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, AKHILESH SINGHANIA, IN PROCEEDINGS OF THE 22ND ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES (SOSP '09), BIG SKY, MT OCTOBER 2007

PRESENTED BY ATAKAN DIRIK
Overview

- Introduction
- Motivations
- Multikernel Model
- Implementation – The Barrelfish
- Performance Testing
- Conclusion
Introduction

- Change and diversity in computer hardware become a challenge for OS designers
  - Number of cores, caches, interconnect links, IO devices, etc.

- Today’s general purpose OS is not be able to scale fast enough to keep up with the new system designs

- In order to adapt with this changing hardware, treat the computer as networked components using OS architecture ideas from distributed systems.

- Multikernel is a good idea
  - Treating the machine as a network of independent cores
  - No inter-core sharing at the lowest level
  - Moving traditional OS functionality to a distributed system of processes

- Scalability problems for operating systems can be recast by using messages
Motivations

• Increasingly diverse systems
  • Impossibility of optimizing general-purpose OS at design or implementation time for any particular hardware configuration
  • In order to use modern hardware efficiently, Oses such as Window 7 are forced to adopt complex optimizations. (6000 lines of code in 58 files)

• Increasingly diverse cores
  • Cores can vary within a single machine
  • A mix of different kinds of cores becoming popular

• Interconnection (connection between different components)
  • For scalability reasons, message passing hardware replaced the single shared interconnect
  • Communication between hardware components resembles a message passing network
  • System software has to adapt to the inter-core topology
Motivations

Messages vs Shared memory

- Trend is changing from shared memory to message passing
- Messages cost less than shared memory
- When 16 cores are modifying the same data it takes almost 12,000 extra cycles to perform the update.

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

• Cache coherence is not always a solution
  • Hardware cache-coherence protocols will be increasingly expensive because of the growth in the number of cores and complexity of the interconnect
  • Future Oses will either have to handle non-coherent memory or be able to realize substantial performance gains bypassing the cache-coherence protocol
The Multikernel Model

• **Three Design Principles:**
  - Make all inter-core communication explicit
  - Make the Operating system structure hardware-neutral
  - View state as replicated instead of shared

![Diagram](image)

Figure 1: The multikernel model.
The Multikernel Model

• Explicit inter-core communication:
  • All communication is done through explicit messages
  • Use of pipelining and batching
    • Pipelining: Sending a number of requests at once
    • Batching: Bundling a number of requests into one message and processing multiple messages together
The Multikernel Model

- Hardware-neutral Operating System structure
  - Separate the OS from the hardware as much as possible
  - Only 2 aspects that are targeted at machine architectures
    - Interface to hardware devices (CPUs and devices)
    - Message passing mechanisms
  - Messaging abstraction is used to avoid extensive optimizations to achieve scalability
  - Focus on optimization of messaging rather than hardware/cache/memory access
The Multikernel Model

Replicated state:

- Maintain state through replication rather than shared memory
- Replicating data and updating by exchanging messages
  - Improves system scalability
  - Reduces:
    - Load on system interconnect
    - Contention for memory
    - Overhead for synchronization
- Brings data closer to the cores that process it which leads to lowered access latencies.
Implementation

• **Barrelfish:**
  • A substantial prototype operating system structured according to the multikernel model

• **Goals:**
  • Perform as well as or better than existing commodity operating systems on future multicore hardware.
  • Be re-targeted and adapted to different hardware
  • Demonstrate evidence of scalability to large numbers of cores
  • Be able to exploit message passing abstraction to achieve good performance (pipelining and batching messages)
  • Exploit the modularity of the OS to place OS functionality according to hardware topology
Implementation

Figure 5: Barrelich structure
Implementation

• **CPU Drivers**
  • Performs authorization, time-slices user-space processes
  • Shares no data with other cores
    • Completely event driven, single-threaded and nonpreemptable

• **Monitors**
  • Performs all the inter-core coordination
  • Single core, user-space processes and schedulable
  • Keeps replicated data structures consistent
  • Responsible for inter-process communication setup
  • Can put the core to sleep if no work is to be done
Implementation

• Process Structure:
  • Collection of dispatcher objects
  • Communication is done through dispatchers
  • Scheduling done by the local CPU drivers
  • The dispatcher runs a user-level thread scheduler

• Inter-core communication:
  • Most communication done through messages
  • For now cache-coherent memory is used
  • Carefully tailored to the cache-coherence protocol to minimize the number of interconnect messages
  • Uses a user-level remote procedure call between cores:
    • Shared memory used as a channel for communication
    • Sender writes message to cache line
    • Receiver polls on the last word of the cache line to read message
Implementation
Memory Management

- User-level applications and system services might use shared memory across multiple cores

- Allocation of physical memory must be consistent
  - OS code and data is itself stored in the same memory

- All memory management is performed explicitly through system calls
  - Manipulate capabilities that are user level references to kernel objects or regions of memory
  - The CPU driver is only responsible for checking the correctness of manipulation operations
Implementation

Memory Management

• All virtual memory management performed by the user-level code
  • To allocate memory it makes a request for some RAM
    • Retypes the RAM capabilities to page table capabilities
    • Send it to the CPU driver to insert into root page table
    • CPU driver checks the correctness and inserts it
  • However, authors realized that this was a mistake
Implementation

Shared Address Space

• Barrelfish supports the traditional process model of threads sharing a single virtual address space
• Coordination has an effect on 3 OS components:
  • Virtual address space: Hardware page tables are shared among dispatchers or replicated through messages
  • Capabilities: Monitors can send capabilities between cores, guaranteeing that capability is not pending revocation
  • Thread management
    • Thread schedulers exchange messages to
      • Create and unblock threads
      • Move threads between dispatchers (cores)
• Barrelfish only multiplexes dispatchers on each core via CPU driver scheduler
Implementation
Knowledge and Policy Engine

• System Knowledge Base to keep track of hardware
  • Contains information gathered through hardware discovery
    • ACPI tables, PCI buses, CPUID data, URPC latency, Bandwidth..
  • Allows brief expressions of optimization queries to select appropriate message transports
Evaluation

TLB shootdown

• Maintains TLB consistency invalidating entries

• Linux/Windows(IPI) vs Barrelfish (message passing):
  • In Linux/Windows, through IPI, a core sends an interrupt to each core so that each core traps, acks the IPI, invalidates the TLB entry and resumes.
    • It could be disruptive when every core takes the cost of a trap (800 cycles)
  • In Barrelfish,
    • Local monitor broadcasts invalidate messages and waits for a reply
    • Are able to exploit knowledge about the specific hardware platform to achieve very good TLB shootdown performance
Figure 6: Comparison of TLB shutdown protocols
Evaluation

TLB Shootdown

- Allows optimization of messaging mechanism
- Multicast scales much better than unicast and broadcast
- Broadcast: good for AMD/Hypertransport which is a broadcast network
- Unicast: good for small number of cores
- Multicast: good for shared, on-chip L3 cache
- NUMA-Aware Multicast: scales very well by allocating URPC buffers from memory local to the multicast aggregation nodes and sending messages to highest latency first
TLB Comparison

![Graph showing latency comparison between Windows, Linux, and Barrelfish across different core counts.](image)
Figure 9: Compute-bound workloads on 4x4-core AMD (note different scales on y-axes)
Conclusion

• It does not beat Linux in performance, however…

• Barrelfish is more lightweight and has reasonable performance on current hardware

• Good scalability with core count and easy adaptation to use more efficient communication patterns

• Advantages of pipelining and batching of request messages without reconstructing the OS code

• Barrelfish can be a practicable alternative to existing monolithic systems