al. [FR92] on a supercomputer with dedicated message passing hardware, and reports that gang scheduling improves application performance when application threads synchro-

nize within an interval equivalent to a few hundred instructions on their system. Otherwise, gang scheduling slows down overall progress due to the rigid scheduling policy, requiring each thread of an application to be scheduled concurrently, even when they wait on synchronization primitives, which defies opportunities to schedule other activities in their place. I find that these tradeoffs still hold, albeit at a time scale equivalent to a hundred thousand cycles on the systems examined in this thesis, and thus the space has widened considerably.

A more recent evaluation of gang scheduling appears as a case study [FE06], which uses a similar synthetic workload to ours, and compares to schedulers in several versions of Linux and Tru64 UNIX. It shows that shorter time slices aid synchronization-oblivious schedulers, and shows that parallel applications tend to self-synchronize.

A study evaluating gang scheduling on clusters shows that spin-then-block mechanisms have performance within 35% of gang scheduling, but lack the scheduler synchronization overhead [DAC96]. This is evidence of the common assertion that gang scheduling has high dispatch overhead.

Another study evaluates several gang scheduling optimizations for dynamic workloads on a cluster [FFFP03]. Using synthetic workloads, the authors find that gang scheduling improves application response time and reduces slowdown and that backfilling fragmented timeslices is only useful within a limited multiprogramming level, which can be improved by using applications that do not benefit from co-scheduling to backfill [FFPF05].

This dissertation is not investigating the effects of bulk program IO under gang scheduling. To an extent, this has been researched in the past. For example, conventional gang scheduling has the disadvantage that when processes perform IO or blocking communication, their processors remain idle because alternative processes cannot run independently of their own gangs. To alleviate this problem, Wiseman *et al.* suggest a slight relaxation of this rule: match gangs that make heavy use of the CPU with gangs that make light use of the CPU (presumably due to IO or communication activity), and schedule such pairs together, allowing the local scheduler on each node to select either of the two processes at any instant [WF03]. As IO-intensive gangs make light use of the CPU, this only causes a minor degradation in the service to compute-bound jobs. This degradation is offset by the overall im-

provement in system performance due to the better utilization of resources. Similar techniques can be employed when gang scheduling on Barrelfish.

3.2.2 Scheduling with Information from Applications

Recent work in HPC uses resource demand specifications to take heterogeneous and performance-asymmetric multicore architectures into account. For example, a scheduling method for heterogeneous multicore processors that projects a core's configuration and a program's resource demand into a multi-dimensional Euclidean space and uses weighted distance between demands and provision to guide the scheduling [CJ09] was shown to improve energy and throughput in throughput-oriented scenarios.

A non-work-conserving scheduler has been designed for throughput-oriented scientific workloads executing on simultaneous multi-threading (SMT) processors [FSS06]. Executing several threads concurrently on these processors can often degrade performance, as many critical resources are shared, such as memory access units. The scheduler uses an analytical performance model to determine when to apply a non-work-conserving policy and thus improves performance.

In grid computing, the environment is more heterogeneous, distributed, and has to deal with more constraints, such as ownership and usage policy concerns and thus scheduling happens at a higher level and is mainly concerned with matching tasks to appropriate processing nodes. Grid computing is relying largely on the Matchmaking [RLS98] algorithm. Matchmaking introduces the notion of *classified advertisements*, which contain resource requirement specifications submitted to the grid computing system along with applications. The matchmaking algorithm tries to match these requirements in the best possible way with offered resources, by splitting the problem of resource allocation into matching and claiming phases and scheduling resources opportunistically. This, again, benefits throughput-oriented applications, but does not work well for applications requiring quick service for short time intervals.

Finally, techniques from quality of service architectures, such as Q-RAM [RLLS97], that allow the specification of minimum and optimum required

resources of hardware components from an application as constraints to the system use a similar approach to the previous two.

None of the work presented in this section is concerned with interactive workloads and thus only applies in limited ways to this dissertation. However, we observe that a lot of factors from application workloads have to be taken into account when scheduling parallel applications within multicore and multi-node scenarios. Otherwise, only limited parallelism can be provided to applications.

3.2.3 Commodity Multicore OS Scheduling

Commodity operating systems running on multicore machines are confronted with the problems of dynamically changing workloads, scalability, and the necessity of interactive response times.

The schedulers of mature commodity operating systems, such as Linux, Windows, and Solaris have evolved from uniprocessors. In the Linux kernel, prior to version 2.6.23, per-CPU run-queues and priority arrays were used that provided adequate scalability for small multicore machines [Sid05]. From version 2.6.23, the *completely fair scheduler* (CFS) was used to combat the problem that the old scheduler was unable to correctly predict which applications were interactive and which ones were CPU-intensive background tasks [lina]. In either case, the schedulers were not designed with the specific needs of multicore processors in mind. For example, the CFS scheduler can exert bad cache effects on multicore machines [And]. Linux introduced *scheduling domains* that can be used to provide better performance on NUMA and shared cache systems, by treating closely-coupled components as a unit, which encourages thread migration and thus optimizes load balancing within the unit, but they have to be configured manually [linb].

Windows has similar scheduling features to those of Linux, but also supports cache-locality for threads and tries to keep threads on the core with highest locality to minimize cache miss effects [win].

Solaris also explicitly deals with the fact that parts of a processor's resources might be shared among several of its cores. A *processor group* abstraction is available that allows to model the system processor, cache, and memory

topology to treat closely coupled components as a scheduling unit [Sax05], similar to Linux's scheduling domains.

Gang scheduling has not yet achieved uptake in general-purpose computing, where time slices generally have to be short in order to guarantee good latency, and parallel synchronization-intensive applications are a niche, though notable exceptions include IRIX [BB95], which supported gang-scheduled process groups as a special application class. Unfortunately, there is no evidence whether this feature found great support among users of the IRIX operating system. Finally, some user-level resource managers for cluster environments [YJG03, Jet98] support gang scheduling, but are used predominantly for HPC.

Commodity OS schedulers are good at providing prompt best-effort service to individual threads. This is good for single-threaded applications and those that use threads for concurrency, the current common workload of these systems. However, when presented with multiple parallel workloads that need fine-grain coordination among threads, these schedulers run into pitfalls, like the one presented in Section 3.1.2. These pitfalls stem from the fact that each thread is treated in isolation, while threads involved in parallel computation might require service as a unit.

3.2.4 Studies in Commodity Multicore Scheduling

Several application workload studies have been conducted to shed light on the impact of multicore architectures on the performance of parallel and desktop workloads and to what extent these workloads can be tuned to benefit more from multicore processors.

A study conducted by Bachthalter *et al.* in 2007 suggests that desktop application workloads, as of then, had little parallelism and suggests that more work needs to be done in parallel programming run-time design and speculative parallelism before workloads will truly emerge [BBF07]. A number of parallel run-times have emerged since then and this is a sign that work in parallel programming language run-time design has indeed occurred, suggesting that the desktop application landscape is rapidly changing towards parallelism.

A study on cache sharing within RMS application workloads finds that current RMS algorithms are not optimized for shared caches and cannot leverage performance benefits when two or more cores share a cache [ZJS10]. The authors transform some of these algorithms to make better use of shared caches and observe up to 36% performance increases. This is evidence that RMS workloads can benefit from shared caches in multicore machines when appropriately tuned, but that care needs to be taken when scheduling multiple of these workloads, such that caches are not over-utilized.

Studies on the effectiveness of informing multicore OS scheduling by performance counters found that considering only a single or a small number of per-core metrics, such as instructions per cycle or cache miss rate, are not sufficient to characterize multi-application workloads [ZUP08]. Another study in contention aware scheduling finds that threads in multicore machines do not only contend for cache space, but also on other shared resources, such as memory controllers, memory bus and prefetching hardware [ZBF10].

These studies suggest that a holistic solution to the scheduling problem is required that takes multiple hardware parameters into account. Also, this is evidence that simply trying to infer an applications' requirements by the operating system via the analysis of simple performance metrics is insufficient. Instead, an integrated approach that directly takes into account the application's resource requirements could be a better solution.

3.2.5 Blocking Cost Evaluation

Synchronization-agnostic scheduling can provide good overall system performance by relying on the ability of applications to sacrifice some of their resources by altruistically blocking execution. Blocking and resuming execution, however, takes time as well and as such incurs a cost to the application.

Gang scheduling eliminates this cost by ensuring all threads of an application always execute at the same time, allowing the application to busy-wait instead. This is another significant benefit of gang scheduling. In this section, I examine this tradeoff.

The measurement of blocking overhead includes managing the kernel queues of blocked threads to block and unblock, which involves system calls on Linux, and the context switch to and back from another runnable application (in another address space), but does not include any time spent waiting while blocked.

The measurement of the cost for busy-waiting is the cost of one wait iteration used to acquire a contended spinlock. This includes the cost to read and modify the memory containing the spinlock, which usually involves fetching a cache line from another core that previously held the lock.

I developed two separate benchmarks, one to measure the cost of blocking, the other to measure busy-waiting. For busy-waiting, I pin two threads to two distinct cores on the same die. Both try to acquire the same spinlock in a tight loop. Whenever the lock is acquired, it is released immediately. On one thread, I measure the time taken to acquire the lock by reading the CPU's time-stamp counter once before trying to acquire the lock and, upon successful lock acquisition, once after. I ignore the cases where the lock could not be acquired.

To measure the overhead of blocking, I employ two processes that synchronize via a Linux futex, which is the only reliable mechanism to block a thread on that OS.

For both experiments, I determine the average time over 10,000 individual measurements. On the Linux system, the cost to block is approximately 7.37 microseconds, whereas busy-waiting requires only 40 nanoseconds on average. Blocking thus incurs a substantial additional cost, two orders of magnitude higher than that of busy-waiting.

I thus conclude that gang scheduling can provide performance benefits in current systems, by allowing applications to busy-wait instead of block.

3.3 Example Workloads

In this section, I give two examples of current workloads that do benefit from gang scheduling, virtual machine monitors and parallel garbage collection.

Finally, I argue why future workloads might greatly extend this set and give an example such workload.

3.3.1 Virtual Machine Monitors

The first example scenario is brought about by virtualization. Virtualization is pervasive today through the wide dispersal of cloud computing platforms that offer compute infrastructure as a service and use virtualization for server consolidation through virtual machine monitors, such as Xen [BDF+03]. Typical virtual machine monitors today offer multiple processors to guest operating systems in order to allow them to make use of parallelism and concurrency.

When para-virtualization [WSG02] is not employed, operating systems employ spin-waiting techniques to synchronize cores. While spin-waiting, a synchronizing core is never relinquished to a higher-level executive, such as a hypervisor. This is a reasonable technique when the OS is running on the bare metal and those cores synchronized with are always executing at the same time, leading to fast hand-over of the synchronization primitive to the waiting core.

On the other hand, spin-waiting techniques can slow down an entire system if virtualization is employed: This happens in those cases where the synchronization peer is currently not scheduled and results in spin-waiting for a potentially long time, using up CPU cycles that could otherwise have been spent by another guest operating system running on the same core. Some commodity operating systems even require that all cores are executing at the same time and periodically check for this condition.

Gang scheduling ensures that synchronization peers are always executing at the same time and thus avoids wasted cycles through spin-waiting. For this reason, some virtual machine monitors employ gang scheduling for performance and correctness reasons [GTHR99, WCS06].

3.3.2 Parallel Garbage Collection

A second example scenario involves parallel garbage collection in modern programming language runtimes. These garbage collectors are typically implemented in a "stop the world" fashion, where a synchronization barrier is executed at the end of a computation phase, before garbage collection can commence. Another barrier is executed when garbage collection is finished to start the next phase of computation.

Parallel run-times typically employ spin-then-block synchronization primitives to implement barriers that spin for a limited amount of time in order to speed up short synchronization wait times and then relinquish the core to another running task by invoking a blocking system call. While using spin-then-block synchronization typically works well with best-effort scheduling, such as employed by the Linux operating system, it will still incur application slowdown if invoked often enough, due to the overhead of a blocking system call [PBAR11], as we have seen in Section 3.2.5.

The frequency of garbage collection depends on memory pressure and the aggressiveness of the garbage collector. Under memory pressure, collection occurs at a frequency close to or shorter than the time slice length of the scheduler. In this case, barrier synchronization will slow down the program under garbage collection if other threads are running concurrently. Gang scheduling can avoid this slow-down by making sure all threads execute concurrently.

3.3.3 Potential Future Workloads

These scenarios are important, but also uncommon, explaining the lack of gang scheduling support in today's general-purpose operating systems. In particular, workloads that combine multiple parallel applications using fine-grain synchronization are especially rare.

However, looking forward, emerging computationally-intensive and potentially interactive "recognition, mining, and synthesis" (RMS) workloads are widely anticipated [SATG+07, ABC+06, BKSL08]. These workloads are attractive to run in an interactive setting. Parallelization allows their use on

ever broader sets of data, while keeping run-time within the attention span of the user.

As an example of a potentially important future workload, I investigate applications employing data stream clustering algorithms. Data stream clustering is considered an important aspect of future RMS workloads, and a version of such an algorithm is included in the PARSEC benchmarking suite [BKSL08], which is designed to be representative of next-generation applications for multicore computers. In particular, I consider an interactive network monitoring application that employs data stream clustering to analyze incoming TCP streams, looking for packets clustered around a specific attribute suggesting an attack. I argue that such workloads will become increasingly important and useful.

Such an interactive application will use a small workload size, which it will try to process quickly using the parallel clustering algorithm to update an interactive display of attack patterns. This leads to more fine-grained synchronization. I confirmed this experimentally on the 4×4-core AMD machine: When run for 3 seconds on a small workload (specifically, the "simlarge" dataset) using 16 threads, the benchmark executes 16,200 barriers with an average processing time of 403 microseconds, and average barrier wait time of 88 microseconds. In this case, synchronization occurs frequently enough to benefit from gang scheduling.

The stream clustering algorithm uses mainly barriers to synchronize. Less than 1% of all synchronization is carried out using locks and condition variables. A good barrier implementation is key to performance in this case. I use a preemption-safe, fixed-time spin-block competitive barrier [KWS97]. While it is suggested to ignore competitive spinning and always block on a barrier synchronization primitive when there is no information about other runnable processes—a feasible technique under synchronization-oblivious schedulers, such as in Linux—it obviates all benefits of gang scheduling. This is why gang scheduled systems typically pick a value that is somewhat larger than the expected average thread arrival time at a barrier. For example, in flexible co-scheduling [FFPF05] a spin time of 120 microseconds is chosen. Scheduler-conscious barrier implementations could potentially further improve performance, but are outside the scope of this dissertation.

I fine-tuned the implementation to use a spin time of 50 microseconds,

which is enough for 16 threads to pass the barrier without having to block on an otherwise idle system at equal barrier arrival times.

To show application performance achieved under gang scheduling, I developed a simple gang scheduler for Linux. This runs as a user-space application in the real-time scheduling priority class, ensuring that it is always scheduled when runnable. It raises gang dispatched processes into the real-time class and lowers de-scheduled processes into the time-sharing class in a round-robin manner. Between dispatches, it sleeps for 15 milliseconds, the effective time slice duration, which I determined to be equal to the average Linux time slice duration. When threads block, the Linux scheduler will pick a process from the time-sharing class—including threads of another gang—to fill the time until the thread becomes runnable again. This approach is similar to cluster gang scheduling solutions previously implemented for Linux [YJG03].

Application performance under this scheduler is below the achievable performance of a kernel-space gang scheduler due to the overhead of running in user-space and using system calls to control program execution, but suffices for comparisons with other scheduling strategies. I have measured the overhead of the gang scheduler indirectly as less than 1%, by measuring iterations of a tight loop composed of two identical processes executing once under Linux and once under gang scheduling.

I compare the response time of the parallel network monitoring application under the default Linux scheduling and under the gang scheduling regime under different rates of network packet burst. Burst in this case refers to the amount of packets arriving in quick succession on a stream. For the purpose of this benchmark, I simplify the metric and send a number of random TCP packets in multiples of 16,000 from a client machine via a Gigabit Ethernet connection to the 4×4-core AMD machine, using a separate port for each stream. I call this multiple the *burst factor*. Each TCP packet contains a random payload of size 64 bytes. The client packet generator will iterate in a round-robin fashion through the streams and send 16,000 packets for each TCP stream. This means that after 16,000 packets have been generated for the first stream, it will generate 16,000 packets for the next, and so on. These bursts, according to the burst factor, are sent out within a fixed interval of 10 seconds to give the packet processor on the 4×4-core AMD machine time to process the clusters.

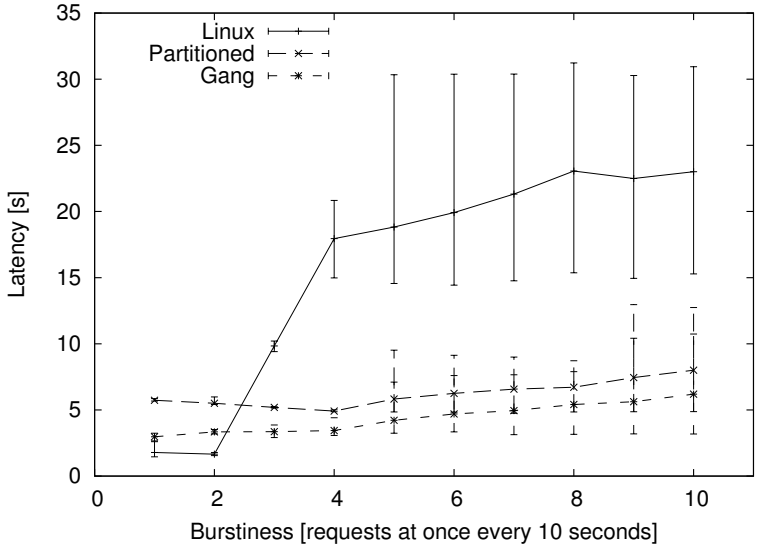


Figure 3.2: Parallel network monitoring under default Linux scheduling vs. gang scheduling and space partitioning over a total run-time of 50 seconds under different rates of packet burst. Error bars show min/max measured over 10 runs.

In this workload, for each of 4 TCP streams, cluster centers are updated every 16,000 packets using a parallel computation. 8 threads are used for each parallel computation, leading to a maximum of 32 threads under full load. When no parallel computation is being executed for a TCP stream, only one thread is being used to collect packets on the stream. When enough packets are collected, this thread will start the 8 other threads for the parallel computation and then wait for all of them to finish computing before continuing to collect packets.

I also compare against a static space partitioning of the workload over the cores of the machine, where only 4 cores can be provisioned for parallel cluster computation of each packet stream, filling up all 16 cores of the 4×4-core AMD machine.

The results of this application benchmark are presented in Figure 3.2. The

graph shows the latency between the arrival of all 16,000 packets on a stream, and the update of the stream's clusters as the burstiness of requests increases. Gang scheduling allows the application to cope with increased burstiness, maintaining an average compute time of 5 seconds with a small variance. Performance under the Linux scheduler degrades as burstiness increases. Furthermore, variance in performance increases considerably. Gang scheduling also outperforms a static partitioning of the machine, because in order to handle the maximum burst rate, only 4 cores could be provisioned for each stream, which slows down the parallel computation of cluster centers.

We conclude that gang scheduling is an viable technique to enable parallel interactive applications with response times adequate of user attention spans. While static space partitioning of a machine might provide similar response times in some cases, it is not elastic enough to provide equivalent response times in bursty scenarios, as it exerts the problem of underutilization when partitions are not used as allocated.

3.4 Design Principles

In this section I present five scheduler design principles that are important for supporting the mix of interactive, parallel workloads we envision to find on a general-purpose multicore OS. While the principles are independent of particular scheduling algorithms, policies or performance metrics, and are applicable to any viable approach which aims at addressing all the layers of the scheduling stack, they were conceived to be implemented in the Barrelfish OS and thus also serve to achieve the design goals presented in Chapter 2.

3.4.1 Time-multiplexing Cores is Still Needed

It is plausible that hardware resources will continue to be time-multiplexed, rather than using spatial partitioning on its own. There are three reasons for this:

First, unlike many HPC and server systems, machines will not be dedicated to individual applications. A desktop computer may run a parallel web browser alongside an application that uses a managed runtime system (such as the Java virtual machine). If parallel programming models are successful, then any of these applications could potentially exploit all of the resources of the machine.

Second, even if a machine contains a large number of cores in total, these may vary greatly in capabilities. Hill and Marty's analysis [HM08] suggests that future multicore architectures might consist of a small number of "big" cores that are useful to allow sequential phases of an application to execute as quickly as possible (and reduce the Amdahl's-law impact of these phases), while a large number of "small" cores can facilitate executing the parallel parts of a program. Access to the big cores will need to be time-multiplexed.

Third, the burstiness of interactive workloads means that the ability to use processing resources will vary according to the user's behavior. Time multiplexing gives a means of providing real-time quality of service to these applications without needlessly limiting system-wide utilization. For instance, an interactive application may use a set of cores in a short burst in response to each piece of user input (say, updating the world in a game), otherwise consuming only a fraction of that much CPU.

Just as in current commodity operating systems, the need to time-multiplex cores means that resource management cannot solely operate by partitioning the resources of a machine at the point when a new application starts. The scheduling regime should be sufficiently nimble to react to rapidly changing resource demands by applications. At the same time however, it must reconcile short-term demands, such as by interactive applications, with coarser-grained requirements, for example a virtual machine monitor needing to gang-schedule the virtual CPUs of a multicore VM. This principle thus serves to achieve the goal of being able to support interactive applications, while being able to respond within short, interactive attention spans.

3.4.2 Schedule at Multiple Timescales

The Multikernel model calls for designs that eschew globally-shared data in favor of decentralized communication. Due to the distributed nature of the system, scheduling in Barrelfish should involve a combination of techniques at different time granularities, much as in grid and cluster scheduling [RLS98]. I argue that scheduling will involve:

- *long-term* placement of applications onto cores, taking into account application requirements, system load, and hardware details – this is where global optimizations and task migration decisions occur;
- *medium-term* resource reallocation, in response to unpredictable application demands, subject to long-term limits;
- *short-term* per-core (or hardware thread) scheduling, including gang-scheduled, real-time and best-effort tasks.

The mix of workloads and the need for interactivity in general-purpose systems means that, in contrast to prior work in HPC systems, gang scheduling will need to occur over timescales typical of interactive timeslices (on the order of milliseconds) to be able to respond to the user promptly. As a result, the overhead and synchronization accuracy of dispatching a gang of threads across decoupled cores becomes much more significant than in the past. Firstly, we would like to avoid costly synchronization mechanisms such as inter-processor interrupts. Secondly, because gangs will be dispatched more frequently and for shorter durations, any penalty from "wasted" overlap time in which some but not all threads in the gang are running will be multiplied, leading to wasted processing time for all cores in the gang.

3.4.3 Reason Online About the Hardware

New processor and system architectures are appearing all the time [Int09]: portability as hardware evolves is as critical as portability across different processor architectures. Both OS and applications must be able to adapt

well to diverse hardware environments. This requires reasoning about complex hardware, beyond what can be achieved by offline autotuners or careful platform-specific coding.

The performance of parallel software is closely tied to the structure of the hardware, and different hardware favors drastically different algorithms (for example, the performance of Dice and Shavit locks [DS09] depends critically on a shared cache and the placement of threads on distinct cores, as opposed to other options [HSS05, SS01]). However, the appropriate choice at runtime is hard to encode in a program.

Adapting scheduling policies to diverse hardware, whether across applications or among threads in a single program, requires

1. extensive, detailed information about the hardware in a usable form, and
2. a means to reason about it online in the scheduler, when system utilization changes. For example, whenever applications are started or stopped.

Limited versions of such functionality exist, e.g. the `proc` and `sys` file systems on Linux, and the CPUID instruction on x86 processors. However, these APIs are complex and specific to particular hardware and OS components, making it non-portable to process their contents. Although performance models such as Roofline [WWP09] and LogP [CKP+93] help by capturing some performance characteristics of available hardware, we would like to explicitly address the broader problem.

3.4.4 Reason Online About Each Application

In addition to the need for applications to exploit the structure of the hardware on which they are running, the OS should exploit knowledge of the structure of the applications which it is scheduling. For example, gang scheduling can eliminate the problems shown in Section 3.1.2 by avoiding preemption of synchronized threads. However, simple gang scheduling of

all threads within applications is overly restrictive. For instance, OpenMP typically only benefits from gang scheduling threads within each team. Similarly, threads performing unrelated operations would favor throughput (allocation of as much time to all threads as possible) over contemporaneous execution. Finally, a single application may consist of different phases of computation, with changing scheduling and resource requirements over its lifetime. The optimal allocation of processor cores and memory regions thus changes over time.

An application should expose as much information about its current and future resource requirements as possible to allow the OS to effectively allocate resources. This especially concerns information about the possible phases of parallel execution they might go through and the resources they project to require within each phase. For example, MapReduce applications follow fixed data-flow phases with different resource requirements and can project these requirements to some degree of accuracy at program start [ZKJ+08]. It is also possible to determine this information at compile-time for programming paradigms like OpenMP [WO09].

3.4.5 Applications and OS Must Communicate

The allocation of resources to applications requires re-negotiation while applications are running. This can occur when a new application starts, but also as its ability to use resources changes (in an extreme example, when a sequential application starts a parallel garbage collection phase), and in response to user input or changes in the underlying hardware (such as reducing the number of active cores to remain within a power budget).

Hints from the application to the OS can be used to improve overall scheduling efficiency, but should not adversely impact other applications, violating the OS scheduler's fairness conditions.

Efficient operation requires two-way information flow between applications and OS. First, applications should indicate their ability to use or relinquish resources. For instance, an OpenMP runtime would indicate if it could profitably expand the team of threads it is using, or if it could contract the team. Secondly, the OS should signal an application when new resources are allocated to it, and when existing resources are preempted. This allows the

application's runtime to respond appropriately; for instance, if the number of cores was reduced, then a work-stealing system would re-distribute work items from the work queue of the core being removed.

De-allocation is co-operative in the sense that an application receives a de-allocation request and is expected to relinquish use of the resources in question. If this is not done promptly then the OS virtualizes the resource to preserve correctness at the expense of performance. For example, if an application is asked to reduce the size of a gang scheduled OpenMP team from 4 threads to 3, but does not respond, then it would find the team being multiplexed over 3 hardware threads rather than gang scheduled.

The need for communication between application and operating system has also been called for in the database community, where database management systems traditionally go to great lengths to second-guess and replace OS functionality, because they have more information about their workload requirements. This does not function well when other applications are executing concurrently in the system, and systems, like Cod [GSAR12], have been proposed to allow for more communication among databases and operating systems to alleviate this problem.

3.4.6 Summary

Due to the interactive environment and heterogeneous workload mix, general-purpose multicore computing faces a different set of challenges from traditional parallel programming for HPC. To tackle these challenges, this dissertation advocates for a principled approach which considers all layers of the software stack. I have outlined a general set of such principles.

3.5 Scheduler Concepts

In this section, I present five scheduler concepts employed in Barrelfish:

1. Dispatcher groups,

2. scheduler activations,
3. deterministic per-core scheduling,
4. phase-locked scheduling, and
5. scheduler manifests.

Together these concepts realize the design goals presented in the previous section. Some of these concepts serve to realize spatial scheduling, such as scheduler manifests and dispatcher groups. Deterministic per-core scheduling serves to realize temporal scheduling. Scheduler activations are needed in support of both temporal and spatial scheduling. Finally, phase-locked scheduling serves to support spatio-temporal scheduling at fine-grained time slices.

3.5.1 Dispatcher Groups

Dispatchers implement a user-space mechanism to multiplex a time fraction of a single physical core among multiple threads of an application. The upcall mechanism is used to notify dispatchers of the start of a timeslice to facilitate user-level thread scheduling and dispatchers can communicate directly to block and unblock threads within an application. Parallel applications typically have many dispatchers, at least as many as physical cores have been allocated to the application and an application is able to control how its threads are scheduled on each core. To facilitate this process, dispatchers can form groups that jointly submit resource requests to the operating system.

For example, in the parallel network monitoring application shown in Section 3.3.3, we have a phase of synchronization-intensive parallel processing and thus can create a dispatcher group for all threads used to execute a new round of this phase and request threads to be gang scheduled across the group. Other threads that are active to collect network packets are running within a different dispatcher group.

Dispatcher groups extend the notion of RTIDs [RLA07] that were conceived in the McRT [SATG+07] parallel run-time system and aggregate requirements such as real-time, gang scheduling and load-balancing parameters.

Membership of dispatcher groups may vary dynamically with workload. For instance, a managed runtime using parallel stop-the-world garbage collection would merge *all* its dispatchers into one group during collection, and then divide them into several groups according to the application's work once garbage collection completes.

Dispatcher groups are an important concept to minimize communication overhead when conducting application execution phase changes and thus serve to aid in fast baseline scheduler response time and scalability.

3.5.2 Scheduler Activations

Scheduler activations [ABLL91] are used to inform applications of changes to resource allocations made by the operating system scheduler and serve to achieve the goal of OS-application communication. Scheduler activations can readily be implemented using Barrelfish's schedule upcall mechanism, as described in Section 2.5. Historically, scheduler activations are employed to inform applications of resource allocation changes and several operating systems had a similar notification mechanism, such as Psyche [SLM+90].

Currently, a data structure in shared memory that is mapped read-only into application domains is employed. There is one such structure for each dispatcher of the domain. The structure is updated by the core-local CPU driver scheduler to show the current core allocation in terms of real-time scheduling parameters, such as scheduling period and allocated CPU time within that period. The application can make use of this information to determine how much service it receives on each core and decide to manage internal resources based on this information.

3.5.3 Deterministic Per-core Scheduling

Deterministic per-core schedulers in the CPU drivers that schedule user-space dispatchers form the basis of the Barrelfish scheduling system. They are required to realize phase-locked scheduling and facilitate the specification of processor requirements within scheduler manifests, as described in the next sections.

A deterministic per-core scheduler ensures that, in the long run, a once defined schedule is being followed deterministically and can be predicted, even when outside events might change it. For example a deterministic scheduler can ensure that a thread is generally scheduled at a fixed rate and with a fixed execution time. This may happen along with deterministically scheduled threads on other cores. Real-time schedulers typically fulfill this requirement and we will use one such scheduler within Barrelfish.

The benefit of deterministic scheduling is that it can reduce inter-core communication overhead by allowing us to specify one longer term schedule for a core that we know it will follow without needing any further communication. This serves to achieve the goals of scalability and the support of efficiently time multiplexing cores.

3.5.4 Phase-locked Scheduling

Phase-locked scheduling is a technique whereby we use the deterministic property of the per-core schedulers to achieve a coherent global schedule without steady inter-core communication requirements. With phase-locked scheduling, cores need to agree on a deterministic schedule only once, at the start of that schedule. It thus serves to achieve the goal of scheduler scalability.

For this to work, all cores need to see the same time. In addition, to support preemptive multi-tasking, all cores need to produce timer interrupts at the same time. Many systems include a system-wide global timer, but it might be too expensive to read when scheduling at fine time granularities. For example, reading the *real-time clock* (RTC) or the *programmable interval timer* (PIT) on the x86 architecture takes ten thousands of cycles. The *high-precision event timer* (HPET) improves read time, but is not ubiquitously available on x86 platforms. The x86 core-local *time-stamp counter* (TSC) is cheap to read (25 cycles were measured on the evaluation systems of this dissertation), but is not always synchronized among cores and can drift if cores go into lower power modes. These issues, for example, were the cause for numerous deadlock problems in the Linux kernel [New]. In addition, the TSC cannot be programmed to interrupt the processor, which is required for preemptive scheduling. Finally, global timers may not always be available

in every system. For example, the SCC does not have such a timer. The core-local APIC timers remain as a source of interrupts. However, these are not synchronized.

We thus synchronize the scheduler timer sources of all cores involved in phase-locked scheduling and set up scheduling parameters, such that a coherent global schedule is achieved. Once a schedule has been agreed upon, no further inter-core communication is required, except for (infrequent) timer re-synchronization if scheduler timer sources experience drift. In the following subsection, I evaluate timer drift and synchronization overheads for the x86 architecture.

Timer Drift

Phase-locked scheduling critically relies on cores always reading identical timer values and getting timer interrupts at the same time. On the test platforms, cores have local timers that need to be synchronized. It is thus reasonable to ask whether these timers exert drift over time, in which case they would need to be periodically re-synchronized when they drift too far apart.

I ran experiments on the 2×2-core AMD and 4×4-core AMD systems to examine whether their core-local APIC timers experience drift over time. In the experiments, I synchronize the core-local APIC timers once, using the scheme presented in Section 3.6.3, set them into periodic mode with a divisor of 1, without interrupts, and then measure APIC timestamps every second on each core.

In this benchmark, cores are organized into a ring. Each timestamp measurement is initiated from a core designated at benchmark start to be the measurement coordinator, which signals a new measurement by flipping a shared bit on an individual cache line in memory, which the next core in the ring polls. Once the bit has changed, the next core will read its local APIC timer and then set another shared bit for the next core in the ring and so on. Each core stores each timestamp measurement individually in a private array in memory. Once the benchmark is stopped, each core will print out its timestamp values and they can be compared after the experiment, checking whether their difference grows over time.

Note that with this method, each core's reading naturally will have an offset of the previous core's reading due to the ring organization of read-outs. I have made sure that this difference is constant over time and can be removed when comparing timestamp values.

I conducted these experiments for a day on each machine and did not encounter an observable growth in difference among APIC timer values, signifying that timers do not drift within this time span. All timestamp values stayed within the baseline fluctuation of measured values and no trend of drift away from this baseline was observable.

If timers do drift over larger time spans, then they have to be re-synchronized periodically. The overhead for an APIC timer sync is currently on the order of several hundreds of milliseconds, a very low overhead if timers need to be synchronized less frequently than daily.

I conclude that, while a global, interrupt-generating timer that can be read with low overhead is required for scalable scheduling of parallel workloads on multicore machines, it is possible to simulate this global timer with low overhead by periodically synchronizing local timers, such as APIC timers, as long as they can be read with low overhead and generate interrupts. This is due to the fact that local timers do not (or, if at all, rarely) drift in a multicore system and the effort required to re-synchronize them is minimal if a global reference clock is available, even if it is expensive to read or does not generate interrupts.

Phase-locked Gang Scheduling

Phase-locked scheduling can be used to realize a variety of different parallel schedules without the need for constant communication among cores to synchronize the same. Here, I describe how phase-locked scheduling can be used to realize gang scheduling. I make use of the phase-locked scheduling capabilities of the Barrelfish scheduler architecture, as follows:

- I synchronize core-local clocks out-of-band of scheduling decisions that must occur with low latency,

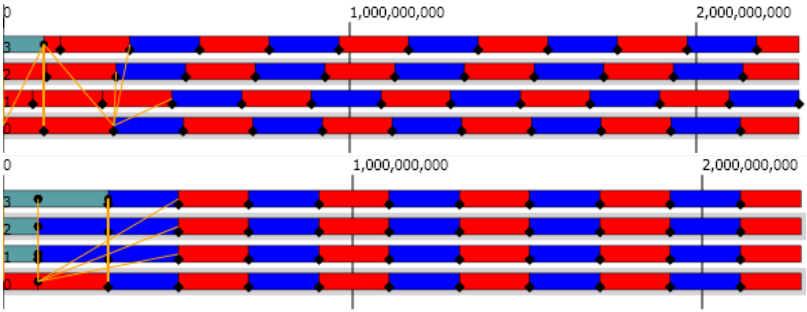


Figure 3.3: Two Aquarium traces of a phase change conducted by the BARRIER (red) application, once into best-effort scheduling and once into gang scheduling, while the CPU-Bound (blue) application that is not gang scheduled is executing concurrently. The x-axis shows time (cycles). The y-axis shows cores. Lines depict messages being sent and received among cores. Black diamonds depict the end of a timeslice.

- I coordinate core-local schedules only at times required by workload changes, and
- I deterministically and independently schedule processes locally on the cores.

This ensures that gangs are co-scheduled without the need for potentially expensive inter-core communication on every dispatch.

When the application switches into a dispatcher group configuration that employs fine-grain synchronization, the involved core-local schedulers communicate to conduct a scheduler requirement phase change to start gang scheduling the affected dispatcher groups.

To illustrate how phase-locked gang scheduling works and how it compares to synchronization-oblivious scheduling, I show an execution trace taken using the Aquarium trace visualization framework [Isa10] and shown in Figure 3.3. Fine-grain execution tracing is a feature in Barrelfish that is able to record system execution events with CPU cycle accuracy through all layers of the system stack (applications and operating system). Aquarium is a program that is able to visualize these traces, by displaying them on a graph

that represents time on the x-axis and cores on the y-axis. Colored boxes in the graph represent the execution of a particular program (according to the color) at the point in the graph and lines represent messages being sent among cores. Finally, diamonds depict the end of a timeslice.

In this example, I use again the BARRIER and CPU-Bound applications described in Section 3.1.2. I show a phase change conducted by the BARRIER application (red), once into best-effort scheduling (upper trace) and once into gang scheduling (lower trace), while the CPU-Bound application (blue) that is not gang scheduled is executing concurrently. Both applications use 4 cores in this case and the phase change is conducted across all cores of the 2×2-core AMD machine, on which this trace was taken. The lines at the left-hand side of the graph show communication among schedulers to conduct the phase change on behalf of the requesting application.

The solid lines in the trace show inter-core communication. We can see that, after the phase change is conducted, no more communication is necessary and local APIC timers are synchronized. In a conventional implementation of gang scheduling (not shown in the Figure), communication would be required at the end of every timeslice (that is, at every row of diamonds in the figure).

3.5.5 Scheduler Manifests

At startup, or during execution, Barrelfish applications may present a *scheduler manifest* to the scheduler to allow the OS to infer information about the applications' resource requirements. A scheduler manifest contains a specification of predicted long-term processor requirements over all cores, expressed as real-time scheduling parameters. In particular, worst-case execution time, scheduling period, deadline and task release time can be specified. This allows the expression of the placement of synchronization-intensive threads on gang scheduled cores. For example, in order to provide a phase-locked gang scheduling policy, we set up identical release time, deadline, worst case execution time and period of all cores involved in gang scheduling a particular process.

Release time is an absolute time value that designates when a particular task in the schedule is to be considered schedulable. If the current time is less

than the release time, the scheduler will ignore the task. Release time allows us to submit tasks to the run-queue early and time precisely when they start being scheduled. This is a requirement for phase-locked scheduling, where it is important that all tasks start their period precisely at the same time.

Scheduler manifests are divided into a number of dispatcher groups, its member dispatchers, and resource requirement specifications divided into several *phases* of execution, which may have different resource requirements. Despite the name, phases do not have to be iterated in any fixed sequential order by applications. Instead, applications can switch between phase configurations at any time using a call to the scheduler API.

Phases are useful for applications that have different execution states and require different processor resources therein. For example, a web browser might use a best-effort thread to manage each web page it is displaying. Some of these web pages might display a video stream when clicked on, in which case they require guaranteed periodic service and can be put in a special real-time scheduling phase for this. Finally, a web page might start a script that uses a parallel algorithm and requires gang scheduling of all threads executing that algorithm. A phase can be allocated for this and the threads be put into that phase.

Figure 3.4 shows an example manifest with two phases: The first phase is a regular best-effort scheduling phase, useful for those threads of applications without special requirements. The second phase specifies deterministic scheduling at a period of 160 milliseconds, a timeslice length of 80 milliseconds (specified by identical worst-case execution time and deadline), and a release time of 480 milliseconds in the future from activation of the phase.

These parameters have to be picked by hand in the current version of the system. In particular, this means that a reasonable value for the release time has to be picked, such that all planners have executed the phase change until the future release time is reached, which is system dependent. In the future, the system should determine the release time automatically and spare the user from specifying this parameter. In order to do so, the planners need a fixed priority in the system to guarantee phase changes to be executed within a certain maximum time span, which should be the lower bound of the release time. We leave this issue for future work.

```
# Best-effort phase with scheduling priority of 1
B 1

# Deterministic scheduling
# WCET  period  deadline  release time
H 80    160     80        480
```

Figure 3.4: Example manifest specifying one single-threaded, best effort phase and one gang scheduled phase. A `B` specifies best-effort scheduling with the scheduling priority as parameter. An `H` specifies deterministic scheduling with worst-case execution time (WCET), period, deadline and release time as parameters, all specified in milliseconds, in that order. Hash characters introduce comments.

Applications will not receive more than best-effort resources if a scheduler manifest is not presented to the operating system. Currently, these manifests are written by application developers. However, I expect that much of the information in manifests could be inferred by compile-time analyses or provided by language runtimes instead, such that application developers would not need to be involved in their specification.

Scheduler manifests resemble similar manifests submitted by applications to grid computing systems or process manifests in the Helios operating system [NHM+09]. Some current commodity operating systems provide scheduler hints, which allow the application to impact OS scheduler policy and are thus related to scheduler manifests. For example, similar to a technique known as "preemption control" in the Solaris scheduler [Mau00], a Barrelfish dispatcher with a thread in a critical section may request to receive more slack in order to finish the section as quickly as possible.

Another alternative to scheduler manifests is embodied by the Poli-C system [And12], which presents a more dynamic way of negotiating resources, by allowing programs to request resources via program annotations that are submitted when reached in the execution. This has the benefit of more dynamism, but also the drawback of less time when calculating a new schedule for the system. I conjecture that it would be possible to replace scheduler manifests with this mechanism and get the same trade-off.

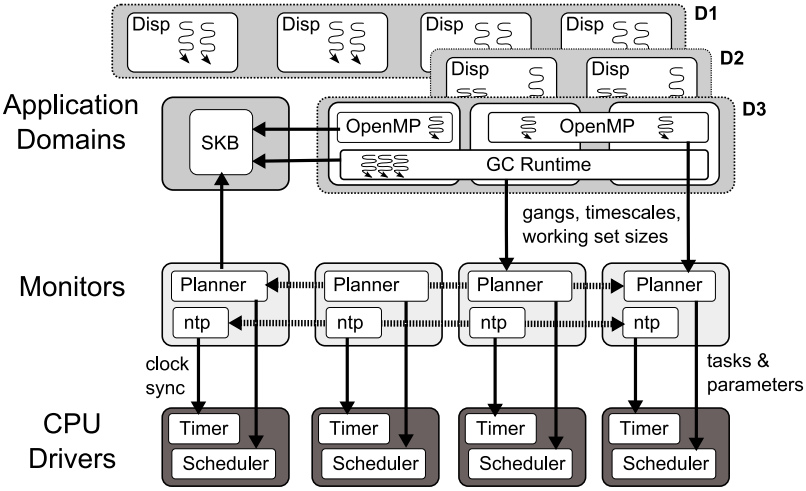


Figure 3.5: Barrelfish scheduler architecture. Arrows represent communication paths among the components. Swirled arrows represent threads and dashed arrows represent multicast communication among planners. *ntp* is short for node time protocol, which is the protocol used to keep timers in sync among cores.

3.6 Barrelfish Scheduler Architecture

In this section, I present the implementation of the processor scheduler subsystem within the Barrelfish general-purpose operating system, applying the concepts introduced in the previous section.

Figure 3.5 illustrates the scheduler architecture on Barrelfish for reference. Two components of the OS run on each core: the CPU drivers in privileged mode, and the monitors in user mode. Also, application domains run dispatchers on every core on which they are executing (labeled 'Disp' in the diagram), and these provide core-local, application-specific, user-level thread scheduling functionality. Dispatchers are scheduled, in turn, by schedulers executing in the CPU drivers via a two-level scheduling hierarchy.

The diagram shows three hypothetical user-application domains: the domain D1 is shown as running on four cores (hence having four dispatchers),

D2 on two cores, and D3 on three. D3 is using two OpenMP parallel sections and a shared garbage-collected run-time. The SKB, shown alongside the other user-level applications, is both queried and populated by user domains as well as by the OS (via the monitors). I will refer back to this diagram as I explain the individual components in the following subsections.

3.6.1 Placement Controller

A centralized placement controller is embedded in the SKB. The placement controller uses the application's scheduler manifest along with knowledge of current hardware utilization to determine a suitable set of hardware resources for the application, which may then create dispatchers and negotiate appropriate scheduler parameters on those cores. In general, an application is free to create a dispatcher on any core, however only by negotiating with the placement controller will it receive more than best-effort resources.

The Barrelfish scheduler API contains a call `rsrc_manifest()` to submit a scheduler manifest to the placement controller as plain text within a variable length character array. It returns an identifier for the submitted manifest, which is used in subsequent API calls to refer to this manifest. Once a manifest is submitted, dispatchers of the same domain can join dispatcher groups via a specially provided `rsrc_join()` system call. Joining a dispatcher group is always possible. Dispatchers are not in any dispatcher group until they join one. This call assumes two parameters: a manifest identifier and a dispatcher group identifier. Manifest and dispatcher identifiers are encoded as 32-bit integers. One dispatcher group was sufficient to conduct the experiments presented in Chapter 4 and so not more than one group has been tested in the current version of the system.

In the version of the system used for experiments in this dissertation, the placement controller executes a load balancing policy to distribute application load over all available cores, based on utilization information that is communicated with the SKB. When a resource request cannot be fulfilled, the placement controller will reduce allocations of all running processes and the requesting process equally until the request can be fulfilled. This global allocation policy can be changed by the system administrator.

The placement controller determines appropriate processor allocations according to the global allocation policy for all different combinations of application phases iteratively and transmits a schedule for each combination separately to the planners, as soon as it is derived. This enables a fast phase change, as all required scheduling information will already be available at the appropriate core-local planners when the phase change occurs and need only be submitted to their respective core-local CPU driver schedulers. It thus serves to achieve the goals of fast scheduler responsiveness and scalability.

3.6.2 Planners

Barrelfish implements per-core user-space *monitors* to implement core operating system services that do not necessarily belong in kernel-space (see Chapter 2 for a description).

I extend this component with a *planner* module that is responsible for the core-local planning of CPU scheduling information for each dispatcher. In vein of the Multikernel design, information about dispatcher groups and gangs is replicated across those planners that are responsible for the per-core resource control of the groups and kept up-to-date via message passing instead of sharing state. When dispatchers form groups across cores, planners communicate to replicate the group's scheduling information.

This design allows us to seamlessly draw long-term resource allocation decisions centrally within the placement controller when this has a clear advantage, like the admittance of new allocations and the allocation of CPU time across a number of CPUs, while keeping performance-critical decisions, like whether a dispatcher is currently in a group requiring gang scheduling, decentralized.

Planners receive schedule information for all different combinations of application phases from the placement controller as soon as these new schedules have been derived. Upon dispatcher group phase changes, planners communicate with corresponding peers on other cores to fulfill the phase change quickly. If scheduling information for a particular combination of phases is not yet available, planners will resort to best-effort scheduling until scheduling information is available.

A phase change is executed by submitting the required real-time scheduling parameters to the core-local CPU driver scheduler independently on each core that is involved in scheduling a dispatcher group that underwent the phase change. When communicating with peer planners to conduct a phase change, only the phase identifier, which has been previously received from the placement controller, needs to be transmitted. Peer planners already have all information required to schedule the phase available locally and need only look up the phase identifier in a local lookup table, which yields the real-time scheduling and phase synchronization parameters. Again, this serves to achieve fast response times and scheduler scalability.

3.6.3 Core-local Timer Synchronization

I have extended the monitor with another module responsible for synchronizing the core-local APIC timers that exist in the x86 and SCC architectures, across all CPUs in the system. This can be done either by referring to a global reference clock, or by using a time synchronization protocol that uses round-trip time of messages, akin to the network time protocol [Mil91].

I have implemented two timer synchronization algorithms that both use a global reference clock to synchronize. The first refers to the core-local time-stamp counters (TSCs) as a global clock source. While TSCs are core-local, they are synchronized on many systems, acting like a global clock. On systems where this is not the case, the programmable interval timer (PIT) is used instead. The PIT is a global clock source in an x86 system, accessible by every CPU core.

In both cases, one core initiates a new *clock synchronization period* by executing the following protocol:

1. Read a time-stamp from a suitable reference clock
2. Add a constant that allows enough time to finish synchronization among all cores
3. Request to synchronize timers by sending the resulting time-stamp to all other cores in the system

Once received, all other cores stop regular system execution, waiting for the reference clock to reach or surpass the transmitted time-stamp. When the time-stamp is reached, all cores synchronously reset their APIC timers, which enforces synchronization.

In the case that the reference clock is already past the time-stamp before the first read-out, some cores will return failure to the initiator of the synchronization period and another period will have to be started by the initiator. In the current system, I start by allowing one timeslice (typically 80ms) to synchronize and use exponential back-off to allow for more synchronization time whenever this is not enough. I stop when synchronization takes longer than one second, upon which I return error to the user and disable phase-locked scheduling.

3.6.4 CPU Driver Scheduler

The CPU driver scheduler is responsible for scheduling dispatchers. Barrelfish supports different types of workloads at the lowest level of CPU allocation. The scheduler in the CPU driver (see Figure 3.5) is based on RBED [BBLB03], a rate-based, earliest deadline first scheduling algorithm, which supports simultaneous scheduling of applications with hard real-time, soft real-time and best effort scheduling requirements, while remaining flexible in distributing the slack (idle CPU time) generated naturally by this scheduling technique back to the tasks it is scheduling. In combination with techniques for scheduling at different timescales this gives the ability to time-multiplex CPUs over a large number of cores while still accommodating for the time-sensitive requirements of some applications.

Within RBED, an earliest-deadline-first scheduling model is used to which all task requirements are fitted. RBED ensures that admitted hard real-time tasks get their requested allocations, while soft real-time tasks get notified about missed deadlines. Finally, best-effort tasks are scheduled equivalent to priority based schedulers. For hard and soft real-time tasks, RBED requires scheduling parameters, such as deadline, worst-case execution time and period. Best-effort tasks are scheduled with priorities identical to UNIX-based schedulers.

I chose RBED for its predictability. RBED produces schedules that repeat periodically under stable resource allocations and is a prerequisite for phase-locked scheduling. For example, gang scheduled tasks are submitted as hard real-time tasks to RBED. For hard real-time tasks, RBED ensures exact service when worst-case execution times, deadlines and periods are adequately chosen. Gang admittance during scheduler manifest submission has already ensured that all gangs are configured so that a combination of gangs that would over-commit the real-time scheduler cannot be submitted.

The Barrelfish implementation of RBED resides entirely within the CPU driver. I have added a release time scheduling parameter, which is not described in the RBED paper. The interpretation of some other scheduler parameters is also different than described in the RBED paper. The execution of a real-time task for a given period ends either if:

1. The task runs out of budget. In this case its executed time is larger or equal to its worst-case execution time (WCET). This may happen for any real-time task.

2. The task yields the CPU. When this happens the task's executed time is reset to 0, while its release time is increased by its period. This prevents it from being re-scheduled within the same period in case slack time is available and restores its budget for the next period. Real-time tasks specify their worst-case execution time for each period and we interpret yielding the CPU before expiration of that time to indicate that the task is finished processing before its worst-case assumption. Thus, it is not necessary to re-schedule the task within that period. Yielding best-effort tasks are immediately re-released at the current time, which ensures that they are re-scheduled whenever slack becomes available, but after all currently runnable best-effort tasks have had their turn.

Figure 3.6 shows a diagram depicting the two situations. If no real-time task has any budget, idle CPU time is not filled with real-time tasks. This decision was arbitrary through the implementation of RBED within Barrelfish and was left unspecified in the RBED paper.

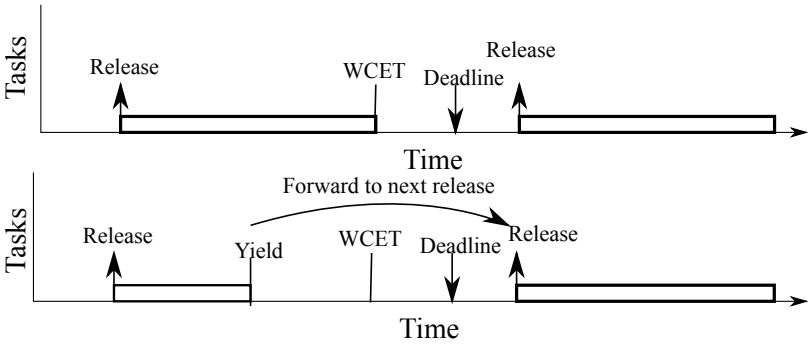


Figure 3.6: RBED scheduling behavior in Barrelfish's implementation. The top graph shows the first case, the bottom graph shows the second case.

3.6.5 User-level Scheduler

Scheduling of application threads is done at user-level via dispatchers that are implemented in the Barrelfish library operating system, which is linked to every application. Dispatchers are put in control of execution via scheduler activations. In particular the Run upcall, first described in Section 2.5.2, at execution resumption from the CPU driver scheduler is used for this purpose.

Upon being entered at the Run entry point, a dispatcher will first poll all its message channels for new messages and keep track of the current timeslice, by incrementing a timeslice counter each time it is entered at the Run entry point. The current timeslice is important in handling yielding threads: It is used for determining when all threads in the current timeslice have yielded, in order to yield the entire dispatcher, instead of switching to the next runnable thread.

After that, the next runnable thread is scheduled using a round-robin policy, by selecting it from the top of a queue of runnable threads. If no threads are runnable, the dispatcher will indicate to the CPU driver that it has no more work to do, by setting the `haswork` field in the shared data structure to zero, unless there are channels to poll periodically, in which case it will simply yield the CPU to the next dispatcher. The CPU driver will activate the

dispatcher again when notifications arrive for it and reset the `haswork` field to one. There are no other scheduling parameters associated with threads in the current version of the system and all threads are treated equally.

Cross-core thread management is also performed in user space. The thread schedulers on each dispatcher exchange messages to create and unblock threads, and to migrate threads among dispatchers (and hence cores). CPU drivers are responsible only for multiplexing the dispatchers on each core.

In order to keep capabilities consistent across cores, a message to send capabilities to a remote dispatcher is implemented. In the current version of the system, capabilities are transferred ad-hoc and only once for those services that need consistent capabilities during setup of the service. A generic service for consistent capabilities does not yet exist and is future work.

3.7 Related Work

User-level parallel programming frameworks have investigated using information from applications to facilitate the scheduling of threads at the run-time level. For example, scheduling in the McRT [SATG+07] integrated language run-time system for large-scale multicore machines involves maximizing resource utilization and providing the user with flexible schedule configurability, which is facilitated by McRT's tight integration with the programming language, by allowing application programmers to specify scheduling preferences within their code [RLA07]. McRT shows improved performance and scalability over previous run-times, especially for RMS workloads.

Cache-affine scheduling is used in user-level run-times, such as process-oriented programming [RSB12]. The message-passing-based structure of these programs allows analysis of communication patterns and subsequent classification of processes into cache-affine work units.

Lithe [PHA09] is a run-time framework to improve the scheduling of several concurrently executing parallel libraries within a large parallel application. It explicitly schedules threads of all libraries to avoid situations of over-commit or adversarial placement of competing threads onto the same core.

These user-level run-times work only from within one application and do not aim to consider the bigger problem of scheduling multiple applications as part of an operating system.

Poli-C [And12] is an exception. Poli-C presents language extensions and a run-time system to allow programmers to specify resource requirements within their program code and enforces these requirements via a central system daemon process among multiple such applications. This is similar to the approach presented in Section 3.5 of this dissertation. However, Poli-C embodies an even more dynamic approach to managing resources that requires information to be negotiated immediately when they are requested by program code, instead of using longer-term reservations to facilitate prompt response upon ad-hoc workload changes. The approach presented in this dissertation allows the OS more time to process resource allocations, but also requires applications to reason in advance about their possible requirements. Nevertheless, Poli-C's ideas can augment the approach presented in this dissertation and should be considered for future work.

Other commodity operating systems, such as Linux, also synchronize their local APIC timers to get synchronized timer interrupts on each core. Linux does this once at boot-up, for each core, by waiting for an HPET or PIT overflow and then resetting the core-local APIC timer. Both the method presented in this dissertation and the one used by Linux are adequate to synchronize local APIC timer interrupt generation. Linux, however, does not use a global clock to synchronize thread release time to conduct phase-locked scheduling.

The Mafia scheduler [Ble10] for the Xen virtual machine monitor uses a similar technique to phase-locked gang scheduling to reduce the overhead of dispatch synchronization among a number of cores. No hard real-time scheduling algorithm is used in the Mafia scheduler. Instead, regular round-robin scheduling is used for each core and dispatch of a gang is started by inserting each gang member once and at the same time into the schedule on each core. The author finds that parallel applications that frequently use barrier synchronization benefit from the gang scheduling technique, while other applications do not. This underlines that it is important to enable gang scheduling only for applications that benefit from the technique.

Work which has proposed new OS architectures, such as fos [WA09] and

Tessellation [LKB+09], has appealed to co-scheduling applications on partitions of a machine, but with little attention so far to the implementation problems to which such architectures lead. For example, Boyd-Wickizer *et al.* call for schedulers that focus on making efficient use of cache memory to avoid expensive RAM access [BWMK09]. Instead of moving shared data among caches, threads that want to access data are moved to the data. The authors show preliminary evidence that this methodology can improve performance on multicore processors.

3.8 Summary

The challenges brought about by multicore commodity computers, namely system diversity, the promise of executing multiple parallel applications concurrently, and interactive workloads have spawned the need for an operating system scheduling architecture that is able to support a dynamic mix of interactive and parallel applications executing concurrently on a multicore machine, all the while being scalable, agile, and able to respond within interactive attention spans.

Scheduling of parallel applications was part of the high-performance computing community historically, where workloads are static and typically receive a dedicated slice of the machine. Hence, this problem was not tackled in that community and the ideas developed there do not apply in a straightforward way to commodity multicore computing.

In order to tackle this mismatch, I have proposed and implemented a scheduler architecture within Barrelfish, a scalable and agile operating system. I find that time-multiplexing of cores is still necessary, despite the number of cores available. Also, I find that scheduling should happen at multiple time scales in order to break down the complex problem of scheduling parallel applications on non-uniform multicore hardware. Additionally, the operating system scheduler should be able to reason online about both the hardware and each application, so to be agile with a changing hardware architecture and to provide applications with the information necessary to achieve the same level of agility. Finally, the operating system and applications should communicate about resource requirements and allocations,

such that a changing mix of parallel, interactive applications that requires dynamic re-allocation of resources can be scheduled effectively.

By applying the design principles of the Multikernel, I derived the concepts of deterministic per-core scheduling, scheduler manifests, dispatcher groups, scheduler activations, and phase-locked scheduling. The application of these concepts brings about new challenges that so far were only exposed in the distributed systems community. These problems include timer synchronization and the problem of efficiently maintaining consistency of scheduler state among cores.

Finally, I have demonstrated that gang scheduling as a technique can improve the responsiveness of a hypothetical interactive parallel application by ensuring that all threads are dispatched at the same time and thus lower cost synchronization primitives can be used and threads do not need to be waited for when synchronizing. Gang scheduler implementations in the literature are developed with batch scheduling of high-performance compute applications in mind and support only coarse-grained time slice lengths, which impacts response time negatively in interactive scenarios that have much shorter timeslice lengths. I have shown how gang scheduling can be combined with phase-locked scheduling to derive phase-locked gang scheduling, which requires less synchronization of schedules among cores and thus better supports the short timescales required in interactive scenarios. I will evaluate these ideas in the following Chapter.

Chapter 4

Evaluation

In this chapter, I first present in **Section 4.1** the evaluation of the overhead and scalability of individual components of the Barrelfish scheduler architecture through various microbenchmarks, as well as application-level benchmarks showing the performance and agility of the entire scheduling subsystem within Barrelfish.

Then, in **Section 4.2**, I am evaluating phase-locked gang scheduling, by comparing its overhead both against a traditional user-level implementation of gang scheduling, as well as a customized, low-level implementation.

In **Section 4.3**, I discuss the results presented in this chapter and conclude.

4.1 Barrelfish Scheduler

I first investigate the Barrelfish scheduler subsystem overhead.

In **Section 4.1.1**, I evaluate the baseline cost and scalability of conducting an application phase change among a number of cores.

In **Section 4.1.2**, I evaluate the baseline performance and scalability of isolated, compute-bound applications and compare it to the same applications running on the Linux operating system to show that Barrelfish can provide comparable application performance and scalability with an existing commodity operating system.

In **Section 4.1.3**, I show that the holistic approach to multicore scheduling can indeed improve application performance by revisiting the example given in Section 3.1.2 and comparing to the same applications running on Barrelfish with an appropriate scheduling manifest.

In **Section 4.1.4**, I show the agility of the Barrelfish scheduler with several different memory architecture layouts, by comparing application performance and scalability with that provided by natively tuned operating systems across a number of different architectures.

4.1.1 Phase Change Cost and Scalability

I measure the average cost of a single application phase change, changing all scheduling parameters, over an increasing amount of cores. Phase changes are expected to take place during program execution, at the boundary of parallel sections with different scheduling requirements; we do not expect them to occur more frequently than on the order of hundreds of timeslices (several seconds). Resetting scheduling parameters more frequently would not result in noticeable performance gains: these phases would be too short to be perceivable by an interactive user of the system.

Phase changes are part of the performance-critical path of program execution and we should pay attention to their overhead. Figure 4.1 shows this overhead for all of the x86 evaluation platforms (the SCC port did not exist when this data was collected, but I expect it to have similar overhead). We observe that phase changes have low overhead compared to the frequency with which they are expected to occur. In the worst case, on 32 cores on the 8×4-core AMD machine, we observe a median overhead of 24,535 cycles, or 12 microseconds. With an increasing number of cores, this overhead is growing, as more cores need to be informed of the phase change. A multicast or broadcast communication mechanism could be employed to improve

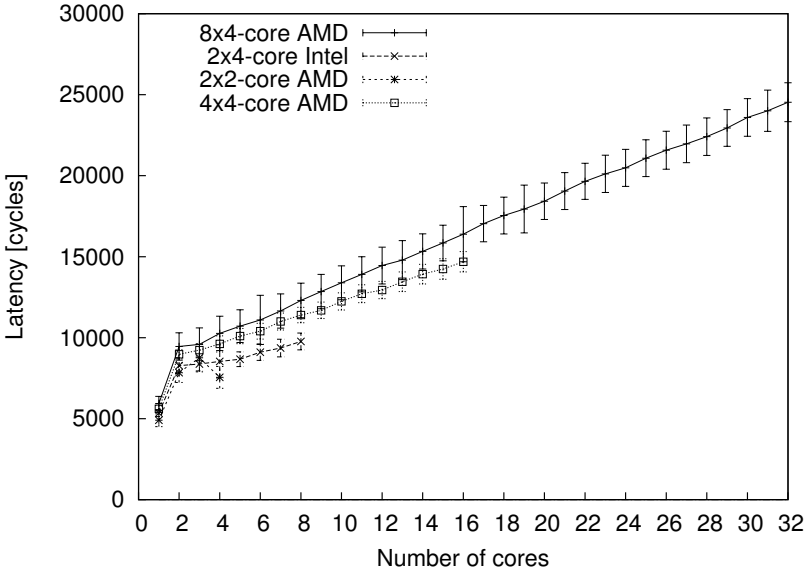


Figure 4.1: Cost of conducting a phase change over a varying number of cores. Median and standard deviation over 1,000 individual measurements are shown.

the scalability of this overhead, but is not necessary for the scale of cores examined in this dissertation.

Conducting a phase-change involves a synchronous message with the phase identifier from the dispatcher initiating the phase-change to the core-local monitor. The monitor retrieves the required scheduling parameters for its local core and which other dispatchers are involved in the phase change. It forwards the phase change message to the core-local monitors of these dispatchers. Once all messages are distributed, each monitor will transmit the new scheduling parameters to their respective CPU driver schedulers. Once this is done, responses are sent back to the initiating monitor, which will forward a single aggregated response to the initiating dispatcher once all responses are received.

The additional overhead involved in conducting phase changes over an increasing number of cores is not as large as the baseline overhead for con-

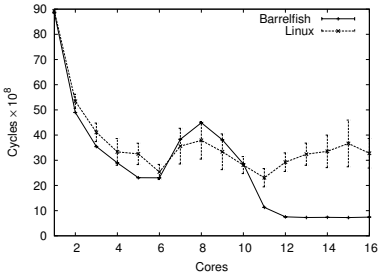
ducting a phase change on just a single core. This is due to planners executing phase changes in parallel, once the request has been submitted to all planners involved.

Phase changes bear resemblance to cell activation and suspension in the Tessellation operating system. The cost to activate a Tessellation cell on 15 cores of an Intel x86 machine is 8.26 microseconds and 17.59 microseconds to suspend it [LKB+09]. Barrelfish cannot be compared directly to the quite different Tessellation OS, but we note here that the phase change overhead agrees with the overhead of similar mechanisms in another multicore operating system.

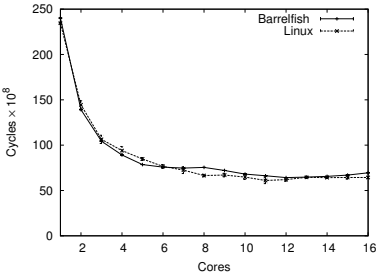
4.1.2 Single Application Performance

In this subsection, I use compute-bound workloads, in the form of computational kernels of the NAS OpenMP benchmark suite [JFY99] and the SPLASH-2 parallel application test suite [SPL], to exercise shared memory, threads, and scheduling. These benchmarks perform no IO and few virtual memory operations but we would expect them to be scalable. I introduce each application briefly here:

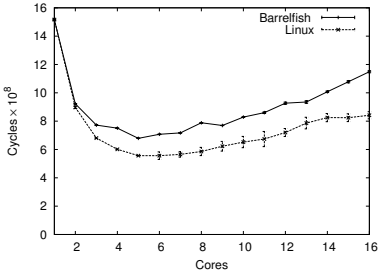
- CG uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries.
- FT contains the computational kernel of a 3-D fast Fourier transform (FFT)-based spectral method. FT performs three one-dimensional (1-D) FFTs, one for each dimension.
- IS is a large integer sort. This kernel performs a sorting operation that is important in "particle method" codes. It tests both integer computation speed and communication performance.
- BarnesHut is conducting n-body simulations in three dimensions using a hierarchical method according to Barnes and Hut. This is useful for galaxy and particle simulation. The algorithm uses an octree



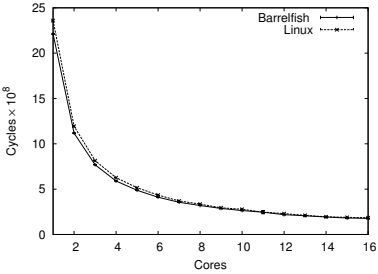
(a) OpenMP conjugate gradient (CG)



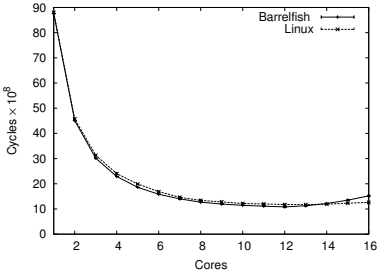
(b) OpenMP 3D fast Fourier transform (FT)



(c) OpenMP integer sort (IS)



(d) SPLASH-2 Barnes-Hut



(e) SPLASH-2 radiosity

Figure 4.2: Average execution time over 5 runs of compute-bound workloads on 4×4-core AMD (note different scales on y-axes). Error bars show standard deviation.

to represent space cells, with leaves containing information on each body. The communication patterns are dependent on the particle distribution and are quite unstructured.

- Radiosity is computing the equilibrium distribution of light in a computer rendered scene using the iterative hierarchical diffuse radiosity method. The structure of the computation and the access patterns to data structures are highly irregular. A task-stealing methodology is used to distribute work over threads.

I compare the baseline performance and scalability of these benchmarks across Barrelfish and Linux 2.6.26 to investigate whether a Multikernel can provide similar application performance and scalability to a state-of-the-art shared-memory operating system. I use GCC 4.3.3 as the compiler, with the GNU GOMP OpenMP runtime on Linux, and a custom OpenMP implementation on Barrelfish. For each benchmark, the time needed to conduct an entire computation over a number of time-steps is measured.

Figure 4.2 shows the results of these five benchmarks from the 4×4-core AMD machine. I plot the compute time in cycles on Barrelfish and Linux, averaged over five runs; error bars show standard deviation. These benchmarks do not scale particularly well on either OS, but demonstrate that despite its distributed structure, Barrelfish can still support large, shared-address space parallel code with little performance penalty. The observable differences are due to Barrelfish's user-space threads library vs. the Linux in-kernel implementation – for example, Linux implements barriers using a system call, whereas the library implementation exhibits different scaling properties under contention (in Figures 4.2a and 4.2c).

4.1.3 Multiple Parallel Applications

To show that the holistic approach to multicore scheduling can indeed improve application performance, I revisit the example given in Section 3.1.2. The experiment in that section was conducted on a Linux system. In order to show a valid comparison, we first need to investigate whether we observe

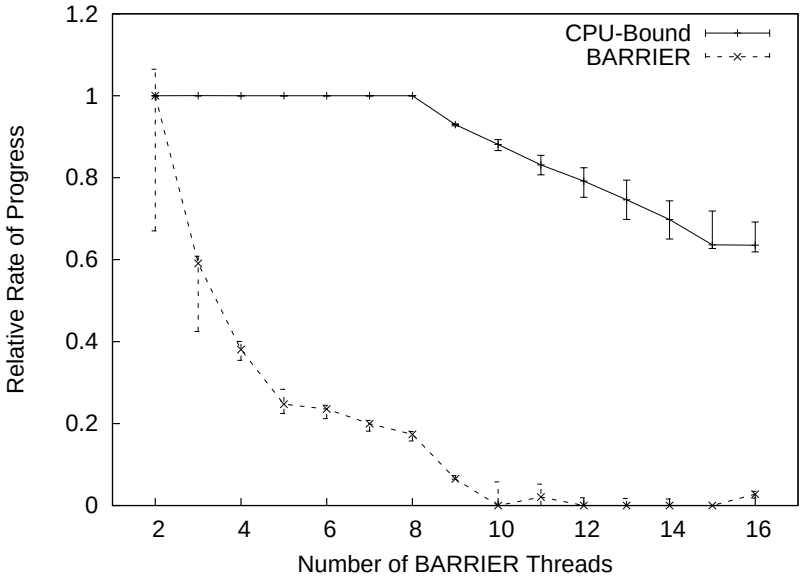


Figure 4.3: Relative progress of the two concurrent OpenMP applications from Section 3.1.2 on the 4×4-core AMD system, running on Barrelfish with best-effort scheduling. Again, mean, minimum, and maximum observed over 20 runs of 15 seconds each are shown.

different performance overheads and bottlenecks when this workload is executed on Barrelfish, where we have a different thread scheduling system and OpenMP run-time library.

Figure 4.3 shows the two applications from Section 3.1.2 running on Barrelfish, with only best-effort scheduling. This configuration is identical to the Linux system. We observe that the BARRIER application does not scale as well as in the Linux example. Through further investigation, I confirm this to be due to the barrier implementation used in Barrelfish's own OpenMP library, which uses a single atomic thread counter shared by all threads entering the barrier. Furthermore, threads wishing to leave a barrier spin on a shared counter to observe a change in the current barrier iteration. This causes contention on barrier entry and exit.

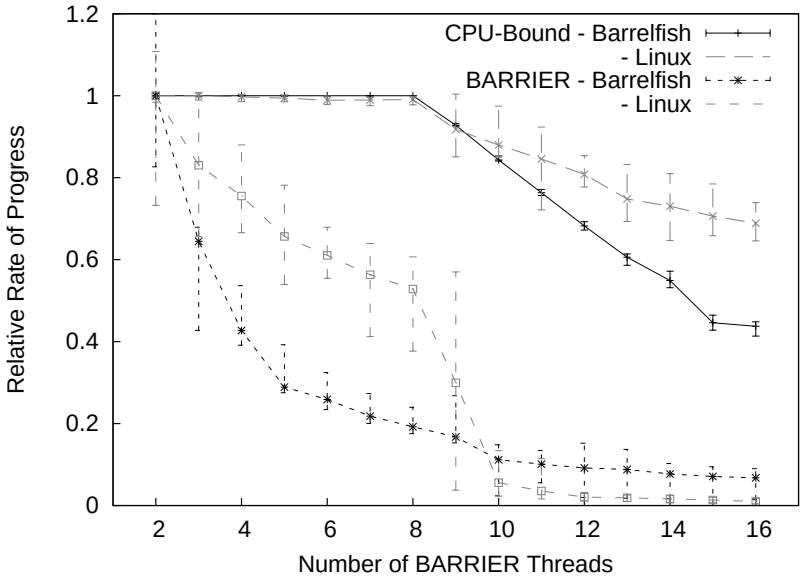


Figure 4.4: Relative progress of the same two OpenMP applications, this time using phase-locked gang scheduling. Also, we show the performance of the same applications running on Linux from Figure 3.1 again, in light-gray, for reference.

Better barrier implementations exist that alleviate some of these contention problems and consequently scale better [MCS91]. However, since we are still able to observe the same pitfall in the progress of the BARRIER application when both applications need to be time-multiplexed, the conditions of the experiment running on Barrelfish are not jeopardized and produce adequately similar results to the Linux version.

We now add a scheduling manifest to the BARRIER application that is submitted to the placement controller on application start. This manifest is in fact the one presented as an example in Section 3.5.5. It requests all cores within the same dispatcher group to be dispatched at exactly the same time, for exactly the same amount of time. Hence, Barrelfish will resort to gang scheduling this application using the phase-locked gang scheduling technique. We run the experiment again.

Figure 4.4 shows the two applications again, this time with the scheduling manifest submitted by the BARRIER application. We observe that the BARRIER application continues its progress, even when applications are time-multiplexed. There is also less noise in the measured progress of the CPU-bound application. Both are due to the gang scheduling technique now enabled by Barrelfish, which ensures that all threads of the BARRIER application are dispatched at the same time on all cores. This technique has the side-effect that other applications tend to self-synchronize on the boundary of timeslices, which is what we observe with the CPU-bound application in this figure. Hence the smaller noise in measured experimental results.

4.1.4 Agility

In order to show that the Barrelfish scheduler is agile with several different memory architecture layouts and can facilitate applications in their agility with the same, I conduct an experiment running a scientific compute application on the Single-Chip Cloud Computer and the 4×4-core AMD x86 platform and compare the performance to the same application tuned manually to the platforms' respective native operating systems. These two platforms have very different memory architectures. However, Linux is the native operating system in both cases.

The scientific compute application in this experiment solves a synthetic system of nonlinear partial differential equations using the lower-upper symmetric Gauss-Seidel method. It is taken from the NAS parallel benchmark suite, where it is called LU. I choose problem size A to conduct this experiment, which is the smallest size that is suggested to be used for benchmarking Supercomputers.

The software environment presently available on the SCC uses a separate instance of the Linux kernel on each core. Above this runs RCCE [MvdW10], a library for light-weight, efficient communication that has been co-designed with the SCC as a research vehicle for message-passing API design on non-cache-coherent manycore chips, and as such is highly optimized for this platform. RCCE runs in user-mode with raw access to the MPBs, providing basic point-to-point message passing functionality as well as a set of higher-level primitives, such as barriers and a reduce operation, akin to those found

in MPI [Mes09]. As part of the RCCE distribution ships a version of the LU benchmark that uses this library for message passing. I use this version as the native contender for the comparison on the SCC architecture.

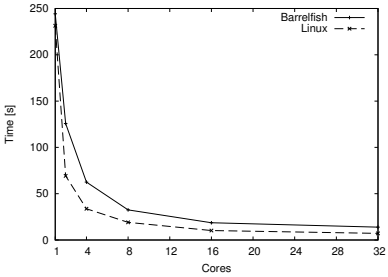
On the 4×4-core AMD machine, I compare against the original MPI implementation that ships with the benchmark suite. I use the OpenMPI [GWS05] message-passing suite to implement the message-passing primitives. This version is highly tuned, even for machine-local message passing. Another version of the LU benchmark exists within the NAS benchmark suite that is using shared memory and the OpenMP parallel programming framework. I compared the performance of the OpenMP version to that of the OpenMPI version and found the performance of the OpenMPI version to be better. This is understandable: the benchmarks were originally conceived as message-passing benchmarks and have subsequently been ported to use shared memory. Hence, I choose the message passing OpenMPI version for comparison.

On Barrelfish, I run a version using message-passing, which I have ported to Barrelfish's inter-domain communication API from the version that shipped with the RCCE message passing library.

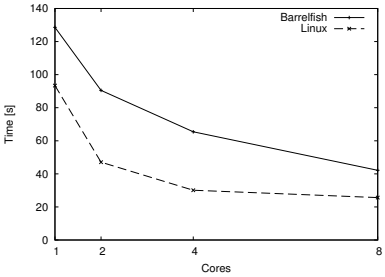
The Barrelfish, RCCE, and MPI implementations all use the same code base, derived from the NAS benchmarks version 3.3. However, the MPI implementation is written in the Fortran language, while the RCCE version has been manually ported to the C language. Nevertheless, both versions show comparable performance when run using just a single process. For example, when run on a single core on the same Linux OS on the 4×4-core AMD machine, I observe that the RCCE version is taking 119.75 seconds to complete, while the MPI version takes 90.01 seconds, a difference of a factor of 1.3.

Figure 4.5 shows the completion time of the LU benchmark running on Barrelfish vs. the respective natively optimized version. We can see that performance and scalability are similar in both cases. On the x86-based systems, we see a higher baseline overhead, which can be attributed to the different versions of the LU benchmark that were used in this comparison, as explained earlier.

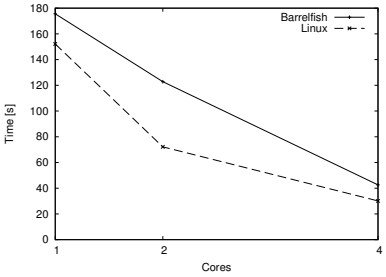
I note that the message-passing-based LU benchmark is designed to run only



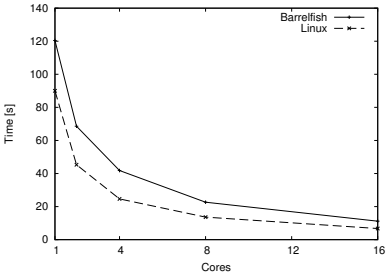
(a) 8x4 AMD machine



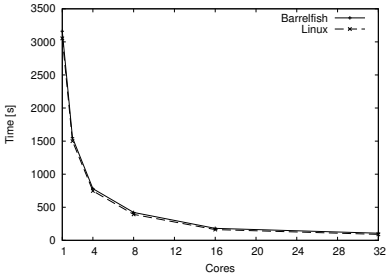
(b) 2x4-core Intel machine



(c) 2x2-core AMD machine



(d) 4x4-core AMD machine



(e) Single-Chip Cloud Computer

Figure 4.5: Completion time of the LU benchmark, when run on Barrelfish vs. run on the native Linux OS on several hardware architectures.

with a number of processing nodes at a power of two, hence I cannot use the maximum number of cores available in the SCC machine, which is not a power of two.

I conclude that the Barrelfish operating system indeed is agile with a changing underlying hardware architecture. It successfully supports performance and scalability comparable to natively tuned message passing and operating system implementations, without requiring changes to the applications and only small changes to the operating system (in this case, the implementation of an interconnect and notification driver). In addition, in the case of the SCC, Barrelfish provides multiplexing of the message passing hardware facilities to multiple concurrently executing applications, a feature not offered by the native RCCE message passing library.

4.2 Phase-locked Gang Scheduling

Now, we turn our attention towards the overhead of phase-locked gang scheduling.

In this section, I first compare the overhead of phase-locked gang scheduling to a traditional gang scheduling solution that is implemented in user-space on the Intel Single-Chip Cloud Computer.

In a second experiment, I am comparing the overhead of phase-locked gang scheduling to an optimized, centralized gang scheduling technique using inter-processor interrupts that is implemented in the Barrelfish CPU driver on the x86 platform.

4.2.1 Scheduling Overhead vs. High-level Solutions

Traditionally, gang scheduling has been realized using user-space resource management software, like the SLURM cluster management suite [YJG03]. In the case of the Single-Chip Cloud Computer, where the programming model is identical to clustered systems, SLURM is indeed the default system-wide scheduler.

The standard software distribution for this computer is designed such that an unmodified Linux operating system is executing in an isolated fashion on every core. In fact, the operating system instances are constrained via hardware techniques, such as memory and IO virtualization, to their own private memory regions and IO interfaces and know nothing about the other OS instances running at the same time on the machine. Communication with other cores is realized via a special device driver that abstracts the memory interconnect as an Ethernet network, accessible via a device by the OS instance. Higher-level application software, such as the SLURM cluster management suite, is expected to manage and coordinate system resources across the entire chip.

With a shared-memory operating system, such as Linux, there is no other way to execute on this computer, as the architecture is not cache-coherent. With the Barrelfish operating system, however, we can run a single-system image across the entire chip and implement resource management, such as processor scheduling, as an operating system service at a lower-level. This allows us to integrate more tightly with other OS resource management and control low-level hardware features, such as synchronizing each core's APIC timer, allowing for a lower total scheduling overhead. As a result we can achieve scheduling at shorter timeslices than is possible at user-level. I am going to show this in the following experiment.

In this experiment, I run the LU NAS benchmark (described in Section 4.1.4) on 16 cores of the SCC alongside CPU stressor programs, running one each on the first 4 cores. The CPU stressor programs spin forever in a tight loop and thus use up their entire time slice. The LU benchmark runs as part of a service that accepts client requests to carry out the LU benchmark computation and return the result. This scenario is supposed to model a parallel interactive application on a system consolidated with other services under CPU load. The decision to use 16 cores for the LU benchmark and 4 cores for the stressors was arbitrary. I tried other configurations with similar results.

I execute multiple runs of the experiment. For each run, I periodically request to compute the LU benchmark result of the class S problem size at an interval of at least 2 seconds. Class S is small enough to be computed in less than 1 second on 16 cores of the SCC, even when run alongside the CPU stressors. It might still be delayed, however, if the overhead of the

scheduler is too high. This means that if the computation does indeed finish before the interval of 2 seconds, I wait until the end of the interval and do nothing. If the computation finishes only after the 2 second interval, I immediately start the next computation. Within a run, I repeat the LU compute time measurement 10 times and take the average, before ending the run.

This setup simulates a parallel computation service, which might be embedded as part of a server-sided script on a webserver, with an incoming request load of one request every 2 seconds. Following the webserver example, this would equate to a person requesting a fresh version of the website once every 2 seconds. Obviously, a faster response of the final result is better than a slower one, as it is a determining factor of the page load time.

Between runs, I shorten the scheduling timeslice length. I try five different lengths between 5 seconds, the default timeslice length for the SLURM cluster management suite, and 60 milliseconds, close to the average timeslice length on Linux for interactive processes. With long timeslices, the likelihood that we issue a request when the LU benchmark is not running is high and the time to finish the computation will be longer. As we move to shorter timeslice lengths, the time to finish computation shortens. However, at the same time, the overhead of the scheduling implementation grows as it is invoked more frequently.

Figure 4.6 shows that phase-locked gang scheduling can provide better response time than the SLURM cluster management software. This is due to lower overhead of the scheduler implementation. The SLURM software is relying on TCP/IP messages sent between user-mode processes on distinct cores, which then modify kernel-level scheduling parameters to synchronize schedules. SLURM cannot leverage synchronized processor clocks to realize phase-locked gang scheduling.

4.2.2 Scheduling Overhead vs. Low-level Solutions

To gauge the overhead of phase-locking schedules when compared against another low-level implementation, I developed a centralized gang scheduling solution that is implemented at the lowest level in the Barrelfish CPU driver. The solution does not rely on synchronized processor clocks, but

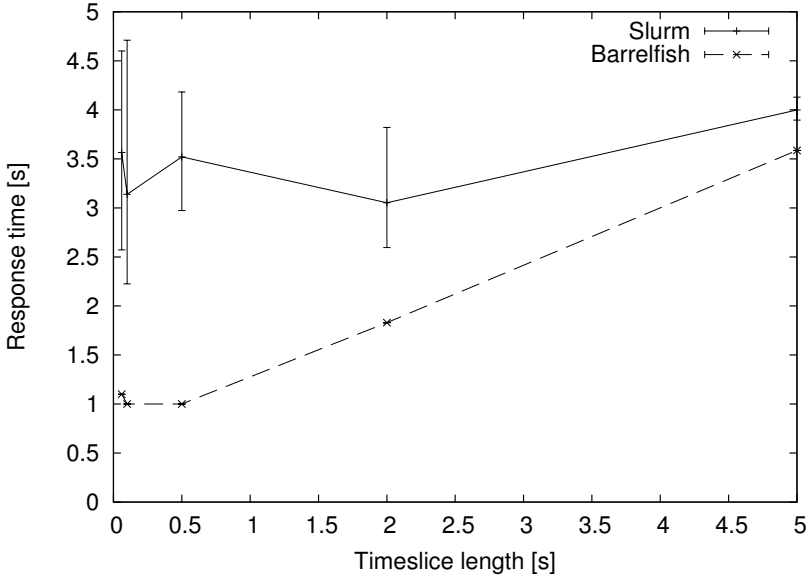


Figure 4.6: Overhead of phase-locked gang scheduling in Barrelfish vs. gang scheduling using the SLURM cluster management software on the SCC. The graph shows average completion time of the NAS LU benchmark, executing on 16 cores, alongside 4 CPU stressor programs. Error bars show minimum and maximum measured over 3 runs of the experiment.

instead designates one core to be the schedule coordinator. I define two distinct inter-processor interrupts: One to activate scheduling of a particular gang and one to deactivate gang scheduling. The coordinator core broadcasts an inter-processor interrupt to context switch a gang to all other cores involved in scheduling a gang upon expiry of each of its own timeslices. The other cores disregard their own local clocks when scheduling a gang and react solely to these broadcast interrupts to undertake a context switch. When not scheduling a gang, the other cores use their local clocks and schedules.

In this experiment, I conduct a limit study where I vary the frequency of context switches to evaluate at what rate phase-locked scheduling provides a benefit to applications over a centralized solution. To measure application progress, I run the BARRIER and CPU-bound applications on the 4×4-core

AMD system and count the accomplished loop iterations of the BARRIER application each second over the course of 10 seconds. The CPU-bound application serves as background load. I configure the scheduler to switch between the gang scheduled BARRIER application and the background load on the boundary of each timeslice. Thus, there is one inter-processor interrupt transmitted at the end of each timeslice.

I repeat the experiment, varying the timeslice length of the system between 80 milliseconds and 20 microseconds, the lower limit at which the system operates reliably. To put the lower limit into perspective, I measure the context switch duration on the 4×4-core AMD system to be 4 microseconds and note that a time slice length of 20 microseconds is only 5 times this duration, resulting in an overhead of one fifth just for context switching on each time slice, which does not leave a lot of processor time for applications.

The results are shown in Figure 4.7, once for time slice lengths on the order of microseconds and once on the order of milliseconds. With time slice lengths in the range of milliseconds, I observe no difference in application progress, regardless of scheduling mechanism, showing that the two mechanisms perform equally well within this range.

Within the microsecond range, between 1 millisecond and 500 microseconds, there is still no difference in application performance among the two mechanisms. With both schemes, we can see a slight drop in performance as I lower the time slice length. This is due to the increased context switching overhead, which I plot on a second axis in the microsecond graph.

We then observe a departure of application performance, starting at 200 microseconds time slice length. Phase-locked scheduling starts to perform better. This is due to the overhead of sending the IPIs on core 0, which does not exist with phase-locked scheduling. I measured this overhead to be on the order of 1000s of cycles, or several microseconds. At 20 microseconds, we observe 36% better performance when using phase-locked scheduling. However, context switching overhead rises exponentially as we lower time slice length. It rises to 20% of the total processor time available within a time slice length of 20us, which explains the slowdown. Thus, this configuration is unrealistic for a commodity operating system like Barrelfish and does not provide any benefit for applications.

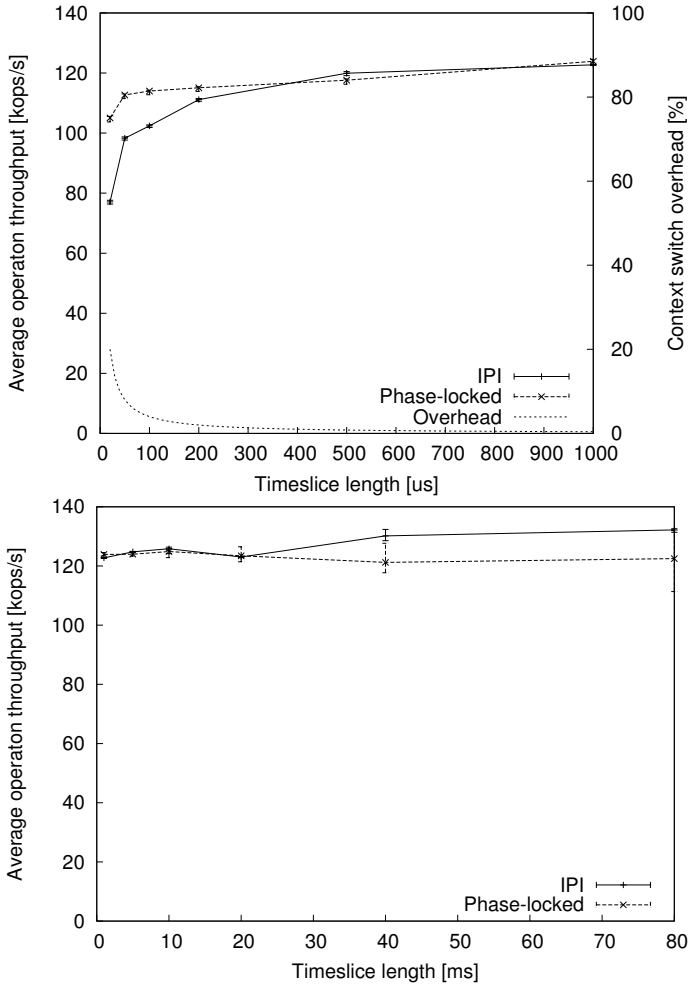


Figure 4.7: Phase-locked scheduling vs. centralized scheduling using inter-processor interrupts at microsecond and millisecond granularity. For each scheduling approach the graphs show average number of accomplished loop iterations per second over a measured run-time of 10 seconds. Error bars show minimum and maximum measured. The microsecond graph also shows context switch overhead.

We conclude that phase-locked gang scheduling does not provide any benefit over centralized gang scheduling when implemented at a low-level within realistic time slice lengths for a commodity operating system that supports interactive applications. Context switching of applications on the order of milliseconds is well within limits of providing responsiveness at interactive attention spans.

4.2.3 Discussion

While phase-locked gang scheduling can provide improved application performance over high-level gang scheduling solutions that are typically employed in cluster and cluster-on-chip scenarios, such as the SCC, it does not show enough promise versus low-level centralized solutions that can be implemented on traditional architectures, like x86.

I note that there exists a small window of opportunity of improved application performance when using phase-locked scheduling for time slice lengths up to 200 microseconds, but that this is outside of the reasonable range for general-purpose commodity operating systems.

Still, the phase-locked approach does consume less hardware resources, by eliminating the need for a coordinator core and inter-processor interrupts. This, in turn, can improve performance on systems, where there is no dedicated hardware bus for inter-processor signaling. For example, on the SCC, all messages, including inter-processor interrupts, are delivered using the mesh interconnect. On the event of congestion of this interconnect, gang scheduling messages sent out centrally would become delayed by other messages on the mesh. This would ultimately result in jittery start and end times of gangs and with that in reduced performance of applications with fine-grain synchronization. Message interconnect congestion is not an issue for phase-locked gang scheduling, where all coordination happens locally on each core.

4.3 Summary

In this chapter, I have shown that the concepts presented in Chapter 3 allowed me to develop a scheduler that is scalable and allows applications to achieve baseline performance and scalability comparable to natively tuned operating systems on a number of different multicore platforms, without requiring vast modifications of the operating system, contributing to the first two goals of this dissertation of scalability and agility. Only the message passing facilities had to be modified.

I hoped that the concept of phase-locked scheduling would improve gang scheduling to fine-grained time slice lengths, by applying the Multikernel principles of core-local state and reduced inter-core communication. I succeeded in developing a gang scheduler that is able to deliver improved parallel application performance in a mix of parallel applications over schedulers used in traditional commodity operating systems, such as Linux, that are oblivious to the synchronization requirements of parallel applications. The scheduler is also built to react to changes in the workload, which might occur due to user input. This is an essential contribution to the goal of supporting interactive response time to the user when running a mix of parallel and interactive applications.

Finally, phase-locked gang scheduling performs better than high-level, cluster-based gang scheduling solutions, such as SLURM. However, a subsequent comparison against a naive low-level implementation showed that this is not due to the technique of phase-locking, but due to the low-level implementation, which can utilize hardware features better than a high-level implementation.

Nevertheless, the technique of phase-locked schedulers consumes less hardware resources than a naive, centralized implementation, by distributing schedule coordination across cores. This can potentially benefit hardware architectures that cannot dedicate resources for inter-core communication techniques, such as inter-processor interrupts. For example, when the available inter-core interconnects become congested with other messages. This is a contribution to the goal of agility, as it does not favor one memory subsystem layout over another.

Chapter 5

Conclusion

With the advent of multicore architectures, commodity computing is becoming increasingly parallel and interactive application software is starting to leverage this parallelism. At the same time, commodity hardware evolution is continuing at its usual pace. Motivated by these developments, this dissertation has presented the design and implementation of operating system mechanisms to support the execution of a dynamic mix of interactive and parallel applications on commodity multicore computers. Scalability, agility, and interactive response times when scheduling a mix of parallel and interactive applications have been the three main goals of the presented design.

In order to achieve these goals, this dissertation has made the following contributions:

- In **Chapter 2**, I have introduced the Multikernel, an operating system model and component architecture that makes all inter-core communication explicit, makes the operating system structure hardware-neutral, and views all operating system state as replicated instead of shared. This model promises scalability and agility for operating system implementations based on it.

I have then reported about a concrete implementation of the Multikernel, Barrelfish, and its *inter-process communication* facilities. I have shown adequate baseline performance and scalability of the communication facilities via a series of messaging microbenchmarks. Barrelfish's communication facilities are also able to provide superior performance for the IP loopback service and a web server, compared to equivalents on a mature commodity operating system.

Finally, I have reported on the simplicity of adding new specialized communication facilities for new types of hardware.

- In **Chapter 3**, I have presented the design and implementation of the *scheduler subsystem* within the Barrelfish Multikernel. The five concepts of deterministic per-core scheduling, scheduler manifests, dispatcher groups, scheduler activations, and phase-locked scheduling help to achieve the goals of scalability and interactive response times.

  Via a benchmark I have motivated why gang scheduling can deliver interactive responsiveness for future parallel interactive applications and presented *phase-locked gang scheduling* as a technique to reduce the overhead of classical gang scheduling implementations found in high-performance computing. I have described the application of the technique within the Barrelfish operating system.

- In **Chapter 4**, I have evaluated the concrete implementation of scheduling within the Barrelfish Multikernel operating system and have shown that Barrelfish is able to efficiently multiplex hardware in time and space, while workloads executing on Barrelfish can perform and scale equivalently to natively tuned operating system implementations. I have demonstrated this across a variety of different machines, requiring minimal changes to Barrelfish's implementation, and thus I have also shown the agility of the approach.

  Via another benchmark, I have shown that, with phase-locked gang scheduling, Barrelfish can avoiding mis-scheduling multiple parallel applications. A surprising result was that, while phase-locked gang scheduling is more efficient than high-level solutions typically employed in cluster-on-chip scenarios, like the SCC, the performance gains versus low-level implementation on the x86 architecture are diminishing.

Together, these contributions constructively prove the thesis stated in Chapter 1: it is possible to design and implement an operating system that achieves the goals of scalability, agility and supporting interactive response times in a commodity multicore computing scenario, by providing inter-process communication and scheduling components that

- multiplex hardware efficiently in time and space,
- react quickly to ad-hoc workload changes,
- reason online about the underlying hardware architecture,
- reason online about each application, and
- allow resource negotiation between applications and the operating system.

Indeed, Barrelfish performs equivalently to, and sometimes better than, natively tuned, mature operating systems.

5.1 Future Work

The Barrelfish operating system implementation presents a significant amount of work. However, it is by no means a finished, mature operating system. Several avenues for future work seem promising and I survey them within this section.

5.1.1 Heterogeneous Hardware Architectures

Our current implementation is based on the Intel and AMD multiprocessors, as well as the SCC research processor. These multiprocessor architectures are homogeneous—all processors are identical. Barrelfish was not evaluated in a truly heterogeneous environment.

Experiments were conducted with architectures comprised of an x86-based host PC and an SCC peripheral device, all executing the same Barrelfish

image [Men11]. However, this just presents a first step and only initial performance results exist.

A port is in progress to the Netronome network accelerator architecture [Sch11], as well as the Intel Many Integrated Core (MIC) architecture, which will allow us to run a single Barrelfish image across a host PC and programmable accelerator cards.

This will also allow us to experiment with specialized data structures and code for different processors within the same operating system. It also opens the door to myriad of scheduling challenges, such as optimizing the utility of accelerators by quantifying which application can make the most use of them and placing computation strategically on accelerators, such that a tradeoff of maximum application speed-up and minimum data movement between system and accelerator memory is maintained.

5.1.2 Distributed Execution Engines

Structuring the operating system as a distributed system more closely matches the structure of some increasingly popular programming models for data center applications, such as MapReduce[DG04], Dryad[IBY+07], and CIEL [MSS+11], where applications are written for aggregates of machines. A distributed system inside the machine may help to reduce the "impedance mismatch" caused by the network interface – the same programming framework could then run as efficiently inside one machine as between many.

In fact, there are efforts to run the CIEL distributed execution engine in this fashion [SMH11]. When coupled with an operating system, such as Barrelfish, one could leverage efficient message passing that is agile with a changing hardware architecture. Challenges that arise in this scenario are again related to resource management when multiple CIEL applications are executing, especially when multiple machines are involved. It is also worthwhile to investigate how resource management changes when a data-parallel program is executing within one multicore machine, as opposed to executing on a cluster.

5.1.3 Systems Analysis

Systems based on explicit communication are amenable to human and automatic analysis. A substantial theoretical foundation exists for reasoning about the high-level structure and performance of a system with explicit communication between concurrent tasks, ranging from process calculi such as communicating sequential processes [Hoa] and the π-calculus [Mil99], to the use of queuing theory to analyze the performance of complex networks [Jai91].

Leveraging this foundation to reason about Multikernel-based systems would allow us to investigate the soundness of the systems built. We could ask questions like: Can we be certain that, by design, the system will be deadlock free? How resilient will it be to faults? And, will the system be secure? With ever more cores being integrated into multicore machines, these questions become ever more important.

5.1.4 Programming Language Integration

Programming language directives can help specify the programmer's intent and communicate application requirements to the operating system. So far, I have only focused on programming language run-times. Going one step further, programming language directives would allow programmers to specify directly the intended resource requirements for program sections and allow for an even better integration.

The Poli-C [And12] language extensions and run-time system are a promising first step in this direction. If integrated with the Barrelfish scheduling framework, more fine-grained resource management could be made possible by leveraging the low-level Barrelfish scheduler implementation. Also, all resources could directly be managed by and negotiated with the operating system scheduler, obviating the need for an OS reserve, which is currently used in Poli-C for all applications not under its control. Finally, dynamic negotiation could be better integrated with the language, by allowing programmers to specify what to do when only a subset of the requested resources is available.

5.2 Summary

As commodity multicore computer architectures become more and more widespread, it is only a matter of time until application software will start to leverage the offered parallelism to speed up computation. Operating systems are an integral part of the system stack and have to support this development to secure performance benefits. This dissertation has presented the design and implementation of operating system mechanisms to support the execution of a dynamic mix of interactive and parallel applications on commodity multicore computers, a scenario that might become commonplace in the near future.

Bibliography

[ABC+06] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[ABLL91] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 95–109, October 1991.

[AD01] Andrea Carol Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[ADK+07] Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Marc Auslander, Michal Ostrowski, Bryan Rosenburg, Amos Waterland, Robert W. Wisniewski, Jimi Xenidis, Michael Stumm, and Livio Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3), 2007.

[And] J. Andrews. Linux: The completely fair scheduler. `http://kerneltrap.org/node/8059`.

[And12] Zachary Anderson. Efficiently combining parallel software using fine-grained, language-level, hierarchical resource management policies. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2012.

[App09] Apple. *Grand Central Dispatch Technology Brief*, 2009. `http://opensource.mlba-team.de/xdispatch/GrandCentral_TB_brief_20090608.pdf`.

[AW07] Krzysztof R. Apt and Marg G. Wallace. *Constraint Logic Programming using ECL^{i}PS^e*. Cambridge University Press, 2007.

[BALL90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[BALL91] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, 1991.

[Bau10] Andrew Baumann. Inter-dispatcher communication in barrelfish. Technical report, ETH Zurich, 2010. Barrelfish technical note 011.

[BB95] James M. Barton and Nawaf Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *Job Scheduling Strategies for Parallel Processing*, volume 949, pages 45–69. Springer-Verlag, 1995. Lecture Notes in Computer Science.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[BBF07] Sven Bachthaler, Fernando Belli, and Alexandra Fedorova. Desktop workload characterization for CMP/SMT and implications for operating system design. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2007.

[BBLB03] Scott A. Brandt, Scott A. Banachowski, Caixue Lin, and Timothy Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, 2003.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, October 2003.

[BDGR97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, 1997.

[BGJ+92] David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohrman. Microkernel operating systems architecture and Mach. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 11–30, April 1992.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 123–138, 1987.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008.

[Ble10] Alexandru S. D. Bleotu. Multicore scheduling on xen. Master's thesis, University of Cambridge, June 2010.

[Bli96] Steve Blightman. Auspex Architecture – FMP Past & Present. Internal document, Auspex Systems Inc., September 1996. `http://www.bitsavers.org/pdf/auspex/eng-doc/848_Auspex_Architecture_FMP_Sep96.pdf`.

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, pages 746–749, 2007.

[BPR+11] Andrew Baumann, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. Barrelfish specification. Technical report, ETH Zurich, July 2011.

[BPS+09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, May 2009.

[Bur06] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

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

[BWCM+10] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on*

Operating Systems Design and Implementation, pages 1–16, 2010.

[BWMK09] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Operating Systems*, May 2009.

[Car10] Tracy Carver. "magny-cours" and direct connect architecture 2.0, March 2010. `http://developer.amd.com/documentation/articles/pages/magny-cours-direct-connect-architecture-2.0.aspx`.

[CB08] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *ACM Queue*, 6(5):16–25, September 2008.

[CCSW10] Yan Cui, Yu Chen, Yuanchun Shi, and Qingbo Wu. Scalability comparison of commodity operating systems on multicores. In *IEEE International Symposium on Performance Analysis of Systems Software*, pages 117–118, March 2010.

[CDL+93] Eliseu M. Chaves, Jr., Prakash Ch. Das, Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Kernel–Kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, 1993.

[CH07] Pat Conway and Bill Hughes. The AMD Opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[CJ09] Jian Chen and Lizy Kurian John. Efficient program scheduling for heterogeneous multi-core processors. In *Proceedings of the 46th ACM Annual Design Automation Conference*, pages 927–930, 2009.

[CKP+93] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[CML01] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Improving gang scheduling through job performance analysis and malleability. In *Proceedings of the 2001 ACM/IEEE Supercomputing Conference*, pages 303–311, 2001.

[CRD+95] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[DAC96] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM SIGMETRICS*, pages 25–36, 1996.

[Dag09] Pierre-Evariste Dagand. Language support for reliable operating systems. Master's thesis, ENS Cachan-Bretagne – University of Rennes, June 2009.

[DBR09] Pierre-Evariste Dagand, Andrew Baumann, and Timothy Roscoe. Filet-o-Fish: practical and dependable domain-specific languages for OS development. In *Proceedings of the 5th Workshop on Programming Languages and Operating Systems*, October 2009.

[DEE06] Philip Derrin, Dhammika Elkaduwe, and Kevin Elphinstone. *seL4 Reference Manual*. NICTA, 2006. `http://www.ertos.nicta.com.au/research/sel4/sel4-refman.pdf`.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 137–150, 2004.

[DM96] Danny Dolev and Dalia Malki. The transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[DS09] David Dice and Nir Shavit. TLRW: return of the read-write lock. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, February 2009.

[Dub05] Pradeep Dubey. Recognition, mining and synthesis moves computers to the era of tera. In *Technology@Intel Magazine*, pages 1–10. Intel Corporation, February 2005.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, December 1995.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, April 2006.

[FE06] Eitan Frachtenberg and Yoav Etsion. Hardware parallelism: Are operating systems ready? In *Proceedings of the 2nd Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 7–15, June 2006.

[FFFP03] Eitan Frachtenberg, Dror G. Feitelson, Juan Fernández, and Fabrizio Petrini. Parallel job scheduling under dynamic workloads. In *Proceedings of the 9th International Workshop on Job Scheduling Strategies for Parallel Processing*, pages 208–227, 2003.

[FFPF05] Eitan Frachtenberg, Dror G. Feitelson, Fabrizio Petrini, and Juan Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel Distributed Systems*, 16(11):1066–1077, 2005.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[FPF+02] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, and Salvador Coll. STORM: Lightning-fast resource

management. In *Proceedings of the 2002 ACM/IEEE Supercomputing Conference*, 2002.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992.

[FSS06] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. A non-work-conserving operating system scheduler for SMT processors. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, June 2006.

[Ger12] Simon Gerber. Virtual memory in a multikernel. Master's thesis, ETH Zurich, May 2012.

[GKAS99] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 87–100, February 1999.

[GMTW08] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, 2008.

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 43–52, 2008.

[Gre11] Alexander Grest. A routing and forwarding subsystem for a multicore operating system. Bachelor's thesis, August 2011.

[GSAR12] Jana Giceva, Adrian Schüpbach, Gustavo Alonso, and Timothy Roscoe. Towards database / operating system co-design. In *Proceedings of the 2nd Workshop on Systems for Future Multi-core Architectures*, April 2012.

[GTHR99] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 154–169, 1999.

[GWS05] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings of the 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *SIGOPS Operating Systems Review*, 40(3):80–89, July 2006.

[HBK06] Jim Held, Jerry Bautista, and Sean Koehl. From a few cores to many: A tera-scale computing research overview. White paper, Intel, September 2006. `ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`.

[HDH+10] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *International Solid-State Circuits Conference*, pages 108–109, February 2010.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, July 2008.

[Hoa] C. A. R. Hoare. Communicating sequential processes. `http://www.usingcsp.com/`.

[HR83] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4), December 1983.

[HSS05] Bijun He, William N. Scherer III, and Michael L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *Proceedings of the 12th International Conference on High Performance Computing*, pages 7–18, 2005.

[Hyp04] The HyperTransport Consortium. *HyperTransport I/O Technology Overview*, June 2004. `http://www.hypertransport.org/docs/wp/HT_Overview.pdf`.

[IBM02] IBM K42 Team. *K42 Overview*, August 2002. `http://www.research.ibm.com/K42/`.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007.

[IKKM07] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 186–197, 2007.

[Int08] Intel Corporation. QuickPath architecture white paper, 2008. `http://www.intel.com/technology/quickpath/whitepaper.pdf`.

[Int09] Intel Corporation. Single-chip cloud computer. `http://techresearch.intel.com/articles/Tera-Scale/1826.htm`, December 2009.

[Int10] Intel Corporation. *Intel Workstation Board s5000XVN Technical Product Specification Revision 1.5*, August 2010. Intel order number D66403-006.

[Isa10] Rebecca Isaacs. Tracing and visualization. Technical Report Barrelfish Technical Note 8, ETH Zurich, 2010.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[Jet98] Morris A. Jette. Expanding symmetric multiprocessor capability through gang scheduling. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 199–216, 1998.

[JFY99] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Advanced Supercomputing Division, Moffett Field, CA, USA, October 1999.

[JLM+09] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanović, and Rastislav Bodik. Parallelizing the web browser. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[JSAM10] F. Ryan Johnson, Radu Stoica, Anastasia Ailamaki, and Todd C. Mowry. Decoupling contention management from scheduling. In *Proceedings of the 15th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2010.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[Kem09] Bettina Kemme. One-copy-serializability. In *Encyclopedia of Database Systems*, pages 1947–1948. Springer US, 2009.

[KLMO91] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55, 1991.

[KPP06] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[KWS97] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15:3–40, February 1997.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. `http://www.cs.washington.edu/homes/levy/capabook/`.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, December 1995.

[lig] lighttpd webserver. `http://www.lighttpd.net/`.

[lina] Introduction of linux 2.6.23 features. `http://kernelnewbies.org/Linux_2_6_23`.

[linb] Linux kernel 2.6.17 changelog. `http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.7`.

[LKB+09] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[LN78] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *Proceedings of the 2nd International Symposium on Operating Systems, IRIA*, 1978. Reprinted in *Operating Systems Review*, 13(2), 1979.

[LSE] Linux scalability effort. http://lse.sourceforge.net/.

[LV03] Peter Lyman and Hal R. Varian. How much information, 2003. http://www.sims.berkeley.edu/how-much-info-2003.

[lwI] lwIP. http://savannah.nongnu.org/projects/lwip/.

[Mau00] Jim Mauro. *The Solaris Process Model: Managing Thread Execution and Wait Times in the System Clock Handler*, 2000. http://developers.sun.com/solaris/articles/THREADexec.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

[Men11] Dominik Menzi. Support for heterogeneous cores for barrelfish. Master's thesis, ETH Zurich, July 2011.

[Mes09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, September 2009. Version 2.2.

[Mic] Microsoft. Receive-side Scaling enhancements in Windows Server 2008. http://www.microsoft.com/whdc/device/network/ndis_rss.mspx.

[Mic10] Microsoft. *C++ Concurrency Runtime*, 2010. http://msdn.microsoft.com/en-us/library/dd504870.aspx.

[Mil91] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Communications*, 39(10):1482 –1493, October 1991.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[MSS+11] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: a universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*, 2011.

[MT98] D. Mosberger and J. Tin. httperf: A tool for measuring web server performance. *Performance Evaluation Review*, 26(3):31–37, December 1998.

[MvdW10] Tim Mattson and Rob van der Wijngaart. *RCCE: a Small Library for Many-Core Communication*. Intel Corporation, March 2010. Version 1.05.

[MVdWF08] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE Supercomputing Conference*, pages 1–11, 2008.

[MW08] Paul E. McKenney and Jonathan Walpole. Introducing technology into the Linux kernel: A case study. *SIGOPS Operating Systems Review*, 42(5):4–17, July 2008.

[Nev12] Mark Nevill. Capabilities in barrelfish. Master's thesis, ETH Zurich, 2012.

[New] Linux Weekly News. Counting on the time stamp counter. `http://lwn.net/Articles/209101/`.

[NHM+09] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 221–234, 2009.

[Ope08] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, 2008. Version 3.0.

[Ous82] John Ousterhout. Scheduling techniques for concurrent systems. In *IEEE Distributed Computer Systems*, 1982.

[PBAR11] Simon Peter, Andrew Baumann, Zachary Anderson, and Timothy Roscoe. Gang scheduling isn't worth it... yet. Technical Report 745, ETH Zurich, November 2011.

[Pet06] Simon Peter. Design and implementation of a flexible framework for replication and its integration into NFSv4. Master's thesis, Systems Software and Distributed Systems Group, Department of Computer Science, Carl von Ossietzky University of Oldenburg, Germany, September 2006.

[PHA09] Heidi Pan, Benjamin Hindman, and Krste Asanović. Lithe: Enabling efficient composition of parallel libraries. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Parallelism*, March 2009.

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[PRB10] Simon Peter, Timothy Roscoe, and Andrew Baumann. Barrelfish on the Intel Single-chip Cloud Computer. Technical report, ETH Zurich, 2010. Barrelfish technical note 006.

[PSB+10] Simon Peter, Adrian Schüpbach, Paul Barham, Andrew Baumann, Rebecca Isaacs, Tim Harris, and Timothy Roscoe. Design principles for end-to-end multicore schedulers. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism*, June 2010.

[PSJT08] Victor Pankratius, Christoph Schaefer, Ali Jannesari, and Walter F. Tichy. Software engineering for multicore systems – an experience report. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*, May 2008.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228–34, 1980.

[PSMR11] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. Early experience with the Barrelfish OS and the Single-Chip Cloud Computer. In *3rd Many-core Applications Research Community Symposium*, Ettlingen, Germany, July 2011.

[Raz10] Kaveh Razavi. Barrelfish networking architecture. Distributed systems lab. Master's thesis, ETH Zurich, 2010.

[Rei07] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, July 2007.

[RLA07] Mohan Rajagopalan, Brian T. Lewis, and Todd A. Anderson. Thread scheduling for multi-core platforms. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, May 2007.

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. pages 298–307, December 1997.

[RLS98] Rajesh Raman, Miron Livny, and Marvin H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of the 7th IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[RSB12] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore scheduling for lightweight communicating processes. *Science of Computer Programming*, 77(6):727–740, 2012.

[Rus08] Mark Russinovich. Inside Windows Server 2008 kernel changes. *Microsoft TechNet Magazine*, March 2008.

[Rus09] Mark Russinovich. Inside Windows 7. *Microsoft MSDN Channel9*, January 2009.

[SATG+07] Bratin Saha, Ali-Reza Adl-Tabatabai, Anwar Ghuloum, Mohan Rajagopalan, Richard L. Hudson, Leaf Peterson, Vijay

Menon, Brian Murphy, Tatiana Shpeisman, Eric Sprangle, Anwar Rohillah, Doug Carmean, and Jesse Fang. Enabling scalability and performance in a large scale CMP environment. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, March 2007.

[Sax05] Eric Saxe. CMT and solaris performance, June 2005. `https://blogs.oracle.com/esaxe/entry/cmt_performance_enhancements_in_solaris`.

[SCC+11] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 61–76, 2011.

[Sch11] Bram Scheidegger. Barrelfish on Netronome. Master's thesis, ETH Zurich, February 2011.

[Sch12] Adrian Schüpbach. *The System Knowledge Base*. PhD thesis, ETH Zurich, 2012.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[Sid05] Suresh Siddha. Multi-core and Linux kernel. Technical report, Intel Open Source Technology Center, 2005.

[SLM+90] Michael L. Scott, T.J. LeBlanc, B.D. Marsh, T.G. Becker, C. Dubnicki, E.P Markatos, and N.G. Smithline. Implementation issues for the Psyche multiprocessor operating system. *Computing Systems*, 3(1):101–137, Winter 1990.

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 104–128, 2008.

[SMH11] Malte Schwartzkopf, Derek G. Murray, and Steven Hand. Condensing the cloud: running CIEL on many-core. In *Proceedings of the 1st Workshop on Systems for Future Multi-core Architectures*, April 2011.

[SNL+03] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 422–433, 2003.

[SPB+08] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the 1st Workshop on Managed Multi-Core Systems*, June 2008.

[SPL] Stanford parallel applications for shared memory (SPLASH-2). `http://www-flash.stanford.edu/apps/SPLASH/`.

[SQL] SQLite database engine. `http://www.sqlite.org/`.

[SS01] Michael L. Scott and William N. Scherer III. Scalable queue-based spin locks with timeout. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 44–52, 2001.

[Sup09] Super Micro Computer, Inc. *Supermicro H8QM3-2 H8QMi-2 User's Manual Version 1.1a*, December 2009. `http://www.supermicro.com/manuals/motherboard/MCP55/MNL-H8QM3i-2.pdf`.

[Tha10] Chuck Thacker. *Beehive: A many-core computer for FPGAs (v5)*. MSR Silicon Valley, January 2010. `http://projects.csail.mit.edu/beehive/BeehiveV5.pdf`.

[TS87] Chuck Thacker and Lawrence Stewart. Firefly: a multiprocessor workstation. *Computer Architecture News*, 15(5), 1987.

[TvR85] Andrew S. Tanenbaum and Robbert van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, 1985.

[TYA06a] TYAN Computer Corporation. *Tyan Thunder n4250QE S4958G4NR Manual Version 1.00*, 2006.

[TYA06b] TYAN Computer Corporation. *Tyan Thunder n6650W Manual Version 1.0*, 2006.

[Uhl07] Volkmar Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. *SIGOPS Operating Systems Review*, 41(4):49–58, July 2007.

[VHR+07] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *IEEE International Solid-State Circuits Conference*, pages 98–589, February 2007.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, January 2009.

[vRT92] Robert van Renesse and Andrew S. Tanenbaum. Short overview of Amoeba. In *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, pages 1–10, April 1992.

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *SIGOPS Operating Systems Review*, 43(2), April 2009.

[WCS06] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 124–133, Seattle, WA, USA, 2006.

[WF03] Y. Wiseman and D.G. Feitelson. Paired gang scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):581–592, June 2003.

[WGH+07] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown, III, and Anant Agarwal. On-chip interconnection architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[win] Windows scheduling. `http://msdn.microsoft.com/en-us/library/ms685096(VS.85).aspx`.

[WO09] Zheng Wang and Michael F. P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical Report 02-02-01, University of Washington, Seattle, WA, USA, February 2002.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[YJG03] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux utility for resource management. In *9th International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, June 2003.

[ZBF10] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, pages 129–142, 2010.

[Zel12] Gerd Zellweger. Unifying synchronization and events in a multicore operating system. Master's thesis, ETH Zurich, March 2012.

[ZJS10] Eddy Z. Zhang, Yunlian Jiang, and Xipeng Shen. Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, January 2010.

[ZKJ+08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, pages 29–42. USENIX Association, 2008.

[ZUP08] Stephen Ziemba, Gautam Upadhyaya, and Vijay S. Pai. Analyzing the effectiveness of multicore scheduling using performance counters. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.