The Multikernel: A new OS architecture for scalable multicore systems

Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Issacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania

Presented by
Sharmadha Moorthy
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
Challenges of future multicore hardware

- Multicore systems exhibit diverse architectural tradeoffs
- Variety of environments and dynamic nature of workloads
- General-purpose OS cannot be optimized at design or implementation time
- OS design tied to particular synchronization scheme or data layout policy
- Adapting OS to new environment is difficult
- Heterogeneous cores cannot share single OS kernel instance
Message-passing over shared memory 1

- Message-passing hardware has replaced shared interconnect for cache-coherent multiprocessors

- Ability to pipeline and batch messages encoding remote operations – greater throughput, reduce interconnect utilization

- Lauer and Needham’s claim that they are duals and choice between them depends on machine architecture
Message-passing over shared memory 2

- Expensive cache coherence protocols as cores and complexity of interconnect increases
- Correctness and performance pitfalls when using shared data structures
- Knowledge for effective sharing encoded implicitly in implementation - cache coherence protocol
- Event-driven systems already applied to monolithic kernels, other programming domains such as GUI, network server
Detour - Cache mapping and associativity

- Direct mapped cache
  - Cache with C blocks, memory with xC blocks
  - Memory block N in cache line N mod C

- Fully associative cache, n-way set associative cache

Source: http://www.cs.nyu.edu/courses/fall07/V22.0436-001/lectures/
Message passing over Shared memory 3

Source: Slides by Tim Harris, Andrew Baumann and Rebecca Isaacs. Joint work with colleagues at MSR Cambridge and ETH Zurich.
Message-passing over shared memory

- Messages cost less than shared memory as more cores are added.
The Multikernel model

- Structure OS as distributed system of cores communicate using messages and no shared memory

- Achieves improved performance, support for hardware heterogeneity, greater modularity and ability to reuse algorithms for distributed systems
Explicit inter-core communication

- Facilitates reasoning about use of system interconnect
- Allows OS to deploy networking optimizations: pipelining, batching
- Enables isolation and resource management on heterogeneous cores, effective job scheduling on inter-core topologies
- Structure can be evolved and refined easily and robust to faults
- Allows operations to have split-phase communication, ex: remote cache invalidations
- Requirement for cores which are not cache-coherent or don’t share memory!
Hardware-neutral OS structure

- Two aspects of OS targeted at specific machine architectures – messaging transport mechanism and interface to hardware
- Distributed communication algorithms are isolated from hardware implementation details
- Different messaging implementations: URPC using shared memory, hardware based channel to programmable peripheral
- Enable late binding of protocol implementation and message transport
- Flexible transports on IO links and implementation fitted to observed workloads
Replication of state

- Shared OS state across cores is replicated and consistency maintained by exchanging messages.
- Updates are exposed in API as non-blocking and split-phase as they can be long operations.
- Reduces load on system interconnect, contention for memory, overhead for synchronization; improves scalability.

- Preserve OS structure as hardware evolves.

Source: Slides by Tim Harris, Andrew Baumann and Rebecca Isaacs. Joint work with colleagues at MSR Cambridge and ETH Zurich.
In reality...

Model represents an ideal which may not be fully realizable in practice

- Certain platform-specific performance optimizations may be sacrificed – shared L2 cache
- Cost and penalty of ensuring replica consistency varies on workload, data volumes and consistency model
Barrelfish

• Goals:
  ▫ Comparable performance to existing commodity OS on multicore hardware
  ▫ Scalability to large number of cores under considerable workload
  ▫ Ability to be re-targeted to different hardware without refactoring
  ▫ Exploit message-passing abstraction to achieve good performance by pipelining and batching messages
  ▫ Exploit modularity of OS and place OS functionality according to hardware topology or load

• It is not the only way to build a multikernel!
System Structure

- Multiple independent OS instances communicating via explicit messages
- OS instance on each core factored into
  - privileged-mode CPU driver which is hardware dependent
  - user-mode Monitor process: responsible for intercore communication, hardware independent

- System of monitors and CPU drivers provide scheduling, communication and low-level resource allocation
- Device drivers and system services run in user-level processes
CPU Drivers

- Enforces protection, performs authorization, time-slices processes and mediates access to core and hardware
- Completely event-driven, single-threaded and nonpreemptable
- Serially processes events in form of traps from user processes or interrupts from devices or other cores
- Performs dispatch and fast local messaging between processes on core
- Implements lightweight, asynchronous (split-phase) same-core IPC facility
Monitors

- Schedulable, single-core user-space processes
- Suited for split-phase, message oriented inter-core communication of messages
- Collectively coordinate consistency of replicated data structures through agreement protocols
- Responsible for IPC setup
- Wakes up blocked processes in response to messages from other cores
- Idle the core when no other processes on the core are runnable, waiting for IPI
Process structure

- Process is represented by collection of dispatcher objects, one on each core which might execute it
- Communication is between dispatchers
- Dispatchers are scheduled by local CPU driver through upcall interface
- Dispatcher runs a core local user-level thread scheduler
- Thread library provides support for model of threads sharing single process address space across multiple cores
Inter-core communication

- Variant of URPC for *cache coherent memory* – region of shared memory used as channel for cache-line-sized messages
- Implementation tailored to cache-coherence protocol to minimize number of interconnect messages
- Dispatchers poll incoming channels for predetermined time before blocking with request to notify local monitor when message arrives
- All message transports are abstracted allowing messages to be marshalled, channel setup by monitors
Memory management

- Manage set of global resources: physical memory shared by applications and system services across multiple cores
- OS code and data stored in same memory - allocation of physical memory must be consistent
- Capability system – memory managed through system calls that manipulate capabilities
- Capabilities are user-level references to kernel objects or regions of physical memory
- CPU driver only responsible for checking correctness of operations through retype and revoke operations
- All virtual memory management performed entirely by user-level code
Memory management 2

- Decentralize resource allocation in interest of scalability
- Unnecessarily complex and requires consistency of local capability lists
- Uniformity – operations requiring global coordination can be cast as instances of capability
- Page mapping and remapping using one-phase commit operation between all monitors
- Capability retyping and revocation using two-phase commit protocol – need to ensure changes to memory usages consistently ordered across processors
Shared address space

- Single virtual address space is shared across multiple dispatchers by coordinating runtime libraries on each dispatcher
- Virtual address space:
  - Sharing hardware page table is efficient
  - Replicating hardware page tables with consistency reduces cross-processor TLB invalidations
- User-level libraries perform capability manipulation, invoke monitor to maintain consistent capability space between cores
- Thread schedulers on each dispatcher exchange messages to create and unblock threads, migrate threads between dispatchers
- Gang scheduling or co-scheduling of dispatchers
Knowledge and policy engine

- System knowledge base (SKB) maintains knowledge of underlying hardware in subset of first-order logic
- Populated with information gathered through hardware discovery, online measurement, pre-asserted facts
- SKB allows concise expression of optimization queries
  - Allocation of device drivers to cores, NUMA-aware memory allocation in topology aware manner
  - Selection of appropriate message transports for inter-core communication
Lessons from Barrelyfish implementation

- Separation of CPU driver and monitor adds constant overhead of local RPC rather than system calls
- Moving monitor into kernel space is at the cost of complex kernel-mode code base
- Differs from current OS designs on reliance on shared data as default communication mechanism
  - Engineering effort to partition data is prohibitive
  - Requires more effort to convert to replication model
  - Shared-memory single-kernel model cannot deal with heterogeneous cores at ISA level
Case study: TLB shootdown

- Process of maintaining TLB consistency by invalidating entries when pages are unmapped
- Short, latency-critical – represents worst-case comparison for multikernel
- Inter-process interrupts has low latency, disruptive and cost of trap to other cores

Case study: TLB shootdown using messages (Barrelfish) - broadcast

- Single URPC channel to broadcast to all other cores
- Remaining cores poll same shared cache, send individual URPC acknowledgements
- Higher latency, messages handled only when “convenient”, data crosses interconnect N times

Source: Slides by Tim Harris, Andrew Baumann and Rebecca Isaacs. Joint work with colleagues at MSR Cambridge and ETH Zurich.
Case study: TLB shootdown using messages (Barrelfish) - n*unicast

- Individual requests sent to all other cores from originating monitor, cache line only shared between two cores

Source: Slides by Tim Harris, Andrew Baumann and Rebecca Isaacs. Joint work with colleagues at MSR Cambridge and ETH Zurich.
Case study: TLB shootdown using messages (Barrelfish) - multicast

- Originating monitor sends URPC message to first core of each processor which forwards to 3 other cores in package
- Cache lines within processor don’t generate interconnect traffic
- 8 processors can send in parallel without interconnect contention

Source: Slides by Tim Harris, Andrew Baumann and Rebecca Isaacs. Joint work with colleagues at MSR Cambridge and ETH Zurich.
Case study: TLB shootdown using messages (Barrelfish) - NUMA-Aware Multicast

- Uses information provided by SKB to allocate URPC buffers from memory local to multicast aggregation nodes
- Master send requests to highest latency nodes first

Source: Slides by Tim Harris, Andrew Baumann and Rebecca Isaacs. Joint work with colleagues at MSR Cambridge and ETH Zurich.
Case study: Comparison of TLB shootdown protocols
Case study: Unmap latency on 8x4-core AMD
Two-phase commit on 8X4-core AMD
Benchmark comparisons of Linux & 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