Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

The Barrelfish operating system for heterogeneous multicore systems

Andrew Baumann

Systems Group, ETH Zurich

Teaser

- Problem: Hardware is changing faster than software
  - More cores
  - Increasing heterogeneity
- Idea: build the OS as a distributed system
  - No sharing by default
  - Explicit message-passing between (heterogeneous) cores
  - Support new and existing applications

Outline

Introduction and goals
- Who's involved

Why should we write a new OS?
- Many cores
- Increasing heterogeneity

The Multikernel architecture

Implementation and results

Direct representation of heterogeneity

Status & Conclusion

Outline

- Introduction and goals
  - Who's involved
- Why should we write a new OS?
  - Many cores
  - Increasing heterogeneity
- The Multikernel architecture
- Implementation and results
- Direct representation of heterogeneity
- Status & Conclusion

Barrelfish goals

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 goals

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
  - some library code reused
- open source, BSD licensed
  - expect a release soon

Dramatis personæ

Systems Group, ETH Zurich, Switzerland:

- Andrew Baumann
- Pierre-Evariste Dagand
- Simon Peter
- Jan Rellermeyer
- Timothy Roscoe
- Adrian Schüpbach
- Akhilesh Singhania

Microsoft Research, Cambridge, UK:

- Paul Barham
- Tim Harris
- Rebecca Isaacs

Outline

Introduction and goals
- Who's involved

Why should we write a new OS?
- Many cores
- Increasing heterogeneity

The Multikernel architecture

Implementation and results

Direct representation of heterogeneity

Status & Conclusion

Many cores

- Sharing within the OS is becoming a problem
  - Cache-coherence protocol limits scalability
  - Tornado/K42, Disco, Corey, …
- Prevents effective use of heterogeneous cores

Scaling existing OSes

- Increasingly difficult to scale conventional OSes
  - Removal of dispatcher lock in Win7 changed 6kLOC in 58 files
- Optimisations are specific to hardware platforms
  - Cache hierarchy, consistency model, access costs

Increasing hardware heterogeneity

1. Non-uniformity
2. Core diversity
3. System diversity

Diversity 1: Non-uniformity

The machine looks different from different cores

- Memory hierarchy becomes more complicated
  - Non-uniform memory access (NUMA), plus …
  - Many levels of cache sharing (L2, L3 caches)
- Device access
  - where are my PCIe root complexes?
- Interconnect increasingly looks like a network
  - Tile64, Intel 80-core
  - Larrabee

Diversity 2: Core diversity

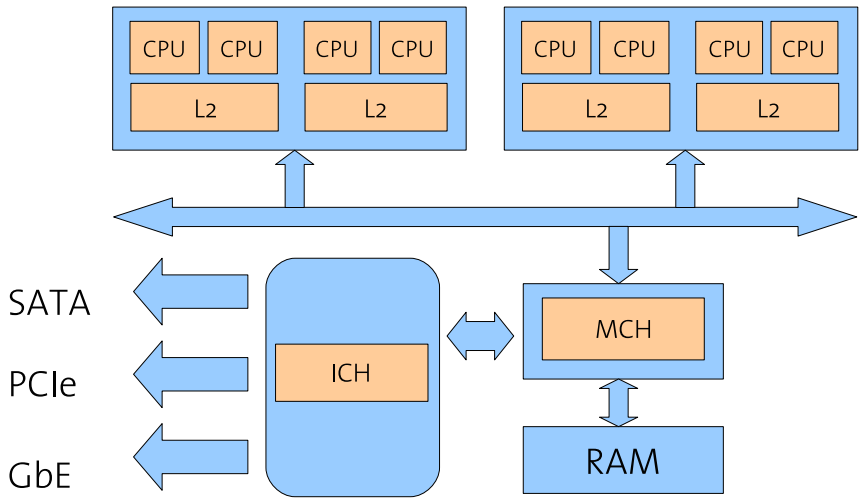
The cores within a box will be diverse

- Architectural differences on a single die:
  - Streaming instructions (SIMD, SSE, etc.)
  - Virtualisation support, power mgmt.
- Within a system
  - Programmable NICs
  - GPUs
  - FPGAs (in CPU sockets)

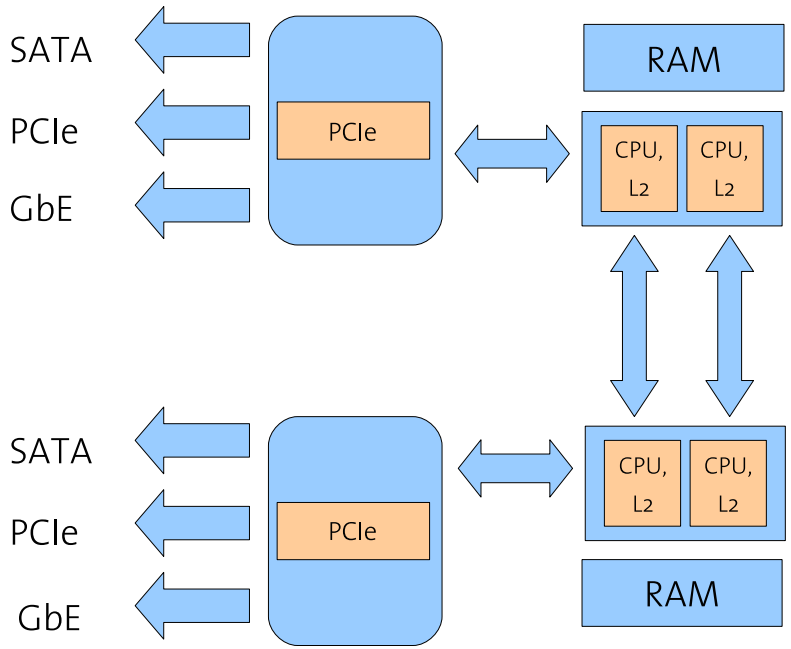
- Already seeing machines with heterogeneous cores
  - Heterogeneity will increase

Diversity 3: System diversity

Machines themselves become more diverse



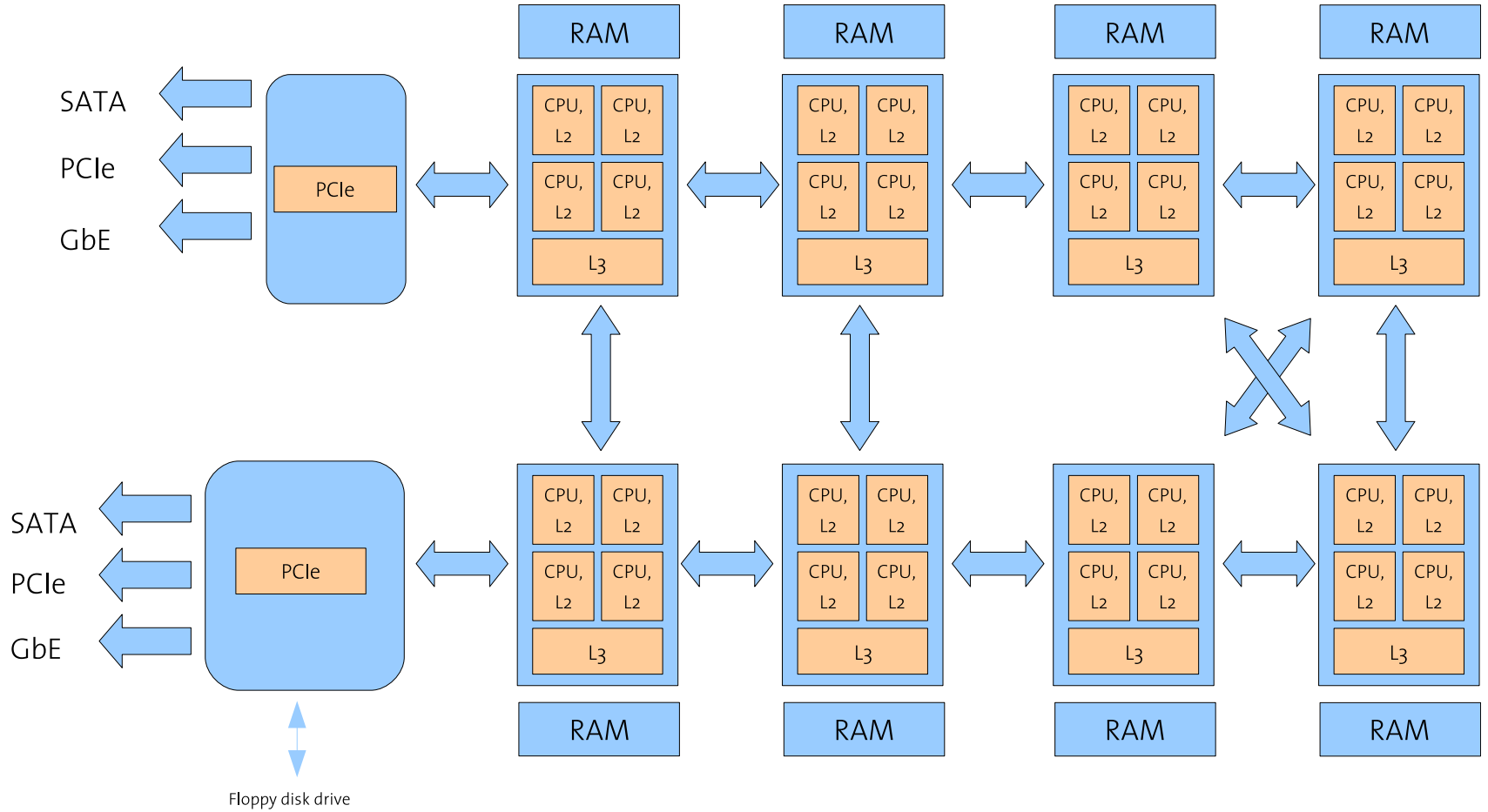
Old 2×4-core Intel system



Old 2×2-core AMD system

Diversity 3: System diversity

Machines themselves become more diverse



8×4-core AMD system

Diversity 3: System diversity

Machines themselves become more diverse

This is new in the mass-market, desktop or server space:

- Specialised code for a certain architecture not possible
  - Unlike with HPC / scientific workloads
- Can't optimise for a particular memory hierarchy
  - If you buy two machines, they may have very different performance tradeoffs.
- Can't manually tune for specific machine

$\Longrightarrow$ system software must adapt at runtime. Hard …

Outline

Introduction and goals
- Who's involved

Why should we write a new OS?
- Many cores
- Increasing heterogeneity

The Multikernel architecture

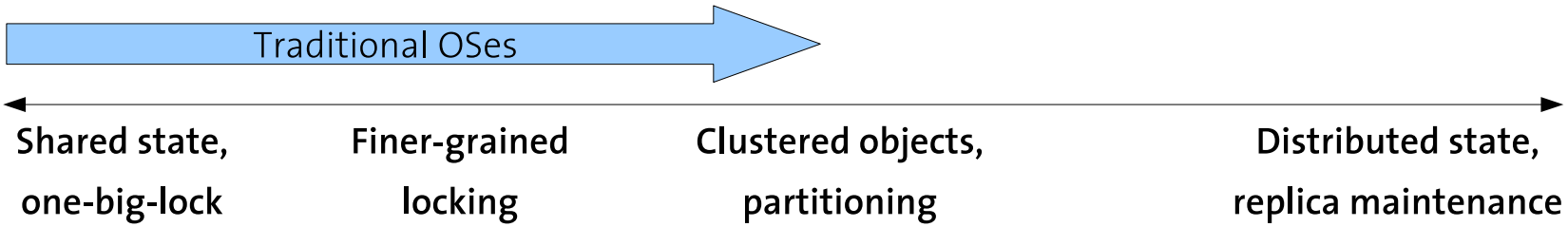
Implementation and results

Direct representation of heterogeneity

Status & Conclusion

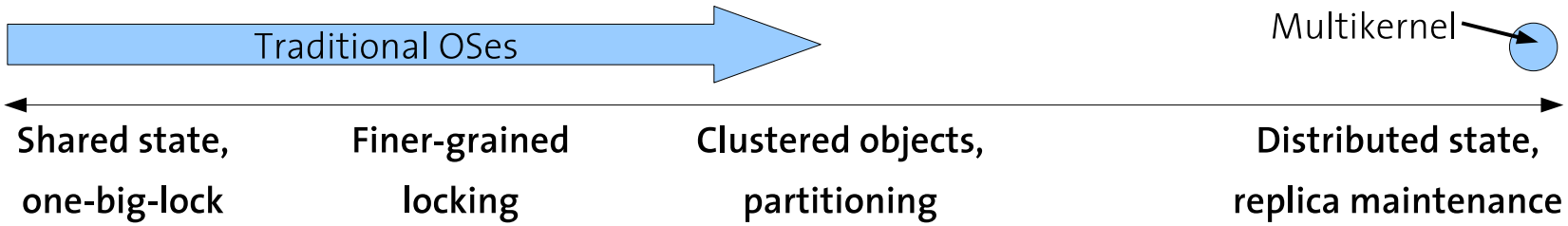
Traditional OS vs Multikernel

- Traditional OSes scale up by:
  - Reducing lock granularity
  - Partitioning state



Traditional OS vs Multikernel

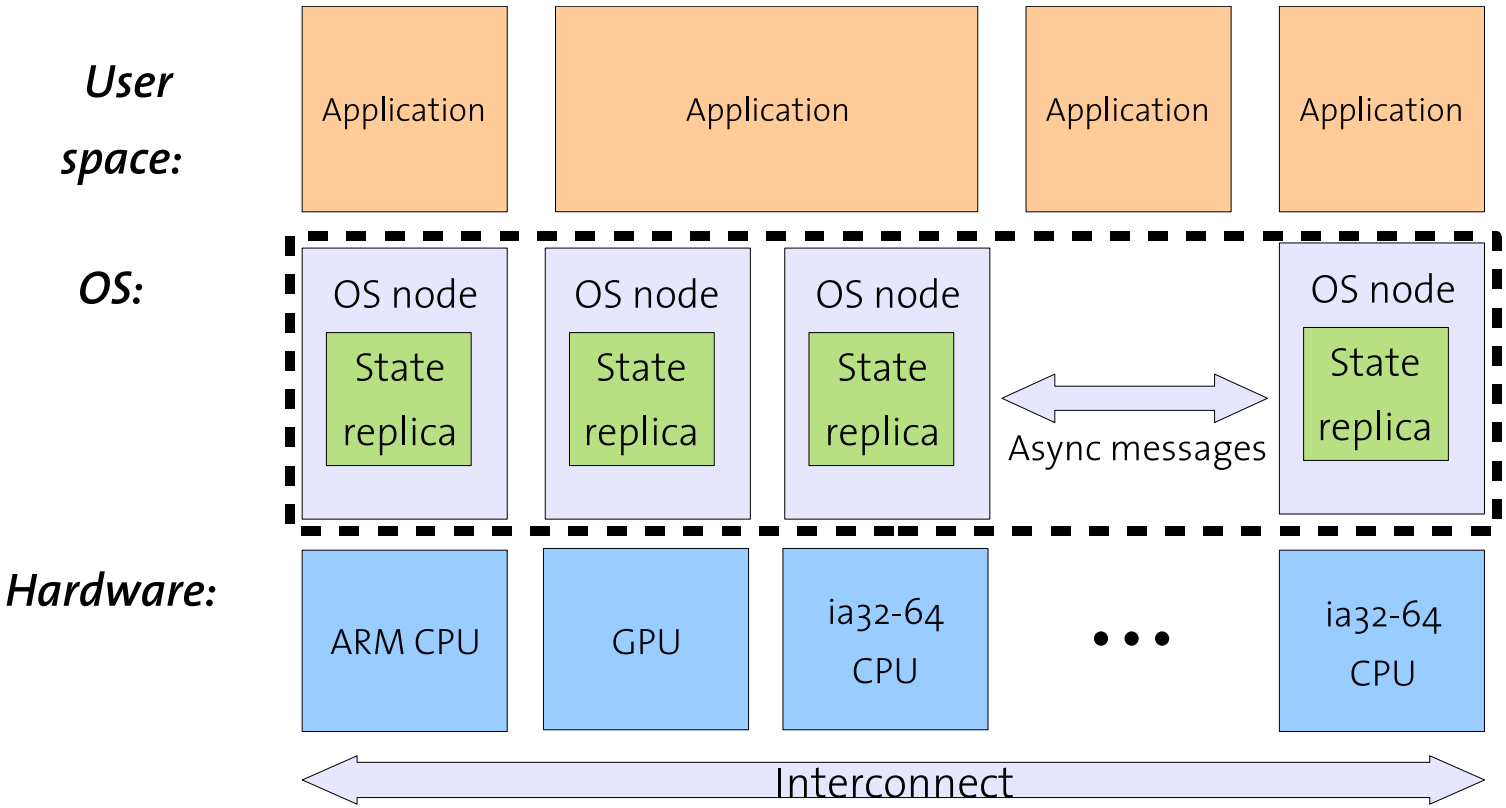
- Traditional OSes scale up by:
  - Reducing lock granularity
  - Partitioning state



Multikernel:

- State partitioned/replicated by default rather than shared
  - Start from the extreme case
  - Cores communicate via message-passing

Multikernel architecture



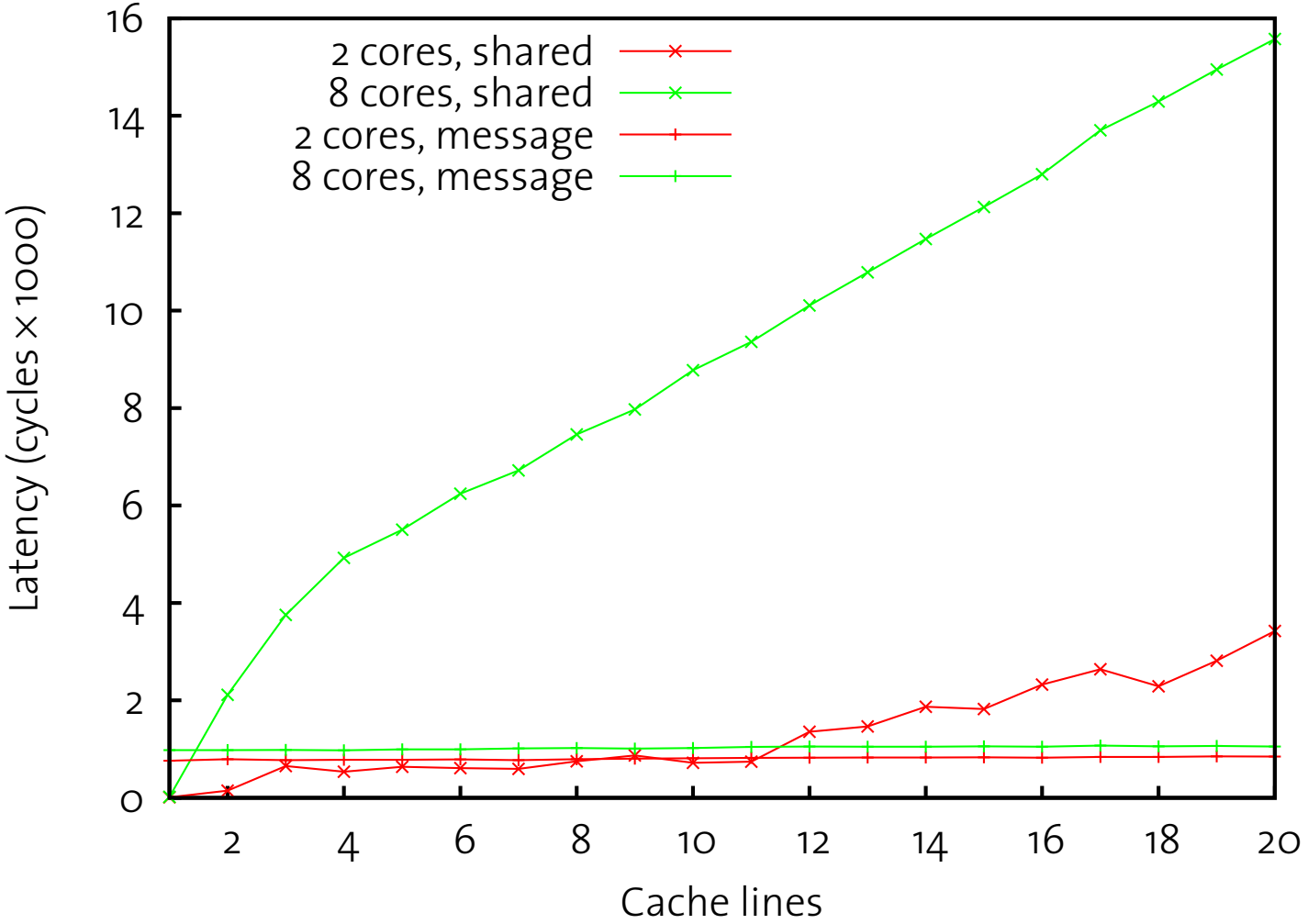
Why message-passing?

- We can reason about it
- Decouples system structure from inter-core communication mechanism
  - Communication patterns explicitly expressed
- Naturally supports heterogeneous cores
- Naturally supports 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)

Message-passing vs. sharing: reduced blocking

- Access remote shared data can be viewed as a blocking RPC
  - Processor stalled while line is fetched or invalidated
  - Limited by latency of interconnect round-trips
- Perf scales with size of data (number of cache lines)
- By sending an explicit RPC (message), we:
  - Send a compact high-level description of the operation
  - Reduce the time spent blocked, waiting for the interconnect
- Potential for more efficient use of interconnect bandwidth

Message-passing vs. sharing: tradeoff



- Shared: Client cores modify shared array (no locking)
- Message: Clients send URPC messages to server core

Change of programming model: why wait?

- In a traditional OS, blocking operations are the norm
- eg: unmap, global TLB shootdown

Idea: change programming model:

- Don't wait: do something else in the meantime
- Make long-running operations split-phase from user space

$\Longrightarrow$ tradeoff latency vs. overhead

Replication

Given no sharing, what do we do with the state?

- Some state naturally partitions
- Other state must be replicated and kept consistent
- How do we maintain consistency?

TLBs (unmap) single-phase commit

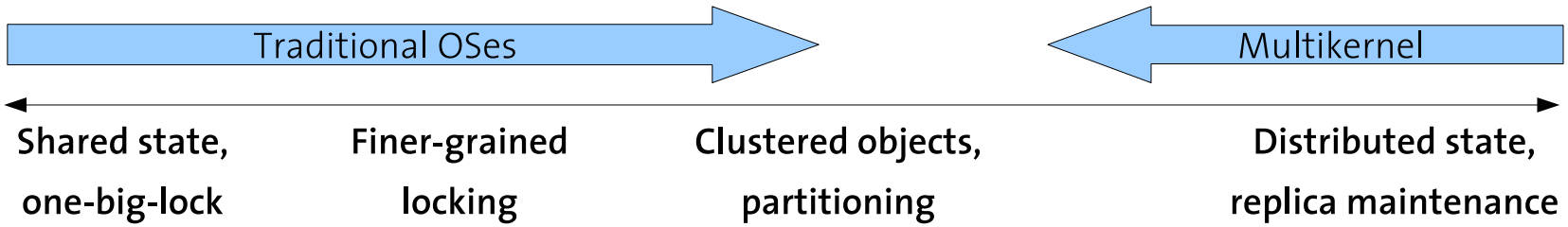
Capabilities (retype/reallocation) two-phase commit

Cores come and go (power management, hotplug) agreement

Optimisation

Sharing as an optimisation in multikernels

- We've replaced shared memory with explicit messaging
- But sharing/locking might be faster between some cores
  - Hyperthreads, or cores with shared L2/3 cache



$\Longrightarrow$ Re-introduce shared memory as optimisation

- Hidden, local
- Only when faster
- Basic model remains split-phase

Outline

Introduction and goals
- Who's involved

Why should we write a new OS?
- Many cores
- Increasing heterogeneity

The Multikernel architecture

Implementation and results

Direct representation of heterogeneity

Status & Conclusion

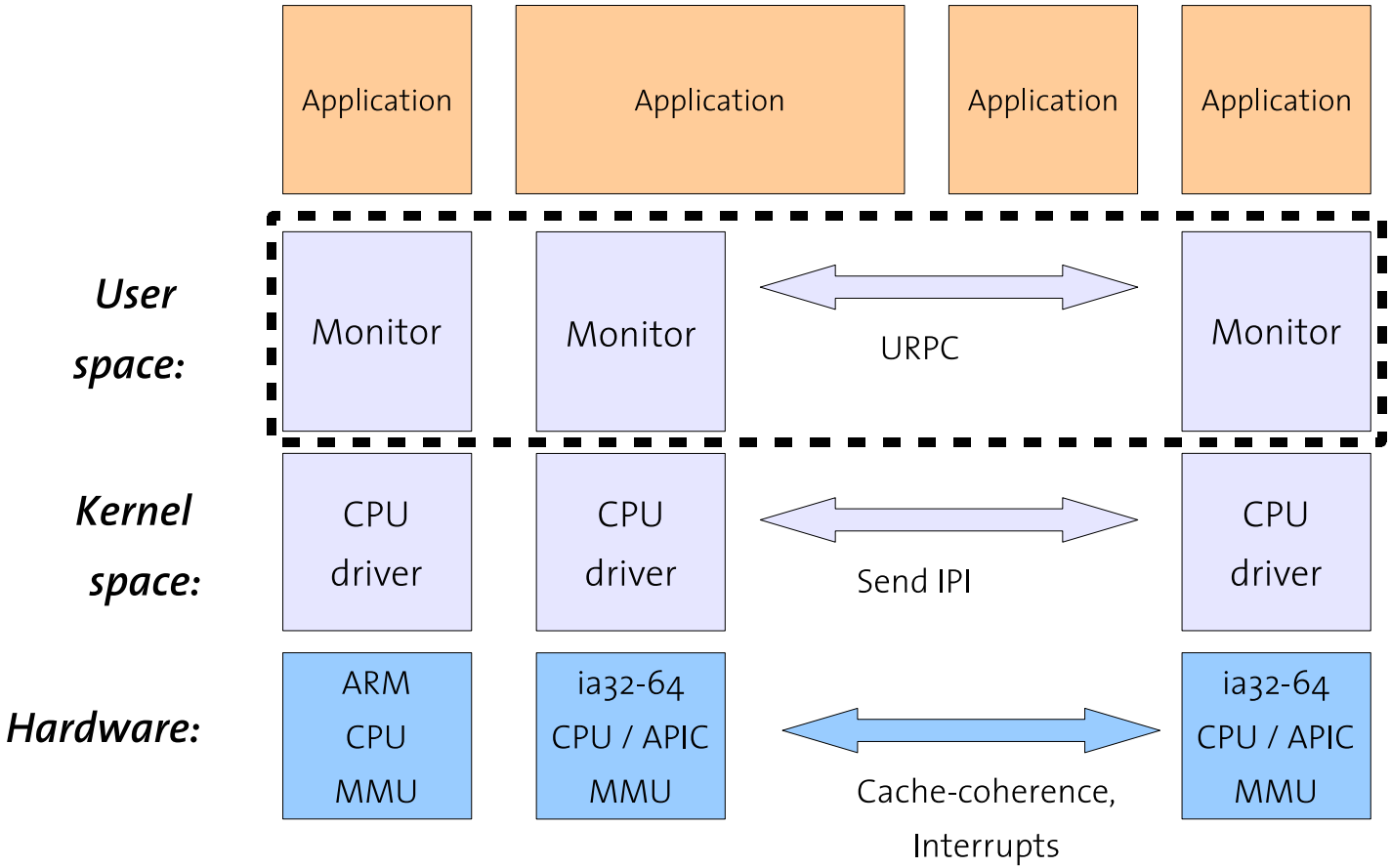
Non-original ideas in Barrelfish

Techniques we liked

- Capabilities for all resource management (seL4)
- Minimise shared state (Tornado, K42, Corey)
- Upcall processor dispatch (Psyche, Sched. Activations, K42)
- Push policy into application domains (Exokernel, Nemesis)
- User-space RPC decoupled from IPIs (URPC)
- Lots of information (Infokernel)
- Single-threaded non-preemptive kernel per core (K42)
- Run drivers in their own domains (μkernels, Xen)
- EDF as per-core CPU scheduler (RBED)
- Specify device registers in a little language (Devil)

Barrelfish structure

Monitors and CPU drivers



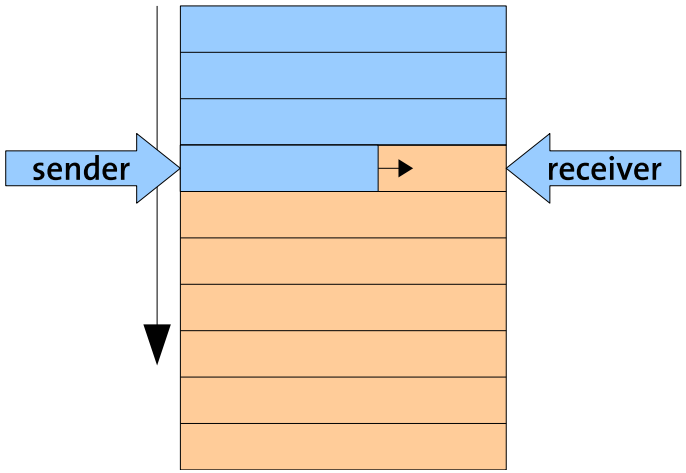
- CPU driver serially handles traps and exceptions
- Monitor mediates local operations on global state

Messaging implementation on current hardware

- Current hardware provides one communcation mechanism: cache-coherent shared memory
- Can we “trick” cache-coherence protocol to send messages?
  - User-level RPC (URPC) [Bershad et al., 1991]

URPC implementation

- Channel is shared-memory ring buffer
- Messages are cache-line sized
- Sender writes message into next line
- Receiver polls on last word
- Marshalling/demarshalling, naming, binding all implemented above



- Slight performance gain (< 5%) possible if sender uses 128-bit SSE instructions
- Buffer placement matters on AMD (NUMA effect)

URPC performance

System	Cache	Latency cycles	Latency ns	Throughput cycles/msg
2×4-core Intel	shared	168	63.2	49
	non-shared	169	63.5	49
2×2-core AMD	shared	450	160.7	145
	non-shared	532	190.0	145
8×4-core AMD	shared	583	291.5	138
	non-shared	623	311.5	137

- Non-shared corresponds to two HyperTransport requests
- Batching/pipelining comes for free

Local vs. remote messaging

2×2-core AMD system

	Latency cycles	Throughput cycles/msg	Cache lines touched Icache	Dcache
URPC	450	145	7	8
LRPC	978	978	33	24
L4 IPC	424	424	25	13

- Barrelfish LRPC could be improved
  - Also invokes user-level thread scheduler
- URPC to a remote core compares favourably with IPC
  - No context switch: TLB unaffected
  - Lower cache impact
  - Higher throughput for pipelined messages

Polling for receive

. . . not as stupid as it sounds

- It's cheap: line is local to receiver until message arrives
  - Memory prefetcher helps
- Some channels need only be polled when awaiting a reply
- If a core is doing something useful, why interrupt it?
  - Tradeoff between timeslicing overhead and message latency

Alternatives to polling

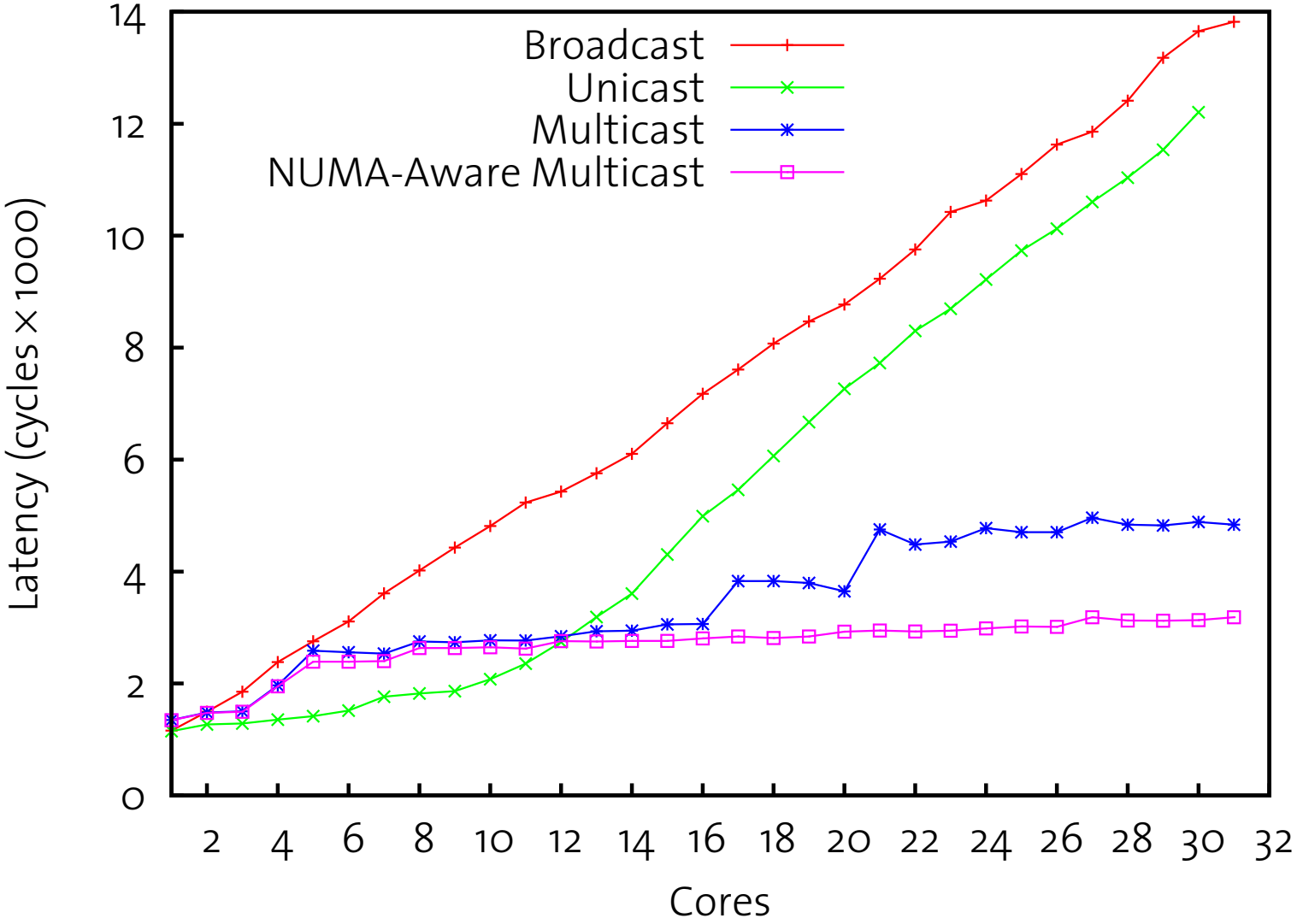
- Alternatives available on current (x86) hardware:
  - IPI: few cycles to send, hundreds to receive
  - MONITOR/MWAIT: core enters low-power state until designated cache line is modified
- General idea:
  - Receiver sends message to indicate they are not polling
  - Sender uses appropriate mechanism to notify receiver core

Unmap (TLB shootdown)

- Send a message to every core with a mapping, wait for all to be acknowledged
- Linux/Windows:
  1. Kernel sends IPIs
  2. Spins on acknowledgement
- Barrelfish:
  1. User request to local monitor
  2. Single-phase commit to remote monitors
- Possible worst-case for a multikernel
- How to implement communication?

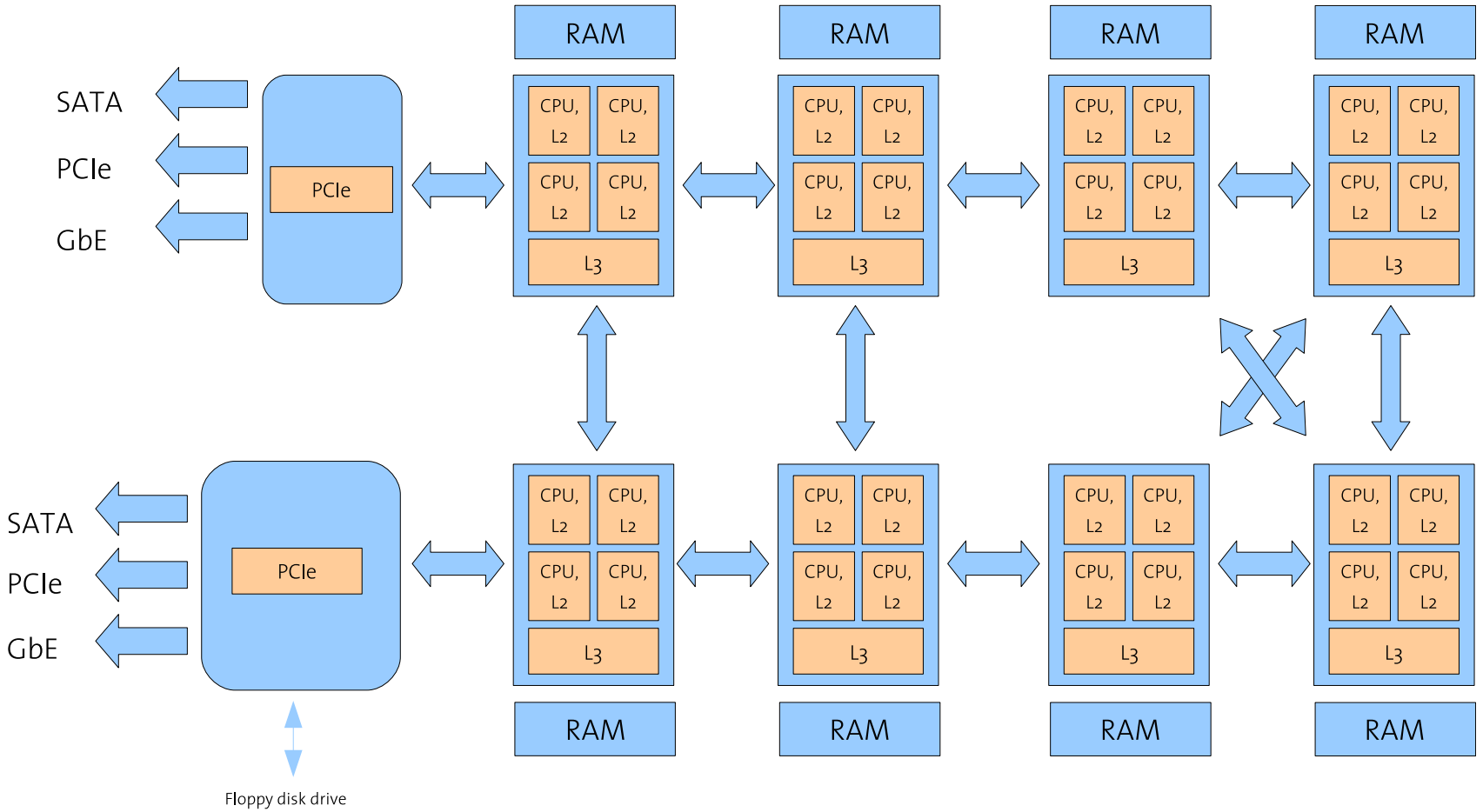
Unmap communication protocols

Raw messaging cost

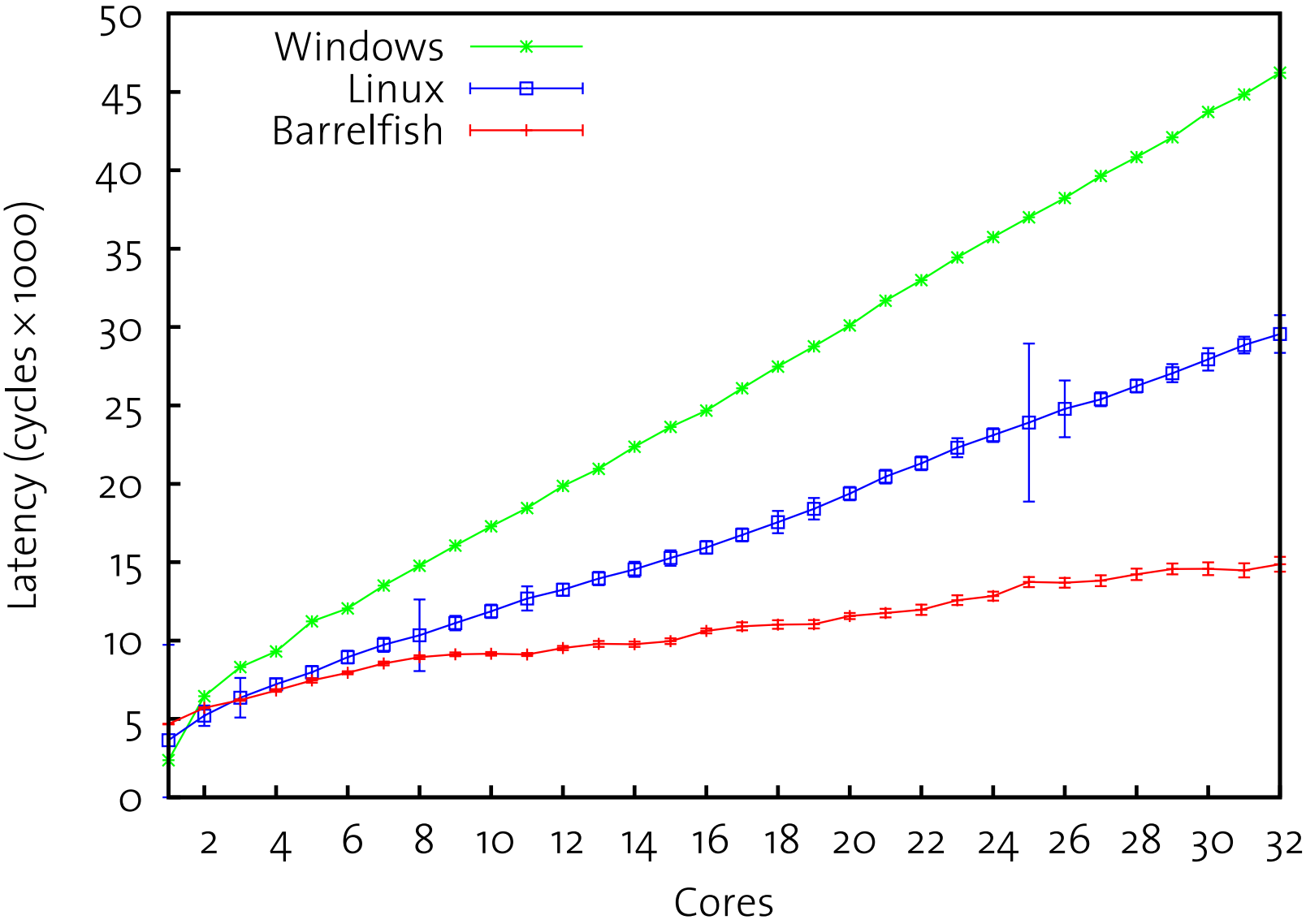


Why multicast?

8×4-core AMD system



Unmap latency



Application-level results

- Shared-memory apps (OpenMP, SPLASH-2) uninteresting
- Network IO: placement on cores matters
  - 887.9 MBit/s vs. 502.7 Mbit/s UDP echo
- Pipelined web server

  - Static: 14180 requests per second vs. 5700 for Apache/Linux
  - Dynamic: ≈1500 requests per second (bottlenecked on SQL)

Outline

Introduction and goals
- Who's involved

Why should we write a new OS?
- Many cores
- Increasing heterogeneity

The Multikernel architecture

Implementation and results

Direct representation of heterogeneity

Status & Conclusion

Managing heterogeneity

- Multikernel architecture handles core diversity
  - Can specialise CPU driver / data structures to cores
- Doesn't deal with heterogeneity in general
- Want to optimise on complex HW representation without affecting fast-paths

Idea: specialise mechanisms,
reason on explicit HW representation for policy

- We deploy a system knowledge base

The system knowledge base

Representing the execution environment

- Representation of hardware and current state in a subset of first-order logic
- Runs as an OS service
- Off the fast-path
- Queried from applications for application level policies
- Used by OS to derive system policies
- Reduces code complexity

Initial implementation: port of the ECL^{i}PS^{e} constraint solver

Populating the SKB

Information sources

- Resource discovery and monitoring
  - Device enumeration
  - ACPI …
- Online measurement and profiling
  - Devices
  - Interconnect links
  - CPU performance counters
  - Application performance and behaviour
- Asserted *a priori* knowledge
  - Data sheets, documentation
  - Device identifiers

SKB example

Uniquely assign IRQ numbers to devices

- Each device supports some IRQ numbers
- Need to find unique allocation

```
device(e1000,...,supported_irqs([1, 3, 4]))
device(SATA,...,supported_irqs([1, 4, 6]))

constrain_irq(L,I) :-
    i(_,supported_irqs(List)) = L,
    I::List.

allocate_irqs(DevIRQ,IRQList) :-
    findall(i(Loc,I),device(_,Loc,_,_,_,I),DevIRQ),
    maplist(constrain_irq,DevIRQ,IRQList),
    alldifferent(IRQList),
    labeling(IRQList).
```

Outline

Introduction and goals
- Who's involved

Why should we write a new OS?
- Many cores
- Increasing heterogeneity

The Multikernel architecture

Implementation and results

Direct representation of heterogeneity

Status & Conclusion

Current status

What's working this week?

- x86-64 CPU/APIC driver, multiple cores
- Capability system & memory management
- Monitor implementation
- Low-level IDC/LRPC/URPC messaging
- High-level OSGi-like component system
- User-space libraries, incl. threads
- Devices: PCI, ACPI, 3 NICs, framebuffer, …
- lwIP, NFS stacks
- SQLite, Python, OpenMP, …
- **currently serving www.barrelfish.org**

Conclusions

1. Treat the machine as a distributed system:
   - Concurrency, communication, heterogeneity
   - Tailor messaging mechanisms and algorithms to the machine
   - Hide sharing as an optimisation

2. Tackle the heterogeneity and complexity head-on:
   - Discover, measure, or just assert it
   - Spend cycles to reason about it

3. Build a real system!