The Multikernel: A New OS Architecture for Scalable Multicore Systems

by Andrew Baumann, Paul Barham, Pierre-Evariste Dagard, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singlania

Presented by Vladimir Solmon

Claim

"The challenge of future multicore hardware is best met by embracing the networked nature of the machine [and] rethinking OS architecture using ideas from distributed systems."

— Baumann, et. al., *The Multikernel: A New OS Architecture for Scalable Multicore Systems*

Why rethink OS architecture for multicore hardware?

- Changes in Hardware
  - Hardware is increasingly diverse while operating systems struggle to keep up
  - Optimization for one hardware design may decrease performance on another
  - Single machines may have a mix of different cores and ISAs making it impossible for them to share a single kernel instance
  - Message-passing hardware is becoming common for communication between cores on cache-coherent multiprocessors

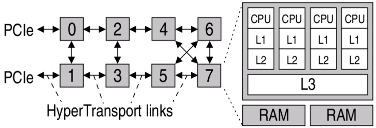


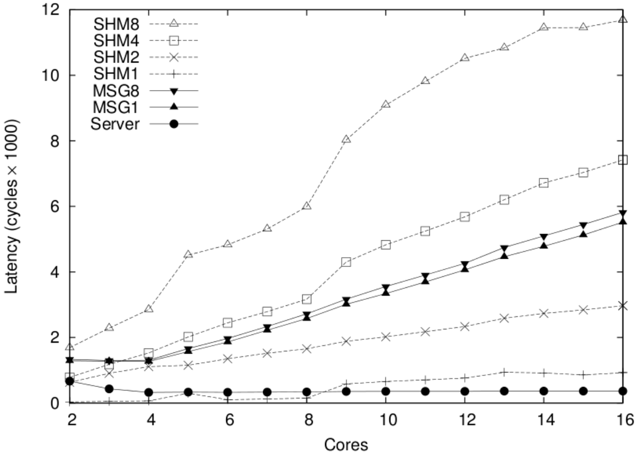
Figure 2: Node layout of an 8×4-core AMD system

Why embrace networking?

- Lauer and Needham argue that message-passing and shared-memory systems are duals and choice should be dependent on hardware
  - Cache coherence increasingly expensive as more cores are added
  - Perception that shared-memory code is more intuitive is belied by the complexity of accurately writing good shared memory code
  - Many programmers are already familiar with message-passing because it is the norm for GUIs
  - Kernel programming already deals heavily in message passing (i.e. interrupts, faults...)

Why embrace networking?

- As more cores are added, messages cost less than shared memory



The Multikernel Model

- Structured as "a distributed system of cores that communicate using messages and share no memory"
- Three design principles:
    1. Make all inter-core communication explicit
    2. Make OS structure hardware-neutral
    3. View state as replicated instead of shared

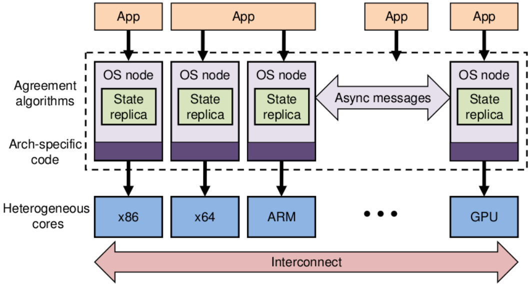


Figure 1: The multikernel model.

Make inter-core communication explicit

- Implicit communication: shared memory
- Explicit communication: message passing
  - No memory is shared between code running on separate cores unless it is used in message passing channels
  - Makes explicit which parts of shared state are accessed, when and by whom
  - Allows OS to use networking optimizations such as pipelining and batching
  - More effective use of the CPU due to “split-phase” (asynchronous) operations – a process sends a request then either moves on to useful work or sleeps until response is returned
  - Communication interfaces lead to a naturally modular system, making it easier to run human or automated analysis using theory built up around complex networks

Make OS structure hardware-neutral

- Only two aspects of a multikernel OS must be targeted to specific machine architectures
  - Messaging transport mechanisms
  - Interface to hardware (CPUs and devices)
- Advantages
  - Limited changes to code base to adapt operating system to run on new platforms
  - Distributed communication algorithms can be isolated from hardware implementation details
  - Enables use of late binding for protocol implementation and message transport, allowing for run-time workload optimizations

View state as replicated

- All state that must be shared across cores is replicated in each core
    - Replicated shared state in a multikernel is treated by each core as though it were local
    - Updates of shared state between cores are passed via messages which may be long-running operations
- Advantages
    - Replicating structures reduces load on system interconnects, reduces memory contention, and reduces local access time
    - Replication provides framework for supporting changes to the set of running cores in the OS

Model meets reality

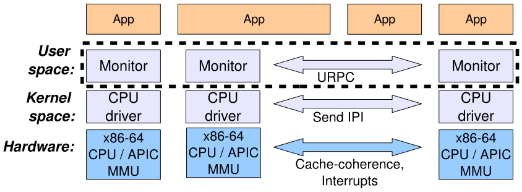
- The model represents an ideal which may not be fully realizable in practice
  - Idealist message-passing approach would mean sacrificing performance optimizations like shared L2 cache between cores
  - Replicated state may lack consistency, particularly under heavy load, forcing the programmer to understand their own consistency requirements and whether they will be met by a particular implementation

BarrelFish — Not the only way to implement a multikernel!

- Goals
    - Give comparable performance to existing commodity OSes on current multicore hardware
    - Demonstrate evidence of scalability to large numbers of cores under large workloads
    - Can be retargeted to different hardware, or use 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 hardware topology or load

System Structure

- OS instance on each core is split into
  - CPU driver, purely local to its core, hardware dependent
  - Monitor, a user-mode process responsible for all inter-core communication, hardware independent
- Collection of CPU drivers and monitors form a distributed system which provides kernel functionality (scheduling, communication, low-level resource allocation)
- Device drivers and system services (network stacks, memory allocators) run in user-level processes as in a microkernel



CPU driver

- Enforce protection, perform authorization, time-slice processes, mediate access to the core
- Shares no state with other cores so it can be completely event-driven, single-threaded, and non-preemptable
- Serially processes events — traps from user processes or interrupts from devices on other cores
- Very small — 7135 lines of C + 337 lines of assembly
- Provides fast local messaging for processes running on its core
- Hardware dependent (current Barrelfish implementation heavily specialized for x86-64 architecture)

Monitors

- Schedulable single-core user-space processes
- Communicate across cores to collectively coordinate system-wide state
- Replicated state on each core is kept globally consistent via an agreement protocol run by the monitors
- Set up interprocess communication
- Wake up blocked local processes when messages come in from other cores
- Idle the core to save power when no other processes are runnable

Process Structure — Dispatchers

- In Barrelfish processes are represented by a collection of "dispatcher" objects
  - Each dispatcher object represents a core on which the process might run
  - Dispatchers are scheduled on each core by the local CPU driver via an upcall interface (similar to Scheduler Activations)
  - Each dispatcher generally runs a user-level thread scheduler which is local to its core

Inter-core Communication

- In the multikernel model, all inter-core communication occurs via messages
  - In reality, the only inter-core communication mechanism available on current hardware platforms is cache-coherent memory
- Barrelfish uses this cache-coherent memory to implement a variant of URPC between cores
  - Region of shared memory mapped between cores is used to transfer cache-line-sized messages
  - Messages received by polling cache and eventually blocking with request to local monitor to wake up when message arrives
  - Implementation built to minimize number of interconnect messages used to send a message by having receiver poll the last word of the cache line and only collect the message when this word is updated

Cost of Polling

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

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

- P is the number of cycles polled before sleeping
- C is the cost of going to sleep
- t is the time at which the message arrives
- On current hardware, C is 6000 cycles, meaning that there is plenty of time for polling

Memory management

- The multikernel is a distributed process but it has to manage physical memory as a global resource
    - User-level applications may run across multiple cores and their memory accesses must be consistent across all cores
    - Since OS code and data is stored in the same memory, inconsistent physical memory allocation could allow user code to overwrite OS objects
- Barrelfish uses a capability system modeled on seL4, an experimental formally verified kernel
    - All memory management is done through system calls that manipulate capabilities
    - This means the CPU driver doesn't have to make memory allocation decisions, it only validates the capabilities of user-level processes and operations that manipulate capabilities

Memory management continued

- All virtual memory management, including allocation and manipulation of page tables, is performed entirely by user-level code
- Steps for a user-level process to allocate and map a region of memory:
  1. Acquire capabilities from the CPU driver for enough RAM to store the needed page tables
  2. Send a request to the CPU driver to retype these RAM capabilities to page table capabilities
- The choice to use capabilities was a trade-off: resource allocation is cleanly decentralized, but the code is much more complex
  - Capability retyping (changing the usage of an area of memory) requires global coordination across all cores
  - Page mapping or remapping requires global coordination across all cores

Knowledge and policy engine

- Barrelfish uses System Knowledge Base (SKB) to provide information about underlying hardware
    - SKB probes hardware to get both static and dynamic information about the system (new components added to the system, URPC latency)
    - SKB allows the OS to make hardware-specific optimization decisions such as how to efficiently allocate NUMA memory

Limitations of the Barrelfish implementation

- Separating the CPU driver and monitor negatively impacted system performance
  - Since most of the OS is in user space, every time a process called into the OS it requires a local RPC call (two context switches) rather than a system call (one context switch), creating a constant overhead of thousands of cycles
  - Moving the monitor into the kernel would improve the speed of these operations but mean significantly more complex kernel-mode code

Evaluation — TLB shootdown

- TLB shootdown occurs when TLB entries must be invalidated when pages are unmapped
  - TLB shootdown requires sending messages to all cores to invalidate their TLBs
- One of the simplest operations in the mulitkernel that requires global coordination, so used as a base case test (like using GetPid() to measure system call overhead)
- Naive algorithm: local monitor that unmapped the page broadcasts invalidate messages to other monitors and waits for all replies
  - This algorithm can be improved on if we have knowledge of the system hardware that allows for optimizations

TLB shootdown — Broadcast and Unicast

- Four different methods for TLB shootdown tried on Barrelfish
  - Broadcast — Monitor uses a single URPC channel to broadcast to all other cores
    - Remaining cores poll the same shared cache waiting for the update
    - Remaining cores send individual URPC acknowledgements.
  - Unicast — Individual requests sent to all other cores from the originating monitor
    - Each messaging cache shared by only two monitors

TLB Shootdown — Multicast and NUMA-Aware

- Multicast — Originating monitor sends URPC call to the first core in each processor and this core then forwards the call to the three cores on the processor.
  - Requires 4-core Opteron processors with shared on-chip L3 cache which appear as a single HyperTransport node
  - Cache lines shared only by cores within a processor don't generate interconnect traffic so all 8 processors can forward to their three cores without interconnect contention
- NUMA-Aware Multicast: Uses information provided by the SKB about machine's NUMA-ness to allocate URPC buffers from memory local to the highest latency nodes first

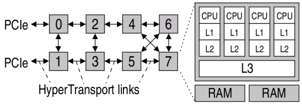
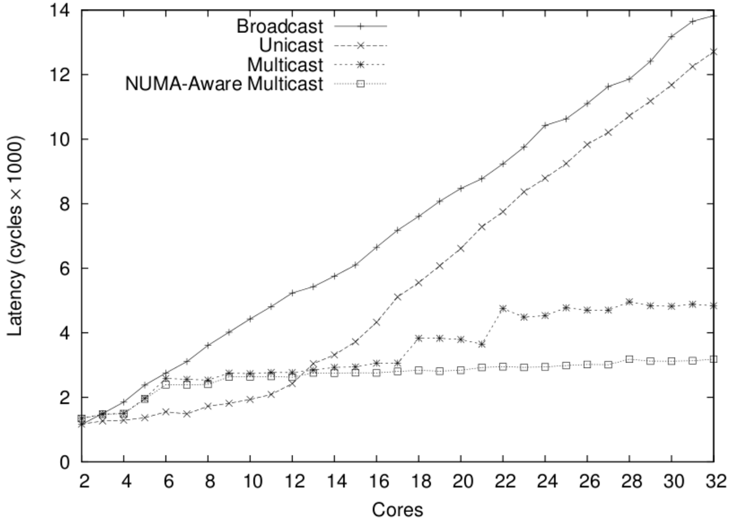
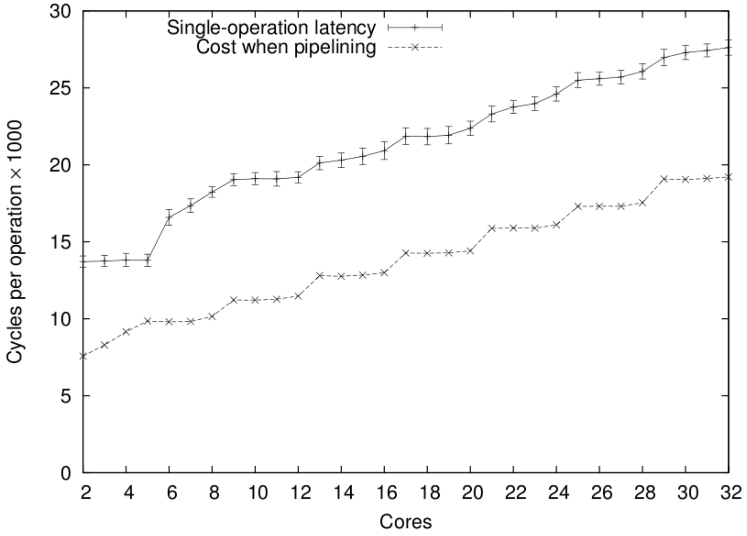


Figure 2: Node layout of an 8×4-core AMD system

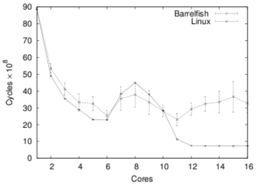
Comparison of TLB shootdown protocols



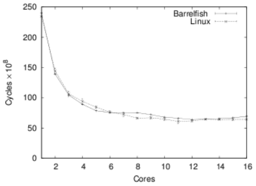
Two-phase commit on 8x4-core AMD



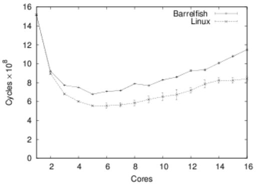
Benchmark Comparisons of Linux and Barrelfish



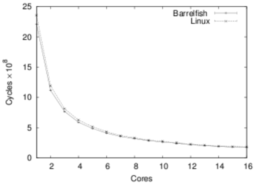
(a) OpenMP conjugate gradient (CG)



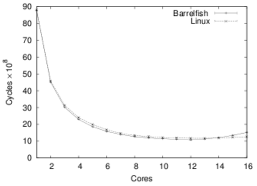
(b) OpenMP 3D fast Fourier transform (FT)



(c) OpenMP integer sort (IS)



(d) SPLASH-2 Barnes-Hut



(e) SPLASH-2 radiosity

Sources

Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüupbach, A., and Singhania, A.
The multikernel: A new os architecture for scalable multicore systems.
In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (2009).

Bershad, B. N., Anderson, T. E., Lazowska, E. D., and Levy, H. M.
User-level interprocess communication for shared memory multiprocessors.
In *ACM Transactions on Computer Systems* (1990).

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S.
sel4: Formal verification of an os kernel.
In *Proceedings of the 22nd ACM Symposium on Operating System Principles* (2009).

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