University of Glasgow | School of Computing Science

# Concurrency 1: Hardware Trends and Programming Models for NUMA Systems

Advanced Operating Systems (M)
Lecture 16

Lecture Outline

- Hardware Trends

- Programming NUMA Systems

  - Cache coherence and messaging

- Implications for kernel design

  - The multi-kernel model

  - Example: Barrelfish

Hardware Trends

- Power consumption limits clock rate: we can't cool faster cores

- Instruction level parallelism is limited

- Moore's "law" continues – transistor counts keep growing exponentially
  - Increase in on-chip memory
  - Increase in number of cores
  - Increase in integration ("system on a chip")

- Systems have more cores and other resources, but we see only a limited increase in performance per core
  - Homogeneous cores in NUMA systems
  - Heterogenous designs
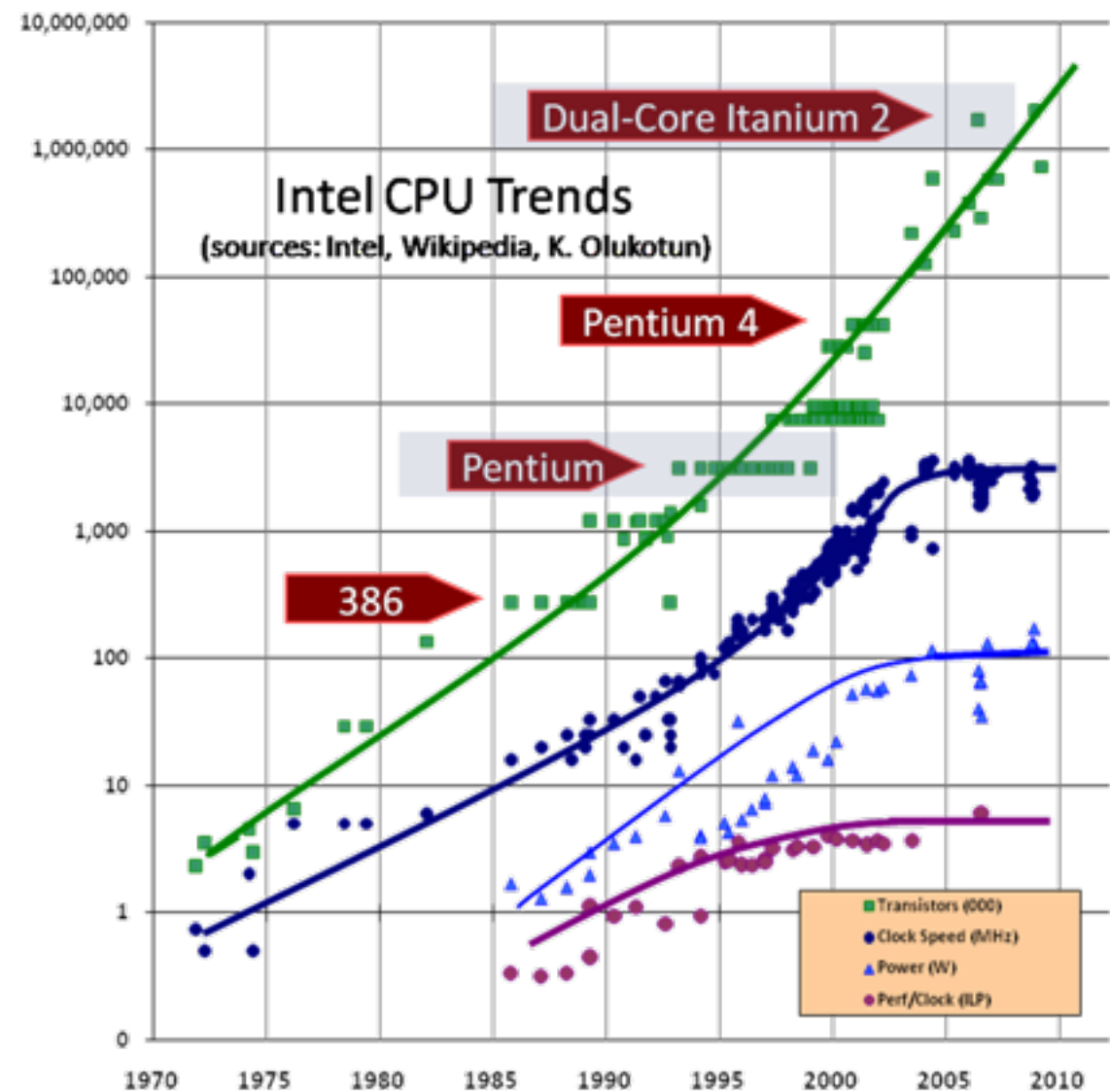
H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", Dr. Dobb's Journal, 30(3), March 2005 (updated with 2010 data) http://www.gotw.ca/publications/concurrency-ddj.htm

Multicore Hardware: NUMA

- Homogeneous and compatible cores – all cores are equivalent and have high-performance

- Memory access is non-uniform (NUMA)

  - Large on-chip cache memory

  - Main memory off-chip, accessed via interconnect

  - Cache coherency protocols maintain random access illusion

  - Memory access latency varies hugely depending on which core is accessing which memory bank

- Typical approach for x86-based systems to date

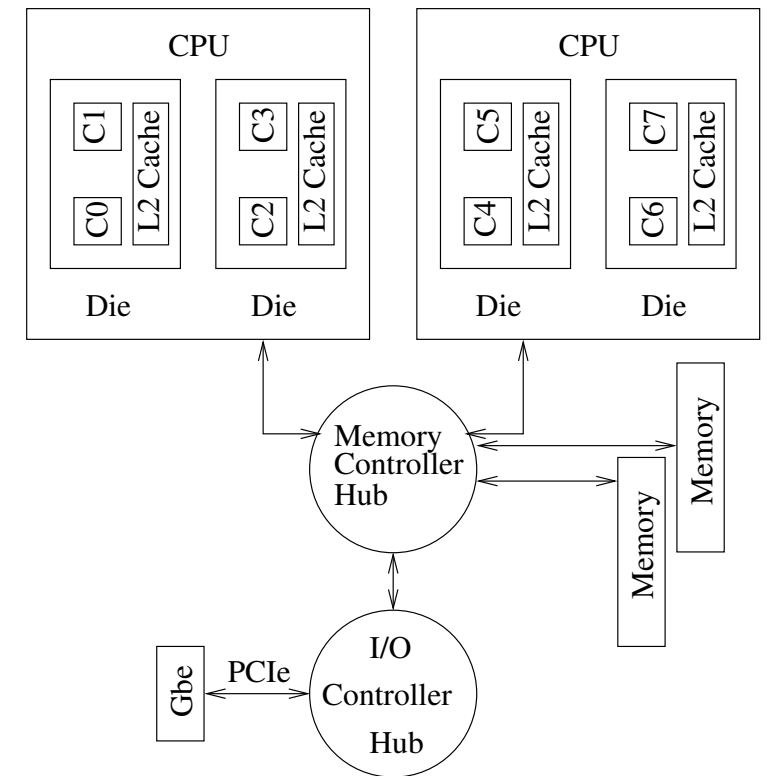 - Obvious evolution of uniprocessor designs to multicore world
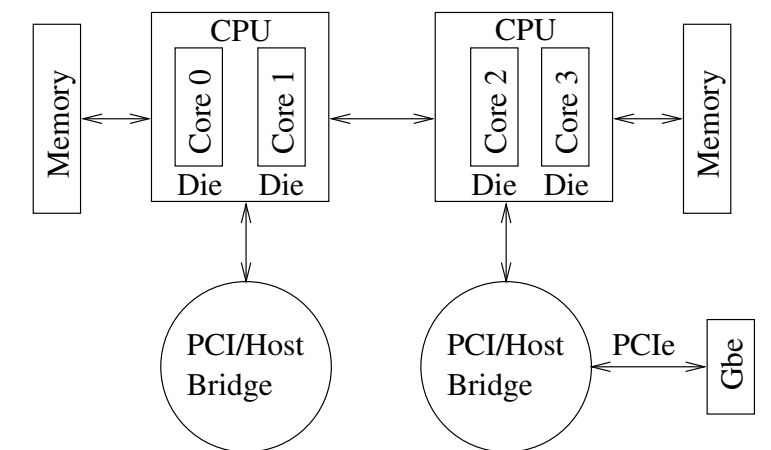

Figure 1. Structure of the Intel system

Figure 2. Structure of the AMD system

Multicore Hardware: Heterogeneous Cores

- Heterogeneous multiprocessor: CPU with multiple special purpose cores
  - IBM Cell Broadband Engine, AMD Fusion, NVIDIA Project Denver, Intel Larrabee, …
- Asymmetric processing capabilities
  - High-performance and low-power cores on a single die (e.g., ARM big.LITTLE model, with both Cortex A7 and A15 cores on-die)
  - GPU-like cores for graphics operations
  - Offload for crypto algorithms, TCP stack, etc.
- Asymmetric memory access models
  - Non-cache coherent
  - Cores explicitly do not share memory
- Common for mobile phones, games consoles, and other non-PC hardware

Cell Broadband Engine Processor
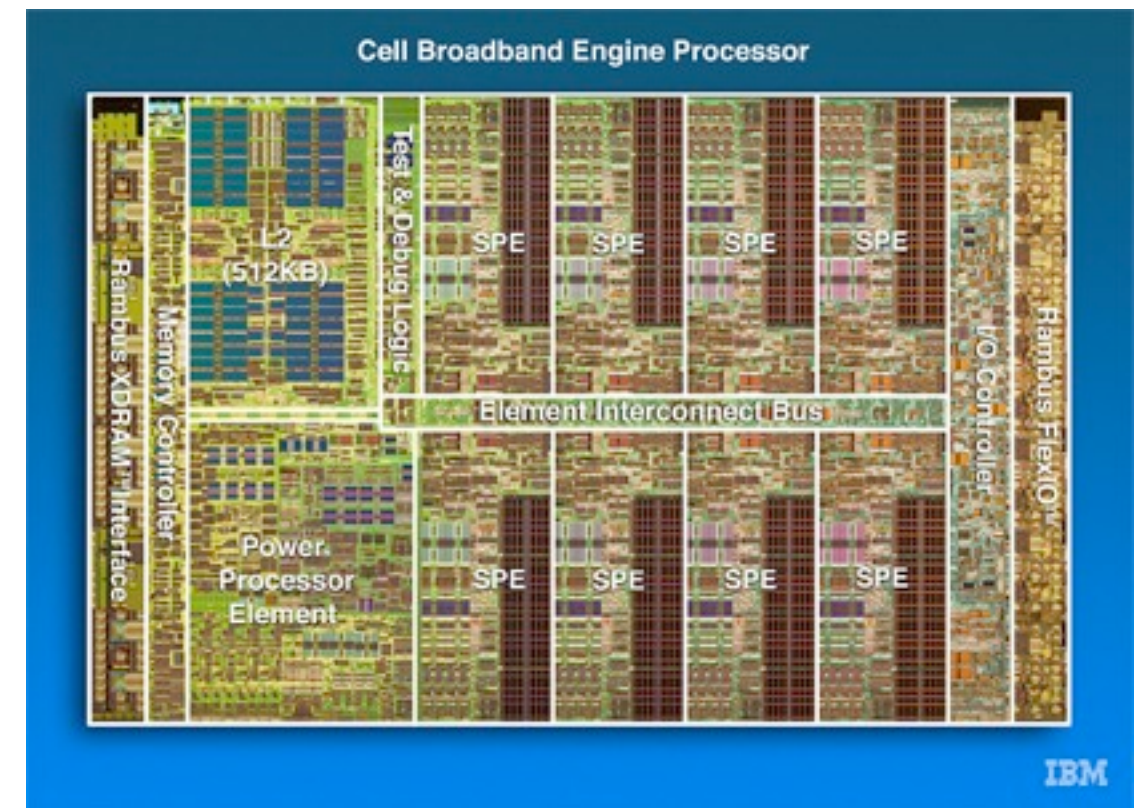
Programming Model

- ## Traditional hardware designs give the appearance of a uniform flat memory, shared between cores

  - ### Complex cache coherency protocols and memory models

  - ### Varying degrees of success in hiding the diversity

- ## Increasingly an illusion, maintained by underlying inter-core network

  - ### e.g., AMD HyperTransport, Intel QuickPath

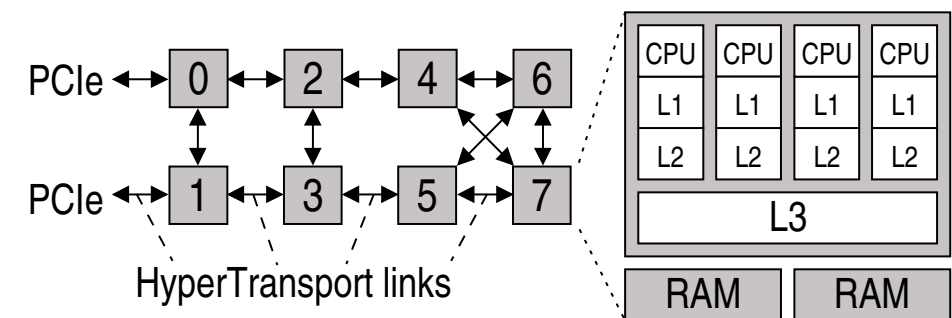 - ### Point-to-point communications/switching network with message passing protocol

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

Baumann *et al*, "The Multikernel: A new OS architecture for scalable multicore systems", Proc. ACM SOSP 2009. DOI 10.1145/1629575.1629579

- ## Newer architectures explicitly expose heterogeneity to programmers

  - ### IBM Cell processor is the canonical example

Design Approaches for Handling Diversity

- ## The range of system designs in increasing

  - Non-uniformity in memory access: multiple levels of partially shared cache is typical; HyperTransport → network-like communication between cores

  - Diversity of cores within a system, or instruction sets between cores:

    - Sony Playstation 3 with IBM Cell processor

    - Systems with CPU and GPGPU

    - Systems with FPGA cards attached as reconfigurable coprocessors,

    - TCP offload onto network adapters

  - Diversity of system designs

    - Server hardware *vs.* smartphone hardware – yet both have to be supported by variants of the same operating system (MacOS X = iOS = modified Mach microkernel with BSD Unix layer)

- ## Two design questions:

  - How to optimise kernel for diverse NUMA architectures?

  - How to design a kernel for heterogenous instruction sets? [→ lecture 17]

NUMA Optimisations: Memory Allocation

- ## Locality-aware memory allocation

  - Memory is discontiguous between nodes – partitioned address space

  - Threads should allocate memory local to the node on which they execute; essentially an independent memory management subsystem per node

    - The `malloc()` API is not sufficient in itself – cannot ensure that related data accessed by multiple threads is allocated in memory that is located on the same node, and cannot place allocations on particular nodes (pinning threads to particular nodes can help here)

- ## Replication of kernel memory on multiple cores

  - Read-only memory regions accessed by multiple threads should be replicated across nodes

    - e.g., the kernel code, shared libraries such as `libc`

  - Requires support from VM system, to map a virtual address to a different physical address on each core

  - Un-copy-on-write to collapse replicas down to a single page if a write occurs

NUMA Optimisations: Scheduling

- Scheduler must be aware of CPU topology, so it can assign threads to physically close processors
  - `struct sched_domain` in Linux

- Three layers of topology
  - Hyper-threading within a core (siblings)
  - Cores within a physical CPU package
  - Physical CPUs within a NUMA node

- Load balancing between nodes
  - CPU intensive tasks should be separate
  - Communicating threads should be close
  - Monitor and periodically rebalance – heuristic driven, hard to formulate a general policy
  - Linux has `sched_setaffinity()` to bind thread to a particular set of CPUs, to allow manual optimisation
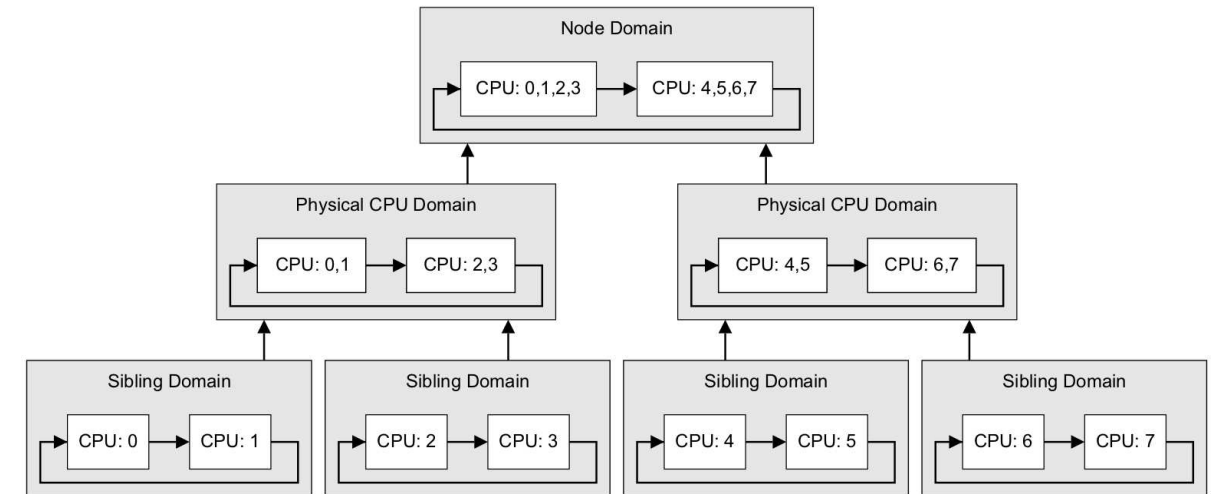
M. Bligh *et al.*, Linux on NUMA systems. Proc. Ottawa Linux Symposium, July 2004

```
struct sched_domain {
        /* These fields must be setup */
        struct sched_domain *parent;            /* top domain must be null terminated */
        struct sched_group *groups;             /* the balancing groups of the domain */
        cpumask_t span;                         /* span of all CPUs in this domain */
        unsigned long min_interval;             /* Minimum balance interval ms */
        unsigned long max_interval;             /* Maximum balance interval ms */
        unsigned int busy_factor;               /* less balancing by factor if busy */
        unsigned int imbalance_pct;             /* No balance until over watermark */
        unsigned long long cache_hot_time;      /* Task considered cache hot (ns) */
        unsigned int cache_nice_tries;          /* Leave cache hot tasks for # tries */
        unsigned int per_cpu_gain;              /* CPU % gained by adding domain cpus */
        int flags;                              /* See SD_* */

        /* Runtime fields. */
        unsigned long last_balance;             /* init to jiffies. units in jiffies */
        unsigned int balance_interval;          /* initialise to 1. units in ms. */
        unsigned int nr_balance_failed;         /* initialise to 0 */
};

struct sched_group {
        struct sched_group *next;               /* Must be a circular list */
        cpumask_t cpumask;
        unsigned long cpu_power;
};
```

Cache Coherence is not a Panacea

- Cost of maintaining cache coherence is increasing rapidly

  - Do all processors need to have a consistent view of memory – even just at synchronisation points defined by the memory model?

  - Potentially significant performance gains to be achieved by partitioning memory between processors – as discussed

    - Scheduling threads that share memory on processors that share cache can result in significant speedup

  - Equally – significant slowdowns can occur if unrelated data accessed by two cores shares a cache line

    - Access by one core invalidates the cache, causing a flush to main memory and reload; the other core then accesses, and the data is flushed back – ping-pong occurs
    - Can causes slowdowns of many orders of magnitude

Messages are Cheap and Easy

- Which is cheaper – messaging or shared memory?

  - On a real shared memory (cores, SHM1...SHM8) and message passing costs (MSG1...MSG8) for a server, with AMD HyperTransport

  - Cost of cache coherency protocols increases with the number of cores – messages can be cheaper, depending on the architecture

- Which is easier to program?

  - Shared-state concurrency is notoriously hard to program (locks, etc.)

  - Systems that avoid shared mutable state are frequently cited as easier to reason about

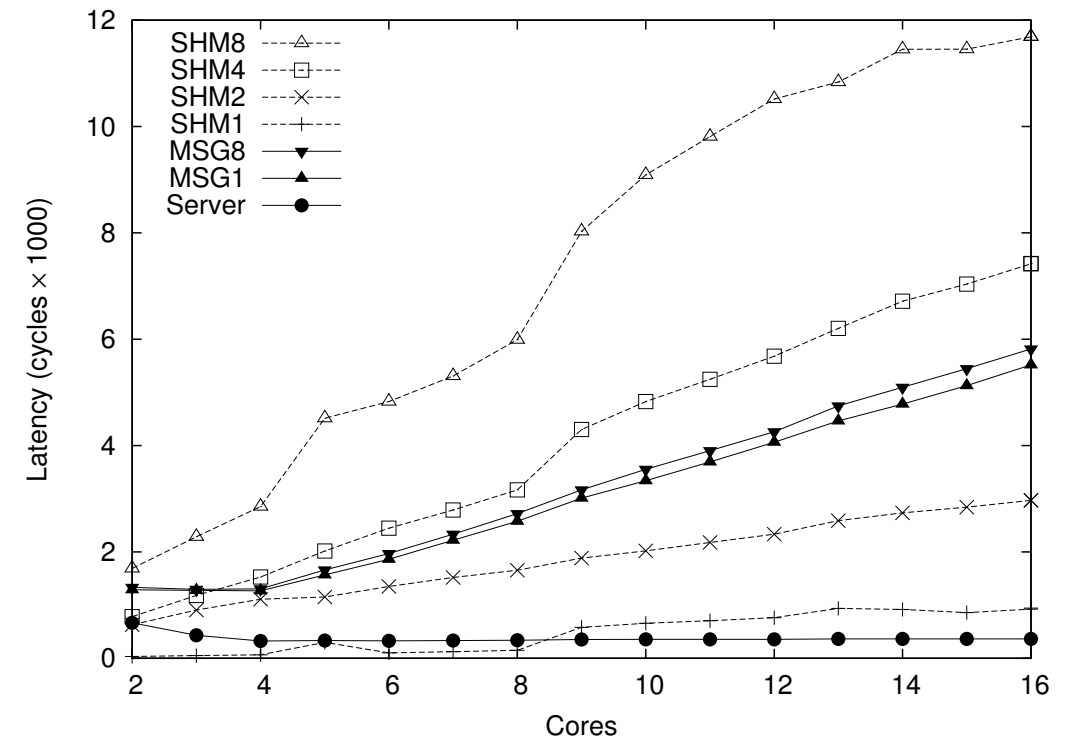- How long will the hardware be able to maintain illusion of shared memory?

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

Implications for Kernel Design

- A single kernel instance may not be appropriate

    - There may be no single "central" processor to initialise the kernel

    - How to coordinate the kernel between peer processors?

- Multicore processors are increasing distributed systems at heart – can we embrace this?

The Multi-kernel Model
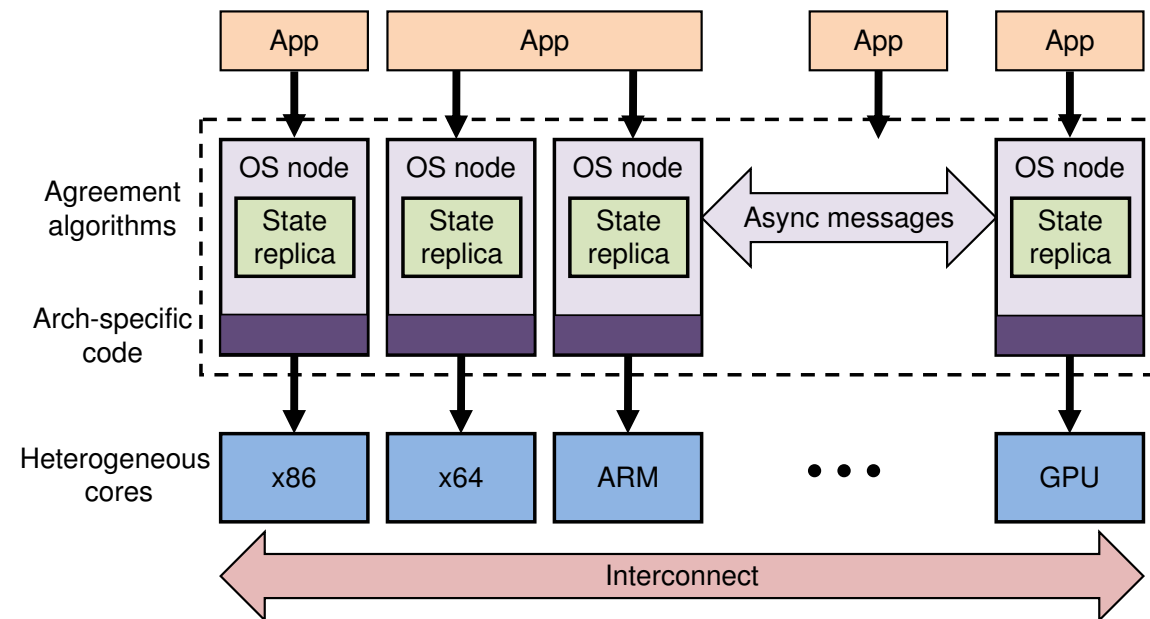
Baumann *et al*, "The Multikernel: A new OS architecture for scalable multicore systems", Proc. ACM SOSP 2009. DOI 10.1145/1629575.1629579

- Three design principles for a multi-kernel operating system
  - Make all inter-core communication explicit
  - Make OS structure hardware neutral
  - View state as replicated instead of shared

- Build a distributed system that can use shared memory where possible as an optimisation, rather than a system that relies on shared memory

- The model is no longer that of a *single* operating system; rather a collection of cooperating kernels

Principle 1: Explicit Communication

- ## Multi-kernel model relies on message passing

  - ### The only shared memory used by the kernels is that used to implement message passing (user-space programs can request shared memory in the usual way, if desired)

    - Strict isolation of kernel instances can be enforced by hardware

    - Share immutable data – message passing, not shared state

  - ### Latency of message passing is explicitly visible

    - Leads to asynchronous designs, since it becomes obvious where the system will block waiting for a synchronous reply

    - Differs from conventional kernels which are primarily synchronous, since latencies are invisible

  - ### Kernels become *simpler* to verify – explicit communication can be validated using formals methods developed for network protocols

Principle 2: Hardware Neutral Kernels

- Write clean, portable, code wherever possible

  - Low-level hardware access is necessarily processor/system specific

  - Message passing is performance critical: should use of system-specific optimisations where necessary

  - Device drivers and much other kernel code can be generic and portable – better suited for heterogeneity

  - Highly-optimised code is difficult to port

    - Optimisations tend to tie it to the details of a particular platform

    - The more variety of hardware platforms a multi-kernel must operate on, the better it is to have acceptable performance everywhere, than high-performance on one platform, poor elsewhere

- Hardware is changing faster than system software

Principle 3: Replicated State

- A multi-kernel does not share state between cores

    - *All* data structures are local to each core

    - Anything needing global coordination must be managed using a distributed protocol

    - This includes things like the scheduler run-queues, network sockets, etc.

        - e.g., there is no way to list all running processes, without sending each core a message asking for its list, then combining the results

    - A distributed system of cooperating kernels, not a single multiprocessor kernel

Multi-kernel Example: Barrelfish

- Implementation of multi-kernel model for x86 NUMA systems

- CPU drivers
  - Enforces memory protection, authorisation, and the security model
  - Schedules user-space processes for its core
  - Mediates access to the core and associated hardware (MMU, APIC, etc.)
  - Provides inter-process communication for applications on the core
  - Implementation is completely event-driven, single-threaded, and non-preemptable
  - ~7500 lines of code (C + assembler)

- Monitors
  - Coordinate system-wide state across cores

- Applications written to a subset of the POSIX APIs
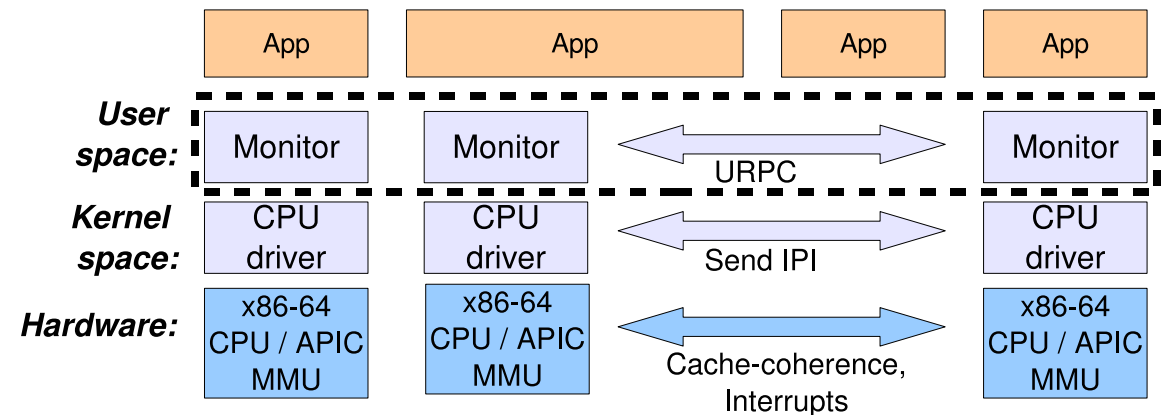
- Microkernel system
  - Network stack, memory allocation via capability system, etc., all run in user space

- Message passing tuned to the details of HyperTransport and x86 cache-coherency protocols
  - Highly system specific – port to ARM is underway

Baumann *et al*, "The Multikernel: A new OS architecture for scalable multicore systems", Proc. ACM SOSP 2009. DOI 10.1145/1629575.1629579

Further Reading and Discussion

- A. Baumann *et al*, "The Multikernel: A new OS architecture for scalable multicore systems", Proc. ACM SOSP 2009. DOI 10.1145/1629575.1629579

- Barrelfish is clearly an extreme: a shared-nothing system implemented on a hardware platform that permits some efficient sharing

  - Is it better to start with a shared-nothing model, and implement sharing as an optimisation, or start with a shared-state system, and introduce message passing?

- Where is the boundary for a Barrelfish-like system?

  - Distinction between a distributed multi-kernel and a distributed system of networked computers?