Multicore

Advanced Operating Systems
(263-3800-00)

Timothy Roscoe

Thursday 25 November 2010

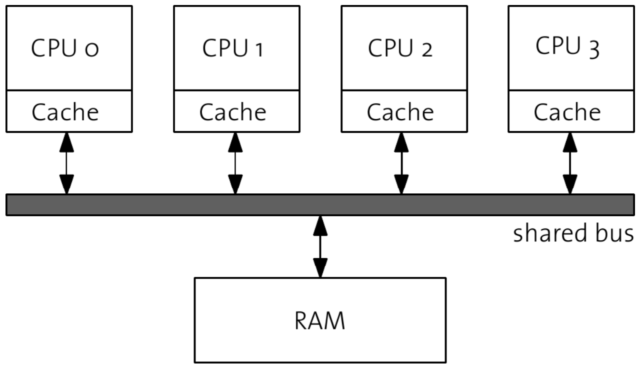
Overview

- Multicore / multiprocessor hardware issues
- General issues for scalable OS design
- Techniques
  – MCS locks
  – Read-Copy-Update
- Research systems exploring different OS structures
  – K42
  – Barrelfish
  – [Disco]

Why multicore?

- Processor designers have hit limits scaling up clock frequencies
  – Energy consumption, heat dissipation
  – no more instruction-level parallelism
- Moore's law holds; what do we do with the extra transistors?
  – put multiple processors (cores) on a chip

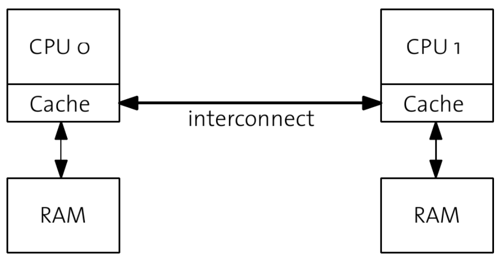
Symmetric multiprocessing (SMP)



Symmetric multiprocessing (SMP)

- All processors are equidistant from RAM
  – e.g.: pre-Nehalem Intel Xeon, many older systems
- Shared bus limits scale:
  – Propagation delays
  – Speed
  – Contention
- **Cache coherence** maintained via hardware support
- Subject to a *memory consistency model*

Non-uniform memory access (NUMA)



- All RAM is accessible, but access to local RAM is faster
- Modern interconnects: HyperTransport, CSI/QuickPath
- Variety of interconnect topologies possible for >2 sockets

Hardware multithreading

- Also: Simultaneous Multithreading (SMT), or HyperThreading
- Core switches between threads to hide effect of stalls
  - E.g. accesses to memory
- Unlike multiple cores, most execution resources are shared
- Two, four, or eight threads per core
- Performance improvements are heavily workload dependent
  - Relatively modest (in the order of 10%)
  - Can be negative
  - Best case: memory-bound transactional (e.g. web services)
- Threads appear to the OS as extra CPUs

Memory consistency models

If one CPU modifies memory, when do others observe it?

- **Strict/Sequential**: reads return the most recently written value
- **Processor/PRAM**: writes from one CPU are seen in order, writes by different CPUs may be reordered
- **Weak**: separate rules for synchronizing accesses (e.g. locks)
  - Synchronising accesses sequentially consistent
  - Synchronising accesses act as a barrier:
    - previous writes completed
    - future read/writes blocked

Memory consistency models

If one CPU modifies memory, when do others observe it?

- **Strict/Sequential**: reads return the most recently written value
- **Processor/PRAM**: writes from one CPU are seen in order, writes by different CPUs may be reordered
- **Weak**: separate rules for synchronizing accesses (e.g. locks)
  - Synchronising accesses sequentially consistent
  - Synchronising accesses act as a barrier:
    - previous writes completed
    - future read/writes blocked

Important to know your hardware!

- x86: processor consistency
- PowerPC: weak consistency

Hardware cache coherence

Example: **MOESI** protocol

- Every cache line is in one of five states:

  Modified: dirty, present only in this cache
  Owned: dirty, present in this cache and possibly others
  Exclusive: clean, present only in this cache
  Shared: present in this cache and possibly others
  Invalid: not present

- May satisfy read from any state
- Fetch to shared or exclusive state
- Write requires modified or exclusive state;
  - if shared, must invalidate other caches
- Owned: line may be transferred without flushing to memory

Hardware cache coherence

Caches must communicate to maintain consistency:

1. Snooping on a shared bus
   - Observe memory accesses of other nodes, broadcast invalidations on bus
   - Used by SMP systems
2. Bus emulation
   - Similar to snooping, but without a shared bus
   - e.g. coherent HyperTransport:
   - Requests and responses (data) are unicast
   - Invalidations and probes are broadcast
   - Limited scalability
   - Latency limited by broadcast
3. Directory-based protocols
   - "Home node" maintains set of nodes that may have line
   - Used by larger multiprocessors, e.g. SCI
   - AMD HTAssist, Intel Beckton QPI

Synchronization

Two ways to synchronize:

1. **Atomic operations** on shared memory
   - e.g.: compare-and-swap, load linked / store conditional, atomic arithmetic
   - Implicit messaging
     (cache-coherence messages, not visible to software)
2. **Interprocessor interrupts** (IPIs)
   - Explicit messaging (invoke interrupt handler on remote CPU)
   - Slow (500+ cycles on Intel), often avoided

- Used for different purposes
  (e.g. locks, vs. asynchronous notification)

Overview

- Multicore / multiprocessor hardware issues
- General issues for scalable OS design
- Techniques
  - MCS locks
  - Read-Copy-Update
- Research systems exploring different OS structures
  - K42
  - Barrelfish
  - [Disco]

Implications for OS design

Assuming cache-coherent shared memory…

- R/W sharing requires communication, limits scaling
- Cache line is the unit of communication and sharing
  - ⇒ Need to avoid shared data
  - ⇒ Need to be careful about data placement in cache lines
    - False sharing: unrelated data structures share a cache line
    - Cache-line bouncing: when shared R/Won many processors
- Physical memory locality also matters for NUMA systems
  - Kernel (or physical memory allocator) needs to know
- Synchronization implies serialization,
  limits parallelism and scalability

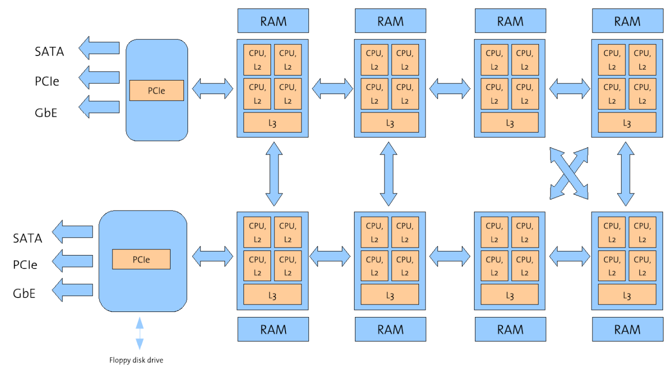
How do you build a scalable OS?

- Fine-grained locking of shared data structures
  - Tradeoff increased parallelism against
    higher overhead for synchronization
  - Many instances of structures won't need fine-grained locking;
    can you have both?
- Avoid sharing and increase locality:
  - Use processor-local data structures
  - Pad data to cache-lines
    - ⇒ higher memory consumption,
      worse (uniprocessor) cache utilization
- Schedule tasks to maintain locality
  - Can lead to more complex, slower scheduler

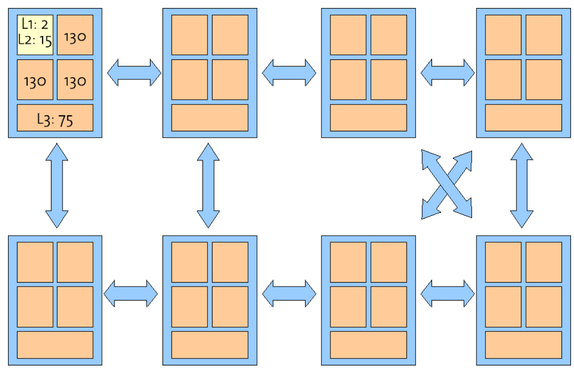
How do you build a scalable OS?

- Fine-grained locking of shared data structures
  - Tradeoff increased parallelism against
    higher overhead for synchronization
  - Many instances of structures won't need fine-grained locking;
    can you have both?
- Avoid sharing and increase locality:
  - Use processor-local data structures
  - Pad data to cache-lines
    - ⇒ higher memory consumption,
      worse (uniprocessor) cache utilization
- Schedule tasks to maintain locality
  - Can lead to more complex, slower scheduler
- **Tradeoff between uniprocessor performance and scalability**

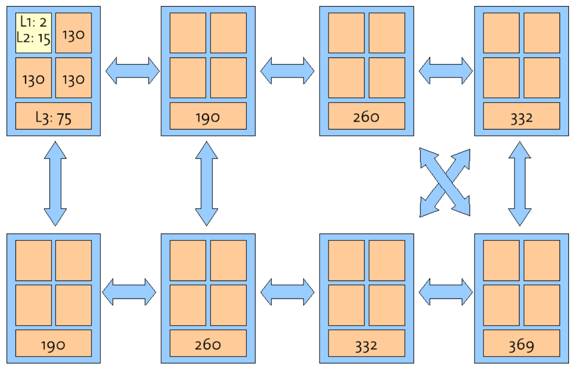
Concrete example: 8-socket 32-core AMD Barcelona



Memory access latency



Memory access latency



Overview

- Multicore / multiprocessor hardware issues
- General issues for scalable OS design
- Techniques
  - MCS locks
  - Read-Copy-Update
- Research systems exploring different OS structures
  - K42
  - Barrelfish
  - [Disco]

MCS locks

[Mellor-Crummey and Scott, 1991]

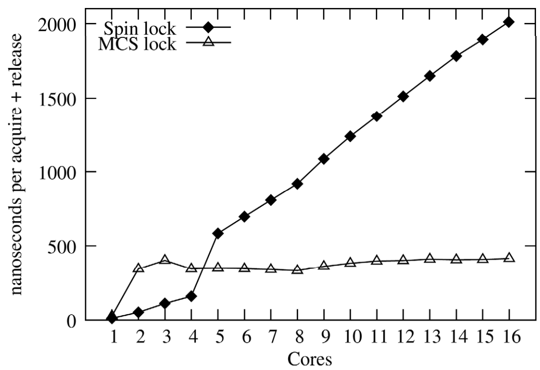
- Cross-CPU locks are typically implemented as a spinlock
  - Processors repeatedly try to atomically update shared value
- **Problem**: cache line containing lock becomes a hot spot
  - Continuously invalidated as every processor tries to acquire it
  - Dominates interconnect traffic
- **Solution**: When acquiring, a processor enqueues itself on a list of waiting processors, and spins on its own entry in the list
- When releasing, only the next processor is awakened

MCS lock pseudocode

```
type qnode = record
         next : ^qnode
         locked : Boolean
type lock = ^qnode

// parameter I, below, points to a qnode record allocated in
// shared memory locally-accessible to the invoking processor

procedure acquire_lock (L : ^lock, I: ^qnode)
         I->next := nil
         predecessor : ^qnode := fetch_and_store (L, I)
         if predecessor != nil // queue was non-empty
                  I->locked := true
                  predecessor->next := I
                  repeat while I->locked // spin

procedure release_lock (L : ^lock, I : ^qnode)
         if I->next = nil // no known successor
                  if compare_and_swap (L, I, nil)
                           return // CAS returns true iff it swapped
                  repeat while I->next = nil // spin
         I->next->locked := false
```

MCS lock performance

4x4-core AMD Opteron



[Boyd-Wickizer et al., 2008]

Read-copy update (RCU)

[McKenney and Slingwine, 1998]

- Mechanism to avoid existence locks on R/W shared data
- In Linux from 2.6 (originated in DYNIX/ptx, K42, ...)
- Readers access a data structure without obtaining a lock
- To write (update) the structure:
  1. Copy data
  2. Modify copy
  3. Update reference to point to modified version
  4. Wait for all previous readers to complete
     - Relies on OS-specific mechanism to detect quiescence (e.g. context switch in Linux, generation count in K42)
  5. Destroy/reclaim previous version

Overview

- Multicore / multiprocessor hardware issues
- General issues for scalable OS design
- Techniques
  - MCS locks
  - Read-Copy-Update
- Research systems exploring different OS structures
  - K42
  - Barrelfish
  - [Disco]

K42

- OS for cache-coherent NUMA systems
- IBM Research, 1997–2006ish
- Successor of Tornado and Hurricane systems (University of Toronto)
- Supports Linux API/ABI
- Aims: high locality, scalability

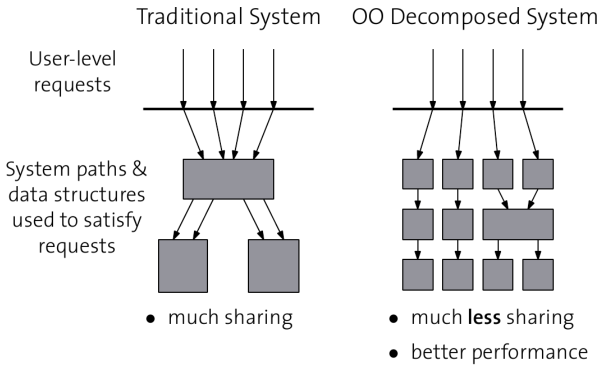
The Answer To . . .

K42

- OS for cache-coherent NUMA systems
- IBM Research, 1997–2006ish
- Successor of Tornado and Hurricane systems (University of Toronto)
- Supports Linux API/ABI
- Aims: high locality, scalability
- Heavily object-oriented
  - Resources managed by set of object instances

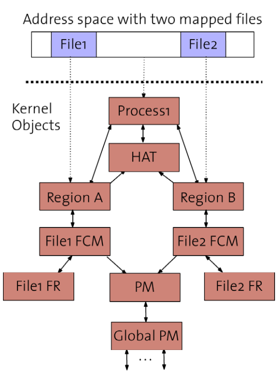
The Answer To . . .

Why use OO in an OS?



[Appavoo, 2005]

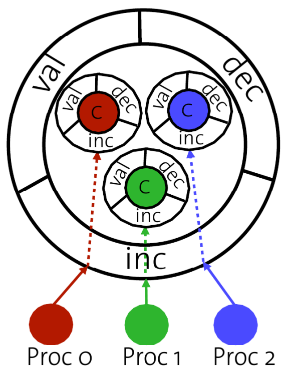
Concrete example: VM objects



- OO decomposition minimizes sharing for unrelated data structures
  - No global locks ⇒ reduced synchronisation
- Clustered objects system limits sharing within an object

Clustered Objects

Example: shared counter



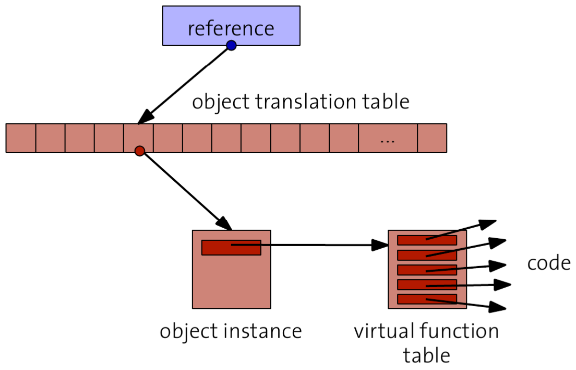
- Object internally decomposed into processor-local representatives
- Same reference on any processor
  - Object system routes invocation to local representative

⇒ Choice of sharing and locking strategy local to each object

- In example, *inc* and *dec* are local; only *val* needs to communicate

Clustered objects

Implementation using processor-local **object translation table**:

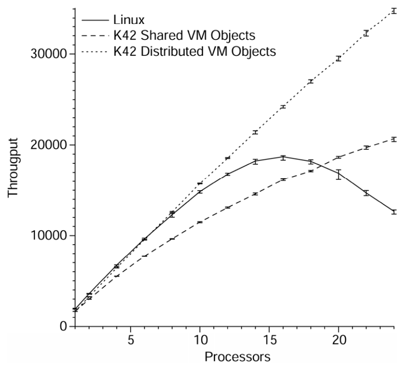


Applying clustered objects

- Distributed versions of core memory-management objects
  - Separate data into local (“rep”) and global (“root”) structures
  - Fast-paths access only per-processor or read-mostly data
  - Fine-grained locking for global data where necessary
  - Cache-line padding of shared data structures
- Other K42 features that help scalability:
  - Deferred deletion (RCU)
  - NUMA-aware memory allocator

Results for SDET benchmark

NB: Linux is version 2.4.19

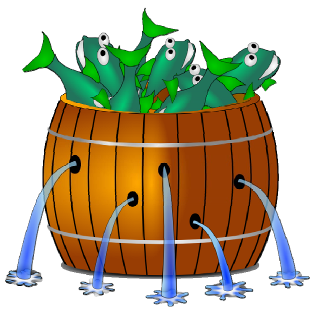


K42 Principles/Lessons

- Focus on **locality**, not concurrency, to achieve scalability
- Adopt distributed component model to enable consistent construction of locality-tuned components
- Support distribution within an OO encapsulation boundary:
  - eases complexity
  - permits controlled/manageable introduction of localized data structures

Barrelfish

- Joint project of ETH Systems Group and Microsoft Research
- We’re exploring how to structure an OS to:
  - scale to many processors
  - manage and exploit heterogeneous hardware
  - run a dynamic set of general-purpose applications
  - reduce code complexity to do this
- Barrelfish is:
  - written from scratch
  - open source



The multikernel model

- Rethinking the default structure of an OS:
  - Shared-memory kernel on every core
  - Data structures protected by locks
  - Anything else is a device
- Our approach: structure the OS as a distributed system
- Design principles:
  - 1. Make inter-core communication explicit
  - 2. Make OS structure hardware-neutral
  - 3. View state as replicated

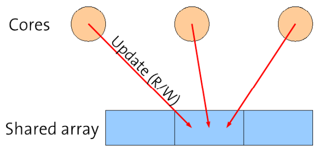
1. Make inter-core communication explicit

- All communication with messages (no shared state)
- Decouples system structure from inter-core communication mechanism
  - Communication patterns explicitly expressed
- Naturally supports heterogeneous cores, non-coherent interconnects (PCIe)
- Better match for future hardware
  - ...with cheap explicit message passing (e.g. Tile64)
  - ...without cache-coherence (e.g. Intel 80-core)
- Allows split-phase operations
  - Decouple requests and responses for concurrency
- We can reason about it

Message passing vs. shared memory: experiment

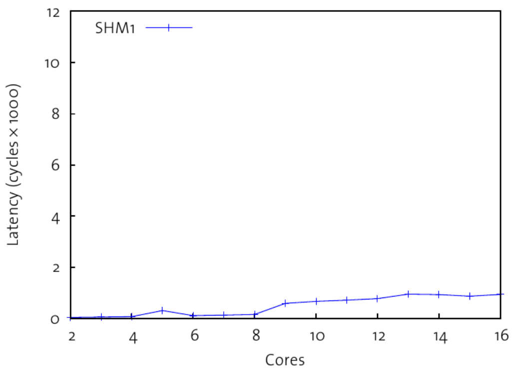
Shared memory (move the data to the operation):

- Each core updates the same memory locations (no locking)
- Cache-coherence protocol migrates modified cache lines
  - Processor stalled while line is fetched or invalidated
  - Limited by latency of interconnect round-trips
  - Performance depends on data size (cache lines) and contention (number of cores)



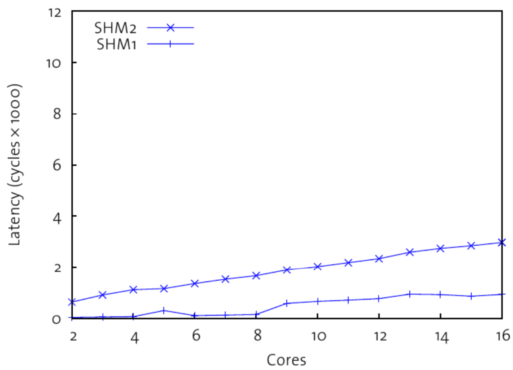
Shared memory results

4x4-core AMD system



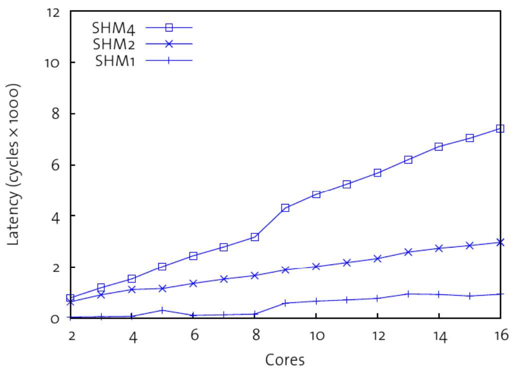
Shared memory results

4x4-core AMD system



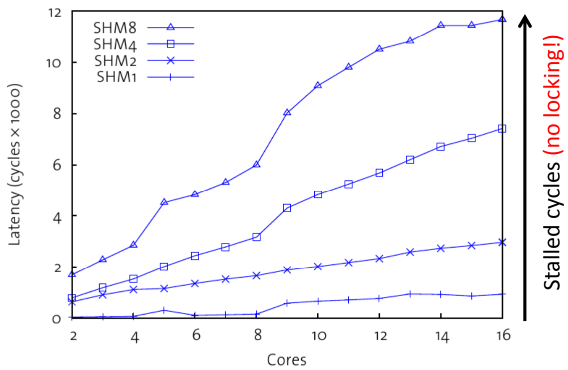
Shared memory results

4x4-core AMD system



Shared memory results

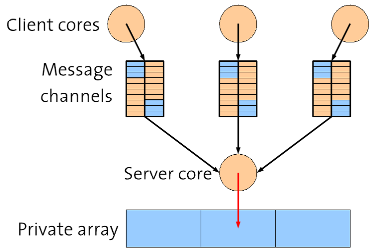
4x4-core AMD system

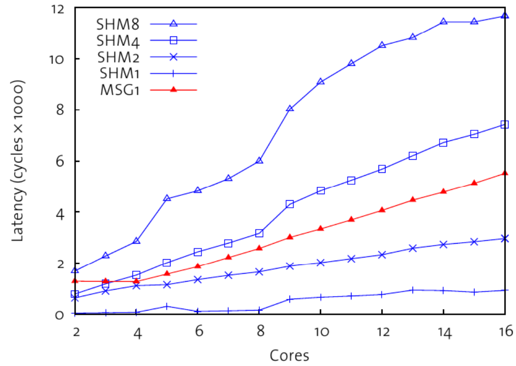


Message passing vs. shared memory: experiment

Message passing (move the operation to the data):

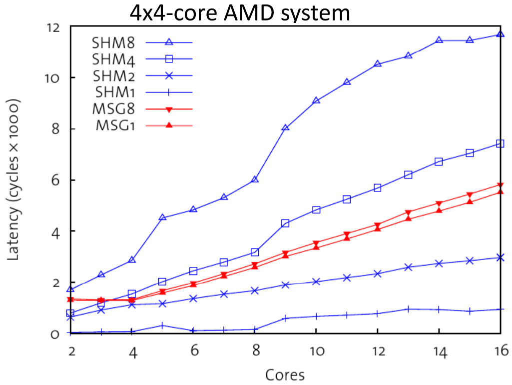
- A single server core updates the memory locations
- Each client core sends RPCs to the server
  - Operation and results described in a single cache line
  - Block while waiting for a response (in this experiment)





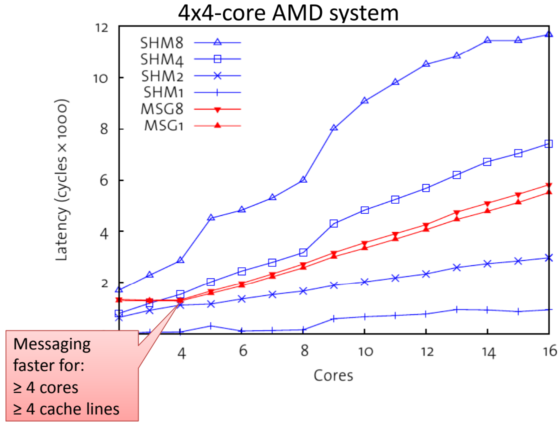
Message passing vs. shared memory: tradeoff

4x4-core AMD system



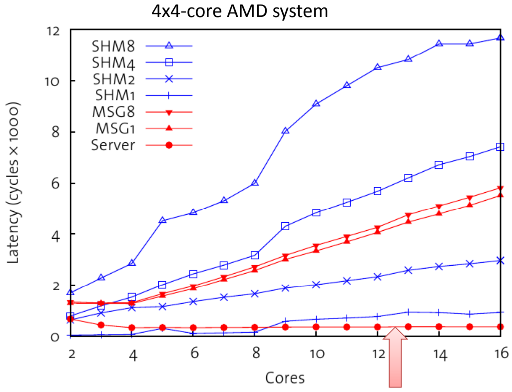
Message passing vs. shared memory: tradeoff

4x4-core AMD system



Message passing vs. shared memory: tradeoff

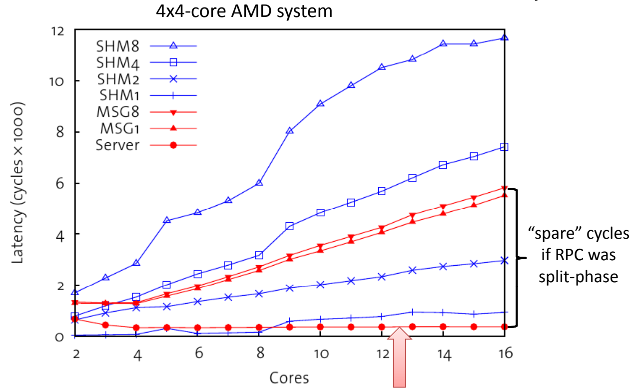
4x4-core AMD system



Actual cost of update at server

Message passing vs. shared memory: tradeoff

4x4-core AMD system



Actual cost of update at server

2. Make OS structure hardware-neutral

- Separate OS structure from hardware
- Only hardware-specific parts:
  - Message transports (highly optimised / specialised)
  - CPU / device drivers

2. Make OS structure hardware-neutral

- Separate OS structure from hardware
- Only hardware-specific parts:
  - Message transports (highly optimised / specialised)
  - CPU / device drivers
- Adaptability to changing performance characteristics
- Late-bind protocol and message transport implementations

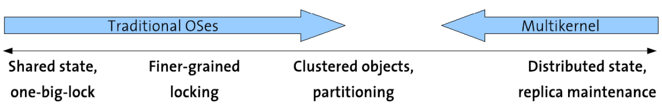
3. View state as replicated

- Potentially-shared state accessed *as if it were a local replica*
  - Scheduler queues, process control blocks, etc.

3. View state as replicated

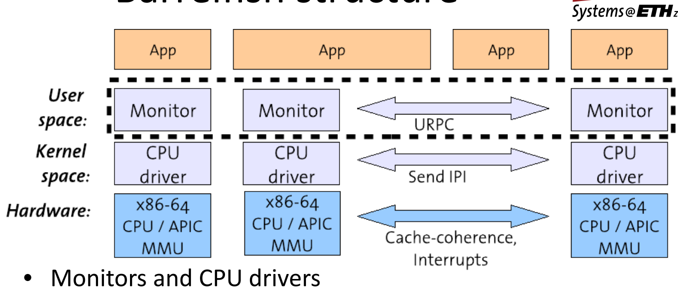
- Potentially-shared state accessed *as if it were a local replica*
  - Scheduler queues, process control blocks, etc.
- Required by message-passing model
- Naturally supports domains that do not share memory
- Naturally supports changes to the set of running cores
  - Hotplug, power management

Replication vs. sharing as default



- In a multikernel, sharing is a local optimisation of replication
- Shared (locked) replica for threads or closely-coupled cores
  - Hidden, local
  - Only when faster, as *decided at runtime*
  - Basic model remains split-phase

Barrelfish structure



- Monitors and CPU drivers
  - CPU driver serially handles traps and exceptions
  - Monitor mediates local operations on global state
- URPC inter-core (shared memory) message transport
  - on *current (cache-coherent) x86 HW*

Case study: Unmap (TLB shootdown)

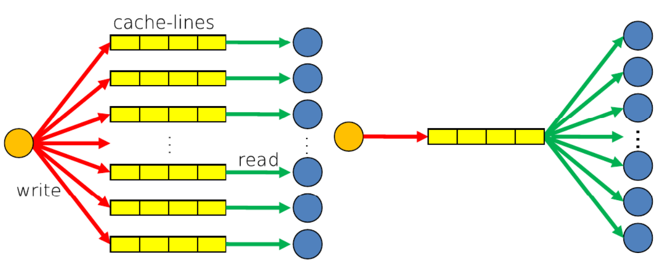
- (Most) HW maintains coherence only of memory caches
- TLBs are left to the OS
- What do you do on unmap or reduced permissions?
- Common operation: process creation (fork), copy-on-write
- Naive approach: global TLB shoot-down
  - IPI to all processors
  - TLB invalidate on all processors
  - Very expensive
- Possible improvements:
  - Track where pages are accessed to avoid global shoot-down
  - Coalesce shoot-downs

Unmap implementation on Barrelfish

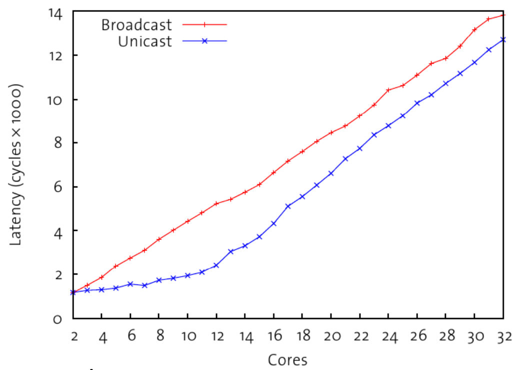
- Send a message to every core with a mapping, wait for all to be acknowledged
- 1. User request to local monitor domain
- 2. Single-phase commit to remote cores
- How to implement communication?

Communication protocols

Unicast **Broadcast**



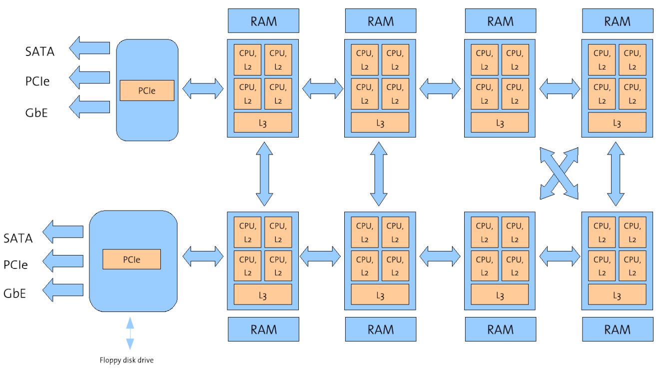
Unmap communication protocols



Raw messaging cost

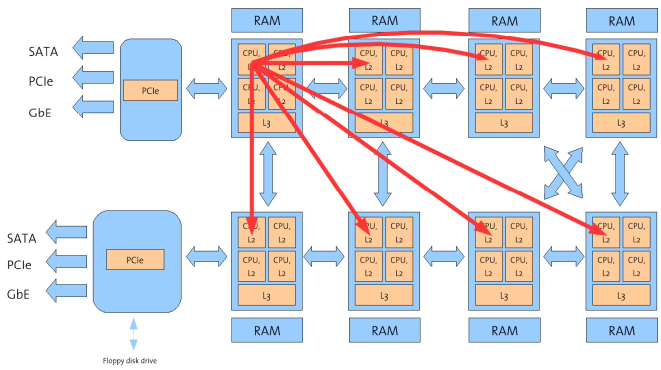
Why use multicast

8x4-core AMD system



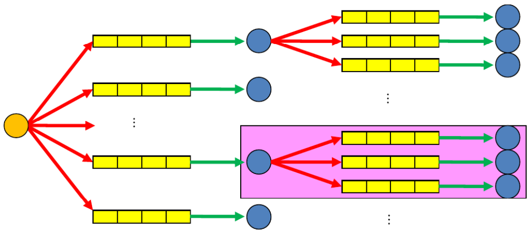
Why use multicast

8x4-core AMD system



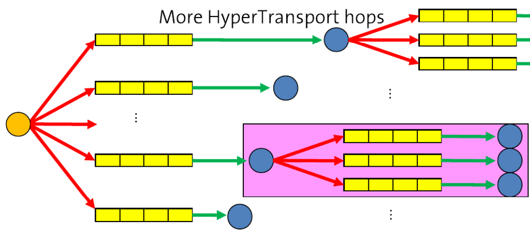
Why use multicast

8x4-core AMD system

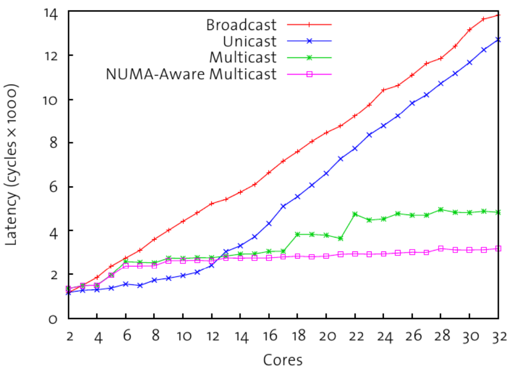


Why use multicast

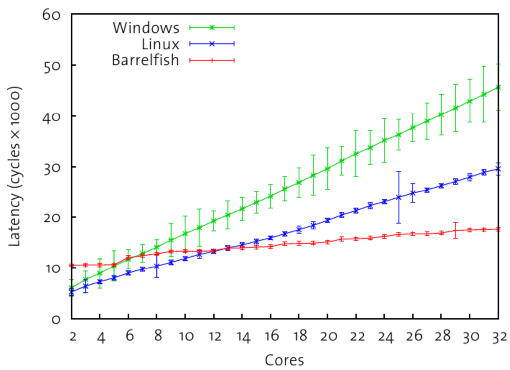
8x4-core AMD system



Raw messaging cost



Unmap latency



Summary

- Different approaches:
  - K42: an OS where it is easy to introduce local data structures
  - Barrelfish: the OS as a message-passing distributed system
- Compare:
  - K42: replicas as optimization of sharing
  - Barrelfish: local sharing as optimization of replicas
- Common ground:
  - Minimizing shared data
  - Increasing locality

References

- Appavoo, J. (2005). **Clustered Objects**. PhD thesis, Department of Computer Science, University of Toronto.
- Appavoo, J., Da Silva, D., Krieger, O., Auslander, M., Ostrowski, M., Rosenburg, B.,Waterland, A., Wisniewski, R.W., Xenidis, J., Stumm, M., and Soares, L. (2007). **Experience distributing objects in an SMMP OS**. *Trans. Comp. Syst., 25(3).*
- Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. (2009a). **The multikernel: A new OS architecture for scalable multicore systems**. In *Proc. 22nd SOSP, Big Sky,MT, USA.*
- Baumann, A., Peter, S., Schüpbach, A., Singhania, A., Roscoe, T., Barham, P., and Isaacs, R. (2009b). **Your computer is already a distributed system. Why isn't your OS**? In *Proc. 12th HotOS, Monte Verità, Switzerland*
- Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L.,Wu, M., hua Dai, Y., Zhang, Y., and Zhang, Z. (2008). **Corey: An operating system for many cores.** In *Proc. 8th OSDI.*
- Bugnion, E., Devine, S., Govil, K., and Rosenblum, M. (1997). **Disco: Running commodity operating systems on scalable multiprocessors**. *Trans. Comp. Syst., 15:412–447.*
- McKenney, P. E. and Slingwine, J. D. (1998). Read-copy update: **Using execution history to solve concurrency problems**. In *Proc. 10th IASTED Int. Conf. Parall. & Distr. Comput. & Syst.*
- Mellor-Crummey, J. M. and Scott, M. L. (1991). **Algorithms for scalable synchronization on shared-memory multiprocessors**. *ACM Trans. Comput. Syst., 9(1):21–65.*

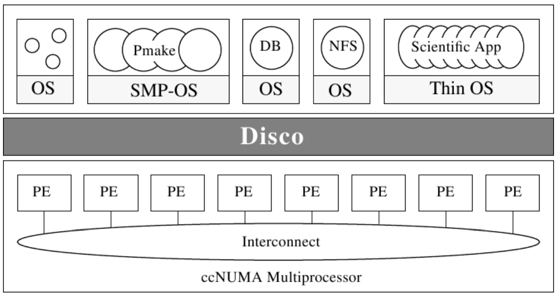
Additional material: Disco

Disco

Running commodity OSes on scalable multiprocessors [Bugnion et al., 1997]

- Context: ca. 1995, large ccNUMA multiprocessors appearing
- Problem: scaling OSes to run efficiently on these was hard
  - Extensive modification of OS required
  - Complexity of OS makes this expensive
  - Availability of software and OSes trailing hardware
- Idea: implement a scalable VMM, run multiple OS instances
- VMM has most of the features of a scalable OS, e.g.:
  - NUMA-aware allocator
  - Page replication, remapping, etc.
- VMM substantially simpler/cheaper to implement
- Run multiple (smaller) OS images, for different applications

Disco architecture



[Bugnion et al., 1997]

Disco Contributions

- First project to revive an old idea: virtualization
  - New way to work around shortcomings of commodity Oses
  - Much of the paper focuses on efficient VM implementation
  - Authors went on to found VMware
- Another interesting (but largely unexplored) idea: programming a single machine as a distributed system
  - Example: parallel make, two configurations:
    1. Run an 8-CPU IRIX instance
    2. Run 8 IRIX VMs on Disco, one with an NFS server
  - Speedup for case 2, despite VM and vNIC overheads