GPU COMPUTING
LECTURE 13 - CONSISTENCY & COHERENCE

Kazem Shekofteh
kazem.shekofteh@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

Inspired from lectures by Holger Fröning

REMINDER: OUR VIEW OF A GPU

Software view: a programmable many-core scalar architecture

- Huge amount of scalar threads to exploit parallel slackness, operates in lock-step
- SIMT: single instruction, multiple threads

IT’S A (ALMOST) PERFECT INCARNATION OF THE BSP MODEL

Hardware view: a programmable multi-core vector architecture

- SIMD: single instruction, multiple data
- Illusion of scalar threads: hardware packs them into compound units

IT’S A VECTOR ARCHITECTURE THAT HIDES ITS VECTOR UNITS

CONSISTENCY AND COHERENCE

Consistency: factual
Coherence: consistency over time

https://www.flickr.com/photos/starwarsblog/632644970/

EXAMPLE #1 (EXPECTATIONS AND REALITY)

Assume a coherent shared memory system

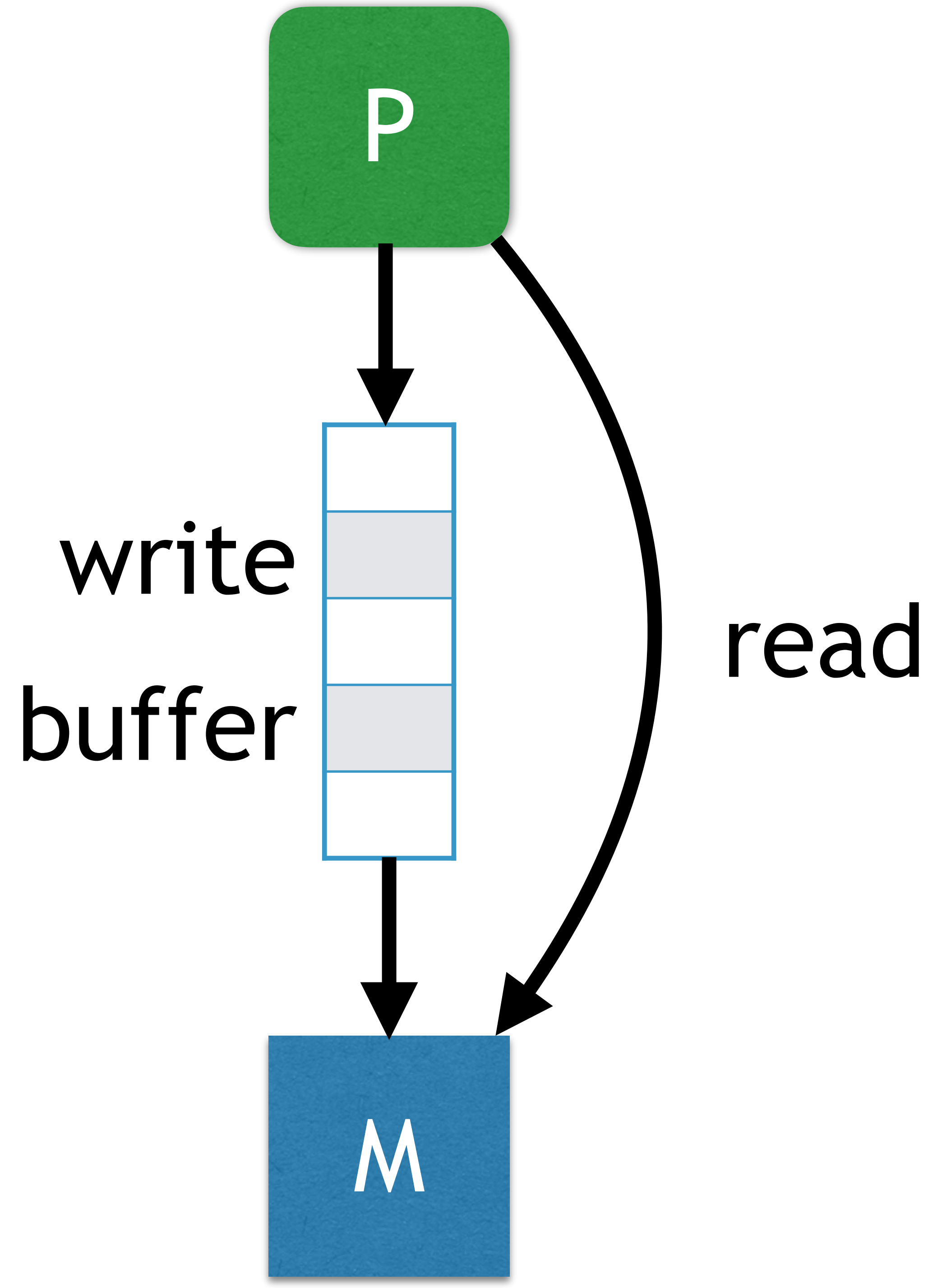
Can both if clauses be evaluated as “true”?

Yes: Assume stores can pass other stores (write buffering)

Other possible sources: out-of-order architecture, compiler optimizations, memory system contention, …

```
Thread 0 on processor 0           Thread 1 on processor 1

a = 0;                            b = 0;
...                               ...
a = 1;                            b = 1;
if ( b == 0 )                     if ( a == 0 )
{                                 {
  ...                               ...
}                                 }
```



EXAMPLE #2 (MORE FRUSTRATIONS)

Producer-Consumer scheme

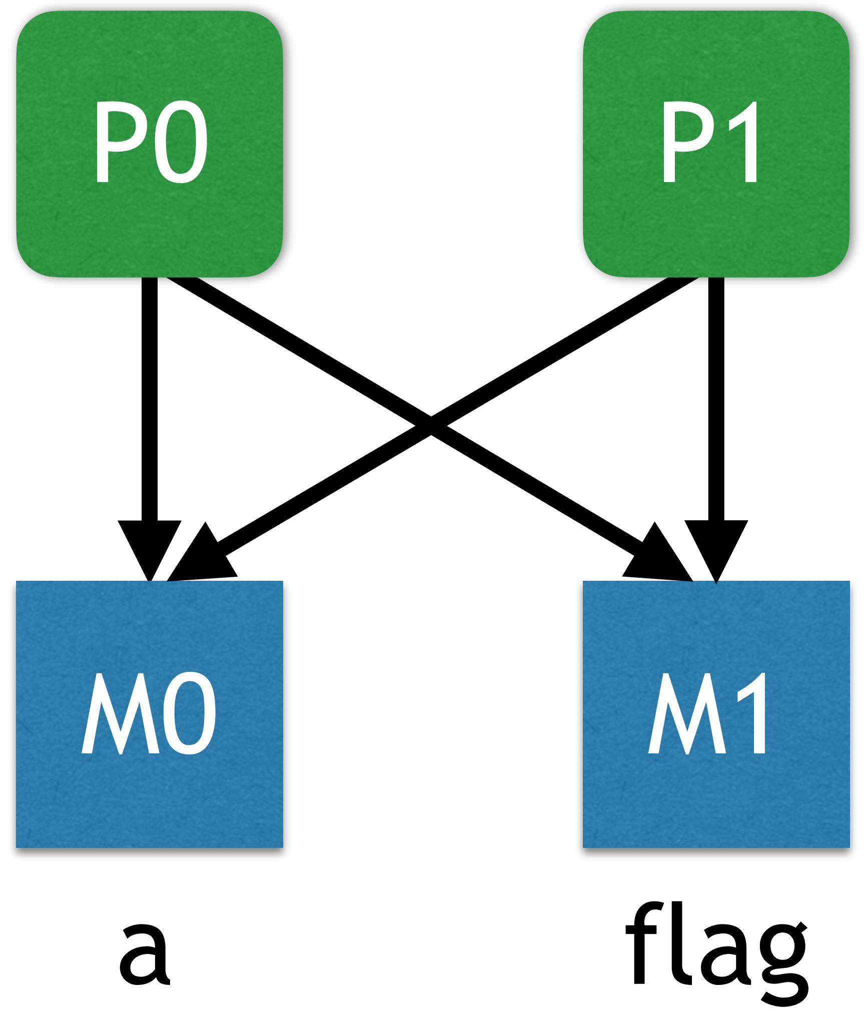
Assume a coherent shared memory system

Variables are initialized to zero

Which values can be printed out?

For modern CPU architectures, both 0 and 1

Thread 0 on processor 0	**Thread 1 on processor 1**
`a = 1;`	`while ( flag == 0 );`
`flag = 1;`	`print a;`



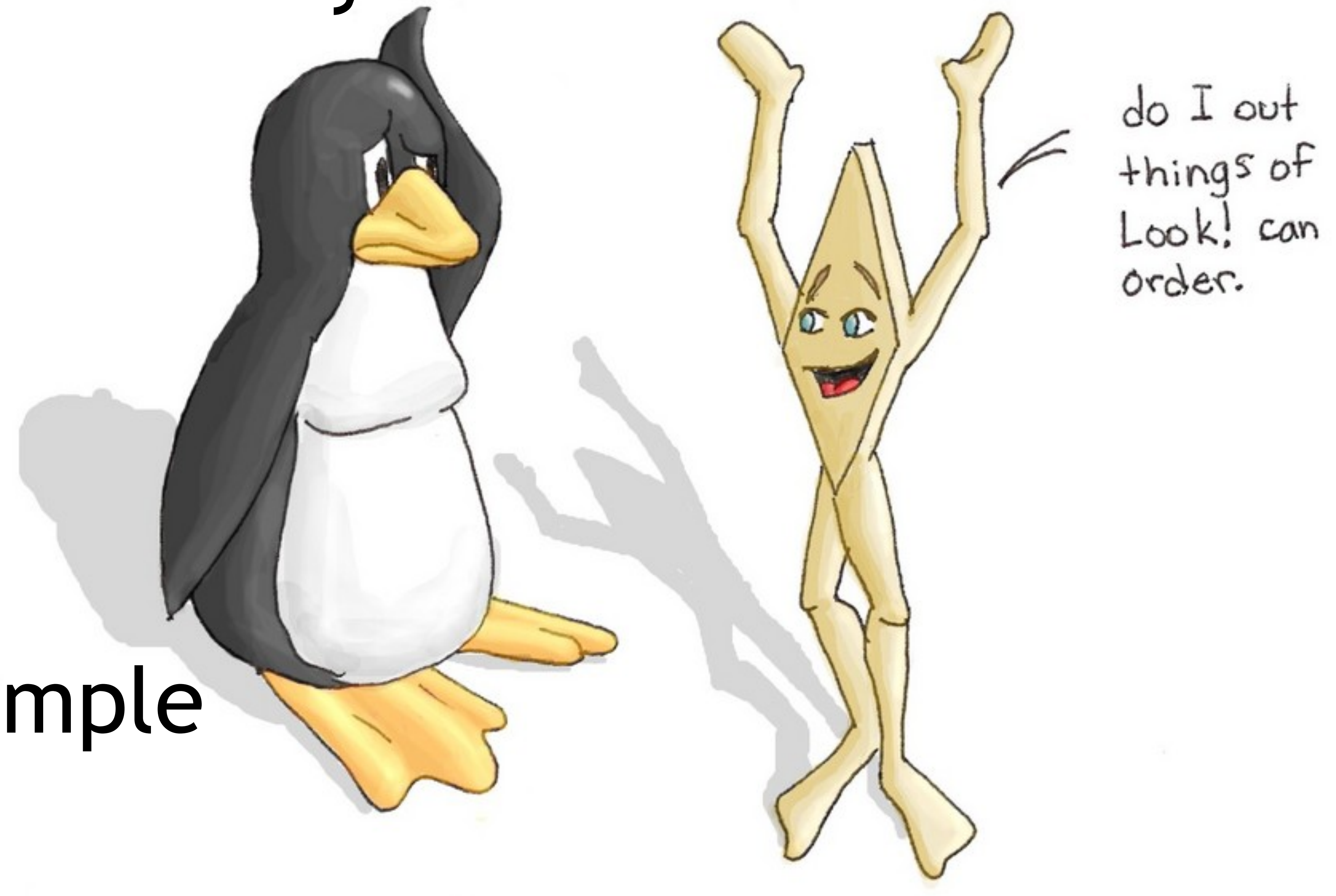
RELAXING CONSISTENCY

We relax consistency for a good reason: performance

Maintaining a strict and global ordering is incredibly expensive and hinders optimizations, e.g.:

- Store buffers
- Out-of-order processor architectures
- Multiple outstanding memory transactions
- Sliced (banked) caches

As we will see, GPUs are a very radical example of such relaxations



http://www.linuxjournal.com

SHARED MEMORY MULTIPROCESSORS

SHARED MEMORY

Similarities to executing multiple processes by time-sharing on a single processor

Process: defined as a single (virtual) address space with one or more threads of control

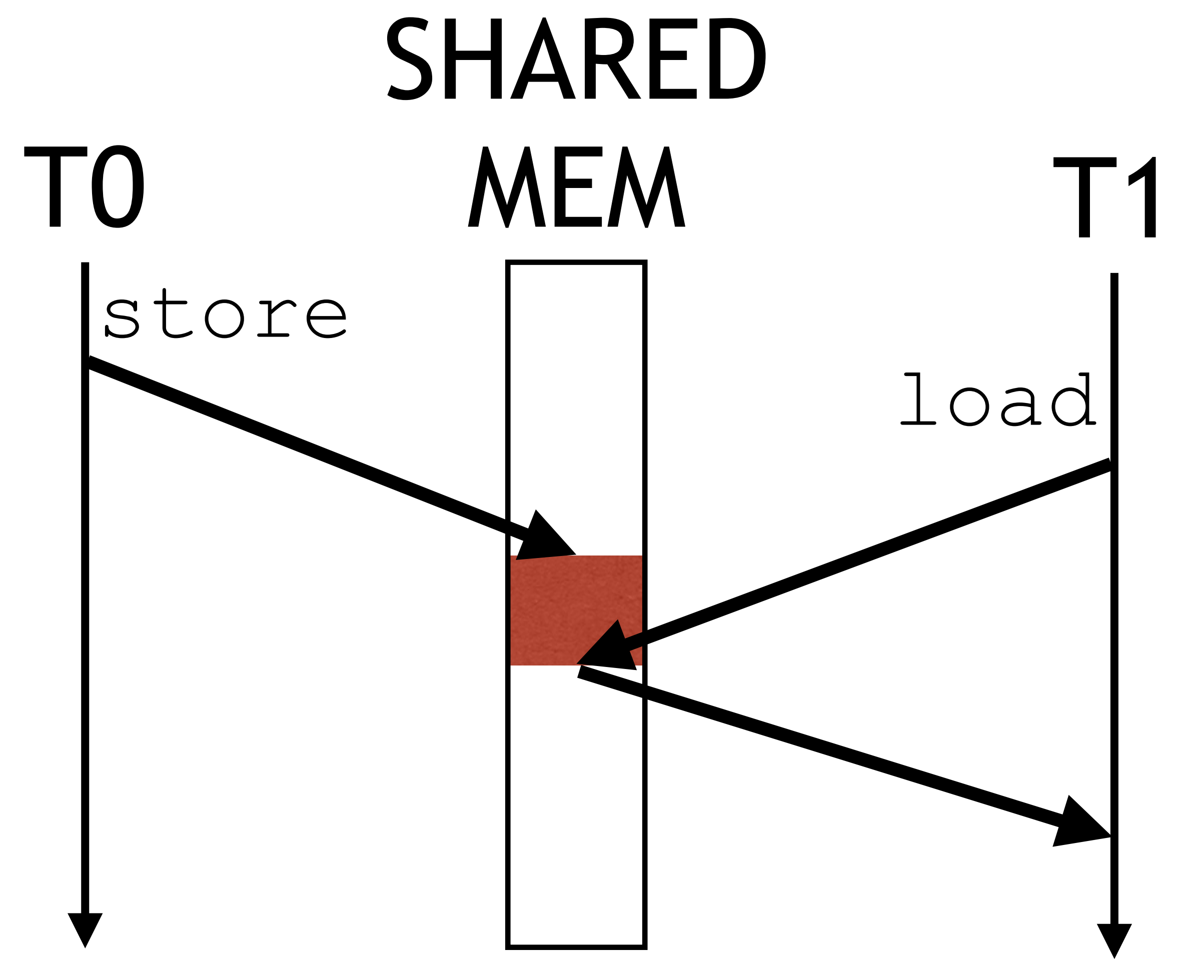
- Multiple threads share one address space by definition
- Portions of the address space can be shared, multiple virtual addresses (VA) map to a single physical address (PA)

Communication and synchronization

- Writes to a logically shared address by one thread are visible to reads of the other threads
- Rely on memory operations, including atomic operations

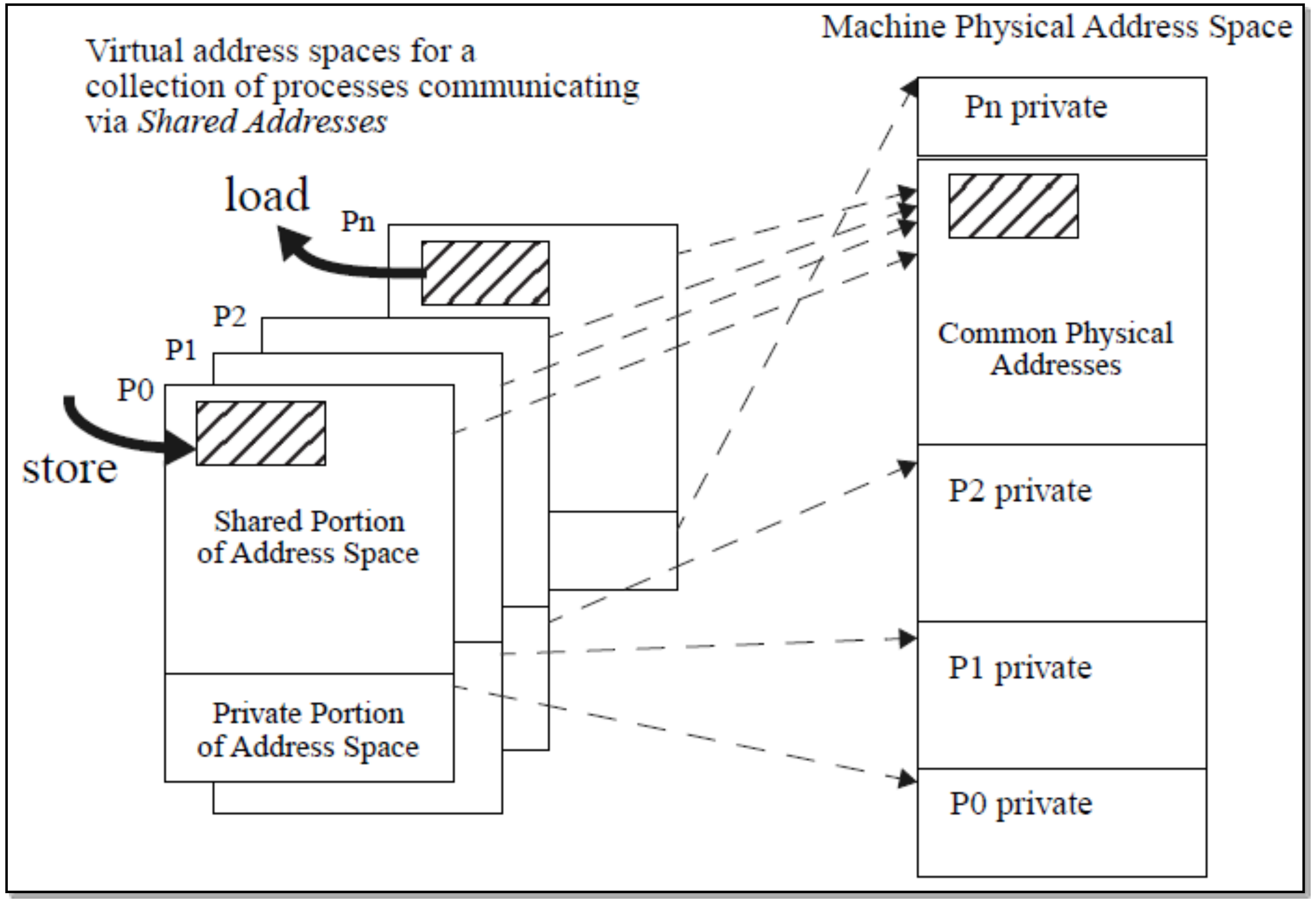
Virtual address space typically quite structured

- Private and shared segments



Perfectly timed load

SHARED MEMORY



Virtual address spaces for a collection of processes communicating via *Shared Addresses*
Machine Physical Address Space
load
store
Pn
P2
P1
P0
Shared Portion of Address Space
Private Portion of Address Space
Pn private
Common Physical Addresses
P2 private
P1 private
P0 private

Culler et al, Parallel Computer Architecture, MK 1999

An address space defines a range of discrete addresses; each address may correspond to a different resource

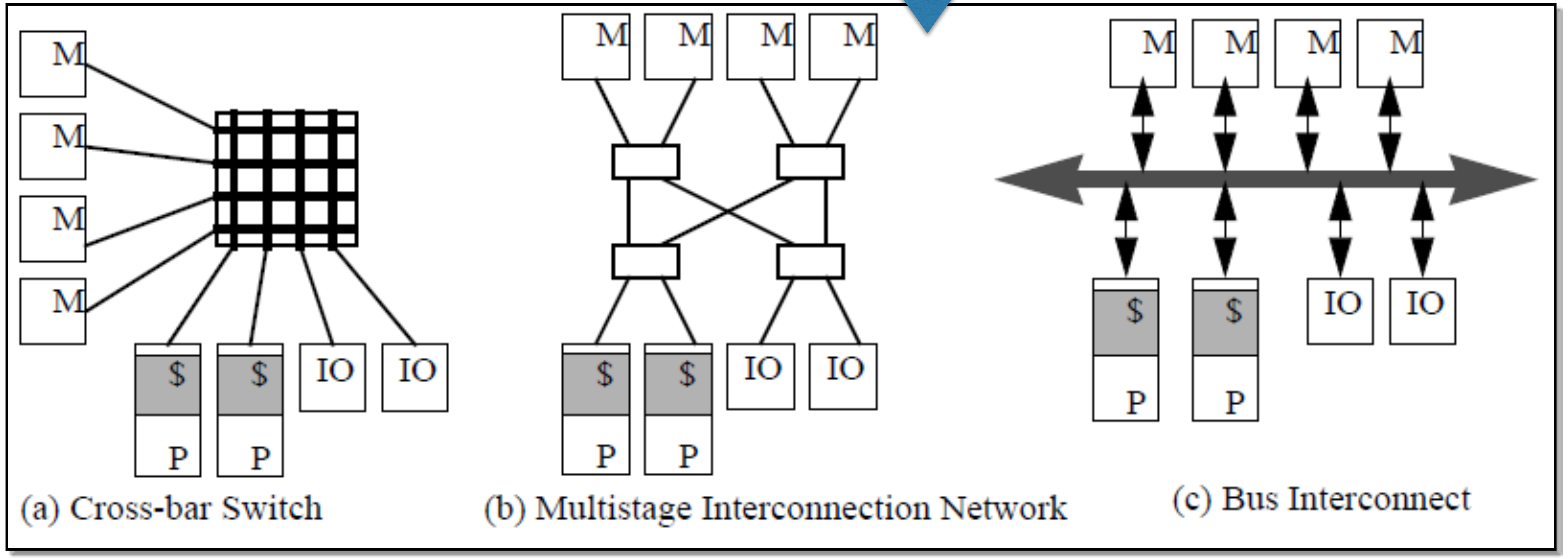
SHARED MEMORY

Extending to a shared-memory multiprocessor by adding processors

- Typical shared memory multiprocessor interconnection scheme
- (Non-) Uniform Memory Access ((N)UMA)

Recent CPU architectures?

(a)Late FSB implementations – illusion of a bus network
(b)Opteron (HT), Intel Nehalem (QPI), Sandy Bridge etc.
(c)Early FSB implementations – true bus networks



Culler et al, Parallel Computer Architecture, MK 1999

FUNDAMENTAL DESIGN ISSUES OF A COMMUNICATION ABSTRACTION

Communication abstraction

Contract, similar to ISA

1. Naming

   What data can be named?

2. Operations

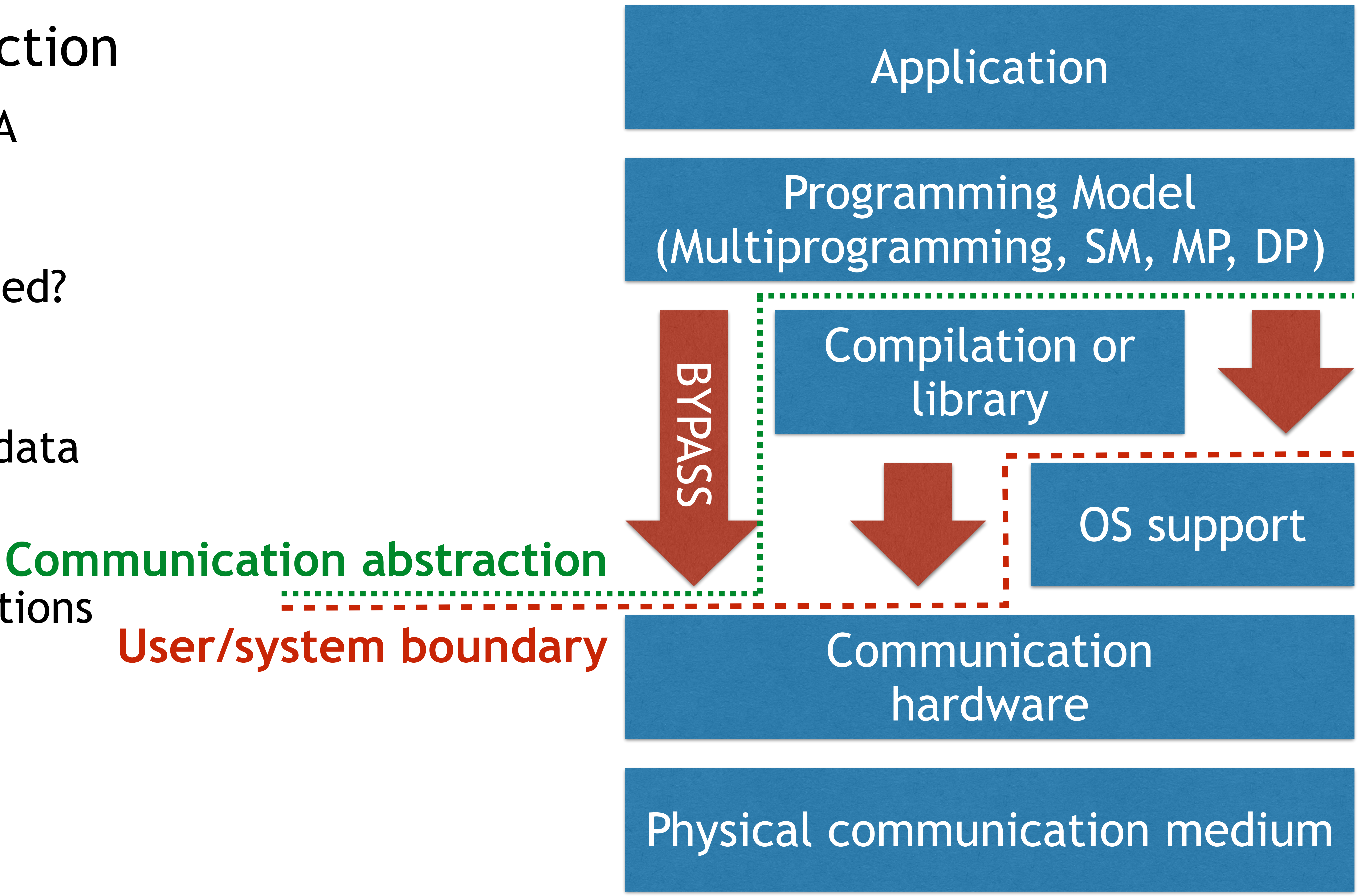
   Operations on named data

3. Ordering

   Ordering among operations

4. Communication/ Replication of data

5. Performance



Application
Programming Model (Multiprogramming, SM, MP, DP)
BYPASS
Compilation or library
OS support
Communication abstraction
User/system boundary
Communication hardware
Physical communication medium

FUNDAMENTAL DESIGN ISSUES: NAMING

- Shared memory
  - Naming: Thread can name locations in the register and the virtual address space
    - Segments for code, stack, heap
  - Access to shared variables mapped to load/store instructions on virtual addresses
  - Global physical address space: shared virtual addresses map to the same physical address
  - Independent local physical address spaces: page faults
- Message passing
  - Message passing in hardware, but matching/buffering in software
- Issue of naming arises at each abstraction level of a parallel architecture

FUNDAMENTAL DESIGN ISSUES: OPERATIONS

- Shared memory
  - Loads and stores on addresses and registers (CISC), only registers (RISC)
  - Reading/writing shared variables
  - Atomic read-modify-write operations on shared variables
- Message passing
  - Sending/receiving on (private) local addresses and process identifiers
  - Collective operations
- Note the complexity difference

FUNDAMENTAL DESIGN ISSUES: ORDERING

Shared memory

- Threads operate independently, so which order to apply?
- Among memory operations: sequential program order
- Variables are read and modified: top-to-bottom, left-to-right order of the program

Message passing

- MPI guarantees strong ordering
- Tag matching, matching results in linear search(es)
- Receive any tag/sender will just return the first matched queue entry

Ordering has big performance impact

- Relaxed ordering models

SHARED MEMORY MULTIPROCESSORS

Multiple execution contexts sharing a single address space

- Multiple processes/threads, multiple data (MIMD)
- Simplification: Single Program Multiple Data (SPMD)

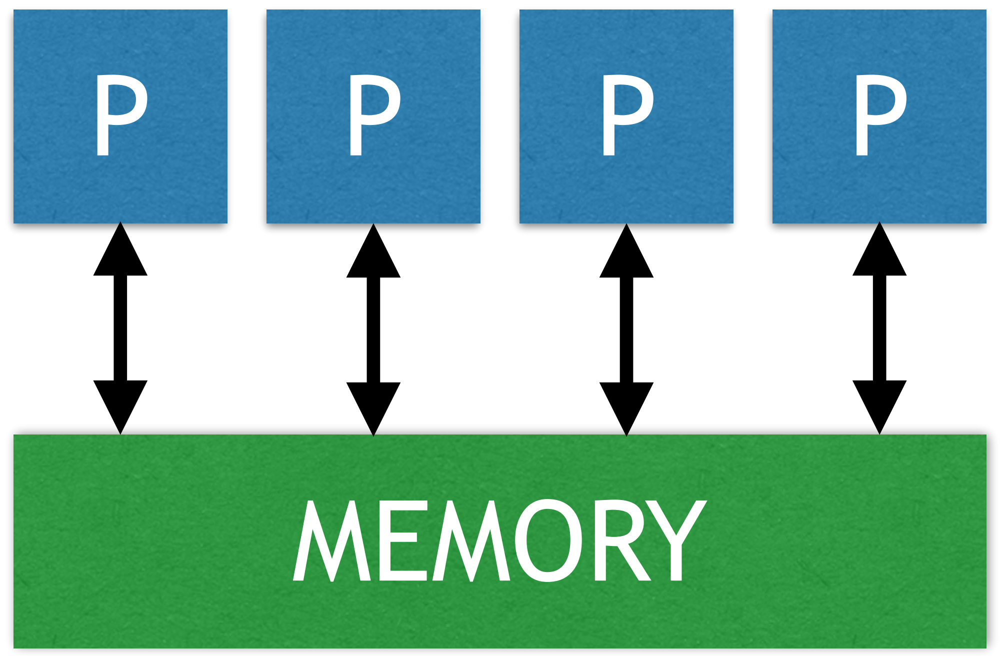
Parallelism type: TLP, DLP

Advantages:

- Applications: looks like multi-threaded uniprocessor
- OS: only evolutionary extensions required
- OS-bypass for communication
- Software development: first correctness, then performance

Disadvantages:

- Synchronization is very difficult
- Implicit communication is harder to optimize



Theoretical foundation:
Parallel Random Access Machine (PRAM)

Symmetric Multiprocessors (SMP) and Chip Multiprocessors (CMP) are the most successful parallel machines ever

COHERENCE AND CONSISTENCY BASICS

RECAP: THE COHERENCE PROBLEM

Caches

- Reduce average memory access latency
- Mind the 3C of cache (in-)effectivity

Caches have to be kept coherent

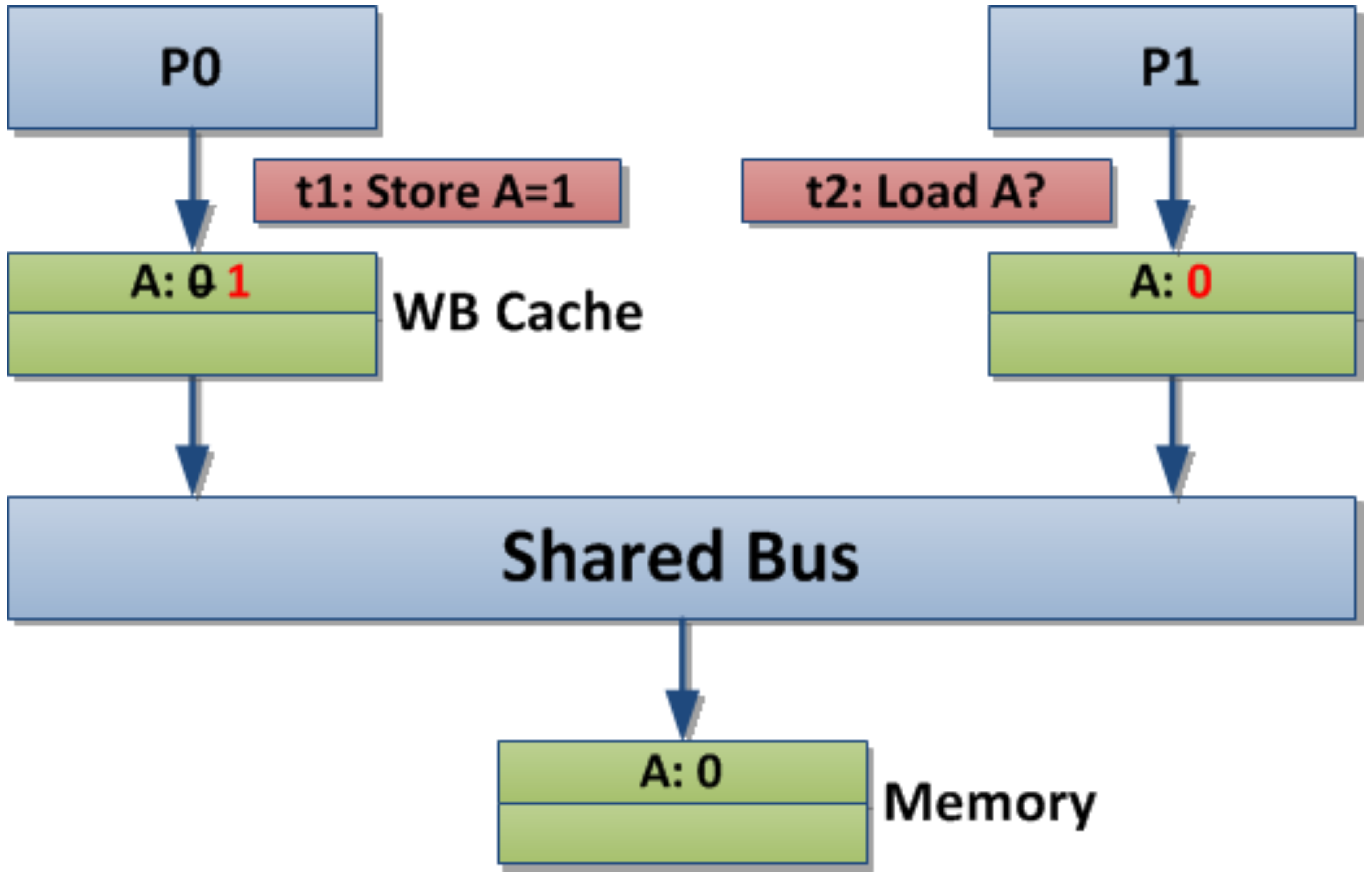
- Ensure that all Ps see the same (most recent) value

Write-back (WB) policy

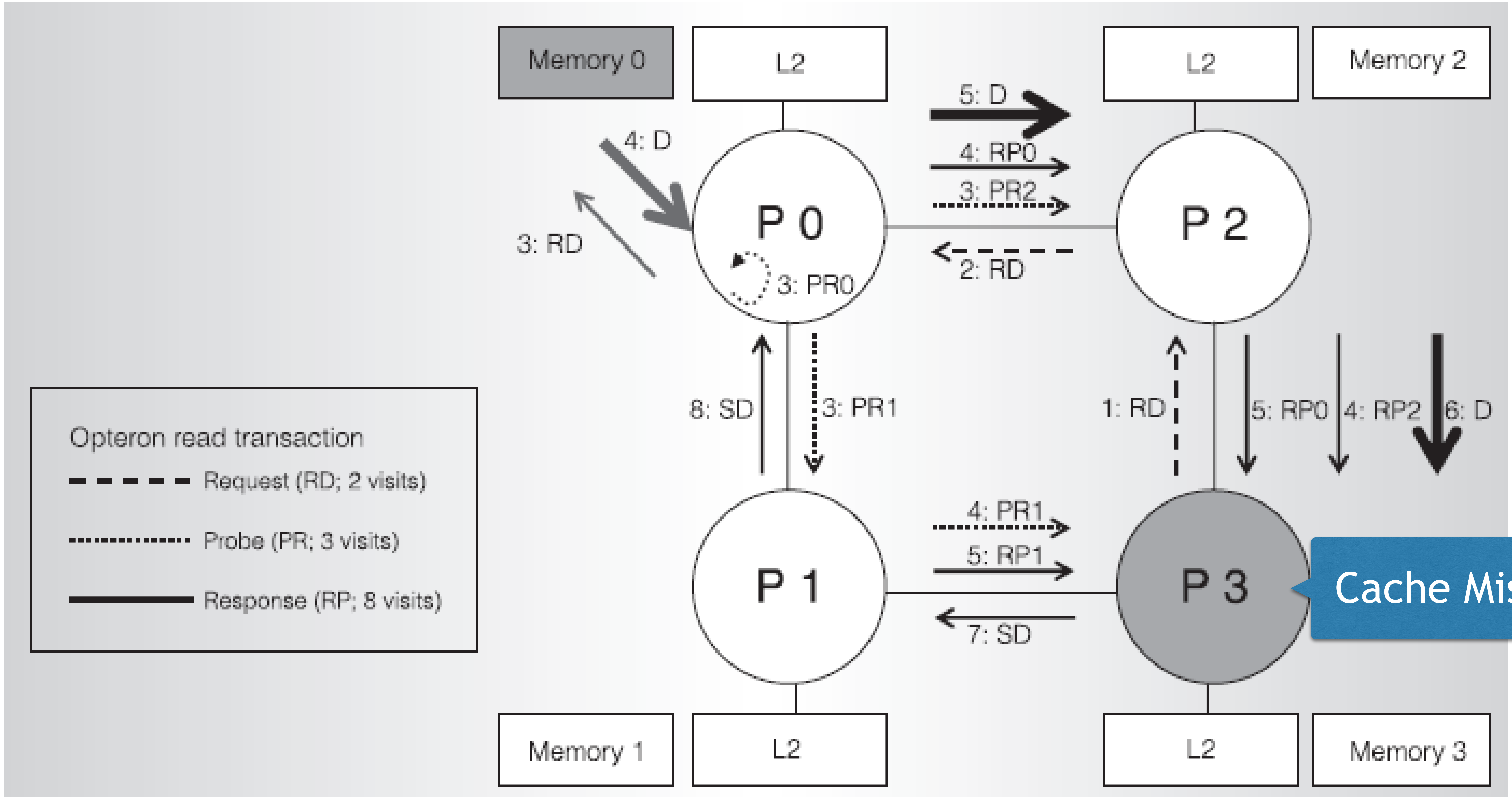
- Coherence problem?

Write-through (WT) policy

- Coherence problem?



COHERENCE PROTOCOL FOR AMD64



Pat Conway and Bill Hughes. 2007. The AMD Opteron Northbridge Architecture. IEEE Micro 27, 2 (March 2007), 10-21.

PROBLEMS WITH SCALABLE CACHE COHERENCE

- Aspect 1: Bandwidth
  - Bus as a shared medium is not scalable at all
  - Replace bus with a switched network (direct or indirect)
- Aspect 2: Snooping overhead
  - Interesting: most snoops result in no action
  - Simply because no copy of the corresponding cache line is present
  - Broadcast protocol is not scalable
  - Revert to a directory protocol, only addressing processors that hold cache line copies (broadcast/multicast)

COHERENCE VS. CONSISTENCY (1)

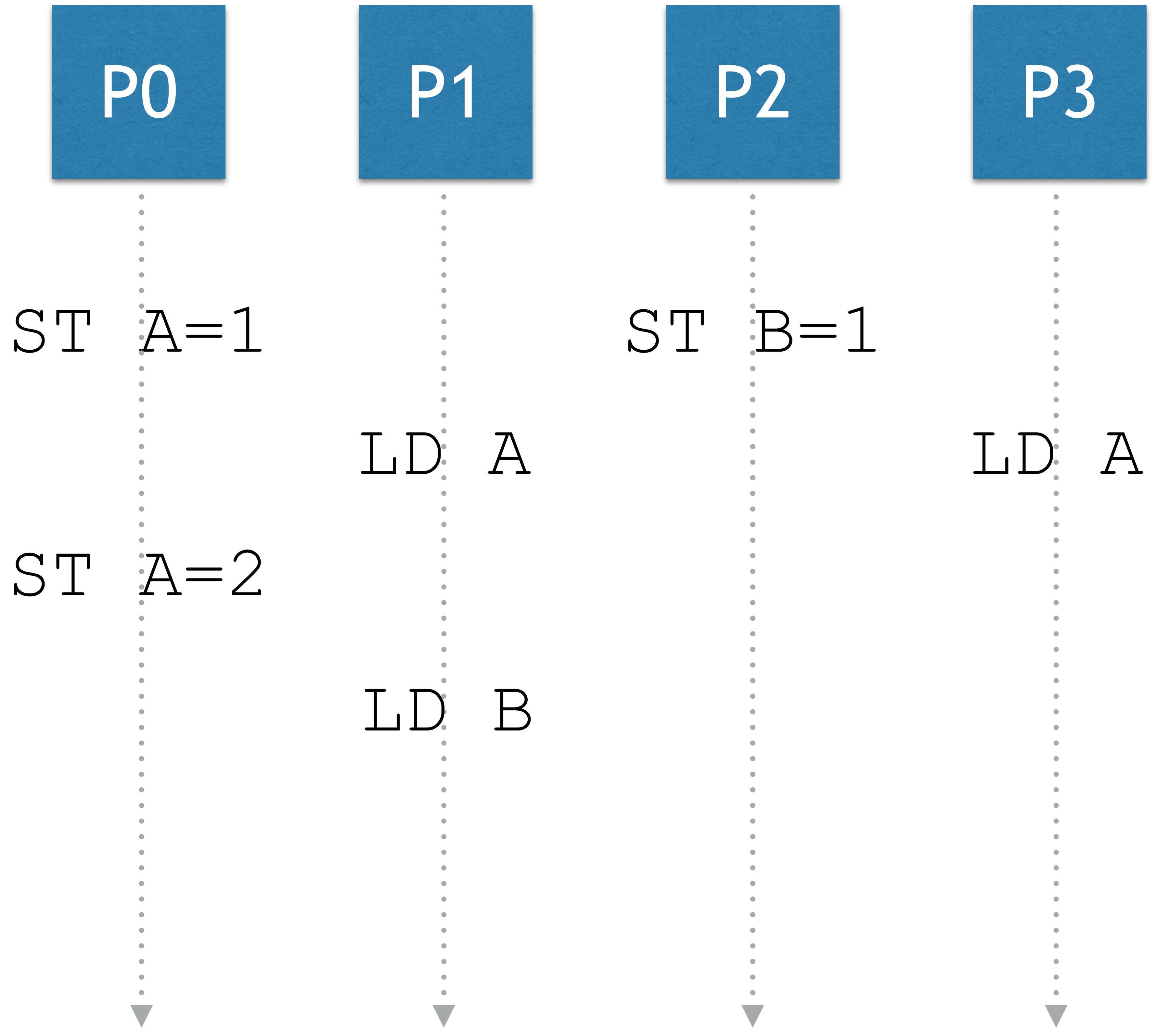
Memory coherence

- Operation serialization
- -> maintained "program order"
- -> read returns last write

Stores to the same address should be seen by all processors in the same order

Writes to an address by a processor will eventually be observed by other processors (question is "when")

Coherence is not visible to software

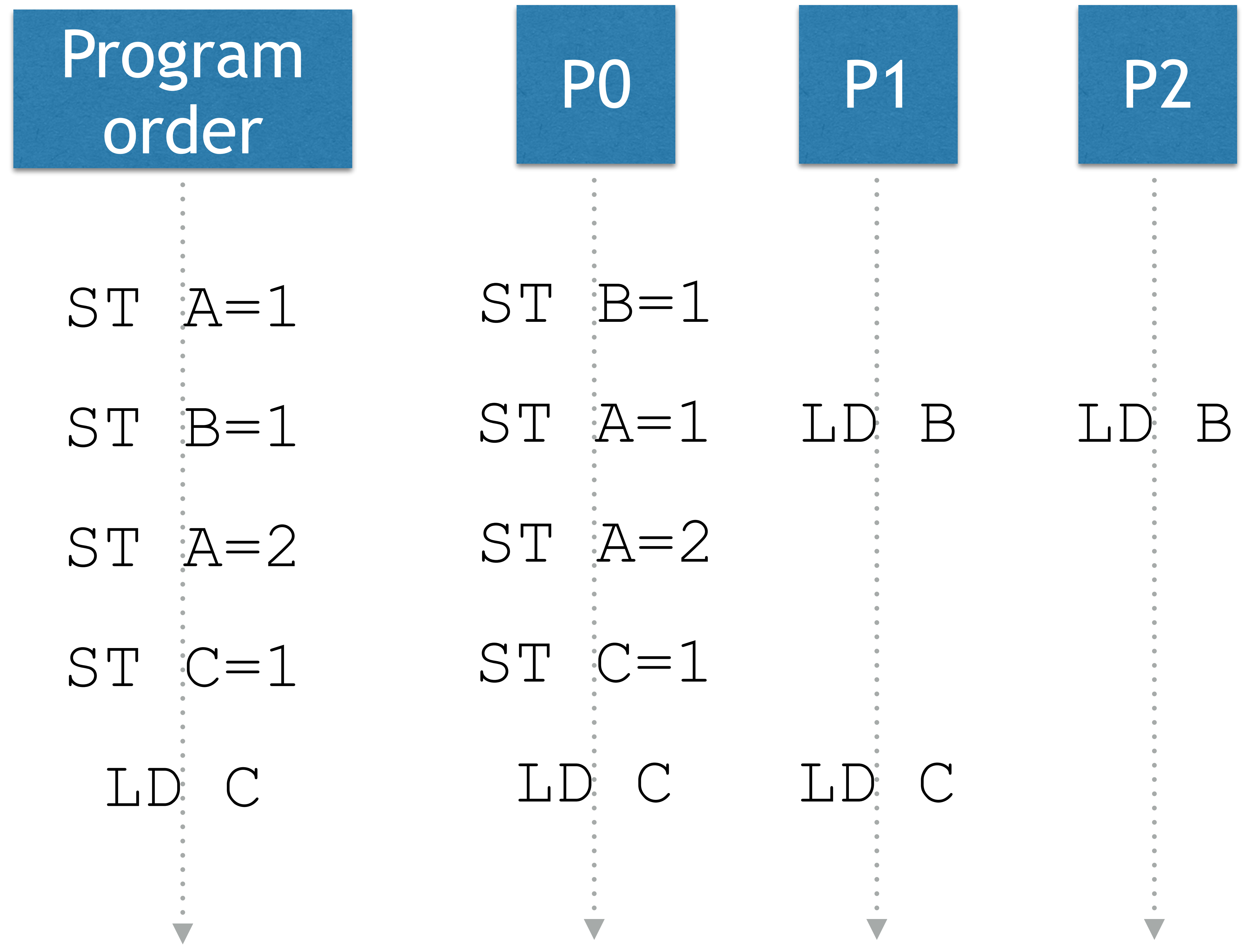


COHERENCE VS. CONSISTENCY (2)

A consistency model defines constraints on the order in which memory operations must appear to be performed (become visible)

Affects operations to the same location (address) and to different locations

Consistency is visible to software



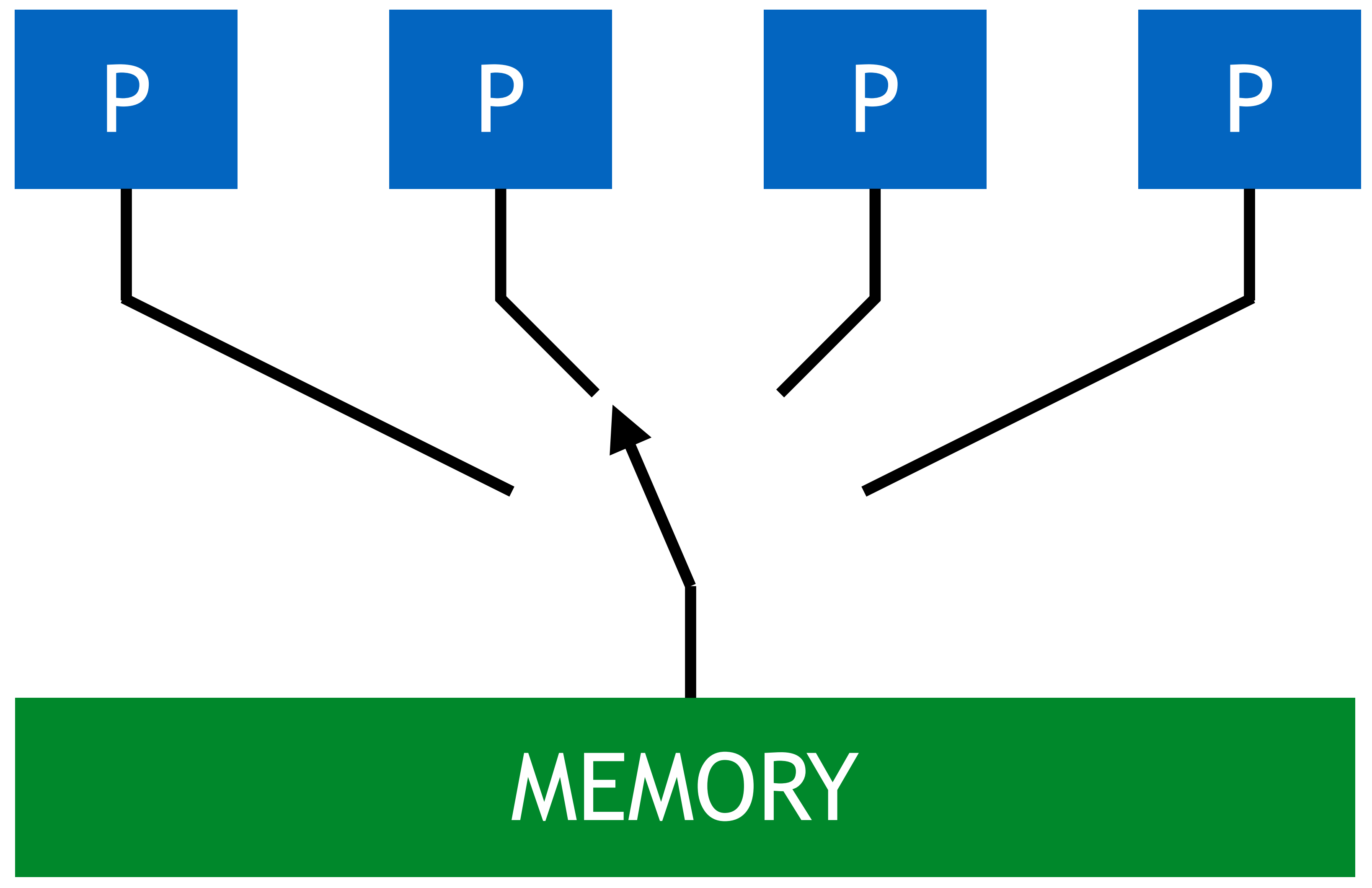
Program order	P0	P1	P2
ST A=1	ST B=1		
ST B=1	ST A=1	LD B	LD B
ST A=2	ST A=2		
ST C=1	ST C=1		
LD C	LD C	LD C	

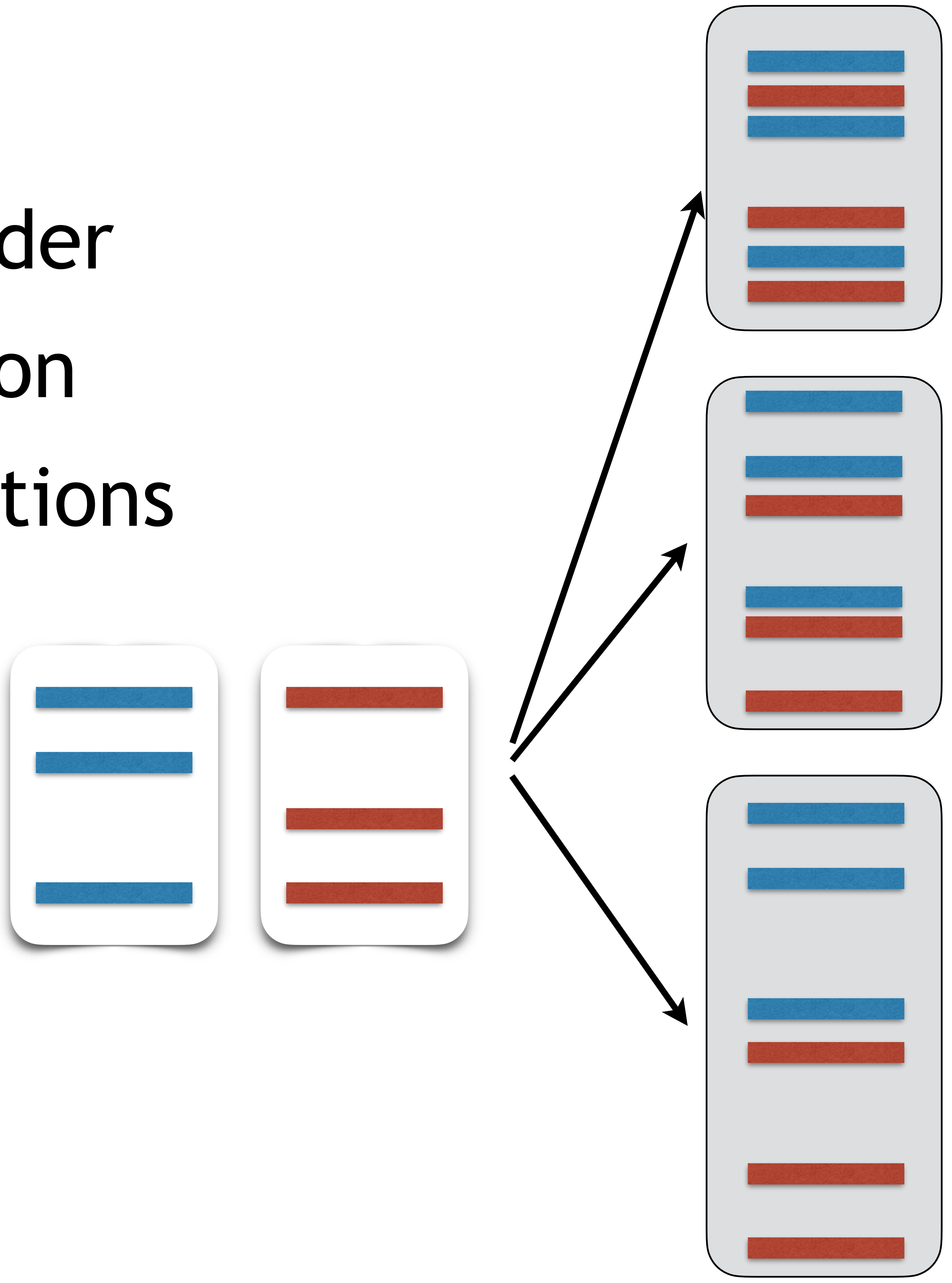
SEQUENTIAL CONSISTENCY

Processors issue memory requests in program order

Switch set randomly after each memory operation

=> Provides sequential ordering among all operations





SEQUENTIAL CONSISTENCY

Sufficient condition for SC:

- *„A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*“ – Lamport, 1979

Every processor issues memory requests in program order

Memory operations happen (start and end) atomically

- Must wait for a store to complete before issuing next operation
- After a load, issuing processor waits for load to complete, before issuing next operation

Easily implemented with a shared bus

- Bus as **synchronization point**, serializing all accesses

PROBLEMS WITH SEQUENTIAL CONSISTENCY

Aspect 1: difficult to implement efficiently in hardware

- No concurrency among memory access
- Strict ordering of memory accesses at each processor (node)
- Essentially precludes out-of-order CPUs

Aspect 2: unnecessarily restrictive

- Most parallel programs won‘t notice out-of-order accesses

Aspect 3: conflicts with latency hiding techniques

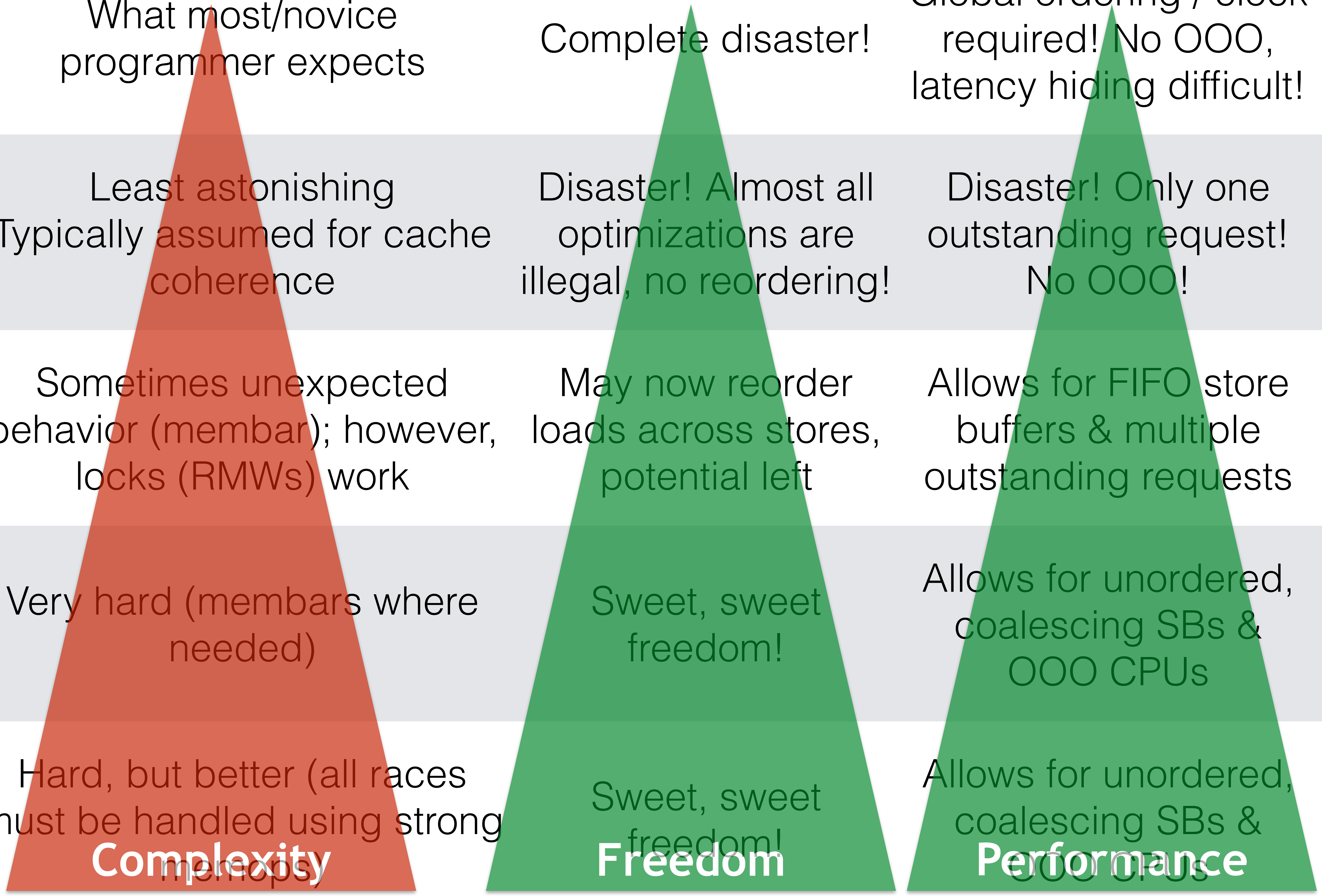
- Which relies on many concurrent outstanding requests

Fixing SC performance

- Revert to a less strict consistency model (relaxed or weak consistency)
- Programmer specifies when ordering matters

	Programmer	Compiler	Hardware	Comment
Strict Consistency	What most/novice programmer expects	Complete disaster!	Global ordering / clock required! No OOO, latency hiding difficult!	Only for uniprocessors
Sequential Consistency	Least astonishing Typically assumed for cache coherence	Disaster! Almost all optimizations are illegal, no reordering!	Disaster! Only one outstanding request! No OOO!	Overkill, most programmers rely on synchronization intrinsics!
Processor Consistency	Sometimes unexpected behavior (membar); however, locks (RMWs) work	May now reorder loads across stores, potential left	Allows for FIFO store buffers & multiple outstanding requests	Typical today x86
Relaxed Consistency (WC,RC,EC)	Very hard (membars where needed)	Sweet, sweet freedom!	Allows for unordered, coalescing SBs & OOO CPUs	
Data-Race-Free (DRF)	Hard, but better (all races must be marked using strong memops)	Sweet, sweet freedom!	Allows for unordered, coalescing SBs & OOO CPUs	Java, C++

	Programmer	Compiler	Hardware	Comment
Strict Consistency	What most/novice programmer expects	Complete disaster!	Global ordering / clock required! No OOO, latency hiding difficult!	Only for uniprocessors
Sequential Consistency	Least astonishing Typically assumed for cache coherence	Disaster! Almost all optimizations are illegal, no reordering!	Disaster! Only one outstanding request! No OOO!	Overkill, most programmers rely on synchronization intrinsics!
Processor Consistency	Sometimes unexpected behavior (membar); however, locks (RMWs) work	May now reorder loads across stores, potential left	Allows for FIFO store buffers & multiple outstanding requests	Typical today x86
Relaxed Consistency (WC,RC,EC)	Very hard (membars where needed)	Sweet, sweet freedom!	Allows for unordered, coalescing SBs & OOO CPUs	
Data-Race-Free (DRF)	Hard, but better (all races must be handled using strong membars)	Sweet, sweet freedom!	Allows for unordered, coalescing SBs & OOO CPUs	Java, C++

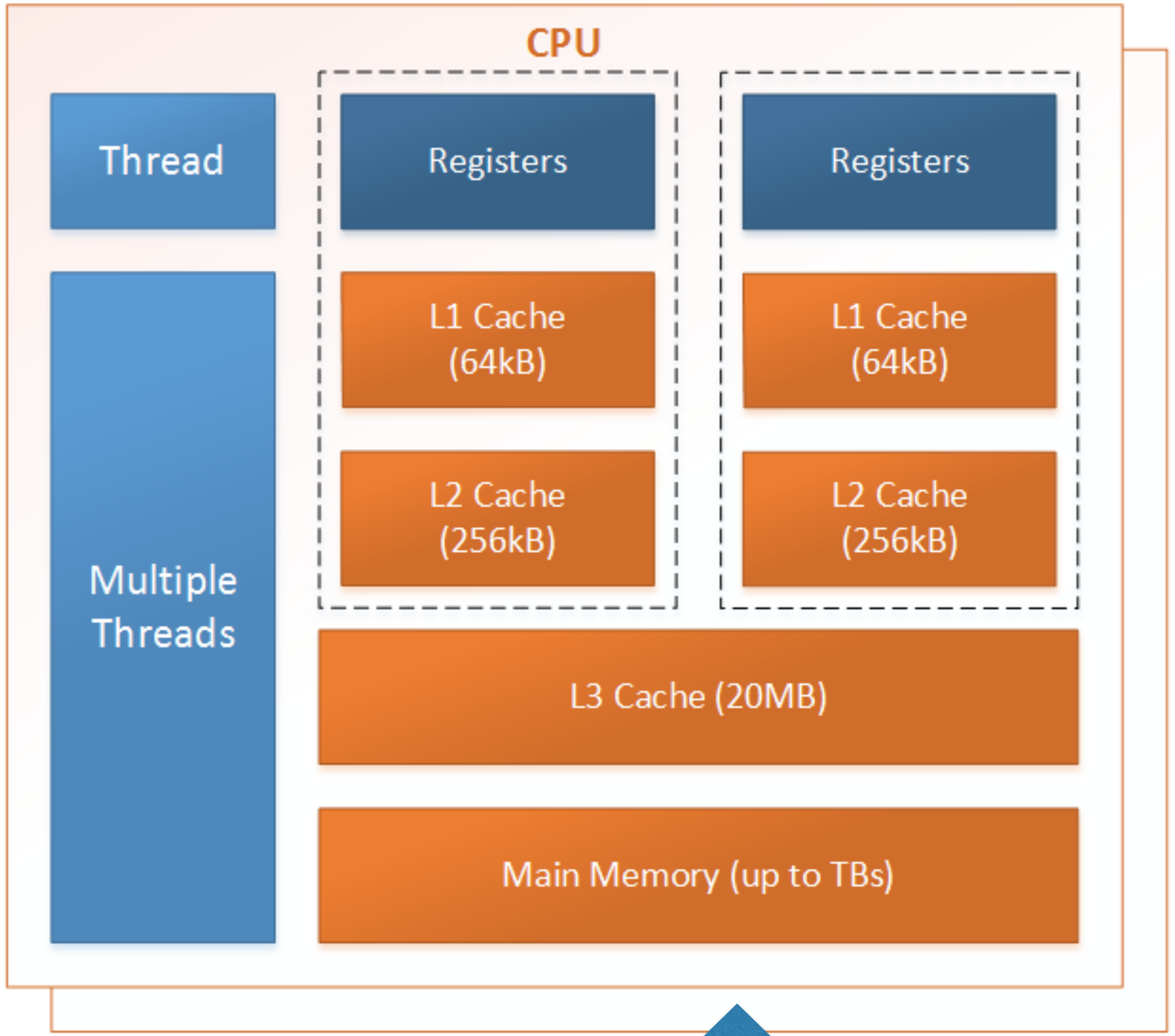


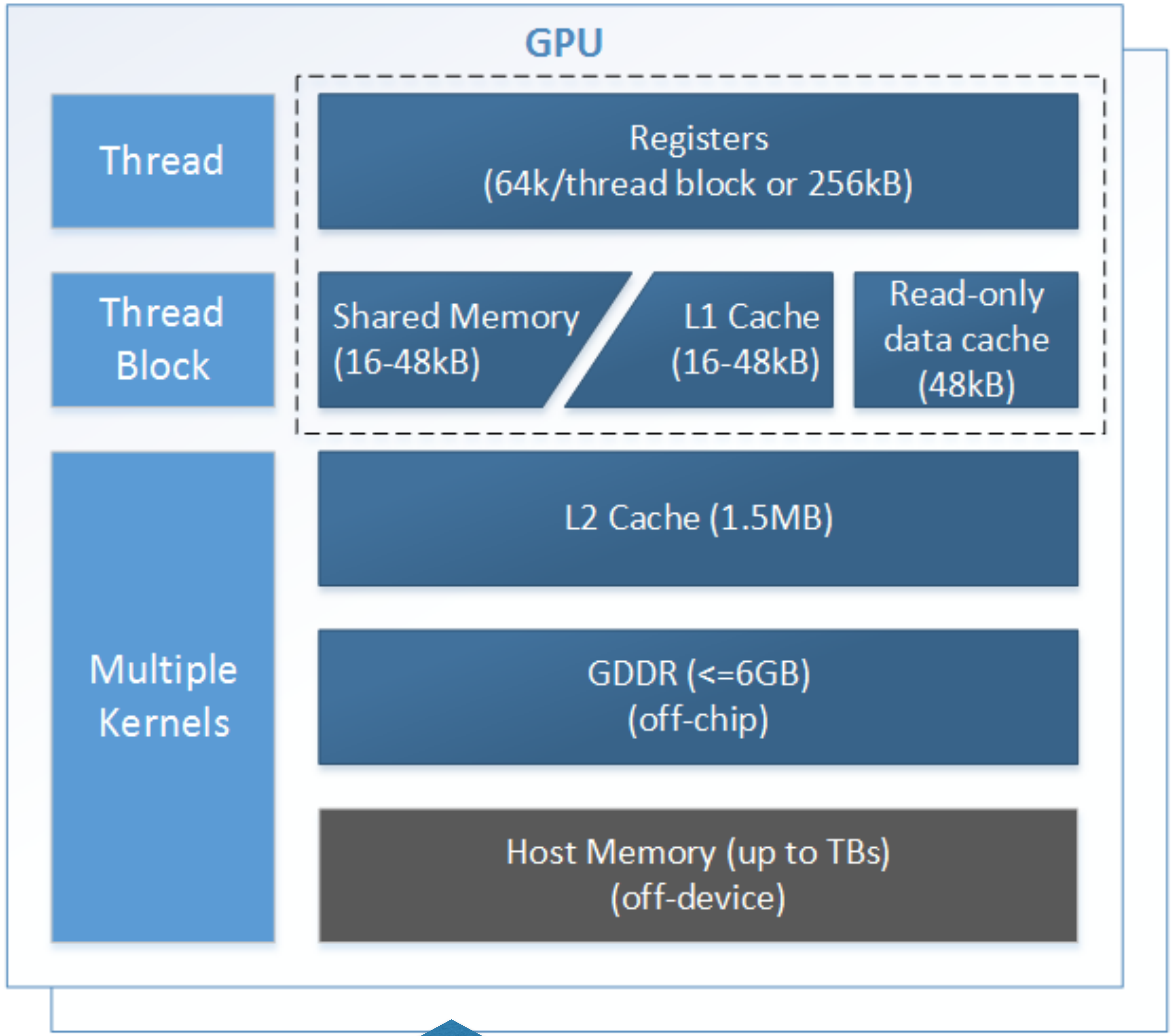
IN A NUTSHELL

- Coherence is a super expensive protocol hiding architecture details
  - Providing the illusion of one big central cache based on physically distributed caches
  - Costs scale with the number of endpoints (processors) -> conflictive with multi-/many-core
- Consistency is a contract in between programmer and architecture
  - Similar to the Instruction Set Architecture (ISA), but regarding ordering and visibility of memory operations
  - Costs of strong consistency scale with number of endpoints (processors) and memory parallelism (memory controllers) -> conflictive with multi-/many-core

GPU COHERENCE & CONSISTENCY MODEL

COHERENCE IN CPUS & GPUS





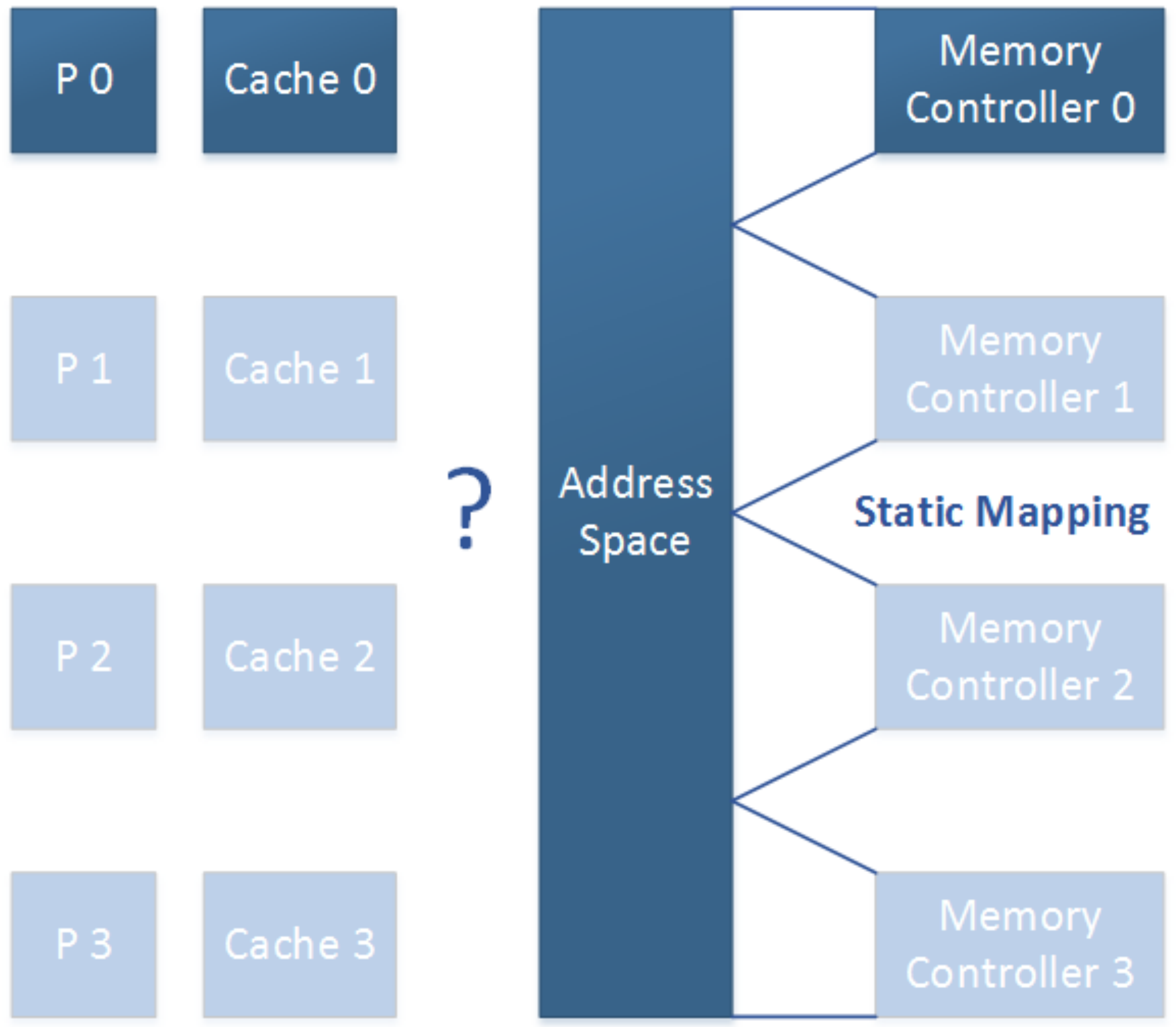
Deep and steep memory hierarchy
Exclusive L1/L2, shared LLC
Requires cache coherence

🟦 not coherent
🟧 coherent

Flat memory hierarchy
Exclusive L1, shared L2
Survives without cache coherence?

ADDRESS SPACE VIEW - UNIPROCESSOR

- Uniprocessor – single memory controller
  - Even with caches no coherence problem
- Homonyms: same name for different data
  - Which cache line belongs to which process
  - On process switch, flush caches (WBINVD) or use ID-tagged cache
  - Address space identifier (ASID)



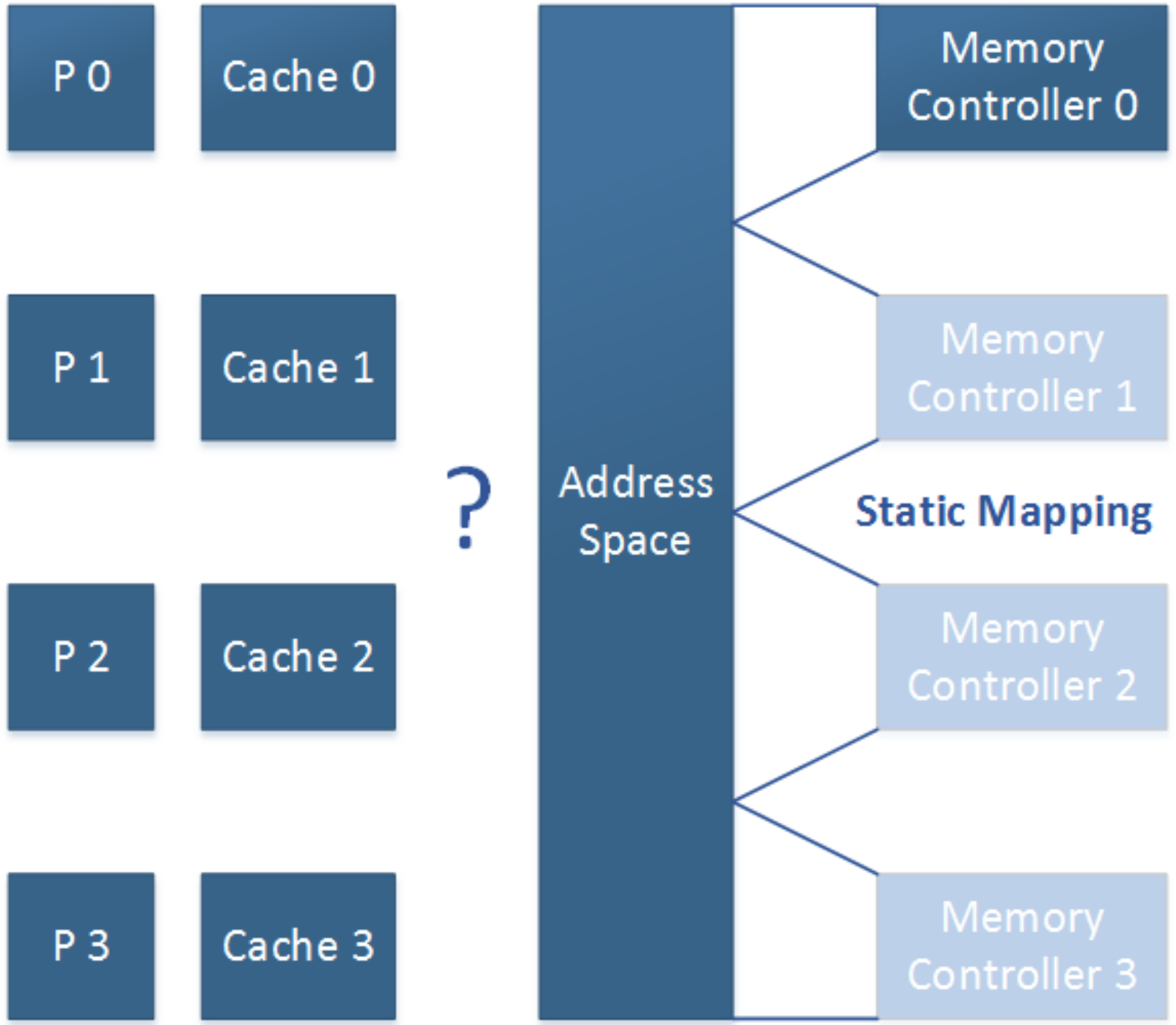
ADDRESS SPACE VIEW - MULTIPROCESSOR

Multiprocessor – single memory controller

- Multiple caches -> multiple copies possible
- Coherence protocol required

Memory controller acts as synchronization point

- Is responsible for appropriate coherence actions



ADDRESS SPACE VIEW - MULTIPROCESSOR

Multiprocessor – multiple memory controllers

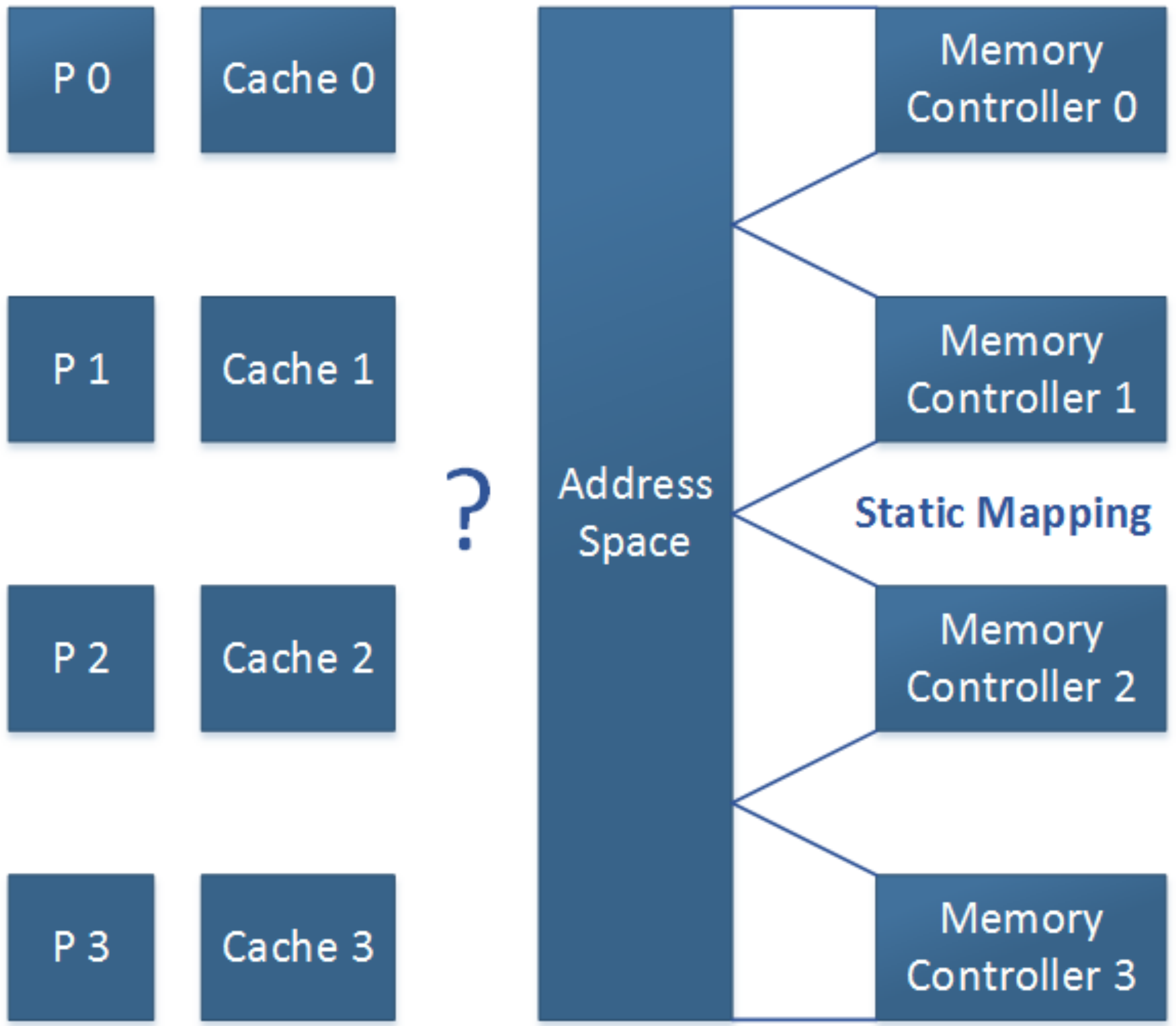
- Now multiple memory controllers, all act as synchronization point

Which MC is responsible?

- Identified by static mapping

Excursion: COMA

- No static mapping, main memory is a giant cache



ADDRESS SPACE VIEW - MULTIPROCESSOR

Multiprocessor – multiple memory controllers

Cache hierarchy

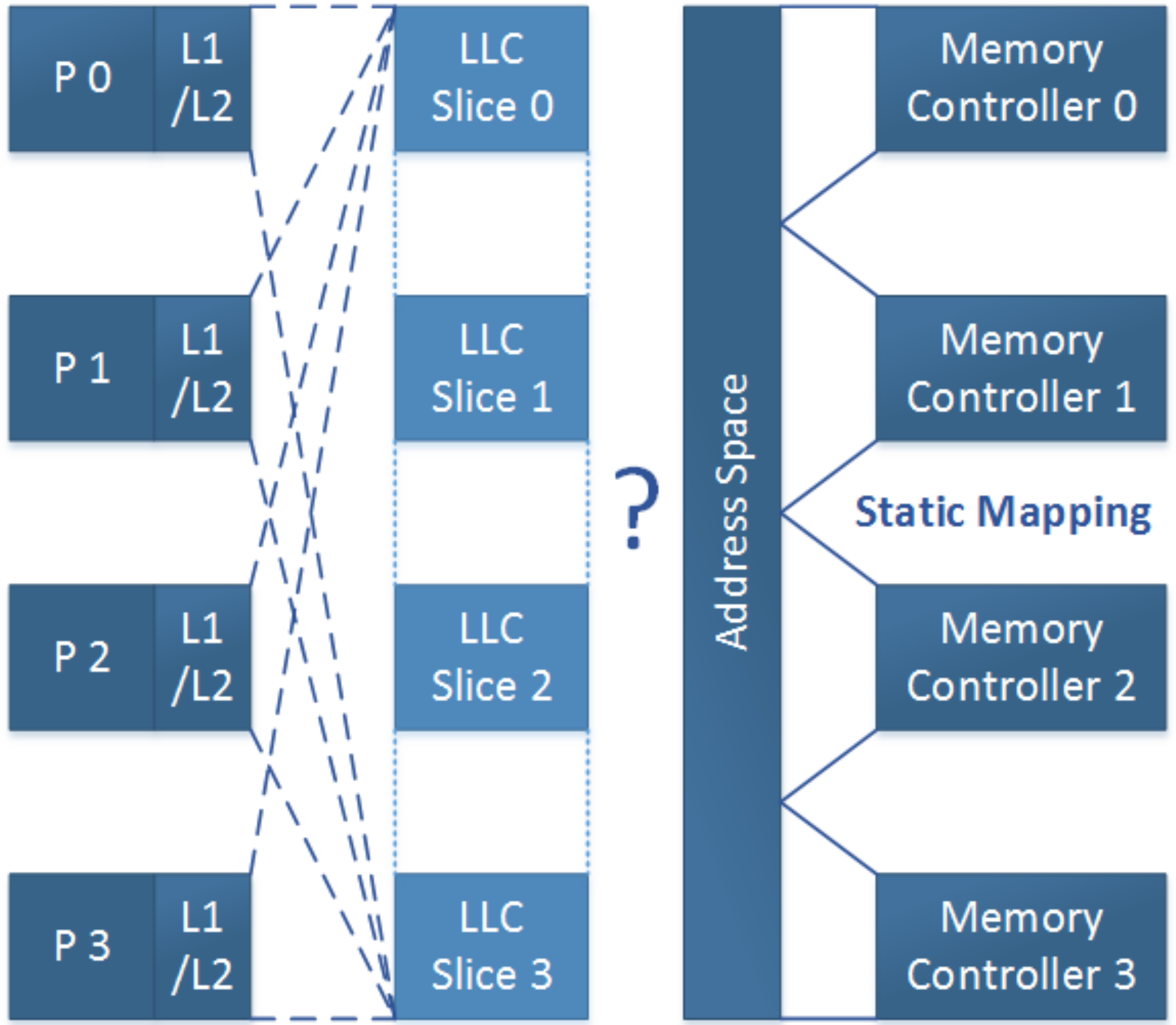
- Exclusive L1/L2 (per core)
- Shared L3/LLC

LLC cache is sliced (banked)

- Multiple concurrent accesses

No implications towards coherence

- Same principle as before

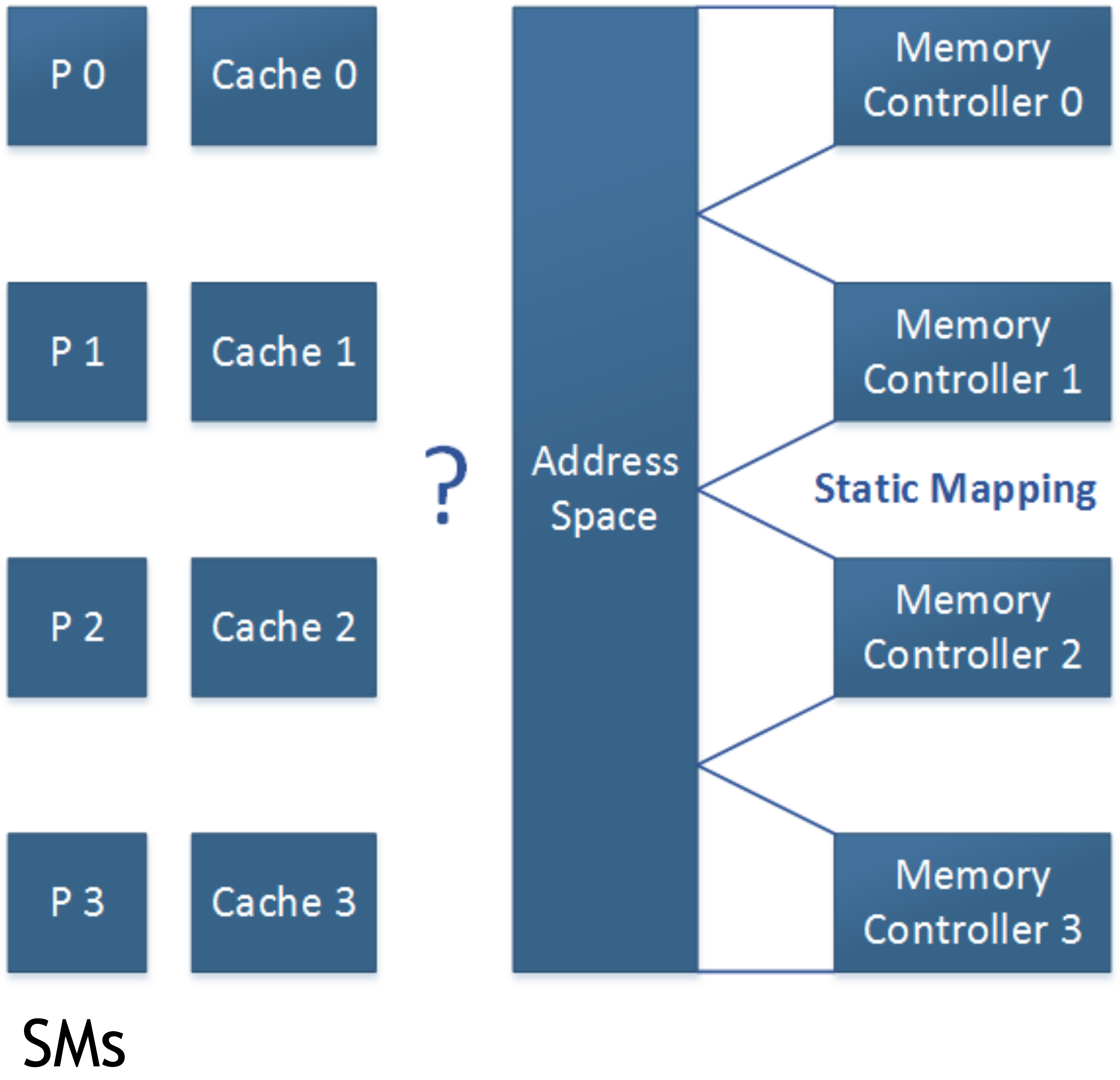


GPU MEMORY HIERARCHY – L2

GPU LLC cache is sliced, too

But why is no coherence required?

Remember that GPUs can *tolerate* memory latency



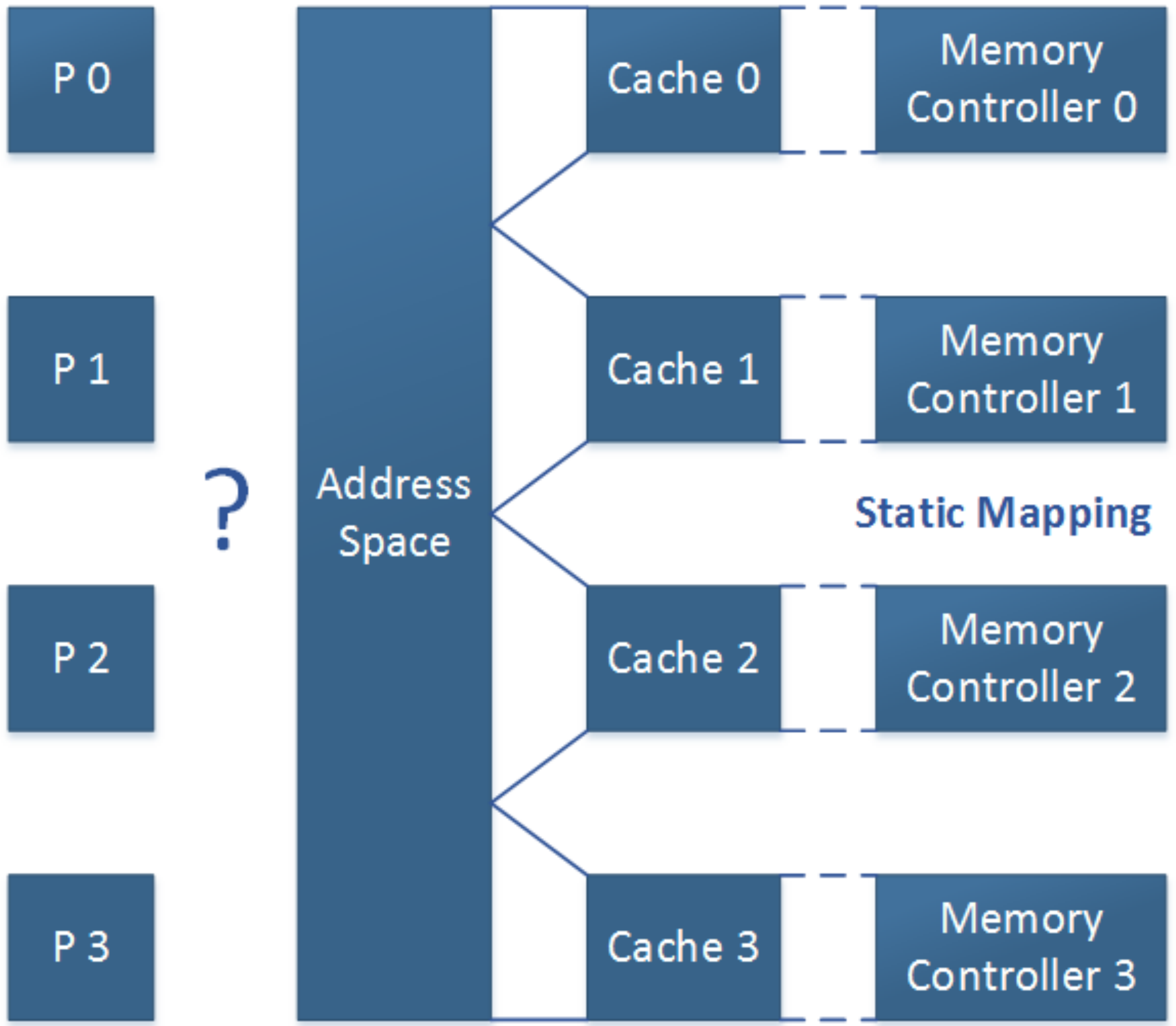
GPU MEMORY HIERARCHY – L2

LLCs are part of the fixed address mapping

- Latency increases significantly
- GPUs actually don't care
- CPUs would care

Effective cache size is reduced if data is not equally distributed among the MCs

- Cache size not as important for GPUs as for CPUs



GPU MEMORY HIERARCHY – L1

So far so good for LLC (L2)

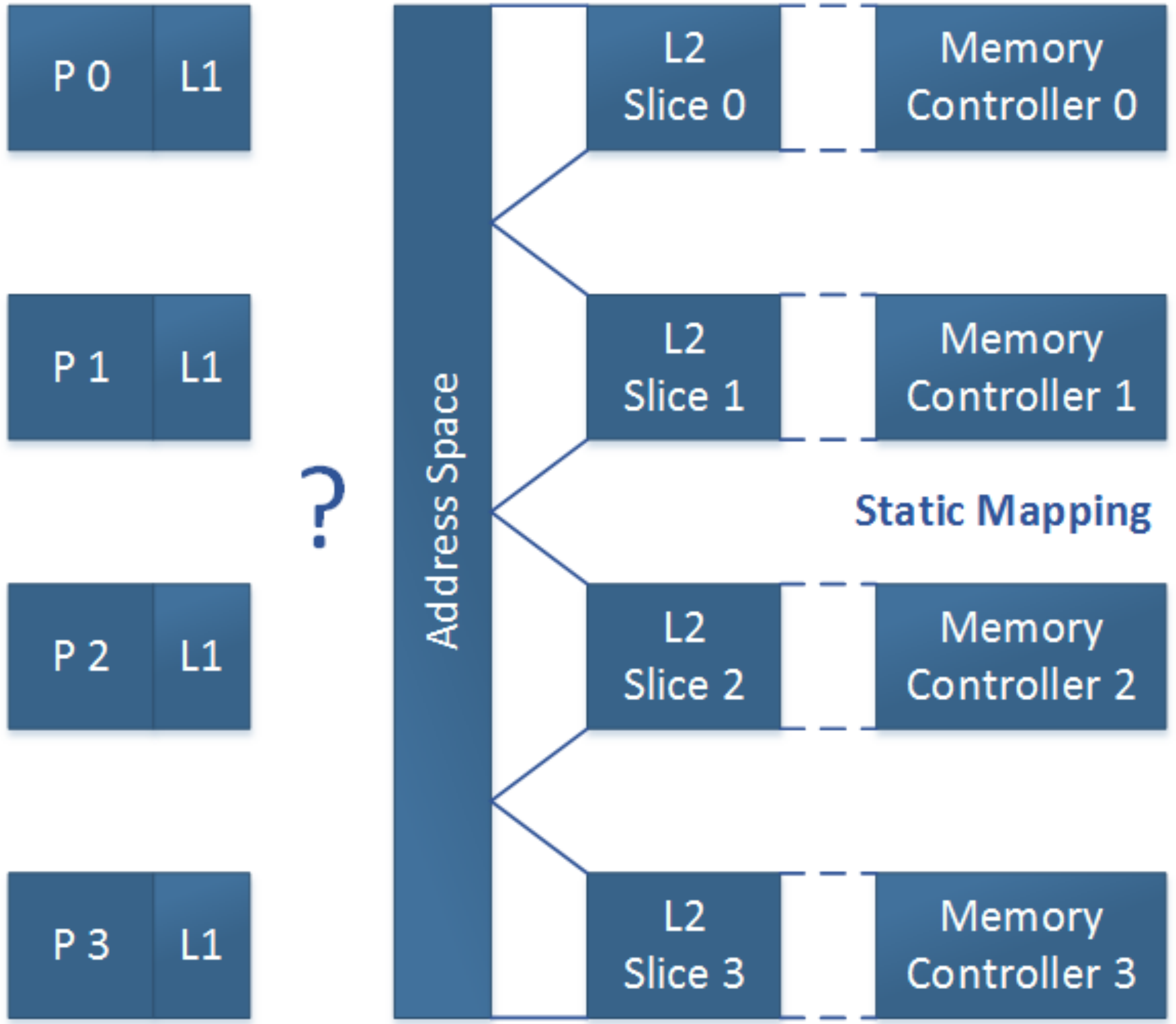
What about L1?

- Local to an SM/thread block

Exclusive cache, coherency guarantees only for the start/end of a thread block

- Making writes globally visible by write-through
- -> No need to write-back caches upon end-of-life
- Invalidating caches at kernel completion boundaries
- -> No memory traffic

-> Software-controlled coherence



CUDA SAFETY NET

Fences as memory barriers for fine-grained consistency control

`void __threadfence()`

- Separates all writes to shared memory and global memory:
- All writes before the call are visible to all threads on the device before the call completes
- Those after the call are not visible until the call is completed

`void __threadfence_block()`

- Same, but only for threads within the same thread block

`void __threadfence_system()`

- Same, but including pinned host memory and visibility for all threads on the device

GPU MEMORY ARCHITECTURE

Address-sliced crossbars

- High-bandwidth, contention-free path into memory

L1 Cache

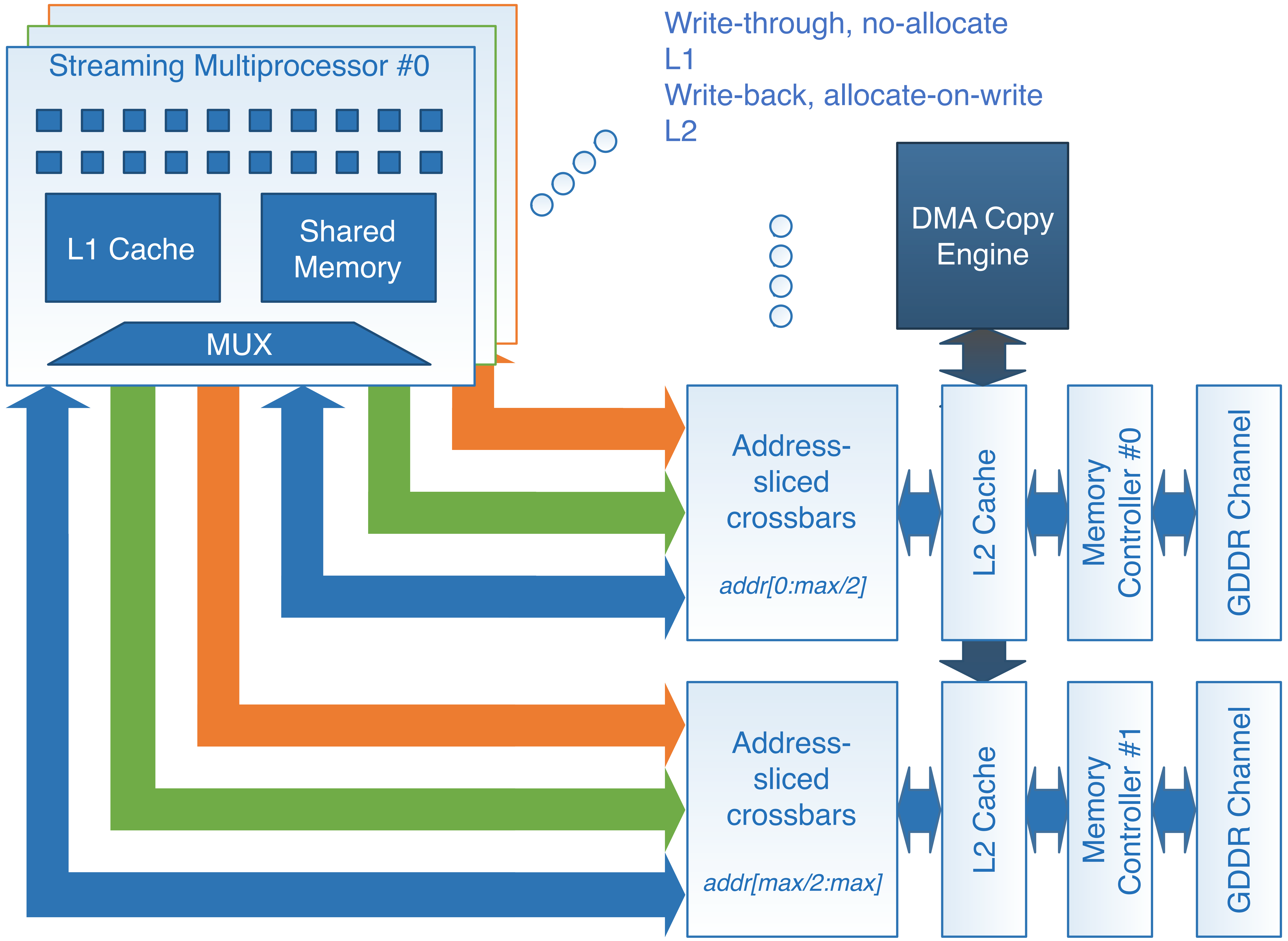
- 128B cache line size == 32 threads x 4B
- Write-invalidate, no write-allocate

L2 Cache (LLC)

- 32B cache line size (stores, over-fetch)
- Write-back, write-allocate

GPU kernels

- Write-once quite common
- -> no need for expensive cache fills



WRAPPING UP

SUMMARY

- It’s not about coherence: coherence is an artificial problem introduced by caches
  - CPUs make many guarantees about coherence, due to reasons including legacy codes, the user, and the unconstrained use model
  - Latency minimization prohibits moving the LLCs towards the memory controller
- It‘s all about consistency
  - Consistency for GPUs is highly relaxed, with few guarantees. Main reasons are that there are no legacy codes, and that the used model is constrained to BSP-like ones.
  - Synchronization points are essentially the start and end of life of a thread block
  - Latency toleration allows to live with little cache capacity and to move the LLCs to the MCs