CS 202
Advanced Operating Systems

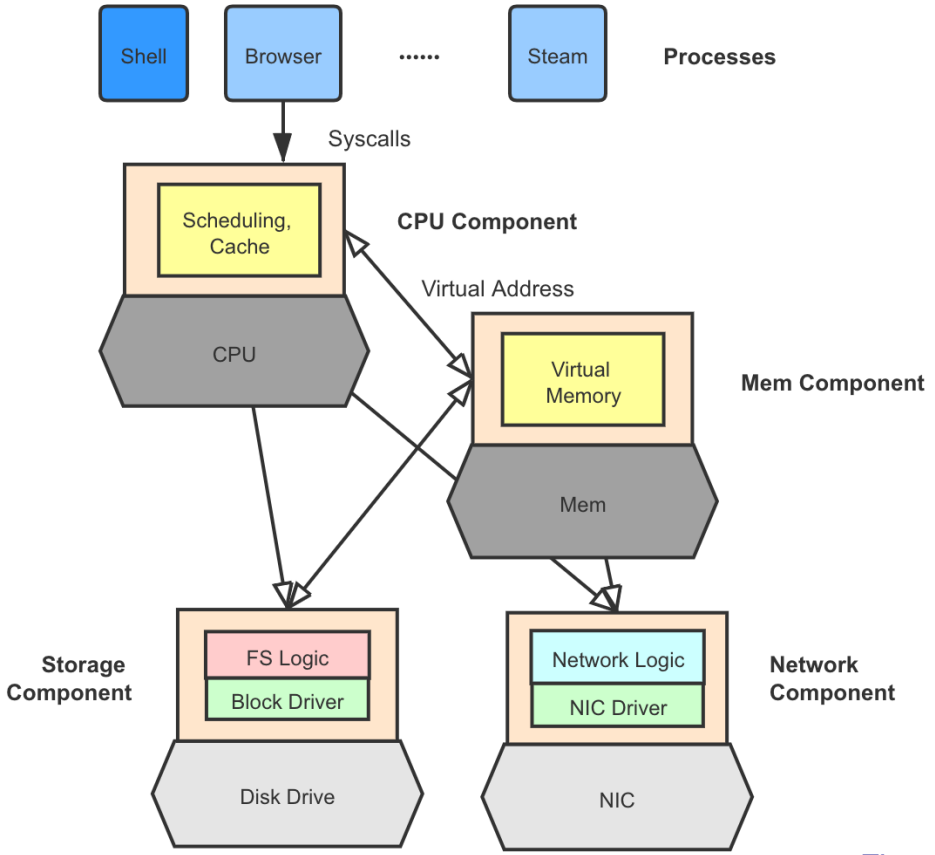
Winter 26

Lecture 3: OS Architecture

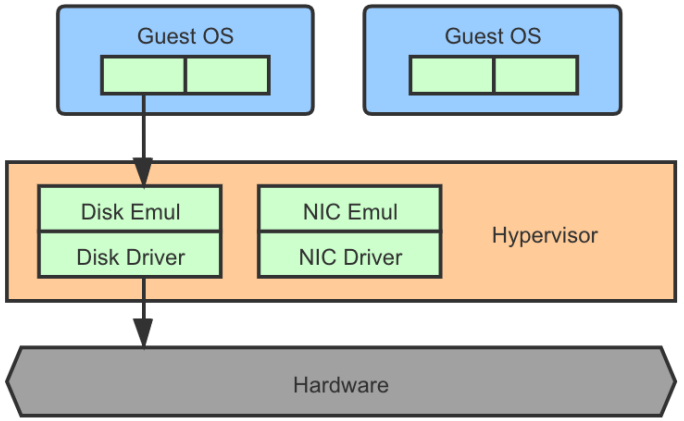
Instructor: Chengyu Song

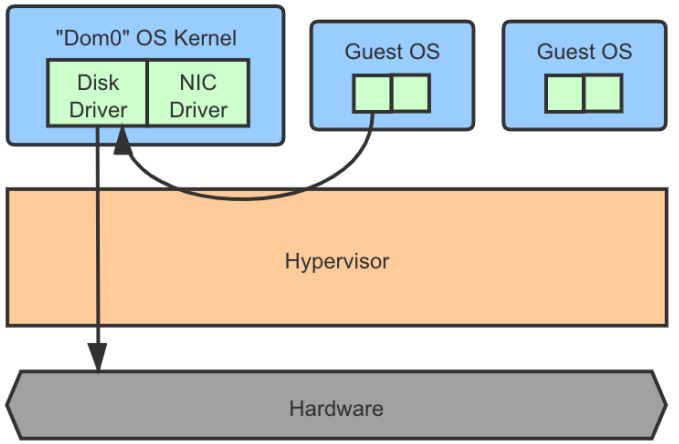
Typical OS architectures

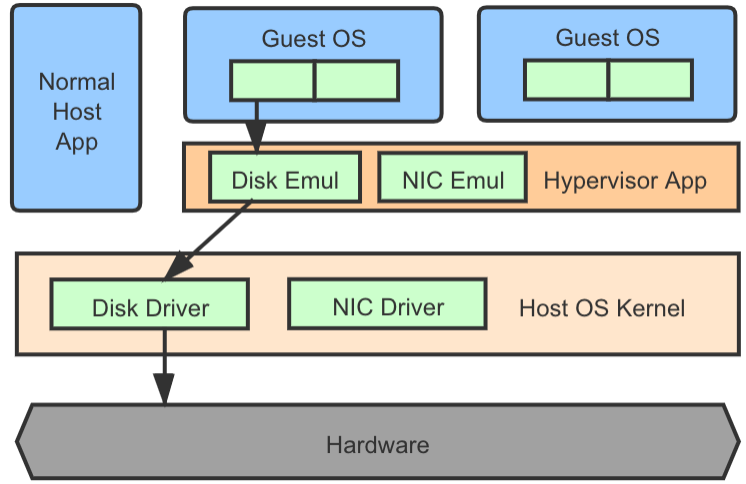
- Multi-kernels
  - Barrelfish, LegoOS



- Hypervisors
  - VMware ESX, Xen, KVM



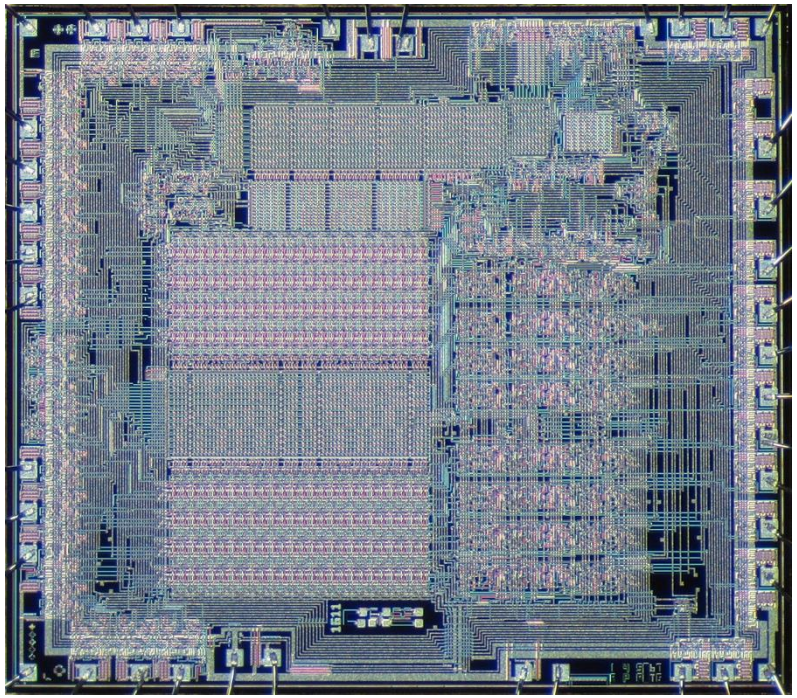




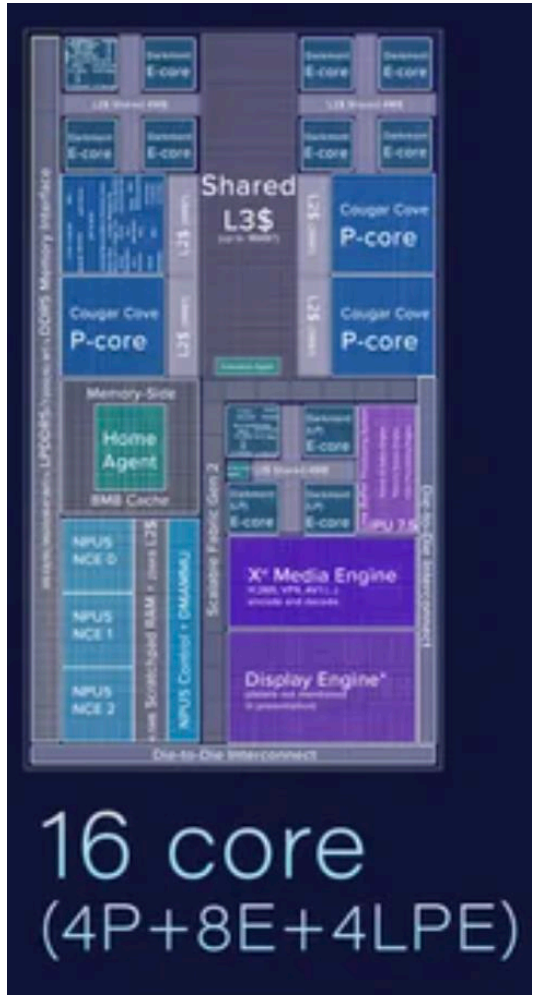
Figures from https://www.josehu.com/technical/2021/05/24/os-kernel-models.html

Barrelfish: context

- UNIX-time CPU

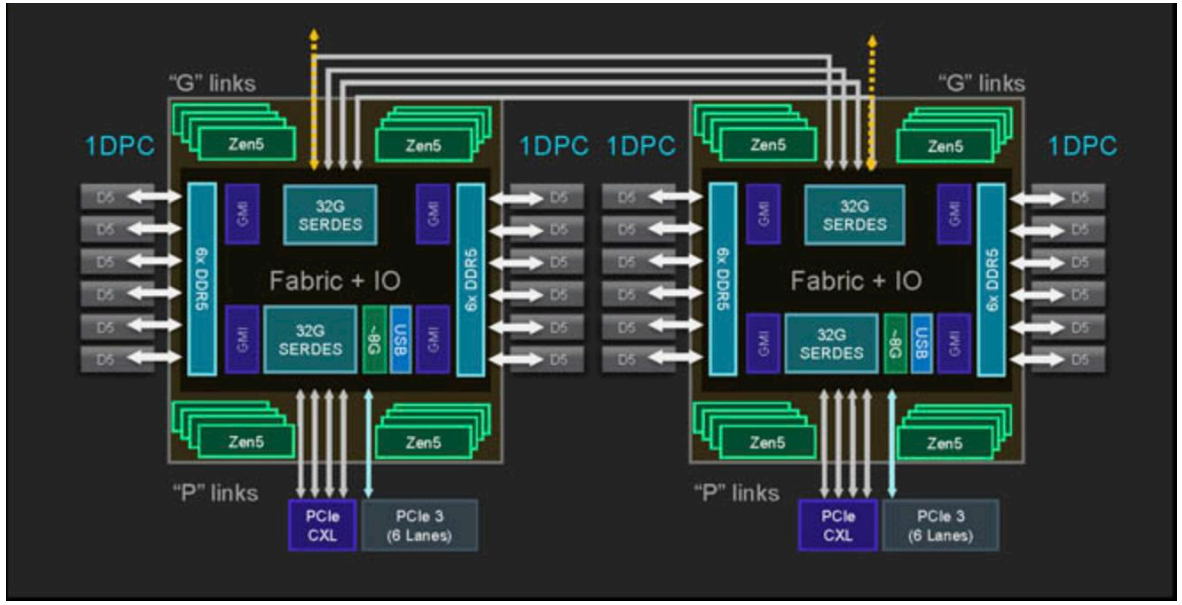


- Today's CPU



Barrelfish: scalability issues

- Systems may have many heterogenous cores
  - Frequency, power consumption, cache size, even ISA
- Systems may have non-uniform memory access (NUMA)
  - Remote memory (including RDMA nowadays)
  - Accessing memory requires help of other CPU/core
- Systems may also have heterogenous I/O
  - Accessing device requires help of other CPU/core
- Systems may have other types of processors and accelerators
  - GPUs, NPUs, DSPs



Barrelfish: limitations of uni-kernel

- Recall: to be efficient, a µ-kernel must be **inherently non-portable**. It should be treated like an optimized code generator, tailored specifically to the processor's architecture (TLB, cache, and register structures)
- For monolithic commodity kernels like Linux, we can configure them to optimize for a specific architecture (e.g., x86-64, aarch64)
- Modern OS cannot take advantage of the hardware on which they run
  - Diverse architectural tradeoffs, including memory hierarchies, inter-connects, instruction sets and variants, and IO configurations
  - Kernel checking which core it’s running on?
- Shared memory model does not scale anymore

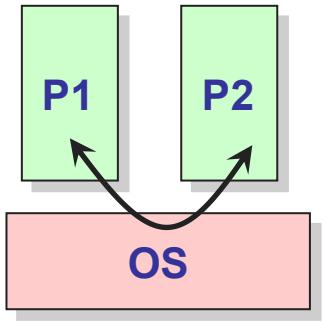
Barrelfish: message passing?

- Wait! Isn’t message passing slower than shared memory?
- Recall: why we introduced the concept of **thread?**

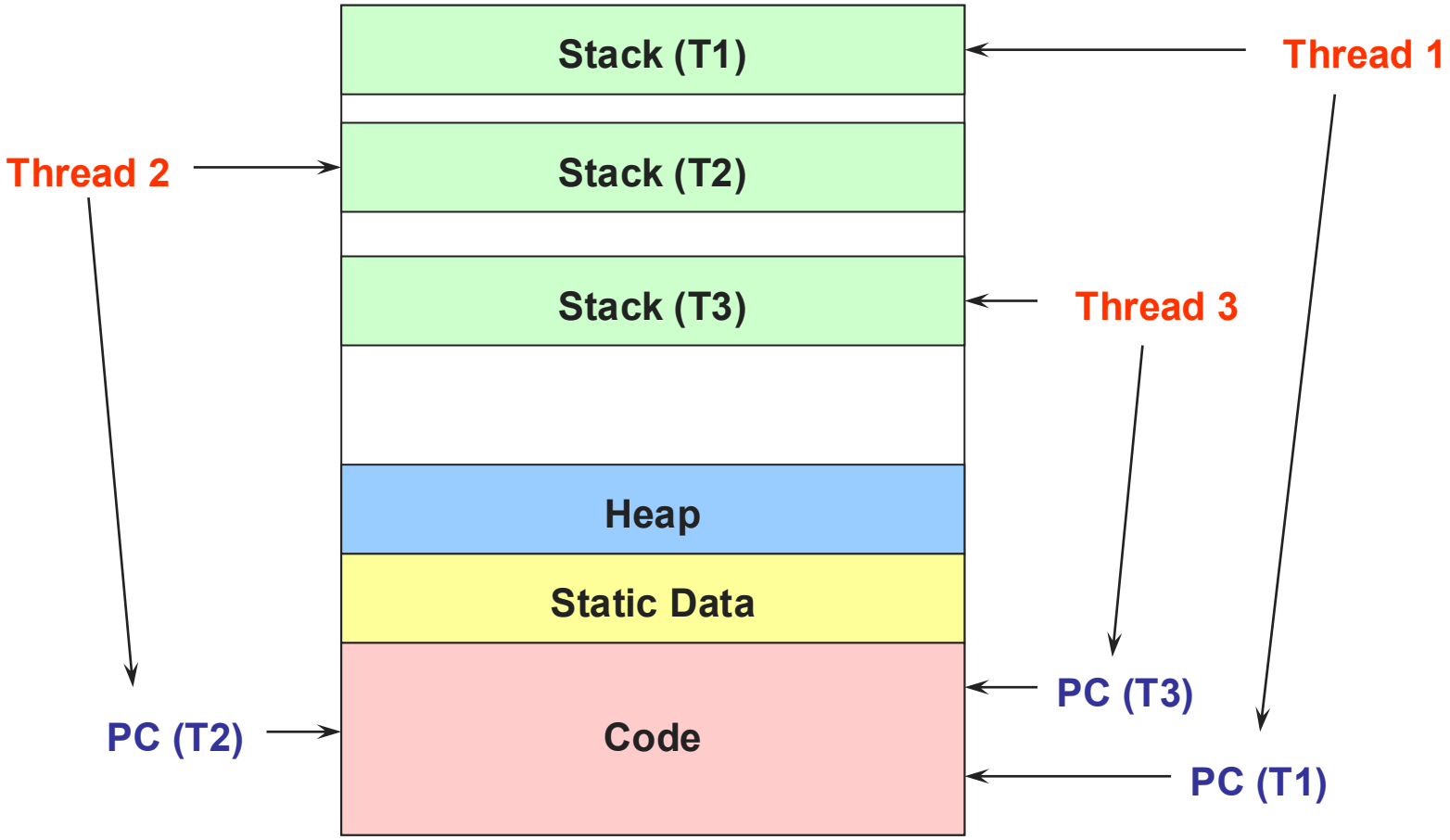
Some issues with processes

- Creating a new process is costly because of new address space and data structures that must be allocated and initialized
  - Recall struct proc in xv6 or Solaris

- Communicating between processes is costly because most communication goes through the OS
  - Inter Process Communication (IPC) – we will discuss later
  - Overhead of system calls and copying data



Threads in a Process



Barrelfish: message passing?

- Wait! Isn't message passing slower than shared memory?
- Recall: why we introduced the concept of **thread?**
- Recall: what is the main **challenge** for a multi-thread shared memory program (e.g., the kernel itself)?

Concurrent Access to Shared Resources

We initially focus on coordinating concurrent access to shared resources

- Basic problem
  - If two concurrent threads are accessing a shared variable, and at least one thread modified/written the variable, then access to the variable must be controlled to avoid erroneous behavior
- Over the next couple of lectures, we will look at
  - Exactly what problems occur
  - How to build mechanisms to control access to shared resources
    - » Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - » Reader-writer, bounded buffer, producer-consumer, etc.

Another Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
        balance = get_balance(account);
        balance = balance – amount;
        put_balance(account, balance);
        return balance;
}
```

- Now suppose that you and your father share a bank account with a balance of $1000
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account

Example Continued

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

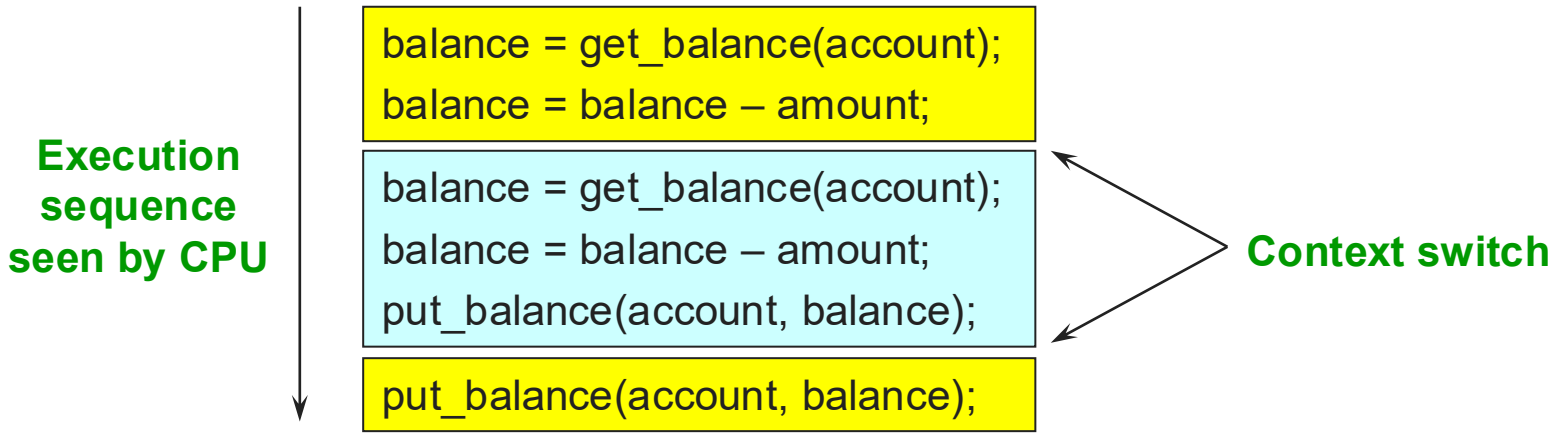
```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What's the problem with this implementation?
  - Think about potential schedules of these two threads

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:



- What is the balance of the account now?

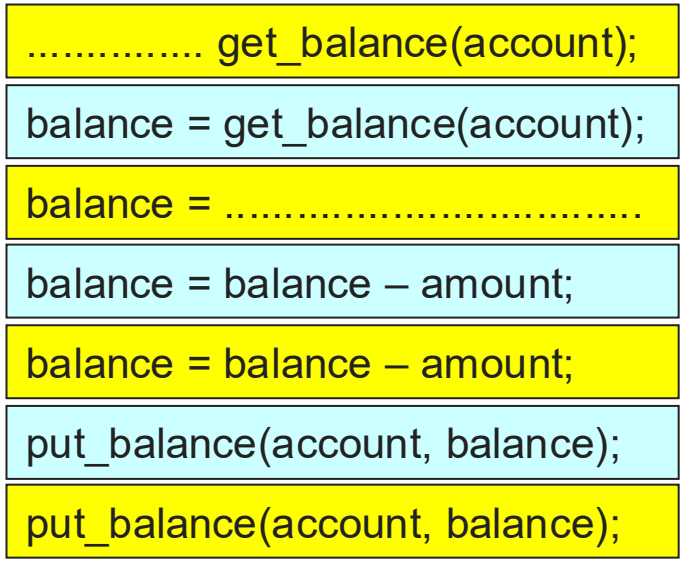
Shared Resources

- Problem: two or more threads accessed a shared resource (and the outcome depends on the order of their executions)
  - Known as a **race condition** (remember this buzzword!)
- Need mechanisms to control this access
  - So we can reason about how the program will operate
- Our example was updating a shared bank account
- Also necessary for synchronizing access to any shared data structure
  - buffers, queues, lists, hash tables, etc.

How Interleaved Can It Get?

How contorted can the interleaving be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever



Synchronization

- For correctness, we need to control this cooperation
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control

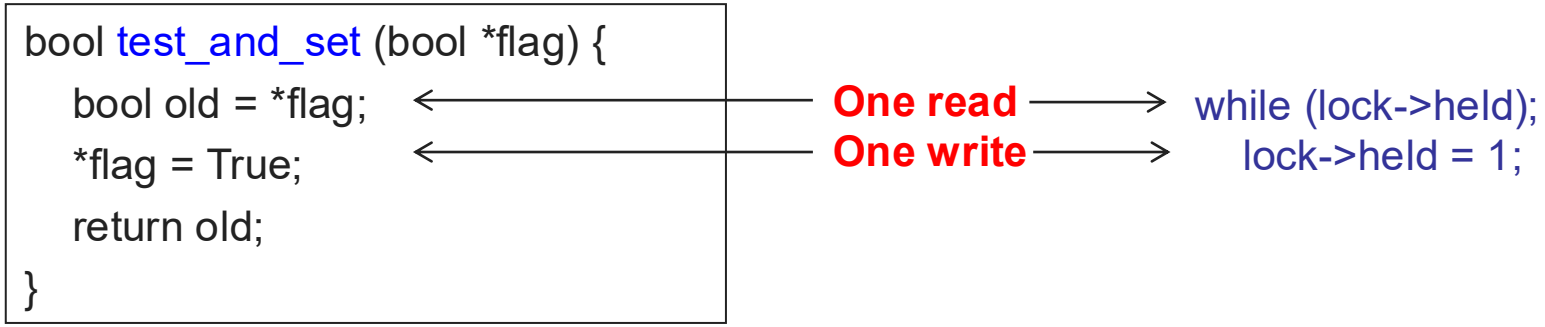
- We control cooperation using synchronization
  - Synchronization enables us to restrict the possible inter-leavings of thread executions

Barrelfish: message passing?

- Wait! Isn't message passing slower than shared memory?
- Recall: why we introduced the concept of **thread?**
- Recall: what is the main **challenge** for a multi-thread shared memory program (e.g., the kernel itself)?
- Recall: how do we **implement synchronization**?

Atomic Instruction: Test-and-Set

- The semantics of test-and-set are:
  1. Record the old value
  2. Set the value to indicate available
  3. Return the old value
- Hardware executes it atomically!



- When executing test-and-set on "flag"
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?

Using Test-and-Set

- Here is our lock implementation with test-and-set:

```
struct lock {
    bool held = 0; // 0 === false
}
void acquire (lock) {
    while (test-and-set(&lock->held));
}
void release (lock) {
    lock->held = 0; // 0 === false
}
```

Does it satisfy critical section requirements?
(mutex, progress, bounded wait, performance?)

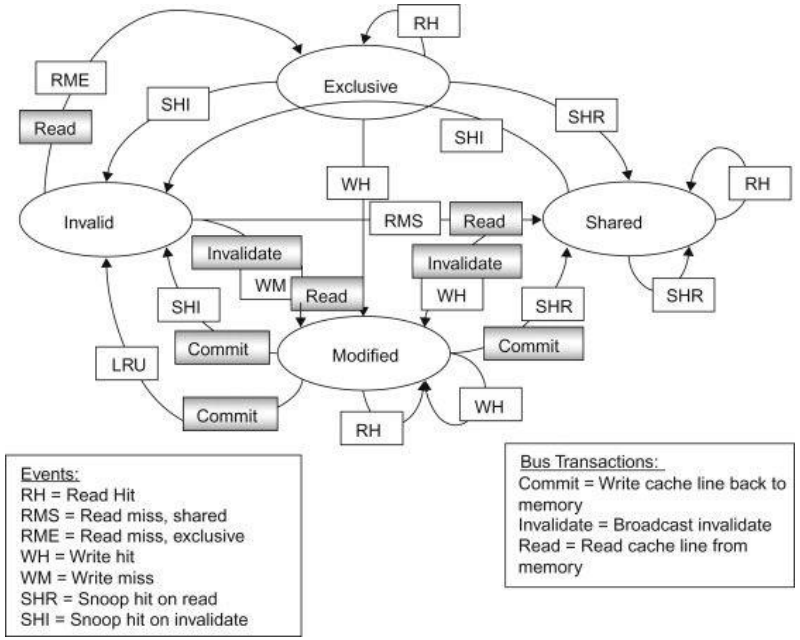
- When will the while return? What is the value of held?

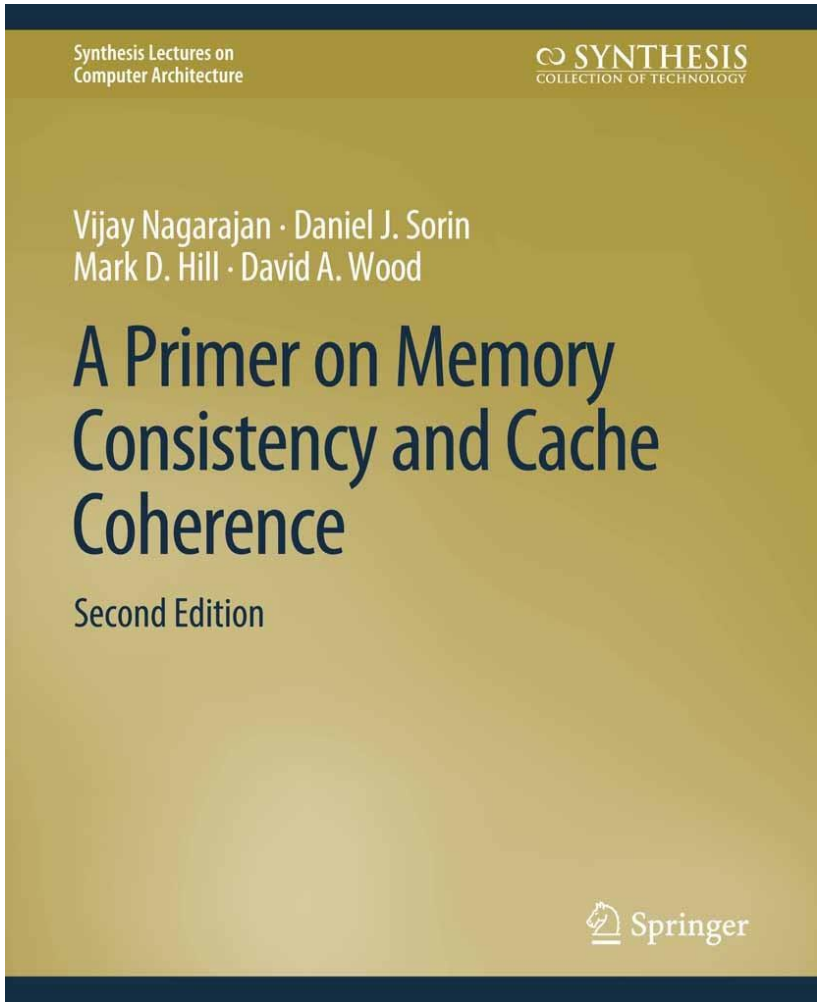
Barrelfish: message passing?

- Wait! Isn't message passing slower than shared memory?
- Recall: why we introduced the concept of **thread?**
- Recall: what is the main **challenge** for a multi-thread shared memory program (e.g., the kernel itself)?
- Recall: how do we **implement synchronization**?
- **Question: anyone knows how atomic operations are implemented?**

*How hardware solve the problem?

- Memory consistency
  - barriers
- Cache coherence
- CPU interconnects





Barrelfish: message passing?

- Wait! Isn't message passing slower than shared memory?
- Recall: why we introduced the concept of **thread?**
- Recall: what is the main **challenge** for a multi-thread shared memory program (e.g., the kernel itself)?
- Recall: how do we **implement synchronization**?
- **Question: anyone knows how atomic operations are implemented?**
  - Cache coherence overhead restricts the ability to scale to many cores

Barrelfish: message passing?

- Hardware is starting to resemble a message-passing network
- Shared memory at scale seems to be plagued by cache misses which cause core stalls
- At scale it has been shown that message passing has surpassed shared memory efficiency

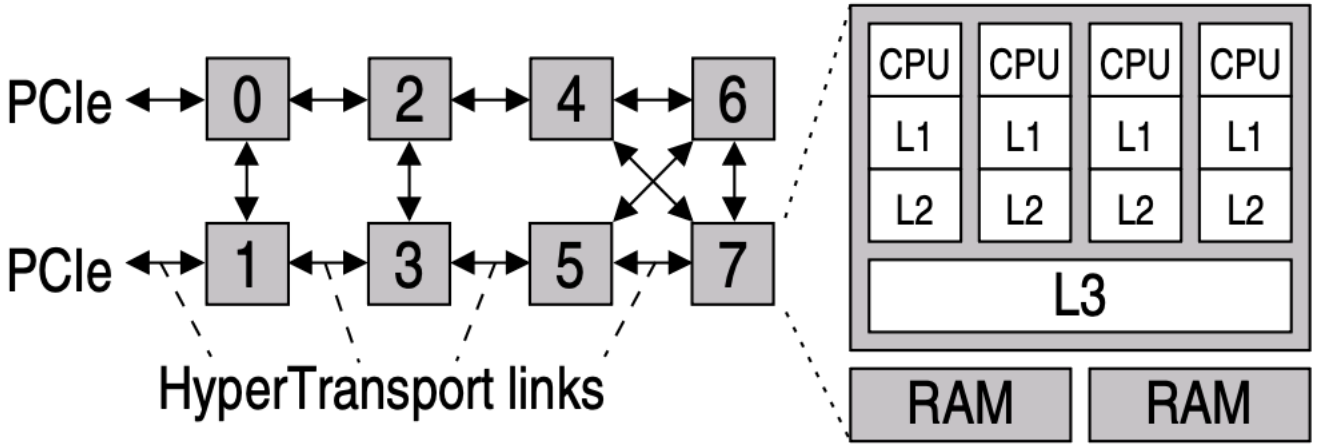


Figure 2: Node layout of an 8×4-core AMD system

Barrelfish: message passing?

- At scale it has been shown that message passing has surpassed shared memory efficiency (no locking)

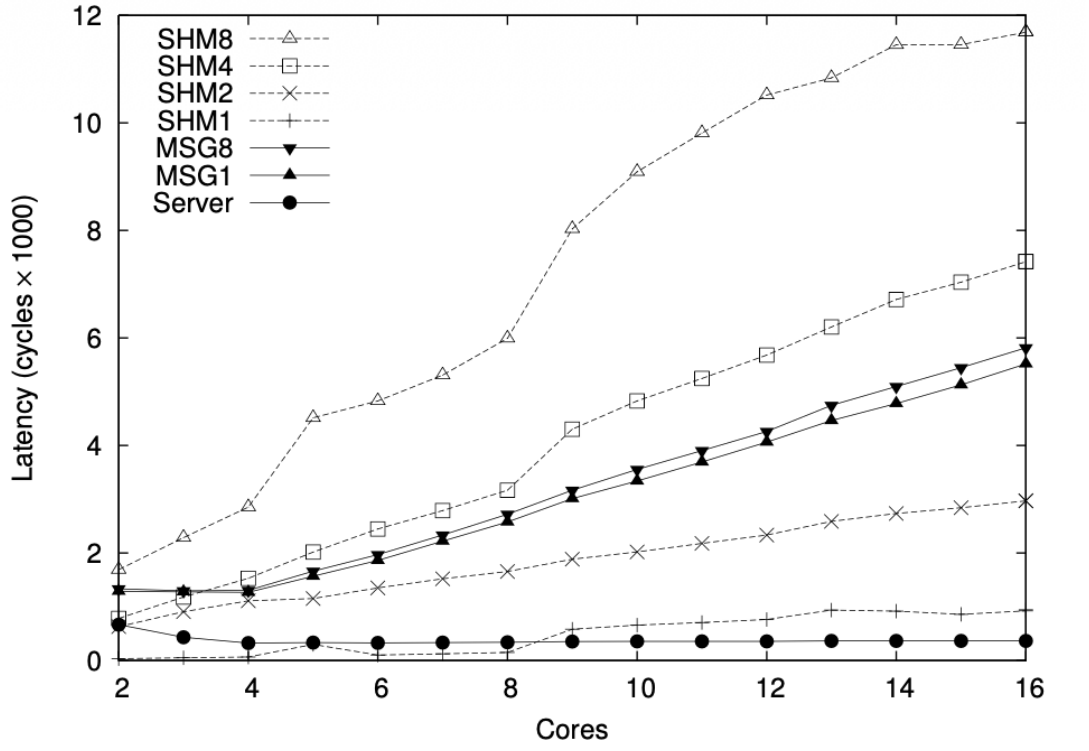


Figure 3: Comparison of the cost of updating shared state using shared memory and message passing.

Barrelfish: the multi-kernel solution

- 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

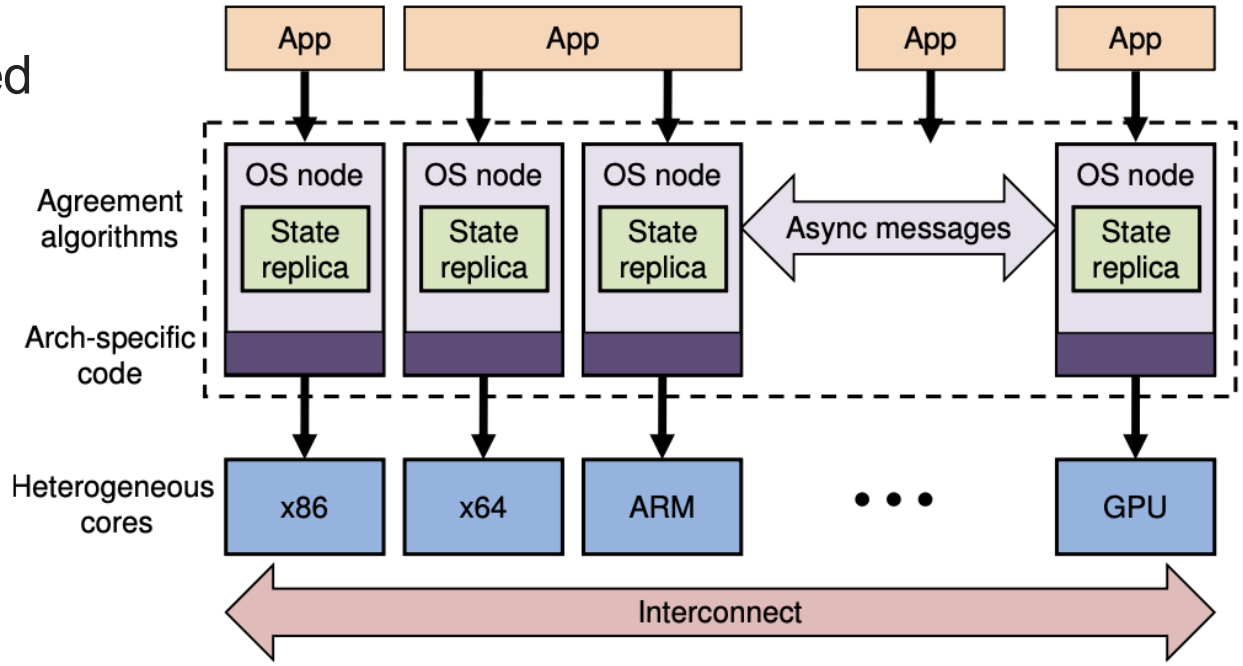


Figure 1: The multikernel model.

Make inter-core communication explicit

- All inter-core communication is performed using explicit messages
  - User-level remote procedure call approach
- No shared memory between cores aside from the memory used for messaging channels
  - shared memory is used as a channel to transfer cacheline-sized messages point-to-point
- Explicit communication allows the OS to deploy well-known networking optimizations to make more efficient use of the interconnect
  - pipelining (having a number of requests in flight at once)
  - batching (sending a number of requests in one message, or processing a number of messages together)

Make OS structure hardware-neutral

- A multi-kernel separates the OS structure as much as possible from the hardware
  - OS instance on each core factored into
    - » privileged-mode CPU Driver which is hardware-dependent
    - » user-mode Monitor process that is responsible for inter-core communication, which is hardware-independent
- Hardware-independence in a multi-kernel 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 is maintained by exchanging messages
- Updates are exposed in APIs as non-blocking and split-phase as they can be long operations
- Reduces load on system interconnects, contention for memory, overhead for synchronization; improves scalability
- Preserve OS structure as hardware evolves

Barrelfish: a multi-kernel implementation

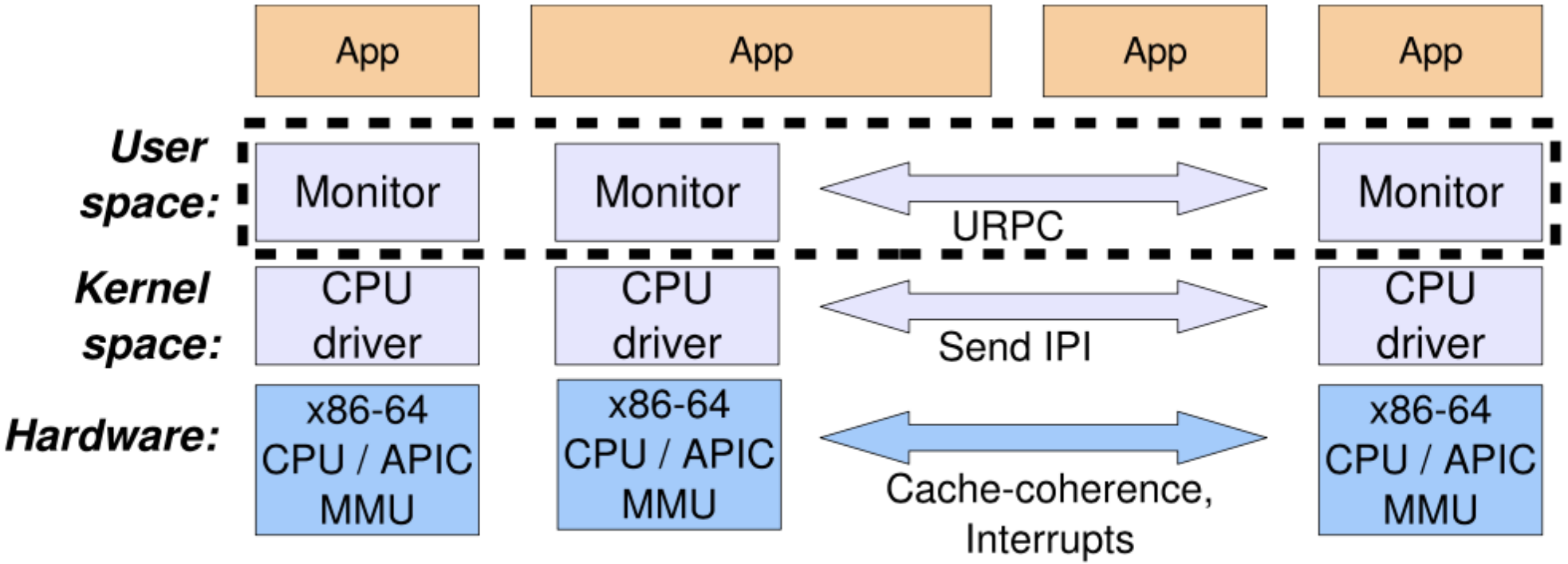


Figure 5: Barrelfish structure

Barrelfish: CPU drivers

- Enforce protection, perform authorization, time- slice processes, and mediate access to core and hardware – hardware-dependent per core
- Completely event-driven, single-threaded, and non- preemptable
- Serially process events in the form of traps from user processes or interrupts from devices or other cores
- Perform dispatch and fast local messaging between processes on core
  - Like exokernel
- Implements lightweight, asynchronous (split-phase) same-core IPC facility
  - Alternative, synchronous operation akin to LRPC or to L4 IPC
- Preserve OS structure as hardware evolves

Barrelfish: Monitors

- Schedulable, single-core, hardware-independent 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
- Device drivers run in user space also

Barrelfish: Process and Thread

- A process in Barrelfish is represented by a collection of dispatcher objects, one on each core on which it might execute
- Dispatchers on a core are scheduled by the local CPU driver
- The threads package in the default Barrelfish user library provides an API similar to POSIX threads

Barrelfish: Memory management

- Barrelfish uses a capability system modeled on that of seL4
  - All memory management is performed explicitly through system calls that manipulate capabilities, which are user-level references to kernel objects or regions of physical memory
  - Remove dynamic memory allocation from the CPU driver
- All virtual memory management, including allocation and manipulation of page tables, is performed entirely by user-level code
- Most operations requiring global coordination can be cast as instances of capability copying or retyping, allowing the use of generic consistency mechanisms in the monitors

Barrelfish: Evaluation

- Calls from the process to the 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

Barrelfish: Evaluation

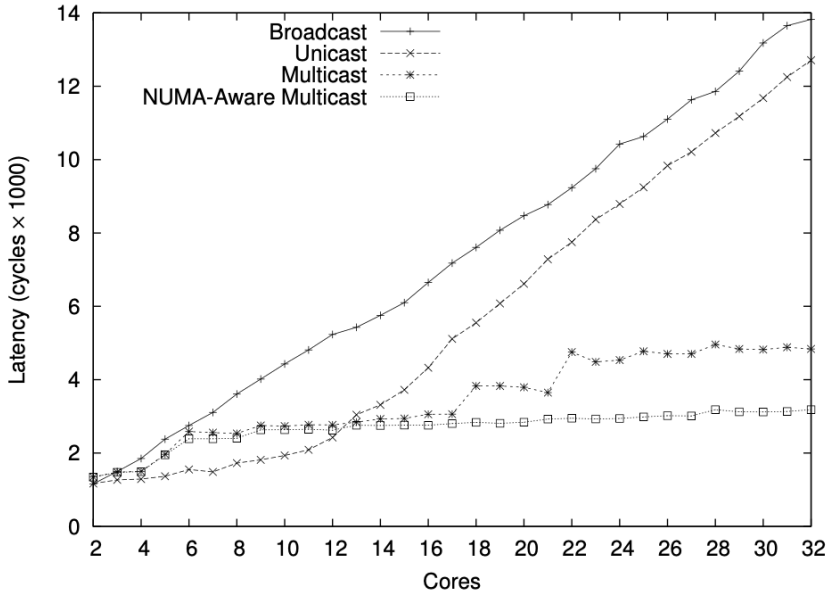


Figure 6: Comparison of TLB shootdown protocols

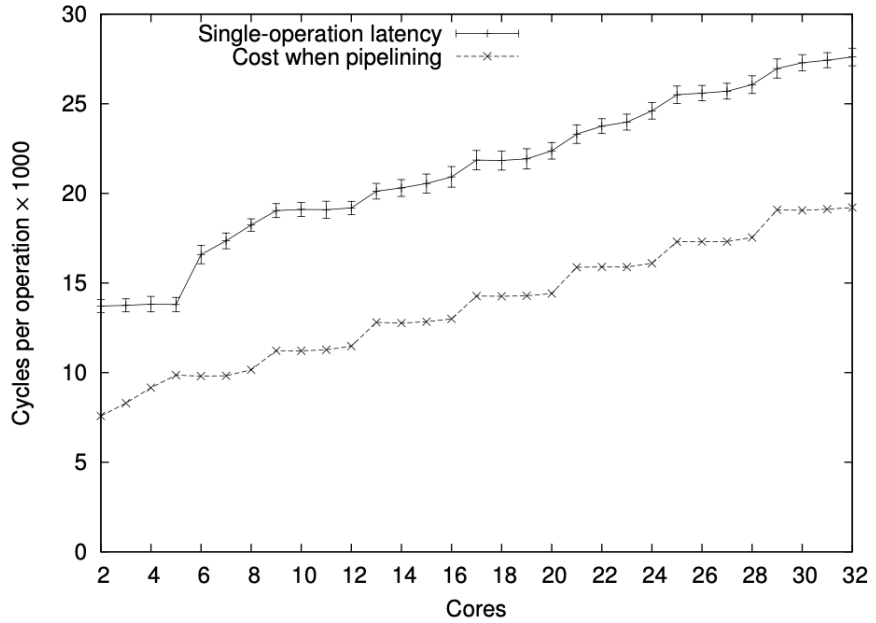


Figure 8: Two-phase commit on 8×4-core AMD

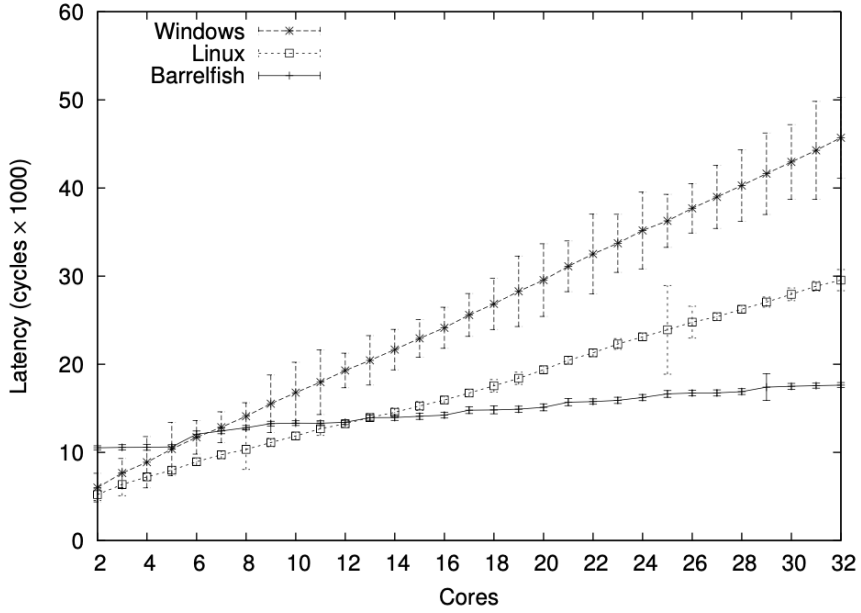
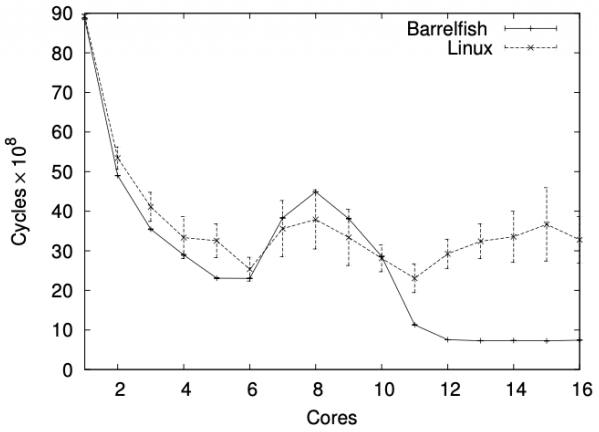
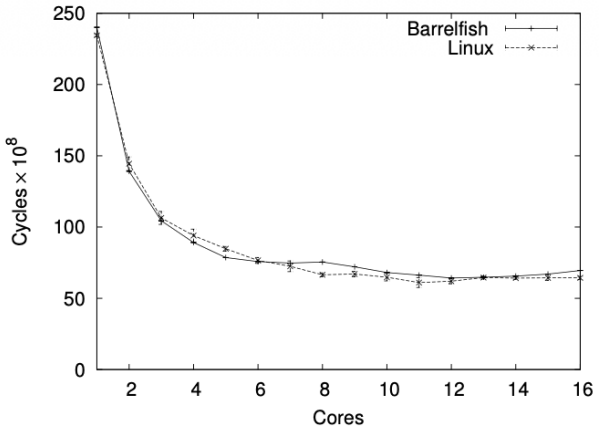


Figure 7: Unmap latency on 8×4-core AMD

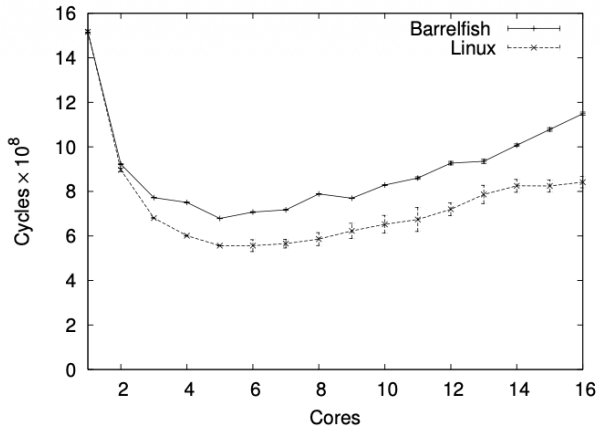
Barrelfish: Evaluation



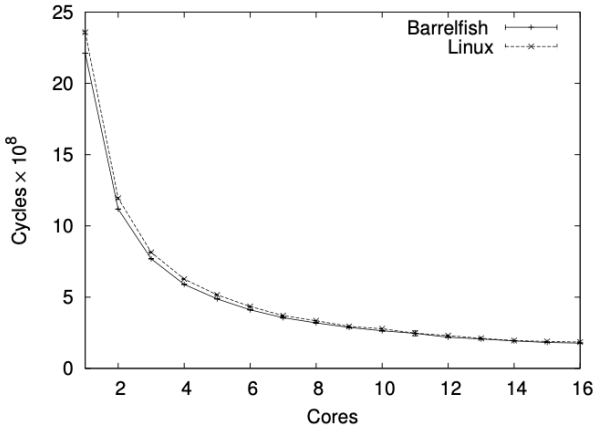
(a) OpenMP conjugate gradient (CG)



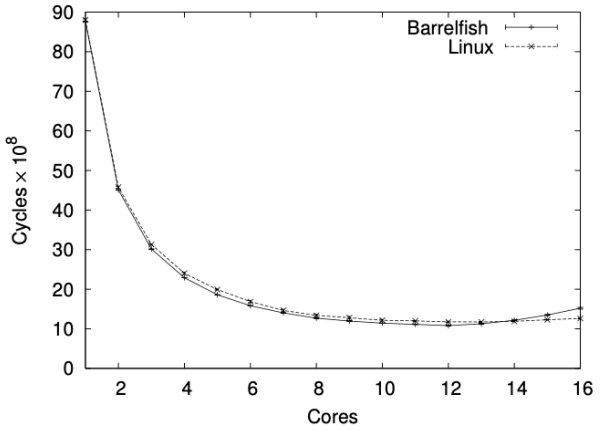
(b) OpenMP 3D fast Fourier transform (FT)



(c) OpenMP integer sort (IS)



(d) SPLASH-2 Barnes-Hut



(e) SPLASH-2 radiosity

Figure 9: Compute-bound workloads on 4×4-core AMD (note different scales on y-axes)