UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

The Multikernel: A New OS Architecture for Scalable Multicore Systems

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

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?

- At scale it has been shown 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.)

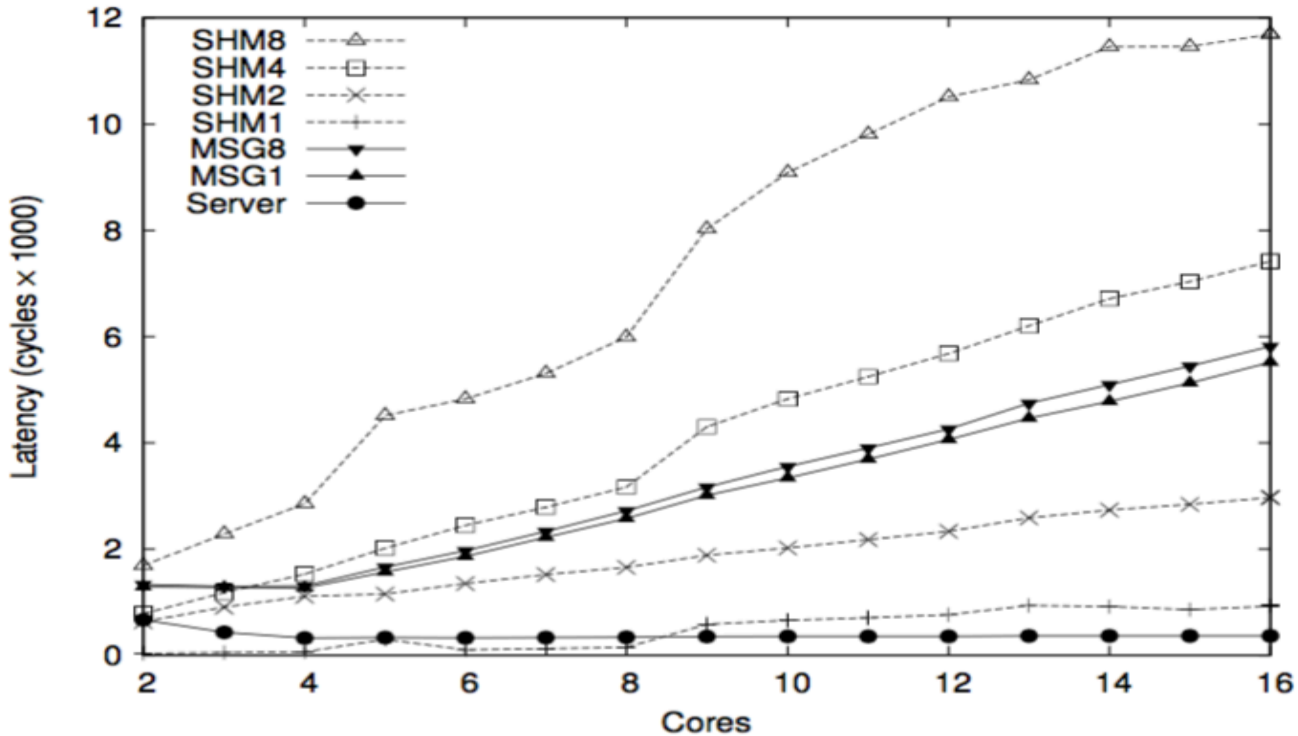


Figure 3: Comparison of the cost of updating shared state using shared memory and message passing.

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

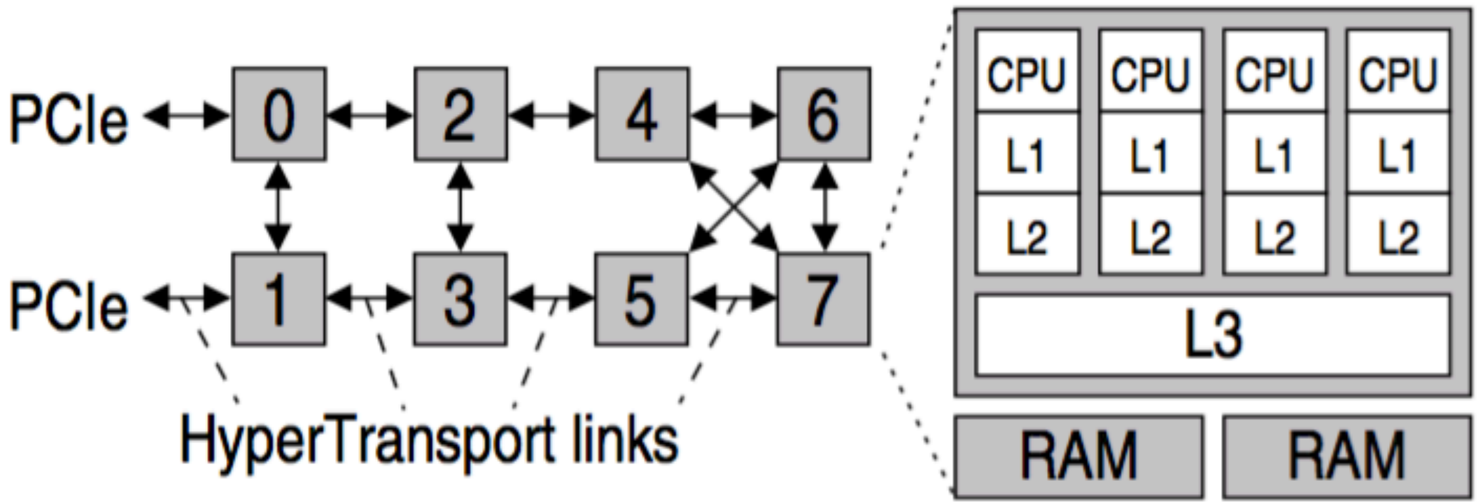


Figure 2: Node layout of an 8×4-core AMD system

The MultiKernel Model

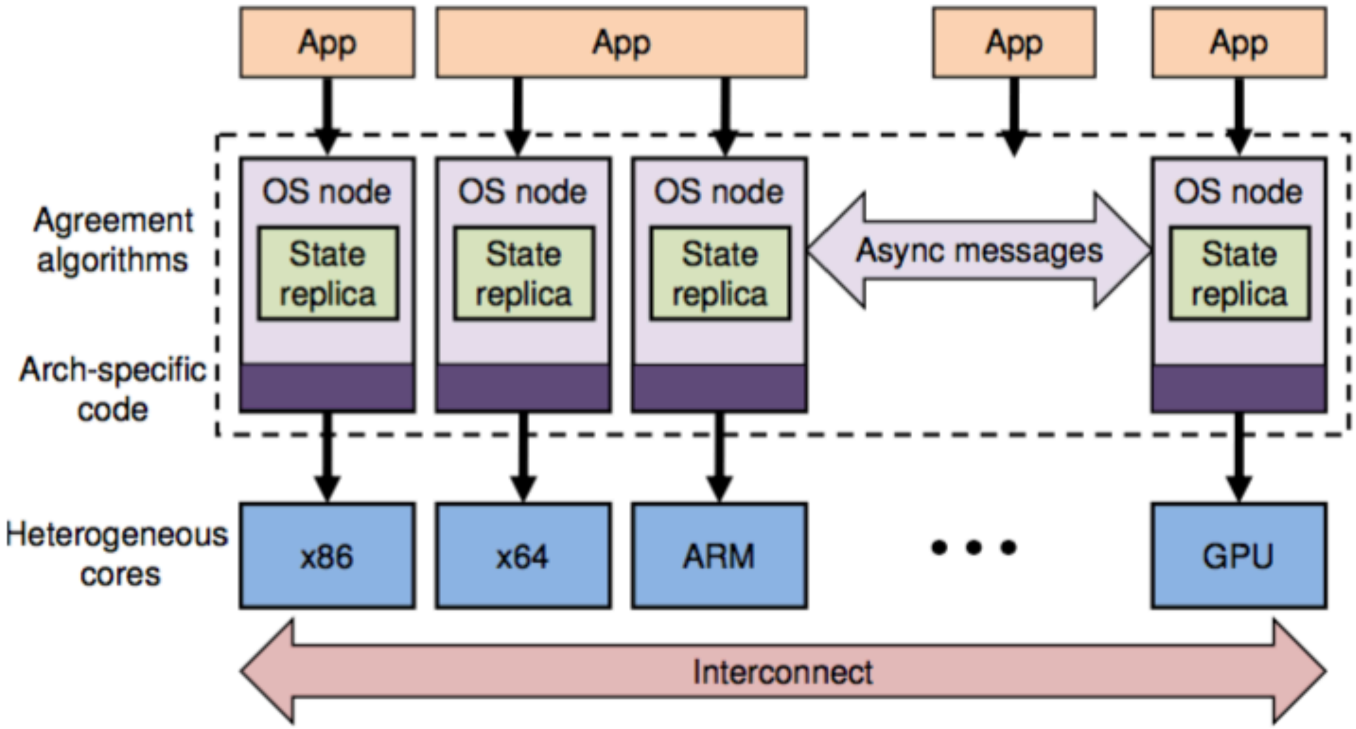


Figure 1: The multikernel model.

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

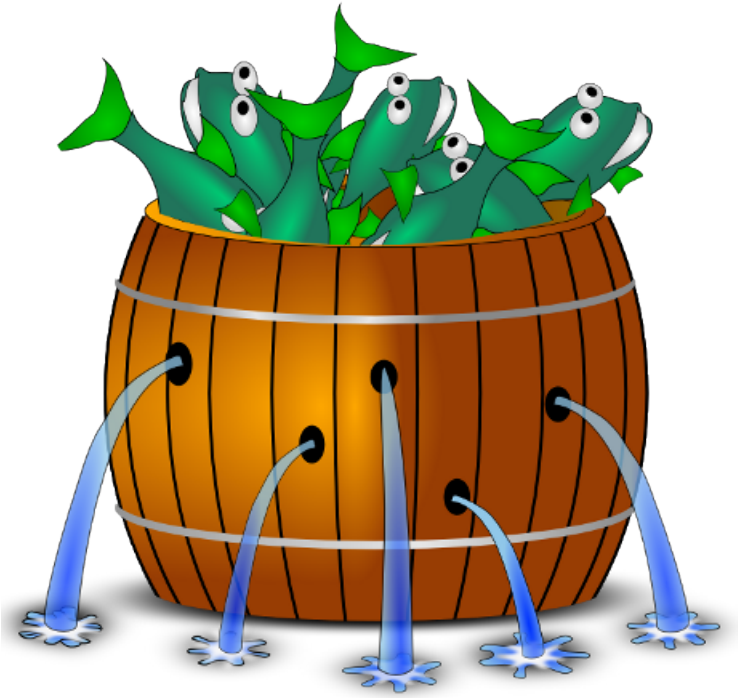
View state as replicated

- 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 idea 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

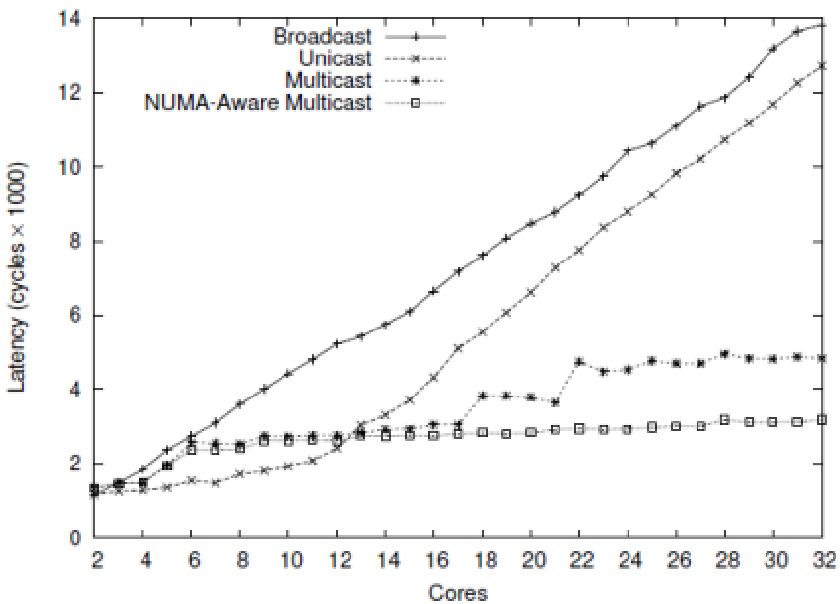


Figure 6: Comparison of TLB shootdown protocols

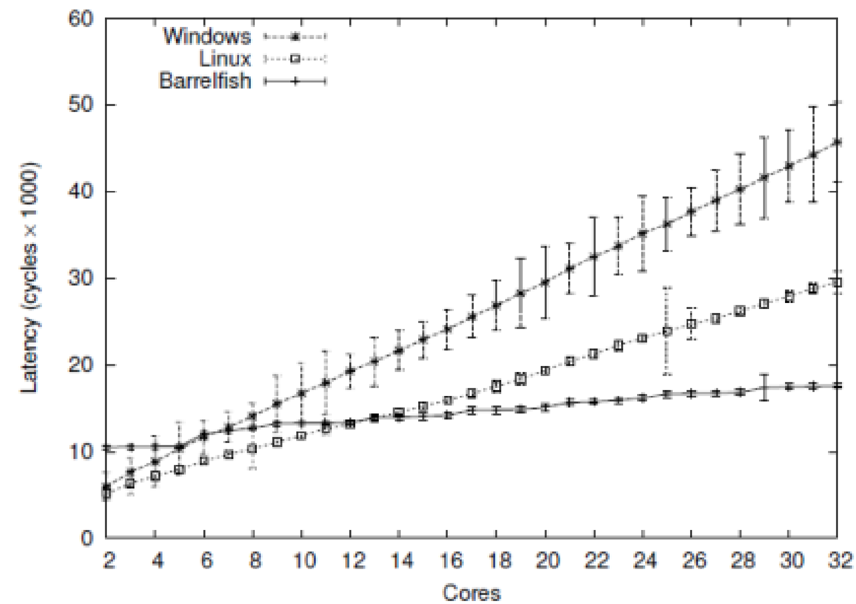


Figure 7: Unmap latency on 8×4-core AMD

An Analysis of Linux Scalability to Many Cores

What are we going to talk about?

- Scalability analysis of 7 system applications running on Linux on a 48-core computer
  - Exim, memcached, Apache, PostgreSQL, gmake, Psearchy and MapReduce
- How can we improve the traditional Linux for better scalability

Amdahl's law

- If α is the fraction of a calculation that is sequential, and $1 - \alpha$ is the fraction that can be parallelized, the maximum speedup that can be achieved by using P processors is given according to Amdahl's Law

Speedup = $\frac{1}{\alpha+\frac{1-\alpha}{P}}$

Introduction

- Popular belief that traditional kernel designs won't scale well on multicore processors
- Can traditional kernel designs be used and implemented in a way that allows applications to scale?

Why Linux? Why these applications?

- Linux has a traditional kernel design and the Linux community has made a great progress in making it scalable
- The chosen applications are designed for parallel execution and stress many major Linux kernel components

How can we decide if Linux is scalable?

- Measure scalability of the applications on a recent Linux kernel
  - 2.6.35-rc5 (July 12,2010)
- Understand and fix scalability problems
- Kernel design is scalable if the changes are modest

Kind of problems

- Linux kernel implementation
- Applications' user-level design
- Applications' use of Linux kernel services

The Applications

- 2 Types of applications
  - Applications that previous work has shown not to scale well on Linux
    - Memcached, Apache and Metis (MapReduce library)
  - Applications that are designed for parallel execution
    - gmake, PosgtreSQL, Exim and Psearchy
- Use synthetic user workloads to cause them to use the kernel intensively
  - Stress the network stack, file name cache, page cache, memory manager, process manager and scheduler

Exim

- Exim is a mail server
- Single master process listens for incoming SMTP connections via TCP
- The master forks a new process for each connection
- Has a good deal of parallelism
- Spends 69% of its time in the kernel on a single core
- Stresses process creation and small file creation and deletion

memcached – Object cache

- In-memory key-value store used to improve web application performance
- Has key-value hash table protected by internal lock
- Stresses the network stack, spending 80% of its time processing packets in the kernel at one core

Apache – Web server

- Popular web server
- Single instance listening on port 80.
- One process per core – each process has a thread pool to service connections
- On a single core, a process spends 60% of the time in the kernel
- Stresses network stack and the file system

PostgreSQL

- Popular open source SQL database
- Makes extensive internal use of shared data structures and synchronization
- Stores database tables as regular files accessed concurrently by all processes
- For read-only workload, it spends 1.5% of the time in the kernel with one core, and 82% with 48 cores

gmake

- Implementation of the standard make utility that supports executing independent build rules concurrently
  - Unofficial default benchmark in the Linux community
- Creates more processes than cores, and reads and writes many files
- Spends 7.6% of the time in the kernel with one core

Psearchy – File indexer

- Parallel version of searchy, a program to index and query web pages
- Version in the article runs searchy indexer on each core, sharing a work queue of input files

Metis - MapReduce

- MapReduce library for single multicore servers
- Allocates large amount of memory to hold temporary tables, stressing the kernel memory allocator
- Spends 3% of the time in the kernel with one core, 16% of the time with 48 cores

Kernel Optimizations

- Many of the bottlenecks are common to multiple applications
- The solutions have not been implemented in the standard kernel because the problems are not serious on small-scale SMPs or are masked by I/O delays

Quick intro to Linux file system

- Superblock - The superblock is essentially file system metadata and defines the file system type, size, status, and information about other metadata structures (metadata of metadata)

- Inode - An inode exists in a file system and represents metadata about a file.

- Dentry - A dentry is the glue that holds inodes and files together by relating inode numbers to file names. Dentries also play a role in directory caching which, ideally, keeps the most frequently used files on-hand for faster access. File system traversal is another aspect of the dentry as it maintains a relationship between directories and their files.

- Taken from: http://unix.stackexchange.com/questions/4402/what-is-a-superblock-inode-dentry-and-a-file

Common problems

- The tasks may lock shared data structures, so that increasing the number of cores increases the lock wait time
- The tasks may write a shared memory location, so that increasing the number of cores increases the time spent waiting for the cache coherence protocol

Common problems - cont

- The tasks may compete for space in a limited size shared hardware cache, so that increasing the number of cores increases the cache miss rate
- The tasks may compete for other shared hardware resources such as DRAM interface
- There may be too few tasks to keep all cores busy

Cache related problems

- Many scaling problems are delays caused by cache misses when a core uses data that other core have written
- Sometimes cache coherence related operation take about the same time as loading data from off-chip RAM
- The cache coherence protocol serializes modifications to the same cache line

Multicore packet processing

- The Linux network stack connects different stages of packet processing with queues
  - A received packet typically passes through multiple queues before arriving at per-socket queue
- The performance would be better if each packet, queue and connection be handled by just one core
  - Avoid cache misses and queue locking
- Linux kernels take advantage of network cards with multiple hardware queues

Multicore packet processing (2)

- Transmitting – place outgoing packets on the hardware queue associated with the current core
- Receiving – configure the hardware to enqueue incoming packets matching a particular criteria (source ip and port) on a specific queue
  - Sample outgoing packets and update hardware's flow directing tables to deliver incoming packets from that connection directly to the core