Advanced Operating Systems (CS 202)

Distributed OS– intro and discussion

UCR
UNIVERSITY OF CALIFORNIA, RIVERSIDE

Overview

- Hardware is changing, so software must too
  - Multicores are here to stay
  - Architectures are heterogeneous
  - Applications are unpredictable unlike specialized systems
- How do operating systems scale?
- Do we need new OS architectures?

Landscape/motivation

- Systems are diverse
  - different implementations require different tradeoffs
    - Some nice examples
- Cores are increasingly diverse
  - Different general purpose cores
  - Accelerators and specialized processors
  - Typically cannot share an OS with such differences
- Interconnects matter: within cores and across cores

What has gone on before?

- Early on, locks were not so expensive
    - Just use them
- Hardware evolved, memory expensive
    - Large caches
    - Cache coherence
    - NUMA machines
    - Increasing gap between memory and processor
    - Shared memory expensive!

Older SMP OS projects

- E.g., Tornado
- Locality matters
- Customize OS to underlying hardware
  - But now we have high diversity
  - Cannot have one size fit all
- Use replication as an optimization
- Still good principles

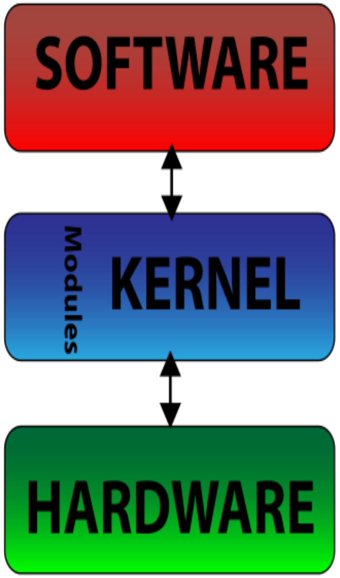
The Multikernel: A New OS Architecture for Scalable Multicore Systems

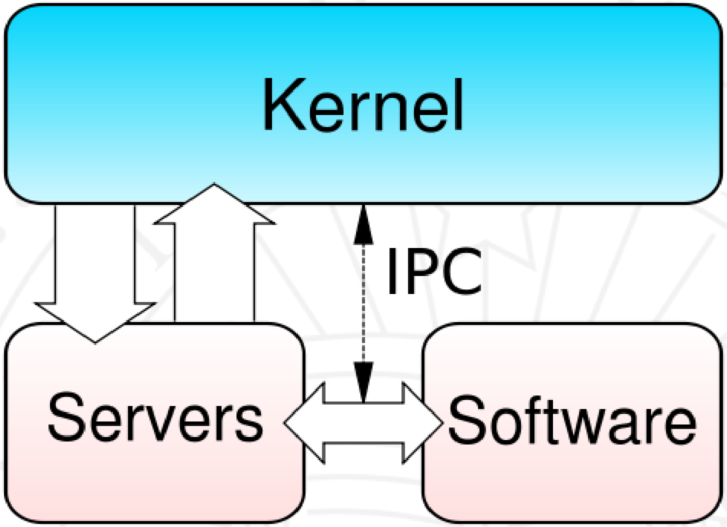
By (last names): Baumann, Barham, Dagand, Harris, Isaacs, Peter, Roscoe, Schupbach, Singhania

Presented by: Clyde, B[illegible]

UNIVERSITY OF CALIFORNIA, RIVERSIDE

The Modern Kernel(s)





The Problem with Modern Kernels

- Modern Operating systems can no longer take serious advantage of the hardware they are running on
- There exists a scalability issue in the shared memory model that many modern kernels abide by
- Cache coherence overhead restricts the ability to scale to many-cores

Solution: MultiKernel

- Treat the machine as a network of independent cores
- Make all inter-core communication explicit; use message passing
- Make OS structure hardware-neutral
- View state as replicated instead of shared

But wait! Isn't message passing slower than Shared Memory?

Not at scale

But wait! Isn’t message passing slower than Shared Memory?

- At scale it has been show that message passing has surpassed shared memory efficiency
- Shared memory at scale seems to be plagued by cache misses which cause core stalls
- Hardware is starting to resemble a message-passing network

But wait! Isn’t message passing slower than Shared Memory? (cont.)

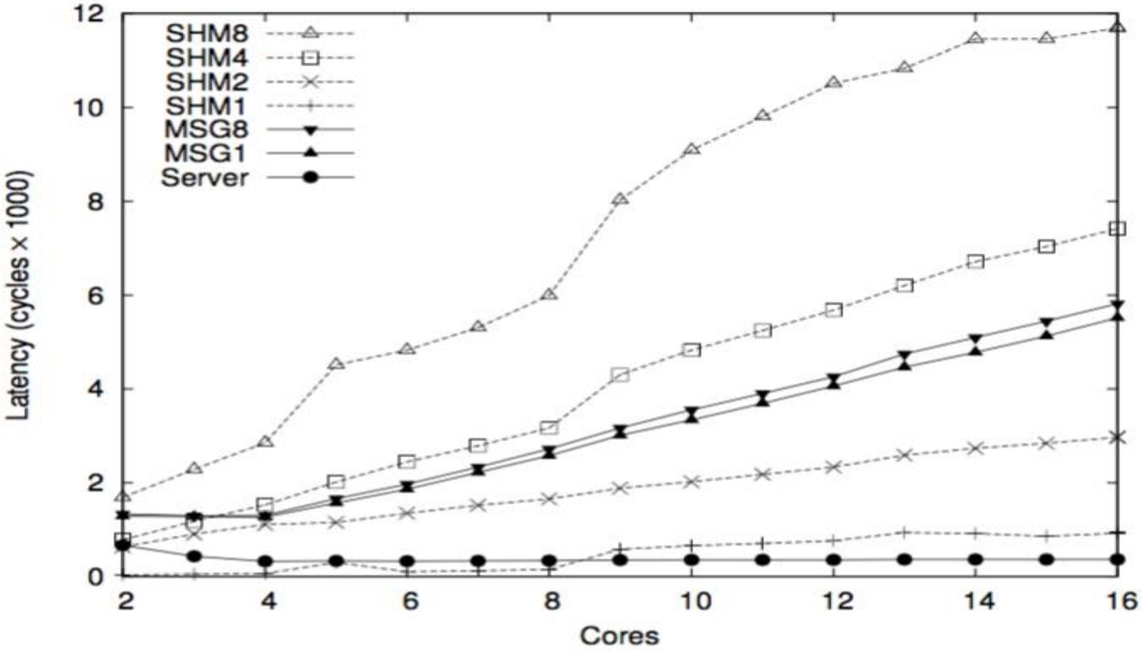


Figure 3: Comparison of the cost of updating shared state using shared memory and message passing.

But wait! Isn't message passing slower than Shared Memory? (cont.)

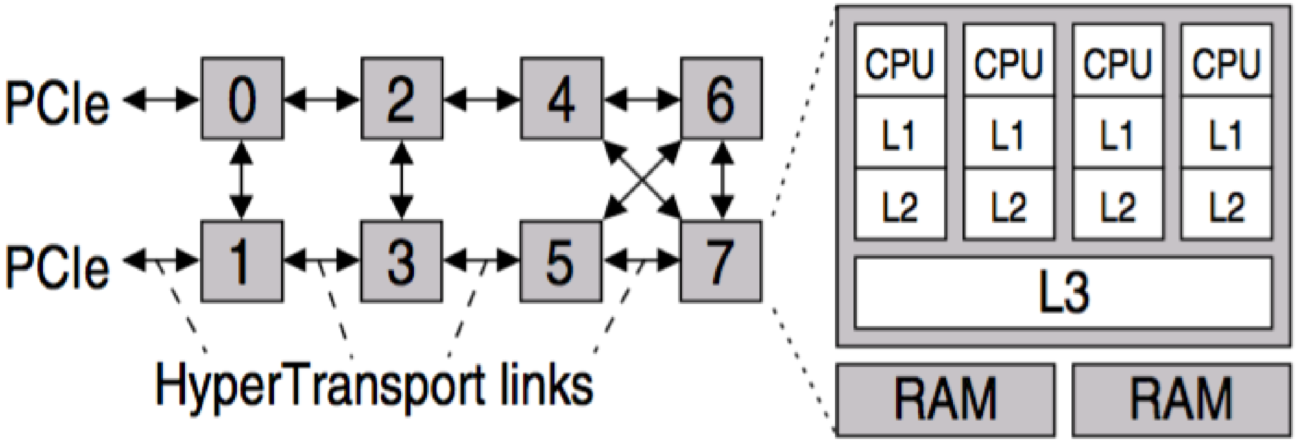


Figure 2: Node layout of an 8×4-core AMD system

The MultiKernel Model

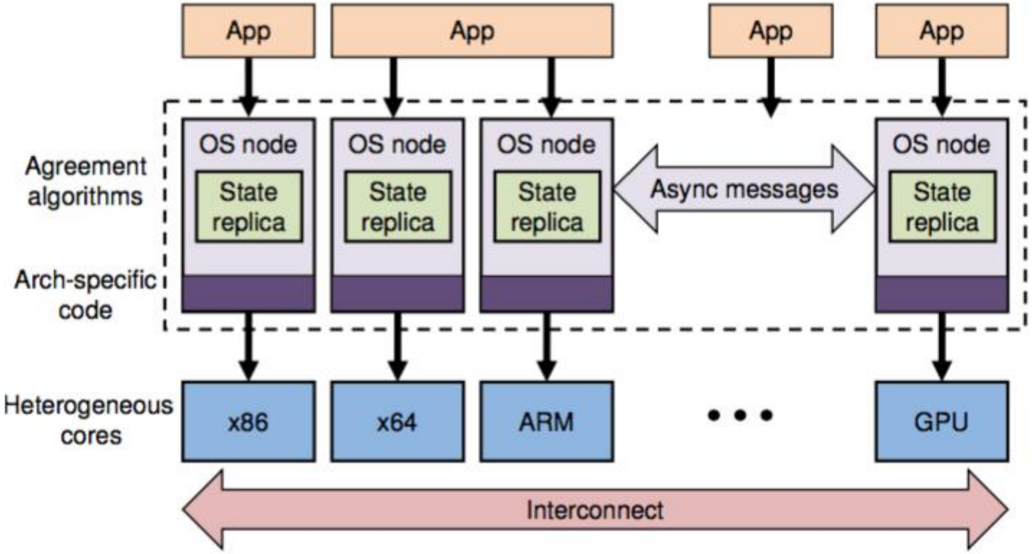


Figure 1: The multikernel model.

Make inter-core communication explicit

- All inter-core communication is performed using explicit messages
- No shared memory between cores aside from the memory used for messaging channels
- Explicit communication allows the OS to deploy well-known networking optimizations to make more efficient use of the interconnect

Make OS structure hardware-neutral

- A multikernel separates the OS structure as much as possible from the hardware
- Hardware-independence in a multikernel means that we can isolate the distributed communication algorithms from hardware details
- Enable late binding of both the protocol implementation and message transport

View state as replicated

- 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

In practice

- Model represents an ideal which may not be fully realizable
- 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

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

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 nonpremptable
- Serially processes events in the 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
- Collectively coordinate consistency of replicated data structures through agreement protocols
- Responsible for IPC setup
- 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

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

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
- All virtual memory management performed entirely by user-level code

System Knowledge Base

- 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

Experiences from Barrelfish 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

Evaluation of Barrelfish

- The testing setup was not accurate
  - making any quantitative conclusions from their benchmarks would be bad
- Barrelfish performs reasonably on contemporary hardware
- Barrelfish can scale well with core count
- Gives authors confidence that multikernel can be a feasible alternative

Evaluation

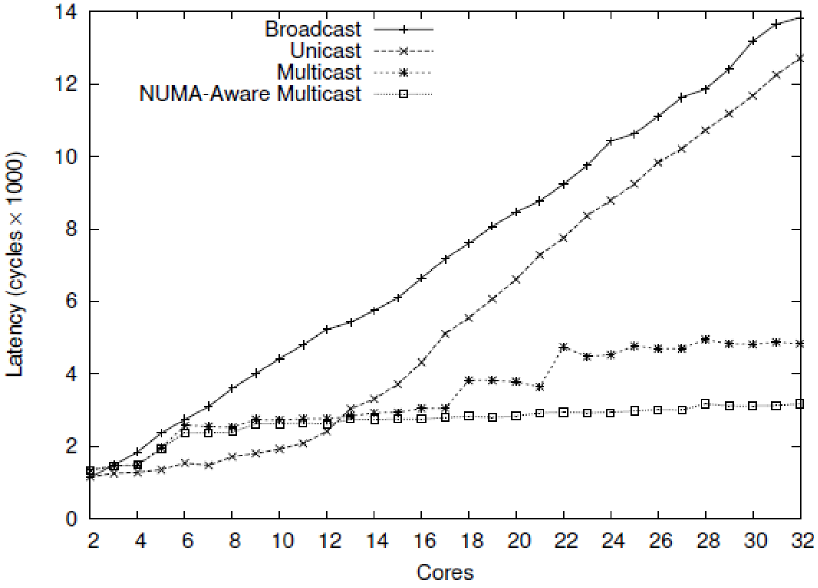


Figure 6: Comparison of TLB shootdown protocols

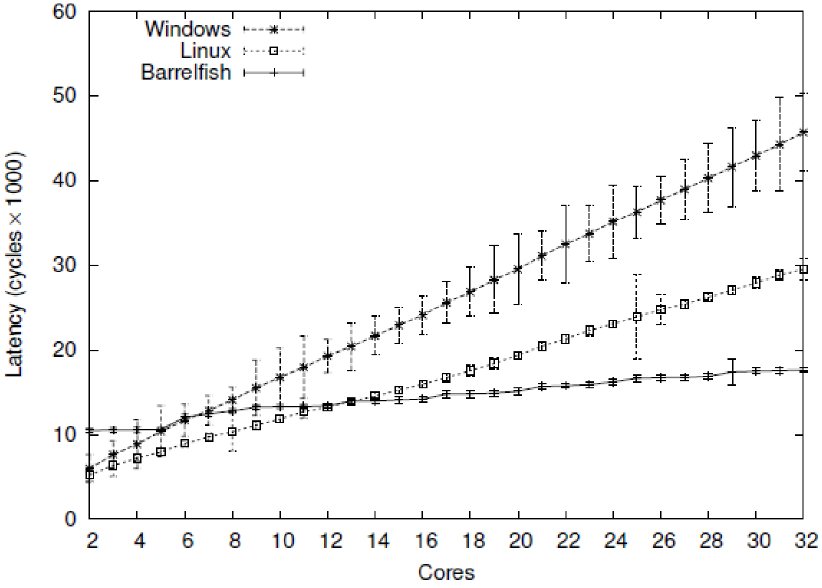


Figure 7: Unmap latency on 8×4-core AMD