The Multikernel: A new OS architecture for scalable multicore systems

Andrew Baumann[*], Paul Barham[†], Pierre-Evariste Dagand[‡], Tim Harris[†], Rebecca Isaacs[†], Simon Peter[*], Timothy Roscoe[*], Adrian Schüpbach[*], and Akhilesh Singhania[*]

[*] Systems Group, ETH Zurich [†] Microsoft Research, Cambridge [‡] ENS Cachan Bretagne

ABSTRACT

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the *multikernel*, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

Categories and Subject Descriptors: D.4.7 [Operating Systems]: Organization and Design

General Terms: Design, Experimentation, Performance

Keywords: Scalability, multicore processors, message passing

1. INTRODUCTION

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

SOSP'09, October 11–14, 2009, Big Sky, Montana, USA.

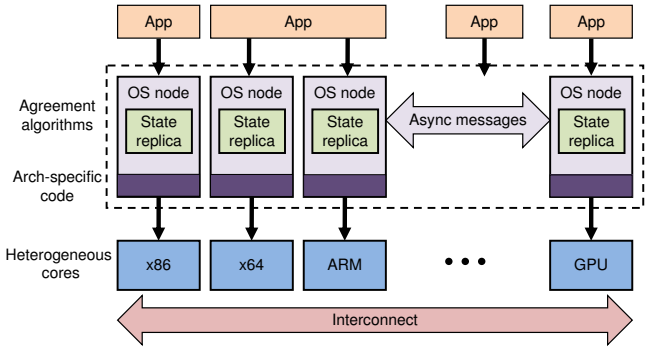


Figure 1: The multikernel model.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit messages. We identify three design principles: (1) make all inter-core communication explicit, (2) make OS structure hardware-neutral, and (3) view state as replicated instead of shared.

The model we develop, called a *multikernel* (Figure 1), is not only a better match to the underlying hardware (which is networked, heterogeneous, and dynamic), but allows us to apply insights from distributed systems to the problems of scale, adaptivity, and diversity in operating systems for future hardware.

Even on present systems with efficient cache-coherent shared memory, building an OS using message-based rather than shared-

data communication offers tangible benefits: instead of sequentially manipulating shared data structures, which is limited by the latency of remote data access, the ability to pipeline and batch messages encoding remote operations allows a single core to achieve greater throughput and reduces interconnect utilization. Furthermore, the concept naturally accommodates heterogeneous hardware.

The contributions of this work are as follows:

- We introduce the multikernel model and the design principles of explicit communication, hardware-neutral structure, and state replication.
- We present a multikernel, Barrelfish, which explores the implications of applying the model to a concrete OS implementation.
- We show through measurement that Barrelfish satisfies our goals of scalability and adaptability to hardware characteristics, while providing competitive performance on contemporary hardware.

In the next section, we survey trends in hardware which further motivate our rethinking of OS structure. In Section 3, we introduce the multikernel model, describe its principles, and elaborate on our goals and success criteria. We describe Barrelfish, our multikernel implementation, in Section 4. Section 5 presents an evaluation of Barrelfish on current hardware based on how well it meets our criteria. We cover related and future work in Sections 6 and 7, and conclude in Section 8.

2. MOTIVATIONS

Most computers built today have multicore processors, and future core counts will increase [12]. However, commercial multiprocessor servers already scale to hundreds of processors with a single OS image, and handle terabytes of RAM and multiple 10Gb network connections. Do we need new OS techniques for future multicore hardware, or do commodity operating systems simply need to exploit techniques already in use in larger multiprocessor systems?

In this section, we argue that the challenges facing a general-purpose operating system on future commodity hardware are different from those associated with traditional ccNUMA and SMP machines used for high-performance computing. In a previous paper [8] we argued that single computers increasingly resemble networked systems, and should be programmed as such. We rehearse that argument here, but also lay out additional scalability challenges for general-purpose system software.

2.1 Systems are increasingly diverse

A general-purpose OS today must perform well on an increasingly diverse range of system designs, each with different performance characteristics [60]. This means that, unlike large systems for high-performance computing, such an OS cannot be optimized at design or implementation time for any particular hardware configuration.

To take one specific example: Dice and Shavit show how a reader-writer lock can be built to exploit the shared, banked L2 cache on the Sun Niagara processor [40], using concurrent writes to the same cache line to track the presence of readers [20]. On Niagara this is highly efficient: the line remains in the L2 cache. On a traditional multiprocessor, it is highly inefficient: the line ping-pongs between the readers' caches.

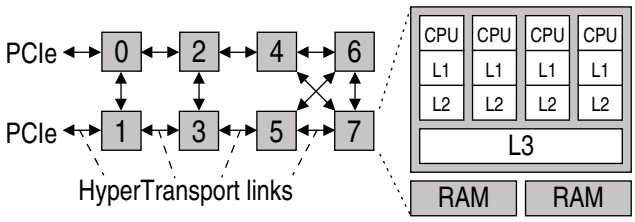


Figure 2: Node layout of an 8×4-core AMD system

This illustrates a general problem: any OS design tied to a particular synchronization scheme (and data layout policy) cannot exploit features of different hardware. Current OS designs optimize for the common hardware case; this becomes less and less efficient as hardware becomes more diverse. Worse, when hardware vendors introduce a design that offers a new opportunity for optimization, or creates a new bottleneck for existing designs, adapting the OS to the new environment can be prohibitively difficult.

Even so, operating systems are forced to adopt increasingly complex optimizations [27, 46, 51, 52, 57] in order to make efficient use of modern hardware. The recent scalability improvements to Windows7 to remove the dispatcher lock touched 6000 lines of code in 58 files and have been described as "heroic" [58]. The Linux read-copy update implementation required numerous iterations due to feature interaction [50]. Backporting receive-side-scaling support to Windows Server 2003 caused serious problems with multiple other network subsystems including firewalls, connection-sharing and even Exchange servers[1].

2.2 Cores are increasingly diverse

Diversity is not merely a challenge across the range of commodity machines. Within a single machine, cores can vary, and the trend is toward a mix of different cores. Some will have the same instruction set architecture (ISA) but different performance characteristics [34,59], since a processor with large cores will be inefficient for readily parallelized programs, but one using only small cores will perform poorly on the sequential parts of a program [31, 42]. Other cores have different ISAs for specialized functions [29], and many peripherals (GPUs, network interfaces, and other, often FPGA-based, specialized processors) are increasingly programmable.

Current OS designs draw a distinction between general-purpose cores, which are assumed to be homogeneous and run a single, shared instance of a kernel, and peripheral devices accessed through a narrow driver interface. However, we are not the only researchers to see an increasing need for OSes to manage the software running on such cores much as they manage CPUs today [55]. Moreover, core heterogeneity means cores can no longer share a single OS kernel instance, either because the performance tradeoffs vary, or because the ISA is simply different.

2.3 The interconnect matters

Even for contemporary cache-coherent multiprocessors, message-passing hardware has replaced the single shared interconnect [18, 33] for scalability reasons. Hardware thus resembles a message-passing network, as in the interconnect topology of the commodity PC server in Figure 2. While on most current hardware the cache-coherence protocol between CPUs ensures that the OS can continue to safely assume a single shared memory, networking problems

[1] See Microsoft Knowledge Base articles 927168, 927695 and 948496.

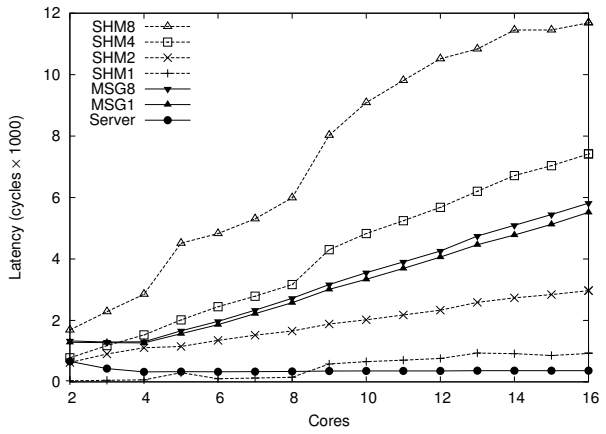


Figure 3: Comparison of the cost of updating shared state using shared memory and message passing.

like routing and congestion are well-known concerns on large-scale multiprocessors, and are now issues in commodity intra-machine interconnects [18]. Future hardware will comprise fewer chips but exhibit network effects inside the chip, such as with ring [38, 61] and mesh networks [68, 70]. The implication is that system software will have to adapt to the inter-core topology, which in turn will differ between machines and become substantially more important for performance than at present.

2.4 Messages cost less than shared memory

In 1978 Lauer and Needham argued that message-passing and shared-memory operating systems are duals, and the choice of one model over another depends on the machine architecture on which the OS is built [43]. Of late, shared-memory systems have been the best fit for PC hardware in terms of both performance and good software engineering, but this trend is reversing. We can see evidence of this by an experiment that compares the costs of updating a data structure using shared memory with the costs using message passing. The graph in Figure 3 plots latency against number of cores for updates of various sizes on the 4×4-core AMD system (described in Section 4.1).

In the shared memory case, threads pinned to each core directly update the same small set of memory locations (without locking) and the cache-coherence mechanism migrates data between 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 threads 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 threads issue a lightweight remote procedure call [10], (which we assume fits in a 64-byte cache line), to a single server process that performs the update on their behalf. The curves labeled *MSG1* and *MSG8*, show the cost of this *synchronous* RPC to the dedicated server thread. 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 threads. 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 threads (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 threads 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 OSes have point-solutions for this problem, which work on specific platforms or software systems, we believe 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 OS kernels.

2.5 Cache coherence is not a panacea

As the number of cores and the subsequent complexity of the interconnect grows, hardware cache-coherence protocols will become increasingly expensive. As a result, it is a distinct possibility that future operating systems will have to handle non-coherent memory [12, 49, 69], or will be able to realize substantial performance gains by bypassing the cache-coherence protocol [70].

It is already the case that programmable peripherals like NICs and GPUs do not maintain cache coherence with CPUs. Furthermore, many multicore processors have already demonstrated the use of non-coherent shared memory [15, 26, 68], and Mattson *et al.* [49] argue that the overhead of cache coherence restricts the ability to scale up to even 80 cores.

2.6 Messages are getting easier

There are legitimate software engineering issues associated with message-based software systems, and one might therefore question the wisdom of constructing a multiprocessor operating system based on a "shared nothing" model as we propose. There are two principal concerns, the first to do with not being able to access shared data, and the second to do with the event-driven programming style that results from asynchronous messaging.

However, the convenience of shared data is somewhat superficial. There are correctness and performance pitfalls when using shared data structures, and in scalable shared-memory programs (particularly high-performance scientific computing applications), expert developers are very careful about details such as lock granularity and how fields are laid out within structures. By fine-tuning code at a low level, one can minimize the cache lines needed to hold the shared data and reduce contention for cache line ownership. This reduces interconnect bandwidth and the number of processor stalls incurred when cache contents are stale.

The same kind of expertise is also applied to make commodity operating systems more scalable. As we have shown above, this

leads to a challenge in evolving the system as tradeoffs change, because the knowledge required for effective sharing of a particular data structure is encoded implicitly in its implementation. Note that this means programmers must think carefully about a shared-memory program *in terms of messages* sent by the cache-coherence protocol in response to loads and stores to data locations.

The second concern with message passing is the resultant "stack ripping" and obfuscation of control flow due to the event-driven nature of such programs. However, traditional monolithic kernels are essentially event-driven systems, even on multiprocessors. OS developers are perhaps more accustomed to thinking in terms of state machines and message handlers than other programmers.

Finally, we note that a message-passing, event-driven programming model is also the norm for many other programming domains, such as graphical user interface programming, some types of network server, and large-scale computation on clusters (where it has completely replaced the "distributed shared virtual memory" paradigm). This has meant that the programmability of message-passing or event-driven systems is an active research area with promising results that seem a natural fit for the multikernel model, such as the Tame/Tamer C++ libraries [41] and the X10 parallel programming language [16]. As the need for multicore programming environments at the user level becomes more pressing, we expect tools like these to support a message-passing abstraction will become widespread.

2.7 Discussion

The architecture of future computers is far from clear but two trends are evident: rising core counts and increasing hardware diversity, both between cores within a machine, and between systems with varying interconnect topologies and performance tradeoffs.

This upheaval in hardware has important consequences for a monolithic OS that shares data structures across cores. These systems perform a delicate balancing act between processor cache size, the likely contention and latency to access main memory, and the complexity and overheads of the cache-coherence protocol. The irony is that hardware is now changing faster than software, and the effort required to evolve such operating systems to perform well on new hardware is becoming prohibitive.

Increasing system and interconnect diversity, as well as core heterogeneity, will prevent developers from optimizing shared memory structures at a source-code level. Sun Niagara and Intel Nehalem or AMD Opteron systems, for example, already require completely different optimizations, and future system software will have to adapt its communication patterns and mechanisms at runtime to the collection of hardware at hand. It seems likely that future general-purpose systems will have limited support for hardware cache coherence, or drop it entirely in favor of a message-passing model. An OS that can exploit native message-passing would be the natural fit for such a design.

We believe that now is the time to reconsider how the OS should be restructured to not merely cope with the next generation of hardware, but efficiently exploit it. Furthermore, rather than evolving an inherently shared-memory model of OS structure to deal with complex tradeoffs and limited sharing, we take the opposite approach: design and reason about the OS as a distributed, non-shared system, and then employ sharing to optimize the model where appropriate.

Figure 4 depicts a spectrum of sharing and locking disciplines. Traditional operating systems, such as Windows and variants of Unix, have evolved from designs at the far left of the continuum towards finer-grained locking and more replication. These changes

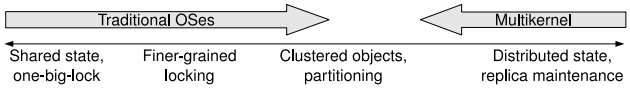


Figure 4: Spectrum of sharing and locking disciplines.

have been driven by hardware developments that exposed scalability bottlenecks, particularly the adoption of multiple processors in commodity PCs. Mainstream OSes are currently moving towards the center, where several "hot" data structures are partitioned or replicated across cores. Research systems take this even further with mechanisms like clustered objects that improve the locality of partitioned data [24]. In contrast, we propose an OS architecture positioned at the extreme right of the spectrum, where all state is replicated by default and consistency is maintained using agreement protocols.

3. THE MULTIKERNEL MODEL

In this section we present our OS architecture for heterogeneous multicore machines, which we call the *multikernel* model. In a nutshell, we structure the OS as a distributed system of cores that communicate using messages and share no memory (Figure 1). The multikernel model is guided by three design principles:

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

These principles allow the OS to benefit from the distributed systems approach to gain improved performance, natural support for hardware heterogeneity, greater modularity, and the ability to reuse algorithms developed for distributed systems.

After discussing the principles in detail below, in Section 4 we explore the implications of these principles by describing the implementation of Barrelfish, a new operating system based on the multikernel model.

3.1 Make inter-core communication explicit

Within a multikernel OS, all inter-core communication is performed using explicit messages. A corollary is that no memory is shared between the code running on each core, except for that used for messaging channels. As we have seen, using messages to access or update state rapidly becomes more efficient than shared-memory access as the number of cache-lines involved increases. We can expect such effects to become more pronounced in the future. Note that this does not preclude *applications* sharing memory between cores (see Section 4.8), only that the OS design itself does not rely on it.

Explicit communication patterns facilitate reasoning about the use of the system interconnect. In contrast to implicit communication (such as distributed shared memory, or the messages used for cache coherence), the knowledge of *what* parts of shared state are accessed *when* and by *who* is exposed. It is, of course, established practice in OS kernels to design point-solution data structures that can be updated using only one or two cache misses on particular architectures, but it is hard to evolve such operations as hardware changes, and such optimizations ignore the wider picture of larger state updates in the OS involving multiple structures.

We have previously argued that as the machine increasingly resembles a network, the OS will inevitably behave as a distributed

system [8]. Explicit communication allows the OS to deploy well-known networking optimizations to make more efficient use of the interconnect, such as pipelining (having a number of requests in flight at once), and batching (sending a number of requests in one message, or processing a number of messages together). In Section 5.2 we show the benefit of such techniques in the case of distributed capability management.

This approach also enables the OS to provide isolation and resource management on heterogeneous cores, or to schedule jobs effectively on arbitrary inter-core topologies by placing tasks with reference to communication patterns and network effects. Furthermore, the message abstraction is a basic requirement for spanning cores which are not cache-coherent, or do not even share memory.

Message passing allows operations that might require communication to be *split-phase*, by which we mean that the operation sends a request and immediately continues, with the expectation that a reply will arrive at some time in the future. When requests and responses are decoupled, the core issuing the request can do useful work, or sleep to save power, while waiting for the reply. A common, concrete example is remote cache invalidations. In a highly concurrent scenario, provided that completing the invalidation is not required for correctness, it can be more important not to waste time waiting for the operation to finish than to perform it with the smallest possible latency.

Finally, a system based on explicit communication is amenable to human or automated analysis. The structure of a message-passing system is naturally modular, because components communicate only through well-defined interfaces. Consequently it can be evolved and refined more easily [23] and made robust to faults [30]. Indeed, 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 Hoare's communicating sequential processes and the π-calculus, to the use of queuing theory to analyze the performance of complex networks

3.2 Make OS structure hardware-neutral

A multikernel separates the OS structure as much as possible from the hardware. This means that there are just two aspects of the OS as a whole that are targeted at specific machine architectures – the messaging transport mechanisms, and the interface to hardware (CPUs and devices). This has several important potential benefits.

Firstly, adapting the OS to run on hardware with new performance characteristics will not require extensive, cross-cutting changes to the code base (as was the case with recent scalability enhancements to Linux and Windows). This will become increasingly important as deployed systems become more diverse.

In particular, experience has shown that the performance of an inter-process communication mechanism is crucially dependent on hardware-specific optimizations (we describe those used in Barrelfish in Section 4.6). Hardware-independence in a multikernel means that we can isolate the distributed communication algorithms from hardware implementation details.

We envision a number of different messaging implementations (for example, a user-level RPC protocol using shared memory, or a hardware-based channel to a programmable peripheral). As we saw in Section 2.5, hardware platforms exist today without cache coherence, and even without shared memory, and are likely to become more widespread. Once the message transport is optimized, we can implement efficient message-based algorithms independently of the hardware details or memory layout.

A final advantage is to enable late binding of both the protocol implementation and message transport. For example, different transports may be used to cores on IO links, or the implementation may be fitted to the observed workload by adjusting queue lengths or polling frequency. In Section 5.1 we show how a topology-aware multicast message protocol can outperform highly optimized TLB shootdown mechanisms on commodity operating systems.

3.3 View state as replicated

Operating systems maintain state, some of which, such as the Windows dispatcher database or Linux scheduler queues, must be accessible on multiple processors. Traditionally that state exists as shared data structures protected by locks, however, in a multikernel, explicit communication between cores that share no memory leads naturally to a model of global OS state replicated across cores.

Replication is a well-known technique for OS scalability [4, 24], but is generally employed as an optimization to an otherwise shared-memory kernel design. In contrast, any potentially shared state in a multikernel is accessed and updated *as if* it were a local replica. In general, the state is replicated as much as is useful, and consistency is maintained by exchanging messages. Depending on the consistency semantics required, updates can therefore be long-running operations to coordinate replicas, and so are exposed in the API as non-blocking and split-phase. We provide examples of different update semantics in Section 4.

Replicating data structures can improve system scalability by reducing load on the system interconnect, contention for memory, and overhead for synchronization. Bringing data nearer to the cores that process it will result in lowered access latencies.

Replication is required to support domains that do not share memory, whether future general-purpose designs or present-day programmable peripherals, and is inherent in the idea of specializing data structures for particular core designs. Making replication of state intrinsic to the multikernel design makes it easier to preserve OS structure and algorithms as underlying hardware evolves.

Furthermore, replication is a useful framework within which to support changes to the set of running cores in an OS, either when hotplugging processors, or when shutting down hardware subsystems to save power. We can apply standard results from the distributed systems literature to maintaining the consistency of OS state across such changes.

Finally, a potentially important optimization of the multikernel model (which we do not pursue in this paper) is to privately share a replica of system state between a group of closely-coupled cores or hardware threads, protected by a shared-memory synchronization technique like spinlocks. In this way we can introduce (limited) sharing behind the interface as an optimization of replication.

3.4 Applying the model

Like all models, the multikernel, while theoretically elegant, is an idealist position: no state is shared and the OS acts like a fully distributed system. This has several implications for a real OS.

We discussed previously (in Section 2.6) the software engineering concerns of message-passing systems in general. Within the operating system kernel, which typically consists of a smaller amount of code written by expert programmers, software development challenges are more easily managed. However, the drawback of a idealist message-passing abstraction here is that certain platform-specific performance optimizations may be sacrificed, such as making use of a shared L2 cache between cores.

The performance and availability benefits of replication are achieved at the cost of ensuring replica consistency. Some operations will inevitably experience longer latencies than others, and the extent of this penalty will depend on the workload, the data volumes and the consistency model being applied. Not only that, the model supports multiple implementations of the agreement protocols used to maintain consistency. This increases the burden on the developer who must understand the consistency requirements for the data, but on the other hand, can also precisely control the degree of consistency. For example, a global flush of the TLB on each CPU is order-insensitive and can be achieved by issuing a single multicast request, whereas other operations may require more elaborate agreement protocols.

From an OS research perspective, a legitimate question is to what extent a real implementation can adhere to the model, and the consequent effect on system performance and scalability. To address this, we have implemented Barrelfish, a substantial prototype operating system structured according to the multikernel model.

Specifically, the goals for Barrelfish are that it:

- 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 OS data structures;
- can be re-targeted to different hardware, or make use of a different mechanism for sharing, without refactoring;
- can exploit the message-passing abstraction to achieve good performance by pipelining and batching messages;
- can exploit the modularity of the OS to place OS functionality according to the hardware topology or load.

In the next section we describe Barrelfish, and in Section 5 we explore the extent to which it achieves these goals.

4. IMPLEMENTATION

While Barrelfish is a point in the multikernel design space, it is not the only way to build a multikernel. In this section we describe our implementation, and note which choices in the design are derived from the model and which are motivated for other reasons, such as local performance, ease of engineering, policy freedom, etc. – we have liberally borrowed ideas from many other operating systems.

4.1 Test platforms

Barrelfish currently runs on x86-64-based multiprocessors (an ARM port is in progress). In the rest of this paper, reported performance figures refer to the following systems:

The *2×4-core Intel system* has an Intel s5000XVN motherboard 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. 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 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.

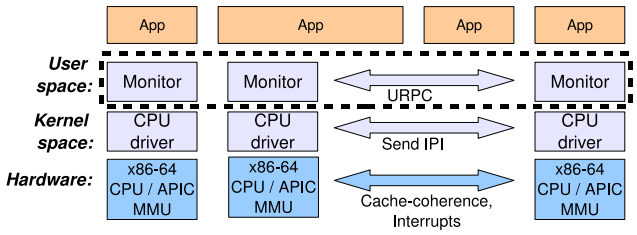


Figure 5: Barrelfish structure

The *4×4-core AMD system* has a Supermicro H8QM3-2 board 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.

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

4.2 System structure

The multikernel model calls for multiple independent OS instances communicating via explicit messages. In Barrelfish, we factor the OS instance on each core into a privileged-mode *CPU driver* and a distinguished user-mode *monitor* process, as in Figure 5 (we discuss this design choice below). CPU drivers are purely local to a core, and all inter-core coordination is performed by monitors. The distributed system of monitors and their associated CPU drivers encapsulate the functionality found in a typical monolithic microkernel: scheduling, communication, and low-level resource allocation.

The rest of Barrelfish consists of device drivers and system services (such as network stacks, memory allocators, etc.), which run in user-level processes as in a microkernel. 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 message.

4.3 CPU drivers

The CPU driver enforces protection, performs authorization, time-slices processes, and mediates access to the core and its associated hardware (MMU, APIC, etc.). Since it shares no state with other cores, the CPU driver can be completely event-driven, single-threaded, and nonpreemptable. It serially processes events in the form of traps from user processes or interrupts from devices or other cores. This means in turn that it is easier to write and debug than a conventional kernel, and is small[2] enabling its text and data to be located in core-local memory.

As with an exokernel [22], a CPU driver abstracts very little but performs dispatch and fast local messaging between processes on the core. It also delivers hardware interrupts to user-space drivers, and locally time-slices user-space processes. The CPU driver is invoked via standard system call instructions with a cost comparable to Linux on the same hardware.

The current CPU driver in Barrelfish is heavily specialized for the x86-64 architecture. In the future, we expect CPU drivers for other processors to be similarly architecture-specific, including

[2]The x86-64 CPU driver, including debugging support and libraries, is 7135 lines of C and 337 lines of assembly (counted by David A. Wheeler's "SLOCCount"), 54kB of text and 370kB of static data (mainly page tables).

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 1: LRPC latency

data structure layout, whereas the monitor source code is almost entirely processor-agnostic.

The CPU driver implements a lightweight, asynchronous (split-phase) 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 [9] or to L4 IPC [44].

Table 1 shows the one-way (user program to user program) performance of this primitive. On the 2×2-core AMD system, L4 performs a raw IPC in about 420 cycles. Since the Barrelfish figures also include a scheduler activation, user-level message dispatching code, and a pass through the thread scheduler, we consider our performance to be acceptable for the moment.

4.4 Monitors

Monitors collectively coordinate system-wide state, and encapsulate much of the mechanism and policy that would be found in the kernel of a traditional OS. The monitors are single-core, 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.

On each core, replicated data structures, such as memory allocation tables and address space mappings, are kept globally consistent by means of an agreement protocol run by the monitors. Application requests that access global state are handled by the monitors, which mediate access to remote copies of state.

Monitors perform some further housekeeping functions in Barrelfish. As described in Section 4.6, monitors are responsible for interprocess communication setup, and for waking up blocked local processes in response to messages from other cores. A monitor can also idle the core itself (to save power) when no other processes on the core are runnable. Core sleep is performed either by waiting for an inter-processor interrupt (IPI) or, where supported, the use of MONITOR and MWAIT instructions.

4.5 Process structure

The multikernel model leads to a somewhat different process structure than a typical monolithic multiprocessor OS. A process in Barrelfish is represented by a collection of *dispatcher* objects, one on each core on which it might execute. Communication in Barrelfish is not actually between processes but between dispatchers (and hence cores).

Dispatchers on a core are scheduled by the local CPU driver, which invokes an upcall interface that is provided by each dispatcher. This is the mechanism used in Psyche [48] and scheduler activations [3], and contrasts with the Unix model of simply resuming execution. Above this upcall interface, a dispatcher typically runs a core-local user-level thread scheduler.

The threads package in the default Barrelfish user library provides an API similar to POSIX threads. We anticipate that language runtimes and parallel programming libraries will take advantage of the ability to customize its behavior, but in the meantime the library provides enough support for implementing the more traditional model of threads sharing a single process address space across multiple cores, as we describe in Section 4.8.

4.6 Inter-core communication

In a multikernel, all inter-core communication occurs with messages. We expect to use different transport implementations for different hardware scenarios. However, the only inter-core communication mechanism available on our current hardware platforms is cache-coherent memory. Barrelfish at present therefore uses a variant of user-level RPC (URPC) [10] between cores: a region of shared memory is used as a channel to transfer cache-line-sized messages point-to-point between single writer and reader cores.

Inter-core messaging performance is critical for a multikernel, and our implementation is carefully tailored to the cache-coherence protocol to minimize the number of interconnect messages used to send a message. For example, on the fast path for a HyperTransport-based system, the sender writes the message sequentially into the cache line, while the receiver polls on the last word of the 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.

As an optimization, pipelined URPC 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 URPC messages is done by polling memory. Polling is cheap because the line is in the cache until invalidated; in addition, keeping the endpoints in an array can allow a hardware stride prefetcher to further improve performance. However, it is unreasonable to spin forever; instead, a dispatcher awaiting messages on URPC 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 time before blocking, however this can be improved by adaptive strategies similar to those used in deciding how long to spin on a shared-memory spinlock [37].

All message transports are abstracted behind a common interface, allowing messages to be marshaled, sent and received in a transport-independent way. As in most RPC systems, marshaling code is generated using a stub compiler to simplify the construction of higher-level services. A name service is used to locate other services in the system by mapping service names and properties to a service reference, which can be used to establish a channel to the service. Channel setup is performed by the monitors.

Table 2 shows the URPC single-message latency and sustained pipelined throughput (with a queue length of 16 messages); hop counts for AMD refer to the number of HyperTransport hops between sender and receiver cores.

Table 3 compares the overhead of our URPC 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

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

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

Table 2: URPC performance

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

Table 3: Messaging costs on 2×2-core AMD

cost, but also have less cache impact and do not incur a TLB flush. They can also be pipelined to trade off latency for throughput.

4.7 Memory management

Although a multikernel OS is itself distributed, it must consistently manage a set of global resources, such as physical memory. In particular, because user-level applications and system services may make use of shared memory across multiple cores, and because OS code and data is itself stored in the same memory, the allocation of physical memory within the machine must be consistent – for example, the system must ensure that one user process can never acquire a virtual mapping to a region of memory used to store a hardware page table or other OS object.

Any OS faces the same problem of tracking ownership and type of in-memory objects. Most systems use some kind of unique identifiers together with accounting information, such as file descriptors in Unix or object handles in Windows, managed by data structures in shared memory. For Barrelfish, we decided to use a capability system modeled on that of seL4 [39]. In this model, all memory management is performed explicitly through system calls that manipulate capabilities, which are user-level references to kernel objects or regions of physical memory. The model has the useful property of removing dynamic memory allocation from the CPU driver, which is only responsible for checking the correctness of operations that manipulate capabilities to memory regions through *retype* and *revoke* operations.

All virtual memory management, including allocation and manipulation of page tables, is performed entirely by user-level code [28]. For instance, to allocate and map in a region of memory, a user process must first acquire capabilities to sufficient RAM to store the required page tables. It then retypes these RAM capabilities to page table capabilities, allowing it to insert the new page tables into its root page table; although the CPU driver performs the actual page table and capability manipulations, its sole task is checking their correctness. The user process may then allocate more RAM capabilities, which it retypes to mappable frame capabilities, and finally inserts into its page tables to map the memory.

We chose capabilities so as to cleanly decentralize resource allocation in the interests of scalability. In hindsight, this was a mistake: the capability code is unnecessarily complex, and no more efficient than scalable per-processor memory managers used in conventional OSes like Windows and Linux. All cores must still keep their local capability lists consistent, to avoid situations such as user-level acquiring a mapping to a hardware page table.

However, one benefit has been uniformity: most operations requiring global coordination in Barrelfish can be cast as instances of capability copying or retyping, allowing the use of generic consistency mechanisms in the monitors. These operations are not specific to capabilities, and we would have to support them with any other accounting scheme.

Page mapping and remapping is an operation which requires global coordination – if an address space mapping is removed or its rights are reduced, it is important that no stale information remains in a core's TLB before any action occurs that requires the operation to have completed. This is implemented by a one-phase commit operation between all the monitors.

A more complex problem is capability retyping, of which revocation is a special case. This corresponds to changing the usage of an area of memory and requires global coordination, since retyping the same capability in different ways (e.g. a mappable frame and a page table) on different cores leads to an inconsistent system. All cores must agree on a single ordering of the operations to preserve safety, and in this case, the monitors initiate a two-phase commit protocol to ensure that all changes to memory usage are consistently ordered across the processors.

4.8 Shared address spaces

We are interested in investigating language runtimes that extend the multikernel model to user-space applications. However, for most current programs, Barrelfish supports the traditional process model of threads sharing a single virtual address space across multiple dispatchers (and hence cores) by coordinating runtime libraries on each dispatcher. This coordination affects three OS components: virtual address space, capabilities, and thread management, and is an example of how traditional OS functionality can be provided over a multikernel.

A shared virtual address space can be achieved by either sharing a hardware page table among all dispatchers in a process, or replicating hardware page tables with consistency achieved by message protocols. The tradeoff between these two is similar to that investigated in Corey [13]; the former is typically more efficient, however the latter may reduce cross-processor TLB invalidations (because it is possible to track which processors may have cached a mapping), and is also the only way to share an address space between cores that do not support the same page table format.

As well as sharing the address space, user applications also expect to share capabilities (for example, to mappable memory regions) across cores. However, a capability in a user's address space is merely a reference to a kernel-level data structure. The monitors provide a mechanism to send capabilities between cores, ensuring in the process that the capability is not pending revocation, and is of a type that may be transferred. The user-level libraries that perform capability manipulation invoke the monitor as required to maintain a consistent capability space between cores.

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 between dispatchers (and hence cores). Barrelfish is responsible only for multiplexing the dispatchers on each core via the CPU driver scheduler, and coordinating the CPU drivers to perform, for example, gang scheduling or co-scheduling of dispatchers. This allows a variety

of spatio-temporal scheduling policies from the literature [62, 65] to be applied according to OS policy.

4.9 Knowledge and policy engine

Dealing with heterogeneous hardware and choosing appropriate system mechanisms is a crucial task in a multikernel. Barrelfish employs a service known as the *system knowledge base* (SKB) [60], which maintains knowledge of the underlying hardware in a subset of first-order logic.[4] It is populated with information gathered through hardware discovery (including ACPI tables, PCI bus probes, and CPUID data), online measurement (such as URPC communication latency and bandwidth between all core pairs in the system), and pre-asserted facts 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). Using this rich repository of data, the SKB allows concise expression of optimization queries, for example to allocate device drivers to cores in a topology-aware manner, to select appropriate message transports for inter-core communication, and to provide the topology information necessary for NUMA-aware memory allocation.

As one example, we describe in Section 5.1 how the SKB is used to construct a cache- and topology-aware network for efficient communication within multicore machines. Space precludes further discussion of policy decisions, but we believe that such a high-level, declarative approach to reasoning about the machine hardware (augmented with online measurements) is an essential part of a multikernel-based system.

4.10 Experiences

Barrelfish is one concrete implementation of the multikernel model in Section 3, and its structure differs substantially from that of a monolithic OS like Linux or indeed a hypervisor like Xen [7]. However, we emphasize that Barrelfish is not the only way to build a multikernel.

In particular, the factoring of the OS node into separate CPU driver and monitor components is not required by the model; our experience is that it is *not* optimal for performance, but has compelling engineering advantages. The downside of the separation is that invocations from processes to the OS 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. However, this is constant as the number of cores increases. Moving the monitor into kernel space would remove this penalty, at the cost of a more complex kernel-mode code base.

In addition, our current network stack (which runs a separate instance of lwIP [47] per application) is very much a placeholder. We are interested in appropriate network stack design for multikernels, and in particular we feel many ideas from RouteBricks [21] are applicable to a scalable end-system multikernel.

The principal difference between current OS designs and the multikernel model is not the location of protection or trust boundaries between system components, but rather the reliance on shared data as the default communication mechanism. Examples from monolithic systems include process, thread, and virtual machine control blocks, page tables, VFS objects, scheduler queues,[5] network sockets and inter-process communication buffers.

[4]The initial implementation of the SKB is based on a port of the ECL^{i}PS^e constraint programming system [5].

[5]Systems with per-processor run queues implement load balancing and thread migration either via a work stealing or work offloading model. In both cases this involves access to the run queues of remote processors.

Some of this state can be naturally partitioned, but (as we saw in Section 1) the engineering effort to do this piecemeal in the process of evolving an OS for scalability can be prohibitive. Furthermore, state which is inherently shared in current systems requires more effort to convert to a replication model. Finally, the shared-memory single-kernel model cannot deal with cores that are heterogeneous at the ISA level.

In contrast, starting with a multikernel model, as in Barrelfish, allows these issues to be addressed from the outset. From a research perspective, Barrelfish can be seen as a useful "proving ground" for experimenting with such ideas in an environment freed from the constraints of an existing complex monolithic system.

5. EVALUATION

In this section we evaluate how well Barrelfish meets the goals in Section 3.4: good baseline performance, scalability with cores, adaptability to different hardware, exploiting the message-passing abstraction for performance, and sufficient modularity to make use of hardware topology-awareness. We start with a detailed case-study of TLB shootdown, and then look at other workloads designed to exercise various parts of the OS.

5.1 Case study: TLB shootdown

TLB shootdown – the process of maintaining TLB consistency by invalidating entries when pages are unmapped – is one of the simplest operations in a multiprocessor OS that requires global coordination. It is also a short, but latency-critical, operation and so represents a worst-case comparison for a multikernel.

In Linux and Windows, inter-processor interrupts (IPIs) are used: a core that wishes to change a page mapping writes the operation to a well-known location and sends an interrupt to every core that might have the mapping in its TLB. Each core takes the trap, acknowledges the IPI by writing to a shared variable, invalidates the TLB entry, and resumes. The initiating core can continue immediately after every IPI has been acknowledged, since by the time each core resumes to user space the TLB entry is guaranteed to be flushed. This has low latency, but can be disruptive as each core immediately takes the cost of a trap (about 800 cycles). The TLB invalidation itself is fast, taking 95-320 cycles on a current x86-64 core.

In Barrelfish, we use messages for shootdown. In the naive algorithm, the local monitor broadcasts invalidate messages to the others and waits for all the replies. We would expect this to show higher latency than the IPI approach, since the remote monitors handle the messages only when "convenient". However, fortunately, the message-passing paradigm allows us to improve on this approach without significantly restructuring the code. In particular, we can exploit knowledge about the specific hardware platform, extracted from the system knowledge base at runtime, to achieve very good TLB shootdown performance.

Figure 6 shows the costs of the raw inter-core messaging mechanisms (without TLB invalidation) for four URPC-based TLB shootdown protocols on the 8×4-core AMD system.

In the *Broadcast* protocol, the master monitor uses a single URPC channel to broadcast a shootdown request to every other core. Each slave core polls the same shared cache, waiting for the master to modify it, and then acknowledges with individual URPCs to the master. This performs badly due to the cache-coherence protocol used by AMD64 processors [1, section 7.3]. When the master updates the line, it is invalidated in all other caches. Each slave core then pulls the new copy from the master's cache. With N

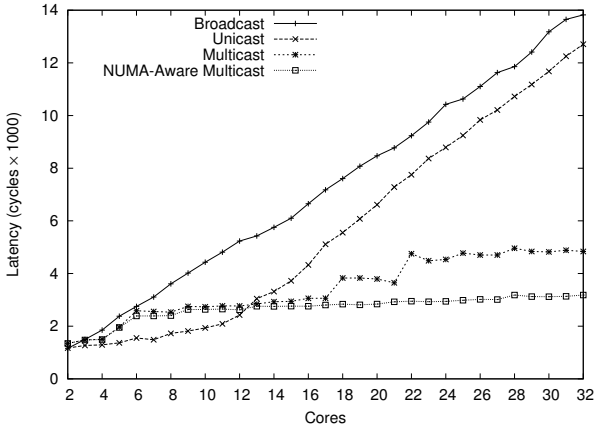


Figure 6: Comparison of TLB shootdown protocols

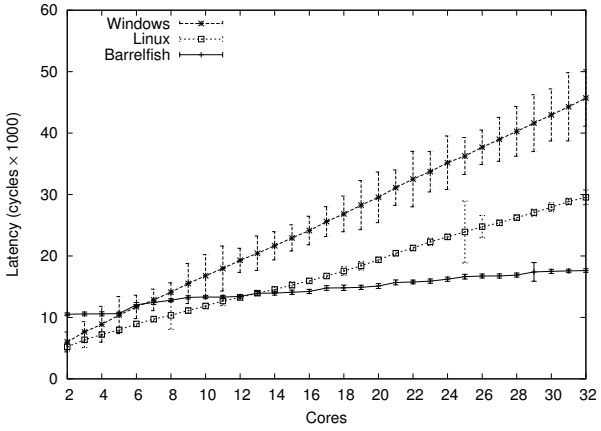


Figure 7: Unmap latency on 8×4-core AMD

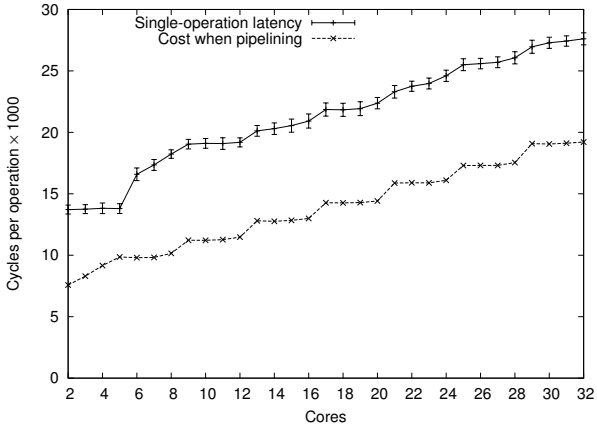


Figure 8: Two-phase commit on 8×4-core AMD

cores, the data crosses the interconnect N times, and latency grows linearly with the number of cores.

The *Unicast* protocol sends individual requests to each slave over unicast URPC so that the cache lines are only shared by two cores. While this performs much better than the broadcast protocol, particularly for a small number of cores, it still has linear scalability. The flatter curve below eight cores is likely to be the processor's hardware "stride prefetcher" predicting correctly which cache lines are likely to be accessed in the master's receive loop.

The HyperTransport interconnect is effectively a broadcast network, where each read or write operation results in probes being sent to all other nodes. However, newer 4-core Opteron processors have a shared on-chip L3 cache and appear as a single HyperTransport node, and so cache lines shared only by these cores will not result in interconnect traffic. This motivates using an explicit two-level multicast tree for shootdown messages. Hence in the *Multicast* protocol, the master sends a URPC message to the first core of each processor, which forwards it to the other three cores in the package. Since they all share an L3 cache, this second message is much cheaper, but more importantly all eight processors can send in parallel without interconnect contention. As shown in Figure 6, the multicast protocol scales significantly better than unicast or broadcast.

Finally, we devised a protocol that takes advantage of the NUMA nature of the machine by allocating URPC buffers from memory local to the multicast aggregation nodes, and having the master send requests to the highest latency nodes first. Once again, there are many analogies to networked systems which motivate these changes. The resulting protocol, labeled *NUMA-Aware Multicast* on Figure 6 scales extremely well across the 32-way system, showing steps only when the number of levels in the tree increases.

This communication protocol has good performance for the 8×4-core AMD system, but relies on hardware knowledge (including the interconnect topology and memory locality) that differs widely between systems, and only works when initiated from the first core. In Barrelfish, we use a query on the system knowledge base to construct a suitable multicast tree at runtime: for every source core in the system, the SKB computes an optimal route consisting of one multicast aggregation node per processor socket, ordered by decreasing message latency, and its children. These routes are calculated online at startup, and used to configure inter-monitor communication.

End-to-end unmap latency: The complete implementation of unmapping pages in Barrelfish adds a number of costs to the baseline protocol, and so in practice is much slower. These include the fixed overhead of LRPC to the local monitor, the per-message cost of marshaling and event demultiplexing, scheduling effects in the monitors, and variable overhead when a core's monitor is not currently its polling channels.

Nevertheless, as Figure 7 shows, the complete message-based unmap operation in Barrelfish quickly outperforms the equivalent IPI-based mechanisms in Linux 2.6.26 and Windows Server 2008 R2 Beta, Enterprise Edition. The error bars on the graphs are standard deviation. We show the latency to change the permissions of a page mapped by a varying number of cores, using `mprotect` on Linux and `VirtualProtect` on Windows. Despite the fact that up to 32 user-level processes are involved in each unmap operation, performance scales better than Linux or Windows, both of which incur the cost of serially sending IPIs. The large overhead on top of the baseline protocol figures is largely due to inefficiencies in the message dispatch loop of the user-level threads package, which has not been optimized.

5.2 Messaging performance

Two-phase commit

As we discussed in Section 4, Barrelfish uses a distributed two-phase commit operation for changing memory ownership and usage via capability retyping. This necessarily serializes more messages

	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 4: IP loopback performance on 2×2-core AMD

than TLB shootdown, and is consequently more expensive. Nevertheless, as Figure 8 shows, using the same multicast technique as with shootdown we achieve good scaling and performance. If there are enough operations to perform, we can also pipeline to amortize the latency. The "cost when pipelining" line shows that a typical capability retype operation consumes fewer cycles than IPI-based TLB-shootdowns on Windows and Linux.

The cost of polling

It is reasonable to ask whether polling to receive URPC messages is a wasteful strategy under real workloads and job mixes. This depends on a number of factors such as scheduling policy and messaging load, but we present a simplistic model here: assume we poll for P cycles before sleeping and waiting for an IPI, which costs C cycles. If a message arrives at time t, the overhead (cost in extra cycles) of message reception is therefore:

$$overhead = \begin{cases} t & \text{if } t \leq P, \\ P + C & \text{otherwise.} \end{cases}$$

– and the latency of the message is:

$$latency = \begin{cases} 0 & \text{if } t \leq P, \\ C & \text{otherwise.} \end{cases}$$

In the absence of any information about the distribution of message arrival times, a reasonable choice for P is C, which gives upper bounds for message overhead at $2C$, and the latency at C.

Significantly, for Barrelfish on current hardware, C is around 6000 cycles, suggesting there is plenty of time for polling before resorting to interrupts. C in this case includes context switch overhead but not additional costs due to TLB fills, cache pollution, etc.

IP loopback

IP loopback (with no physical network card to impose a bottleneck) can be a useful stress-test of the messaging, buffering, and networking subsystems of the OS, since in many systems a loopback interface is often used for communication between networked services on the same host, such as a web application and database.

Linux and Windows use in-kernel network stacks with packet queues in shared data structures, thus loopback utilization requires kernel interaction and shared-memory synchronization. On Barrelfish, we achieve the equivalent functionality for point-to-point links by connecting two user-space IP stacks via URPC. 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 URPC 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 to cores using `libnuma`.

Table 4 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 URPC messages avoids any shared-memory other than the URPC channel and packet payload; conversely, Linux causes more cache-coherence traffic for shared-memory synchronization. Barrelfish also benefits by avoiding kernel crossings.

5.3 Compute-bound workloads

In this section we use compute-bound workloads, in the form of the NAS OpenMP benchmark suite [36] and the SPLASH-2 parallel application test suite [63], 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.

The Barrelfish user-level scheduler supports POSIX-like threads, and allows processes to share an address space across multiple cores as in a traditional OS. We compare applications running under Linux and Barrelfish on the same hardware, using GCC 4.3.3 as the compiler, with the GNU GOMP OpenMP runtime on Linux, and our own implementation over Barrelfish.

Figure 9 shows the results of five applications from the 4×4-core AMD machine[6]. We 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 at least demonstrate that despite its distributed structure, Barrelfish can still support large, shared-address space parallel code with little performance penalty. The differences we observe are due to our user-space threads library vs. the Linux in-kernel implementation – for example, Linux implements barriers using a system call, whereas our library implementation exhibits different scaling properties under contention (in Figures 9a and 9c).

5.4 IO workloads

The experiments in this section are designed to exercise device drivers, interrupts and OS buffering using high-bandwidth IO, with the aim again of exposing any performance penalty due to the multikernel architecture.

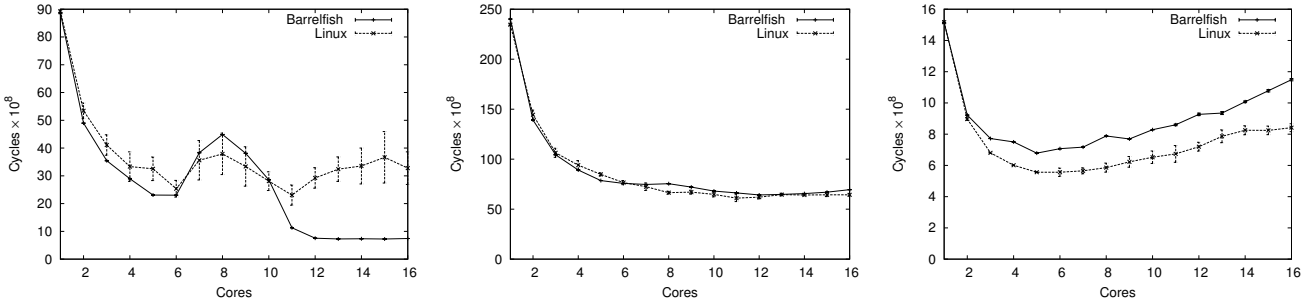
Network throughput

We first measure the UDP throughput of Barrelfish when communicating over Gigabit Ethernet using an Intel e1000 card. This exercises DMA, inter-process communication, networking and scheduling.

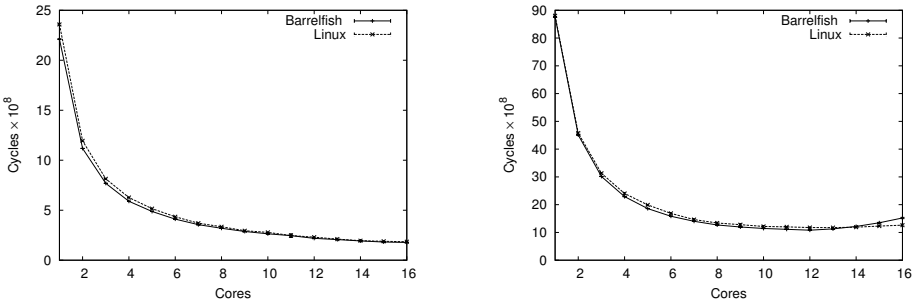
Using Barrelfish on the 2×4-core Intel machine, we ran an e1000 driver and a single-core application that listens on a UDP socket and echos every received packet back to the sender. The network stack is lwIP [47] linked as a library in the application's domain. Receive and transmit buffers are allocated by lwIP and passed to the network card driver, which manages the card's receive and transmit rings. The two processes communicate over a URPC channel, allowing us to vary the placement of the driver and client on cores.

We use two load generators, also with e1000 NICs, running Linux and the ipbench daemon [71], which generates UDP traffic

[6] The remaining OpenMP applications depend upon thread-local storage, which Barrelfish does not yet support.



(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 9: Compute-bound workloads on 4×4-core AMD (note different scales on y-axes)

at a configurable rate and measures the achieved echo throughput. We obtained UDP echo payload throughput of up to 951.7 Mbit/s, at which point we are close to saturating the card. By comparison, Linux 2.6.26 on the same hardware also achieves 951 Mbit/s; note that we pinned the Linux `inetd` server to a single core to prevent sub-optimal process migration in this experiment.

Web server and relational database

Our final IO example is of serving both static and dynamic web content from a relational database. This scenario, while still synthetic, is closer to a realistic I/O bound application configuration.

The 2×2-core AMD machine is used with an Intel e1000 NIC. First, the machine serves a 4.1kB static web page to a set of clients, and we measure the throughput of successful client requests using `httperf` [54] on a cluster of 17 Linux clients.

On Barrelfish, we run separate processes for the web server (which uses the lwIP stack), e1000 driver, and a timer driver (for TCP timeouts). These communicate over URPC, allowing us to experiment with placement of domains 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 [45] 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 checksumming, scatter gather and TCP segmentation offload on the network interface.

The Barrelfish e1000 driver does not yet support the offload features, but is also substantially simpler. It sustained 18697 requests per second (640 Mbit/s), versus 8924 for lighttpd on Linux (316 Mbit/s). The performance gain is mainly due to avoiding kernel-user crossings by running entirely in user space and communicating over URPC.

Finally, we use the same load pattern to execute web-based SELECT queries modified from the TPC-W benchmark suite on a SQLite [64] database running on the remaining core of the machine, connected to the web server via URPC. In this configuration we can sustain 3417 requests per second (17.1 Mbit/s), and are bottlenecked at the SQLite server core.

5.5 Summary

It would be wrong to draw any quantitative conclusions from our large-scale benchmarks; the systems involved are very different. An enormous investment has been made in optimizing Linux and Windows for current hardware, and conversely our system is inevitably more lightweight (it is new, and less complete). Instead, they should be read as indication that Barrelfish performs *reasonably* on contemporary hardware, our first goal from Section 3.4.

We make stronger claims for the microbenchmarks. Barrelfish can scale well with core count for these operations, and can easily adapt to use more efficient communication patterns (for example, tailoring multicast to the cache architecture and hardware topology). Finally we can also demonstrate the benefits of pipelining and batching of request messages without requiring changes to the OS code performing the operations.

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. In the course of this evaluation we ported a web server, network stack [47], and various drivers, applications and libraries to Barrelfish, which gives us confidence that our OS design offers a feasible alternative to existing monolithic systems.

Nevertheless, bringing up a new OS from scratch is a substantial undertaking, and limits the extent to which we can fully evaluate the multikernel architecture. In particular, this evaluation does not address complex application workloads, or higher-level operating system services such as a storage system. Moreover, we have not evaluated the system's scalability beyond currently-available commodity hardware, or its ability to integrate heterogeneous cores.

6. RELATED WORK

Although a new point in the OS design space, the multikernel model is related to much previous work on both operating systems and distributed systems.

In 1993 Chaves *et al.* [17] examined the tradeoffs between message passing and shared data structures for an early multiprocessor, finding a performance tradeoff biased towards message passing for many kernel operations.

Machines with heterogeneous cores that communicate using messages have long existed. The Auspex [11] and IBM System/360 hardware consisted of heterogeneous cores with partially shared memory, and unsurprisingly their OSes resembled distributed systems in some respects. We take inspiration from this; what is new is the scale of parallelism and the diversity of different machines on which a general-purpose OS must run. Similarly, explicit communication has been used on large-scale multiprocessors such as the Cray T3 or IBM Blue Gene, to enable scalability beyond the limits of cache-coherence.

The problem of scheduling computations on multiple cores that have the same ISA but different performance tradeoffs is being addressed by the Cypress project [62]; we see this work as largely complementary to our own. Also related is the fos system [69] which targets scalability through space-sharing of resources.

Most work on OS scalability for multiprocessors to date has focused on performance optimizations that reduce sharing. Tornado and K42 [4, 24] introduced clustered objects, which optimize shared data through the use of partitioning and replication. However, the base case, and the means by which replicas communicate, remains shared data. Similarly, Corey [13] advocates reducing sharing within the OS by allowing applications to specify sharing requirements for OS data, effectively relaxing the consistency of specific objects. As in K42, however, the base case for communication is shared memory. In a multikernel, we make no specific assumptions about the application interface, and construct the OS as a shared-nothing distributed system, which may locally share data (transparently to applications) as an optimization.

We see a multikernel as distinct from a microkernel, which also uses message-based communication between processes to achieve protection and isolation but remains a shared-memory, multithreaded system in the kernel. For instance, Barrelfish has some structural similarity to a microkernel, in that it consists of a distributed system of communicating user-space processes which provide services to applications. However, unlike multiprocessor microkernels, each core in the machine is managed completely independently – the CPU driver and monitor share no data structures with other cores except for message channels.

That said, some work in scaling microkernels is related: Uhlig's distributed TLB shootdown algorithm is similar to our two-phase commit [67]. The microkernel comparison is also informative: as we have shown, the cost of a URPC message is comparable to that of the best microkernel IPC mechanisms in the literature [44], without the cache and TLB context switch penalties.

Disco and Cellular Disco [14, 25] were based on the premise that large multiprocessors can be better programmed as distributed systems, an argument complementary to our own. We see this as further evidence that the shared-memory model is not a complete solution for large-scale multiprocessors, even at the OS level.

Prior work on "distributed operating systems" [66] aimed to build a uniform OS from a collection of independent computers linked by a network. There are obvious parallels with the multikernel approach, which seeks to build an OS from a collection of cores communicating over links within a machine, but also important differences: firstly, a multikernel may exploit reliable in-order message delivery to substantially simplify its communication. Secondly, the latencies of intra-machine links are lower (and less variable) than between machines. Finally, much prior work sought to handle partial failures (i.e. of individual machines) in a fault-tolerant manner, whereas in Barrelfish the complete system is a failure unit. That said, extending a multikernel beyond a single machine to handle partial failures is a possibility for the future that we discussion briefly below.

Despite much work on distributed shared virtual memory systems [2, 56], performance and scalability problems have limited their widespread use in favor of explicit message-passing models. There are parallels with our argument that the single-machine programming model should now also move to message passing. Our model can be more closely compared with that of distributed shared objects [6, 32], in which remote method invocations on objects are encoded as messages in the interests of communication efficiency.

7. EXPERIENCE AND FUTURE WORK

The multikernel model has been strongly influenced by the process of building a concrete implementation. Though not by any means yet a mature system, we have learned much from the process.

Perhaps unsurprisingly, queuing effects become very important in a purely message-oriented system, and we encountered a number of performance anomalies due to extreme sensitivity to message queue lengths. Understanding such effects has been essential to performance debugging. Scheduling based on queue lengths, as in Scout [53], may be a useful technique to apply here.

When implementing transports based on cache-coherent shared memory, we found it important to understand exactly the cache-coherence protocol. We experimented with many URPC implementations, with often unexpected results that were explained by a careful analysis of cache line states and interconnect messages.

Related future work includes efficient multicast, incast and anycast URPC transports. As we found in the case of unmap, there are cases where a multicast tree is more efficient for performing global operations than many point-to-point links, and we expect such transports will be important for scalability to many cores.

Our current implementation is based on homogeneous Intel and AMD multiprocessors, and so does not represent a truly heterogeneous environment. A port is in progress to the ARM processor architecture, which will allow us to run a Barrelfish CPU driver and monitor on programmable network cards. This will also allow us to experiment with specializing data structures and code for different processors within the same operating system.

There are many ideas for future work that we hope to explore. Structuring the OS as a distributed system more closely matches the structure of some increasingly popular programming models for datacenter applications, such as MapReduce [19] and Dryad [35], 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.

Another area of interest is file systems. Barrelfish currently uses a conventional VFS-style file-oriented API backed by NFS. It may be fruitful to draw on techniques from high-performance cluster file systems and the parallel IO models of cloud computing providers to construct a scalable, replicated file system *inside* the computer.

Barrelfish is at present a rather uncompromising implementation of a multikernel, in that it never shares data. As we noted in Section 2.1, some machines are highly optimized for fine-grained sharing among a subset of processing elements. A next step for Barrelfish is to exploit such opportunities by limited sharing behind the existing replica-oriented interfaces. This also raises the issue of how to decide *when* to share, and whether such a decision can be automated.

8. CONCLUDING REMARKS

Computer hardware is changing faster than system software, and in particular operating systems. Current OS structure is tuned for a coherent shared memory 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, we have proposed the multikernel architecture as a way forward. We view the OS as, first and foremost, a distributed system which may be amenable to local optimizations, rather than centralized system which must somehow be scaled to the network-like environment of a modern or future machine. By basing the OS design on replicated data, message-based communication between cores, and split-phase operations, we 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, monolithic kernels. Its source code is available at http://www.barrelfish.org/.

Acknowledgments

We would like to thank our shepherd, Jeff Dean, the anonymous reviewers, and Tom Anderson, Steven Hand, and Michael Scott for their helpful suggestions for how to improve this paper and Barrelfish in general. We would also like to thank Jan Rellermeyer, Ankush Gupta, and Animesh Trivedi for their contributions to Barrelfish, and Charles Gray for the term "CPU driver".

References

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Sept. 2007. Publication number 24593.

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2), 1996.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazoswka, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of threads. *ACM Transactions on Computer Systems*, 10:53–79, 1992.

[4] J. Appavoo, D. Da Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems*, 25(3), 2007.

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

[6] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance evaluation of the Orca shared-object system. *ACM Transactions on Computer Systems*, 16(1):1–40, 1998.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Oct. 2003.

[8] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. 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.

[9] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

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

[11] S. 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.

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

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

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

[15] C. Caşcaval, J. G. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. Warren, Jr. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the 8th IEEE Symposium on High-Performance Computer Architecture*, pages 311–322, 2002.

[16] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 519–538, 2005.

[17] E. M. Chaves, Jr., P. C. Das, T. J. LeBlanc, B. D. Marsh, and M. L. Scott. Kernel–Kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, 1993.

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

[19] J. Dean and S. 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.

[20] D. Dice and N. Shavit. TLRW: return of the read-write lock. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

[22] D. R. Engler, M. F. Kaashoek, and J. 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, Dec. 1995.

[23] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the EuroSys Conference*, pages 177–190, 2006.

[24] B. Gamsa, O. Krieger, J. Appavoo, and M. 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, Feb. 1999.

[25] K. Govil, D. Teodosiu, Y. Huang, and M. 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.

[26] M. Gschwind. The Cell Broadband Engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[27] 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.

[28] S. M. Hand. Self-paging in the Nemesis operating system. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 73–86, Feb. 1999.

[29] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. White paper, Intel, Sept. 2006. ftp://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf.

[30] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *Operating Systems Review*, 40(3):80–89, July 2006.

[31] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7):33–38, 2008.

[32] P. Homburg, M. van Steen, and A. Tanenbaum. Distributed shared objects as a communication paradigm. In *Proceedings of the 2nd Annual ASCI Conference*, pages 132–137, June 1996.

[33] Intel Corporation. QuickPath architecture white paper, 2008.

[34] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 186–197, 2007.

[35] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference*, pages 59–72, 2007.

[36] 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, Oct. 1999.

[37] A. R. Karlin, K. Li, M. S. Manasse, and S. 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.

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

[39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009.

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

[41] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proceedings of the 2007 Annual USENIX Technical Conference*, pages 1–14, 2007.

[42] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 64–75, 2004.

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

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

[45] lighttpd webserver. http://www.lighttpd.net/.

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

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

[48] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, Oct. 1991.

[49] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 International Conference on Supercomputing*, pages 1–11, 2008.

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

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

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

[53] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 153–167, Oct. 1996.

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

[55] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: Heterogeneous multiprocessing with satellite kernels. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[56] J. Protić, M. Tomašević, and V. Milutinović. Distributed shared memory: Concepts and systems. *IEEE Parallel and Distributed Technology*, 4(2):63–79, 1996.

[57] M. Russinovich. Inside Windows Server 2008 kernel changes. *Microsoft TechNet Magazine*, Mar. 2008.

[58] M. Russinovich. Inside Windows 7. *Microsoft MSDN Channel9*, Jan. 2009.

[59] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. 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.

[60] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, June 2008.

[61] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3):1–15, 2008.

[62] D. Shelepov and A. Fedorova. Scheduling on heterogeneous multicore processors using architectural signatures. In *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[63] Stanford parallel applications for shared memory (SPLASH-2). http://www-flash.stanford.edu/apps/SPLASH/.

[64] SQLite database engine. http://www.sqlite.org/.

[65] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proceedings of the EuroSys Conference*, pages 47–58, 2007.

[66] A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, 1985.

[67] V. Uhlig. *Scalability of Microkernel-Based Systems*. PhD thesis, Computer Science Department, University of Karlsruhe, Germany, June 2005.

[68] 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 *International Solid-State Circuits Conference*, pages 98–589, Feb. 2007.

[69] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *Operating Systems Review*, 43(2), Apr. 2009.

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

[71] I. Wienand and L. Macpherson. ipbench: A framework for distributed network benchmarking. In *AUUG Winter Conference*, Melbourne, Australia, Sept. 2004.