CS-590.26, Spring 2014

High-Speed Memory Systems: Architecture and Performance Analysis

Coherence and Consistency

The Problem is Multi-Fold

Cache Consistency (taken from web-cache community)

**In the presence of a cache,
reads and writes behave (to a first order)
no differently than if the cache were not there**

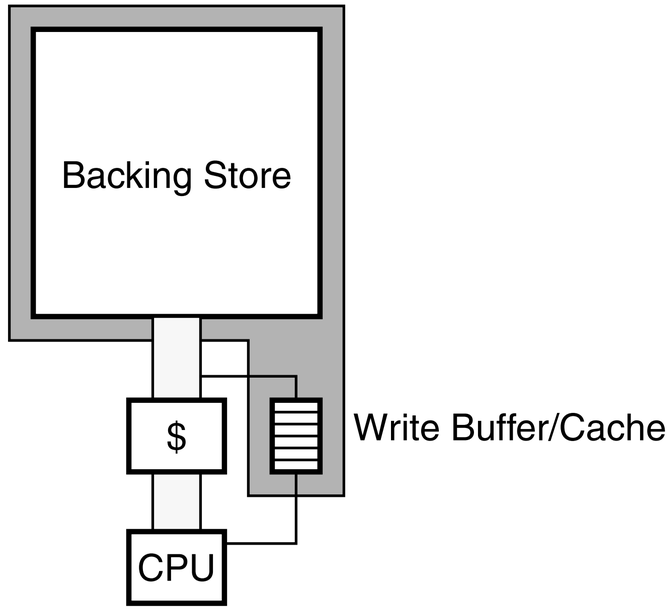
Three main issues:

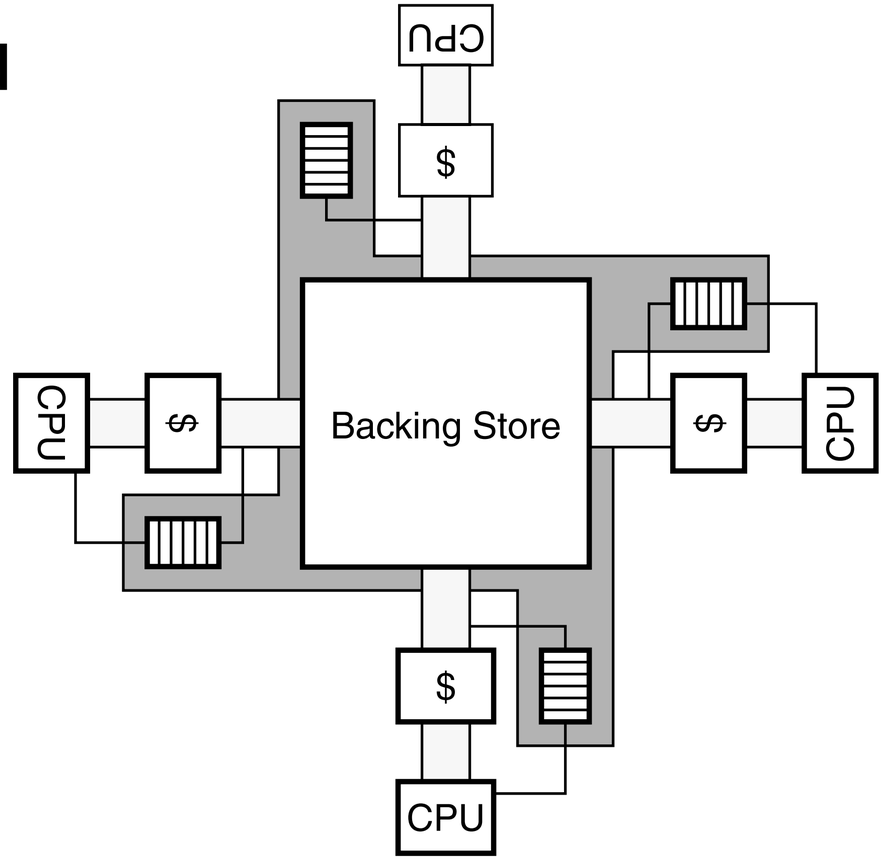
- **Consistent with backing store**
- **Consistent with self**
- **Consistent with other clients of same backing store**

Consistency w/ Backing Store

For example, write-through vs. write-back

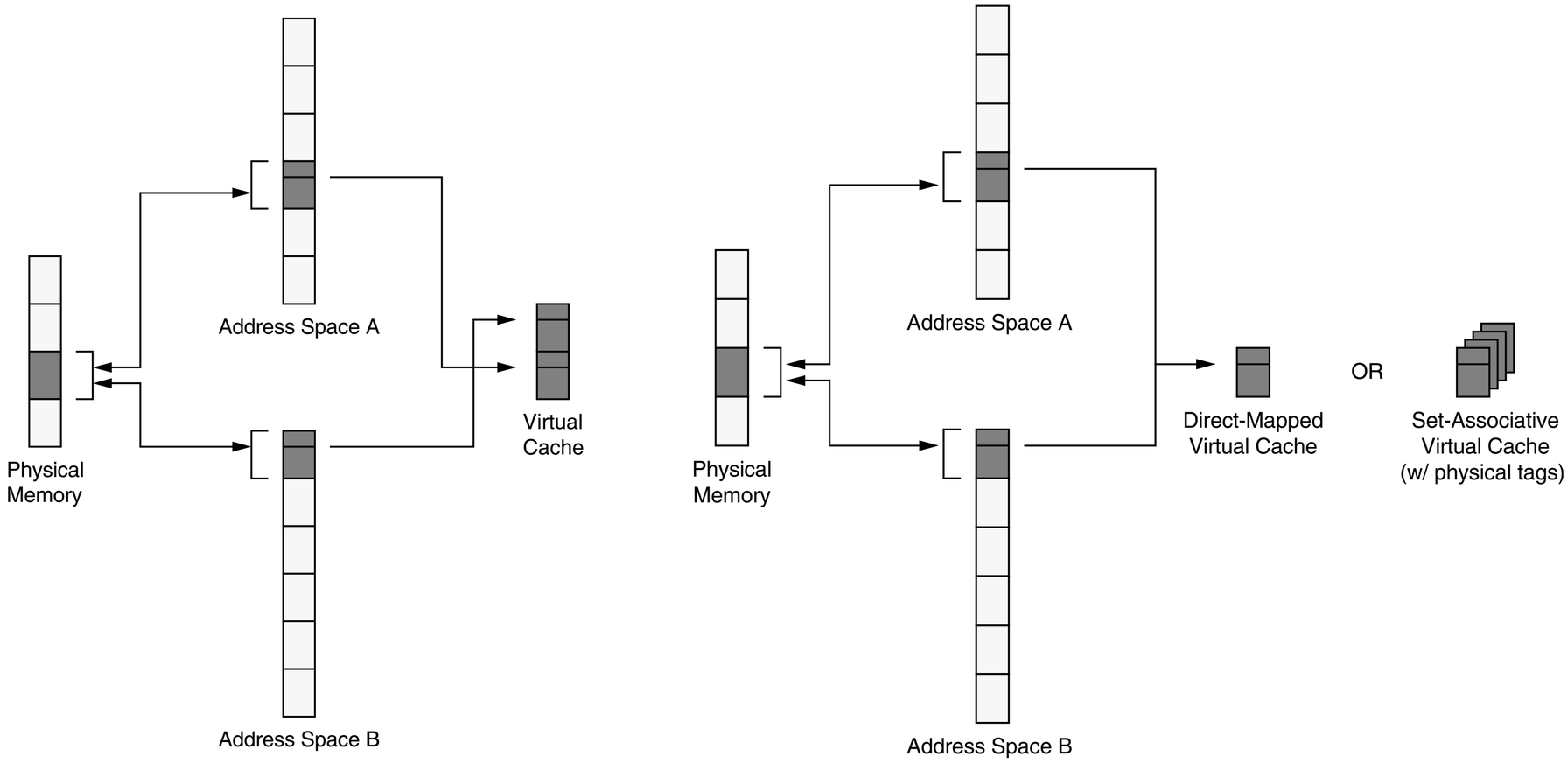
Write buffer commonly used in write-through caches:





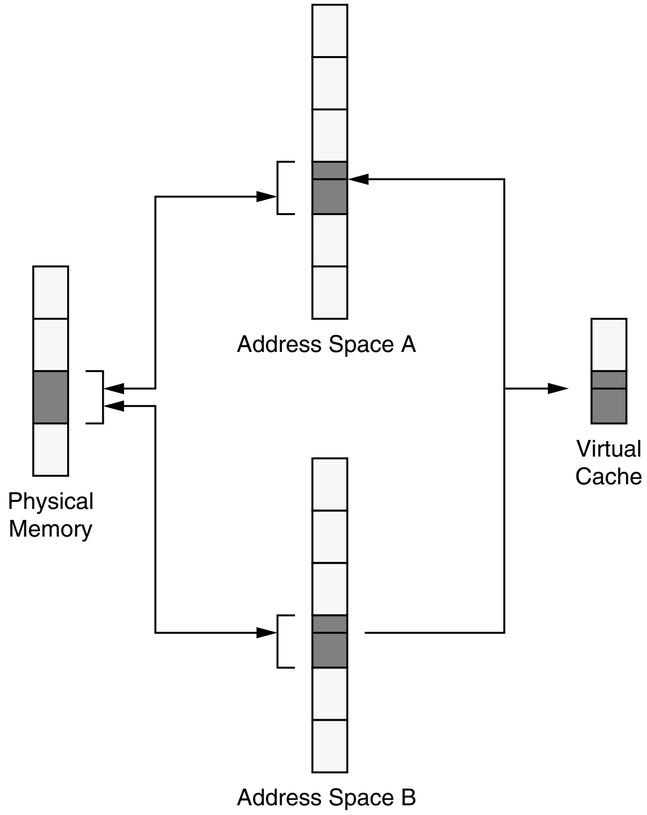
Consistency w Self

Virtual cache synonym problem & hardware solutions

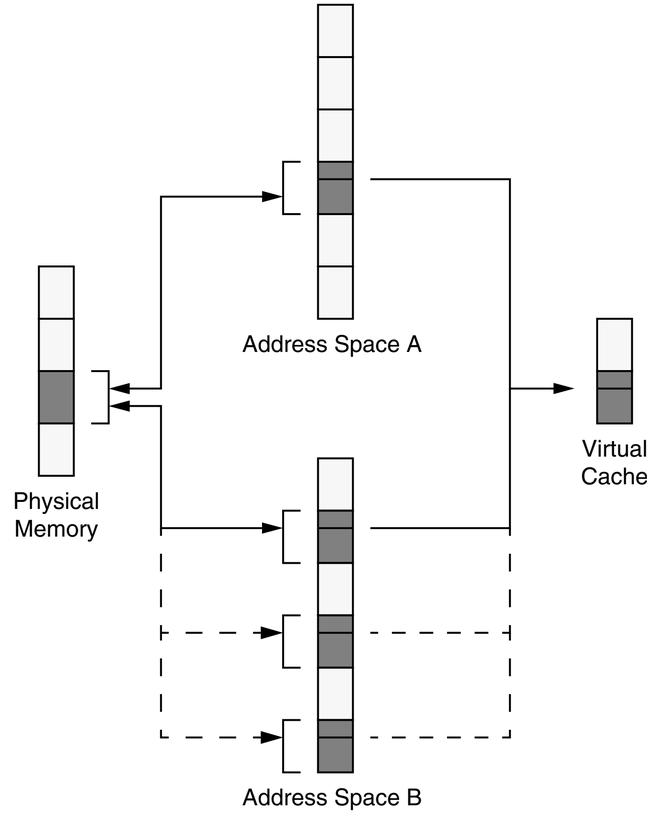


Consistency w Self

Operating system solutions to aliasing problem



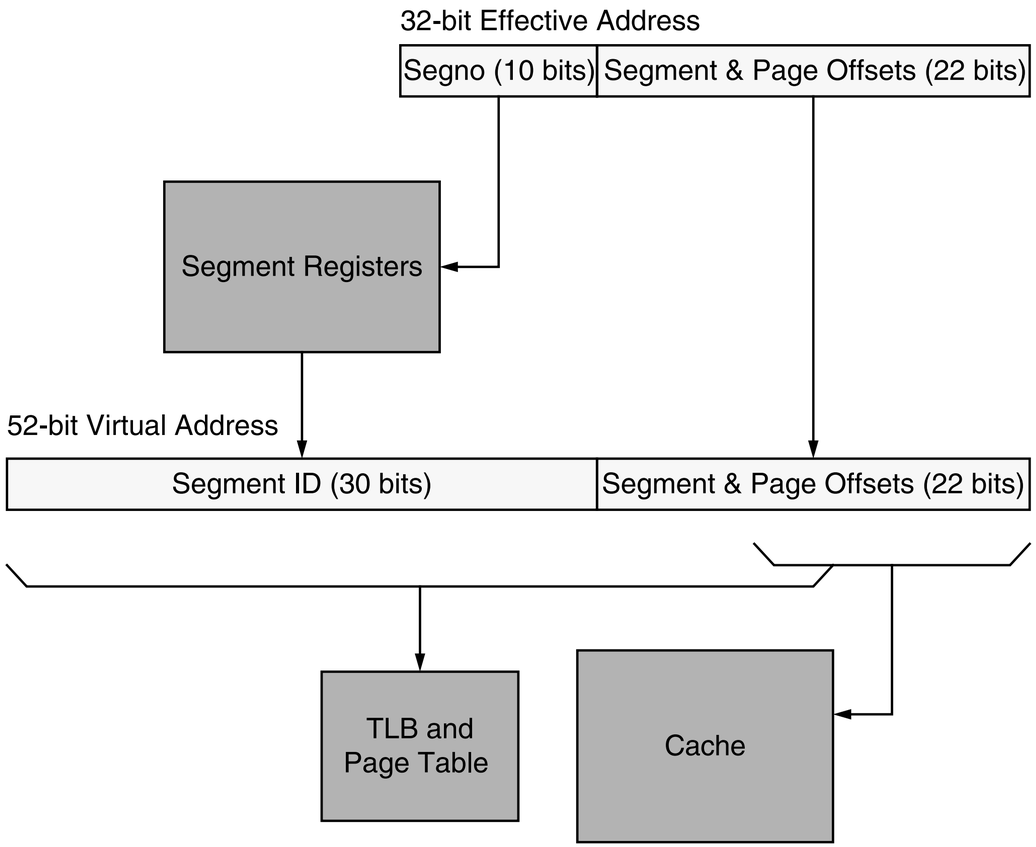
(a) SPUR and OS/2 solutions



(b) SunOS solution

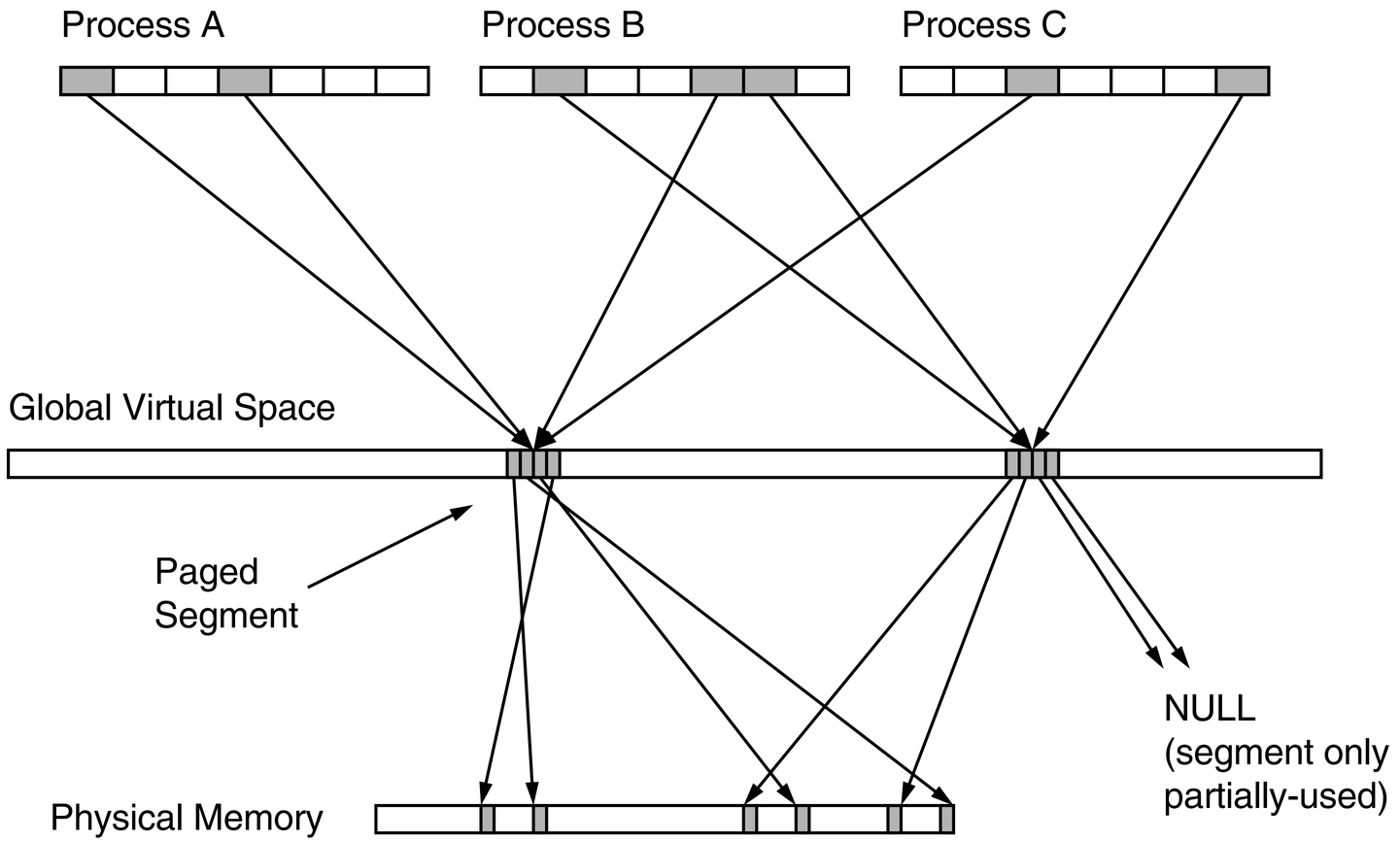
Consistency w Self

Segmentation as a solution to the aliasing problem



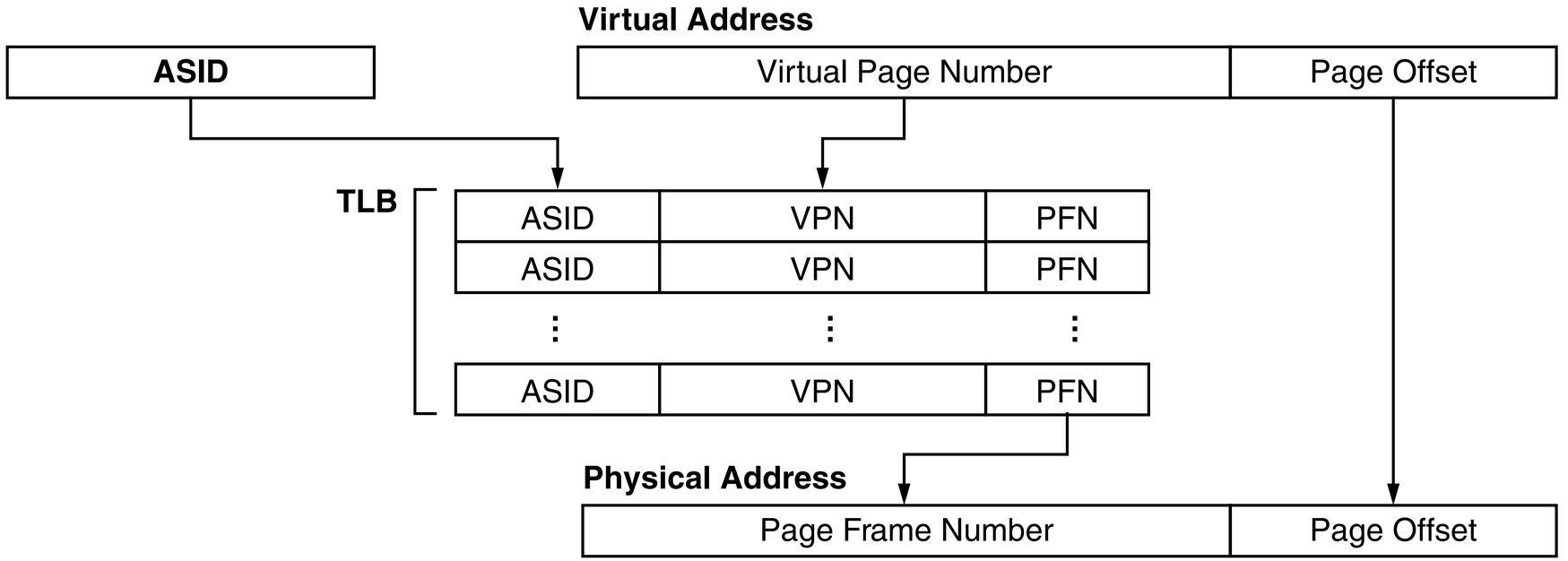
Consistency w Self

Segmentation as a solution to the aliasing problem



Consistency w Self

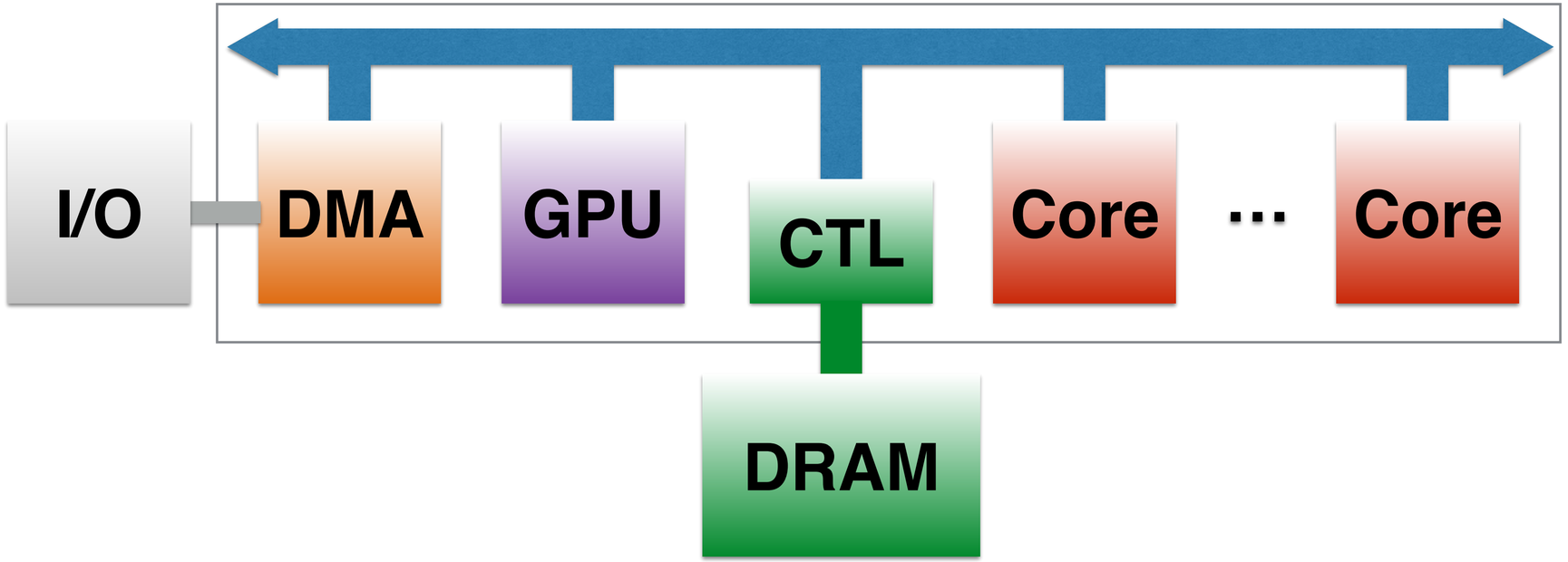
ASID remapping



Consistency w Other Clients

i.e. Cache Coherence & various Consistency Models

First, a look at some of the things that can go wrong, just inside a SINGLE CHIP:



Proc B reads data from dev A, signals proc C when done (producer-consumer pair)

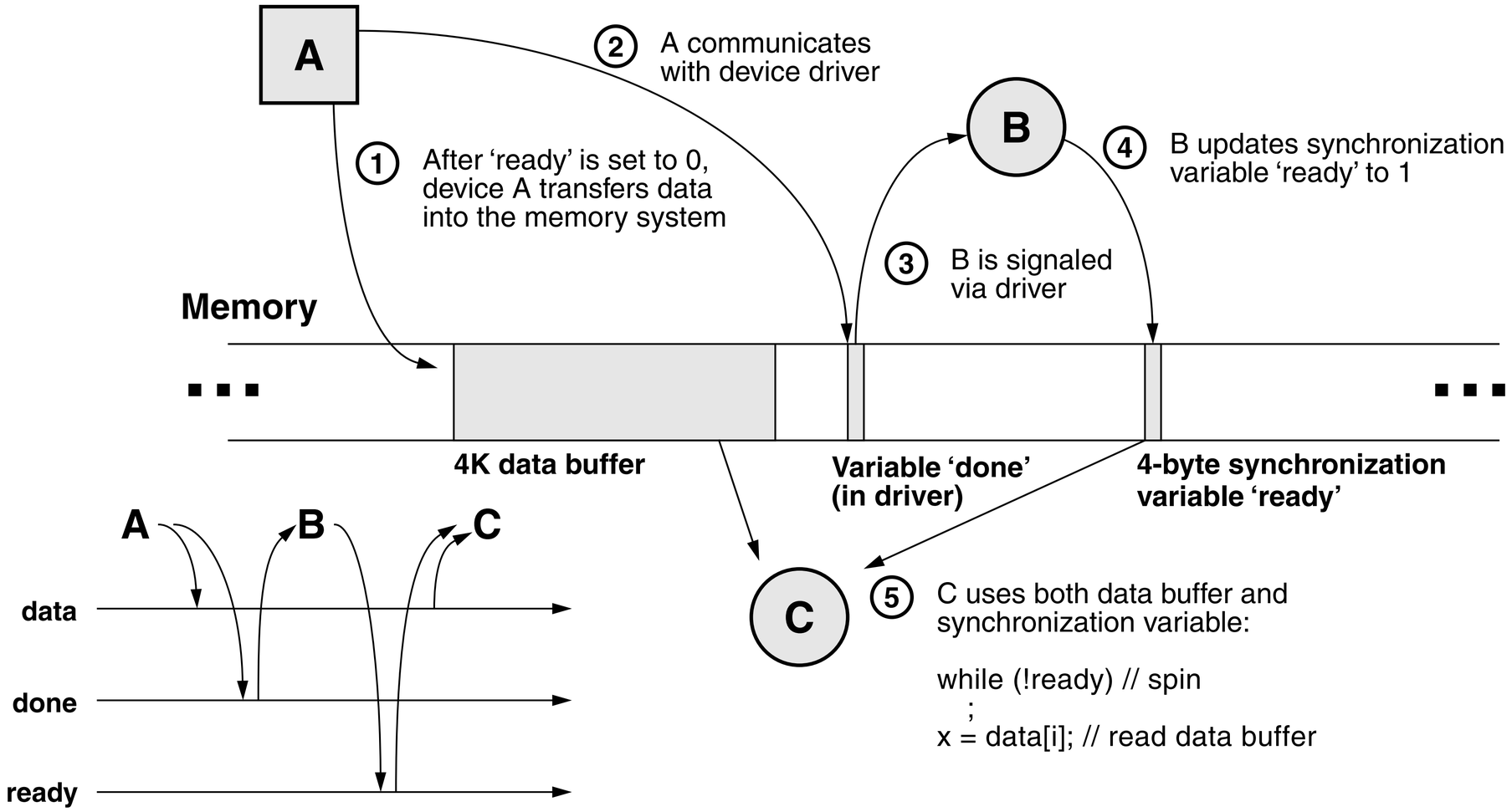
Process C (consumer):

```
global char data[SIZE];
global int ready=0;

while (1) {

   while (!ready)
      ;

   process( data );
   ready = 0;

}
```

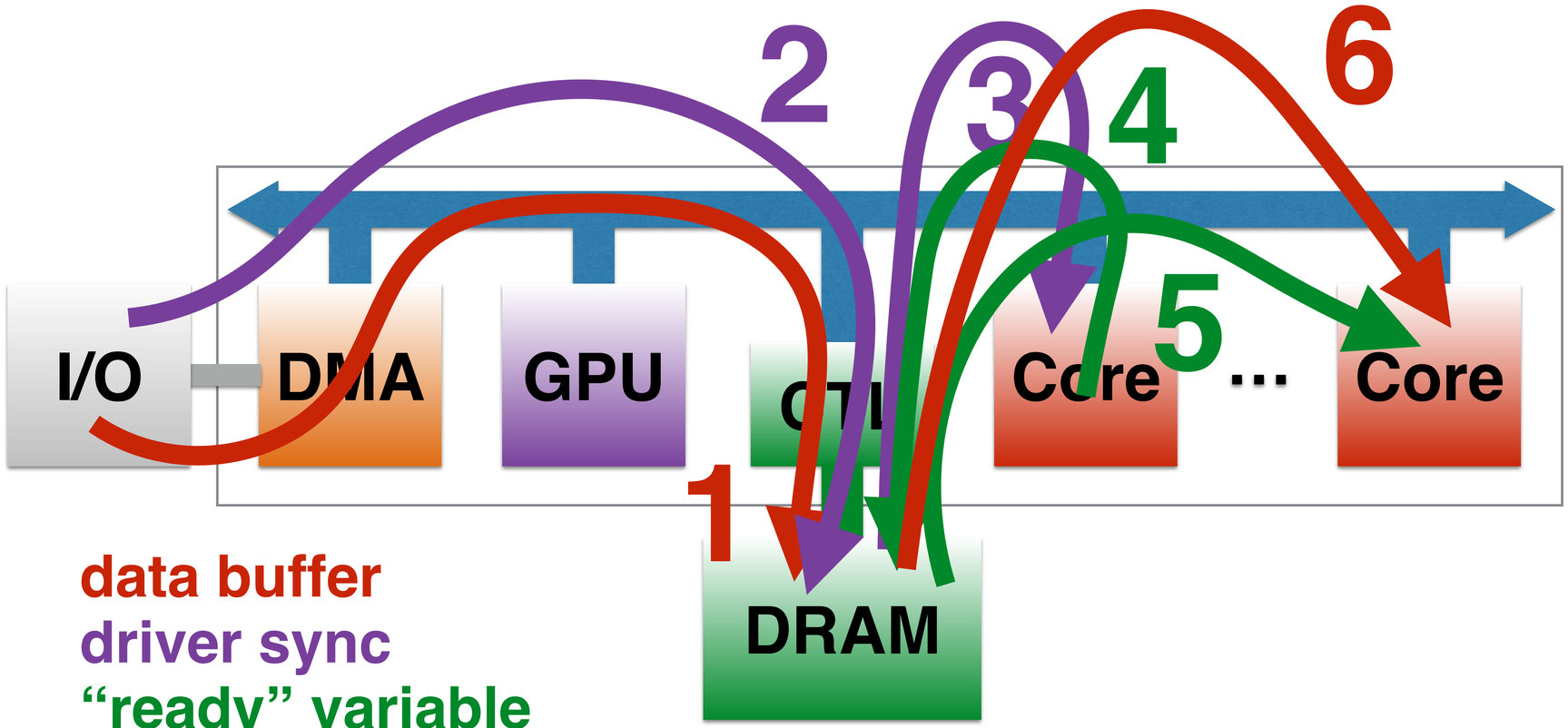
Process B (producer):

```
global char data[SIZE];
global int ready=0;

int fd = open("dev A");

while (1) {

   while (ready)
      ;

   dma( fd, data, SIZE );
   ready = 1;

}
```

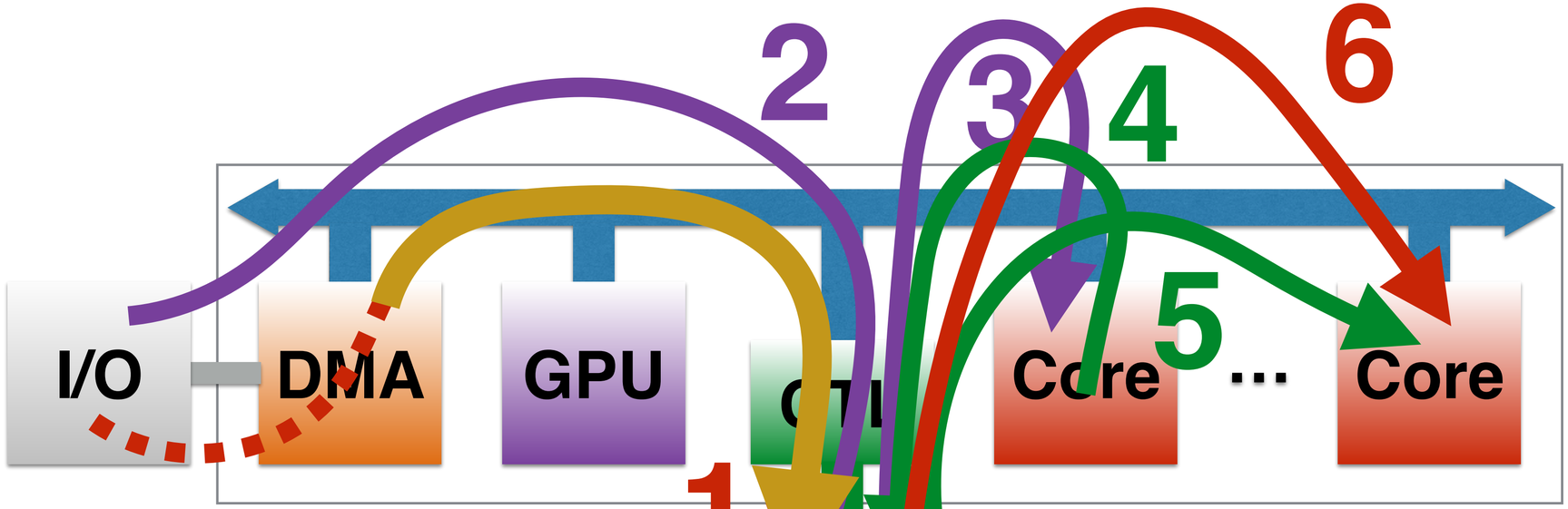
Proc B reads data from dev A, signals proc C when done (producer-consumer pair) — more detail



Proc B reads data from dev A, signals proc C when done (producer-consumer pair)



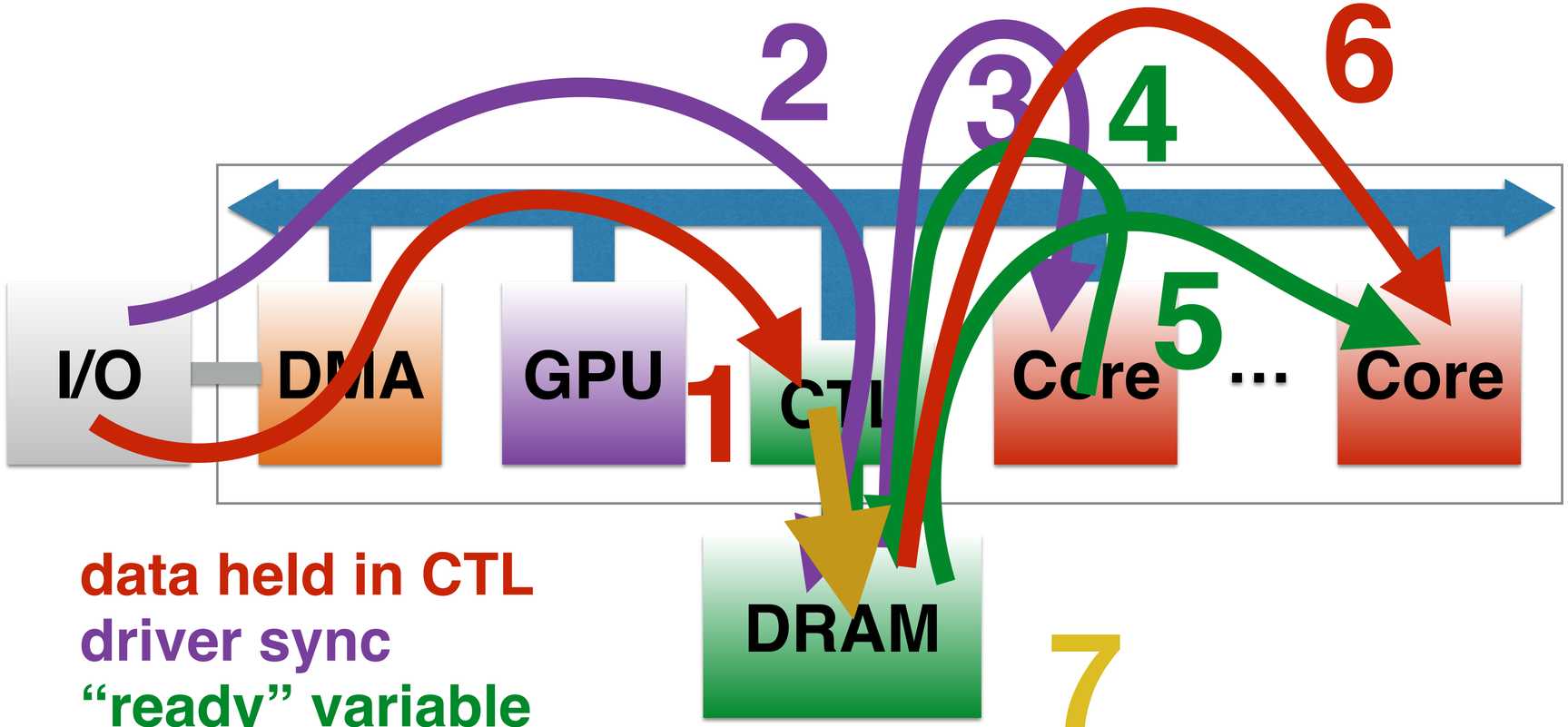
Proc B reads data from dev A, signals proc C when done (producer-consumer pair)



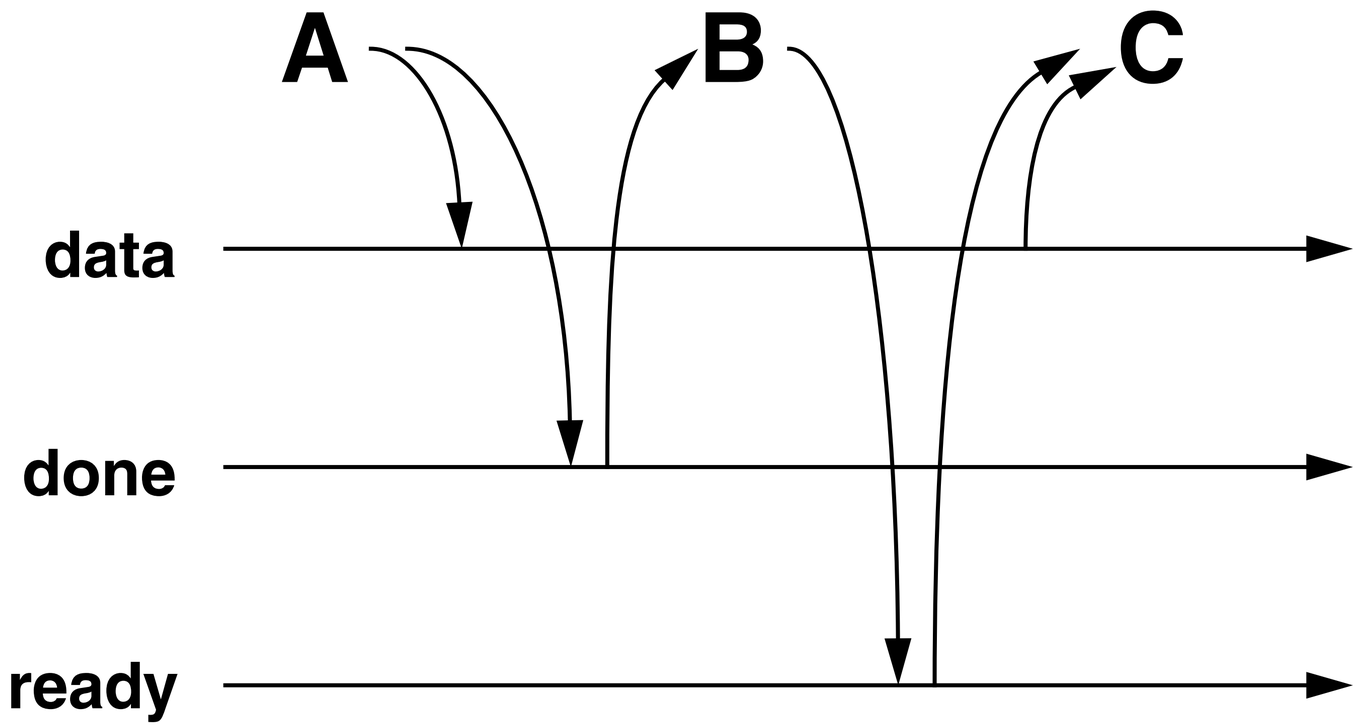
data - delayed in DMA
driver sync
“ready” variable
data - stale

7
data sent by DMA
… arrives too late

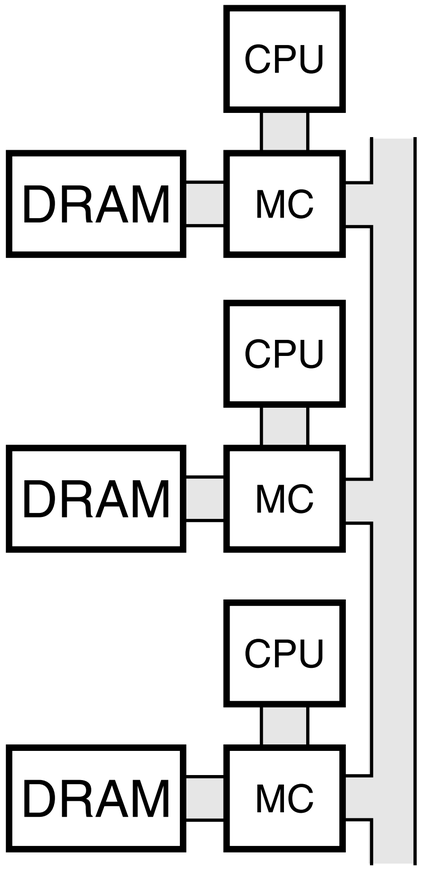
Proc B reads data from dev A, signals proc C when done (producer-consumer pair)



Problem: causal relationships



Problem scales with the system size



Solve system linear eqs: $x_{i+1} = Ax_i + b$

```
while (!converged) {
   doparallel(N) {
      int i = myid();
      xtemp[i] = b[i];
      for (j=0; j<N; j++) {
         xtemp[i] += A[i,j] * x[j];
      }
   }
   // implicit barrier sync
   doparallel(N) {
      int i = myid();
      x[i] = xtemp[i];
   }
}
```

Some Consistency Models

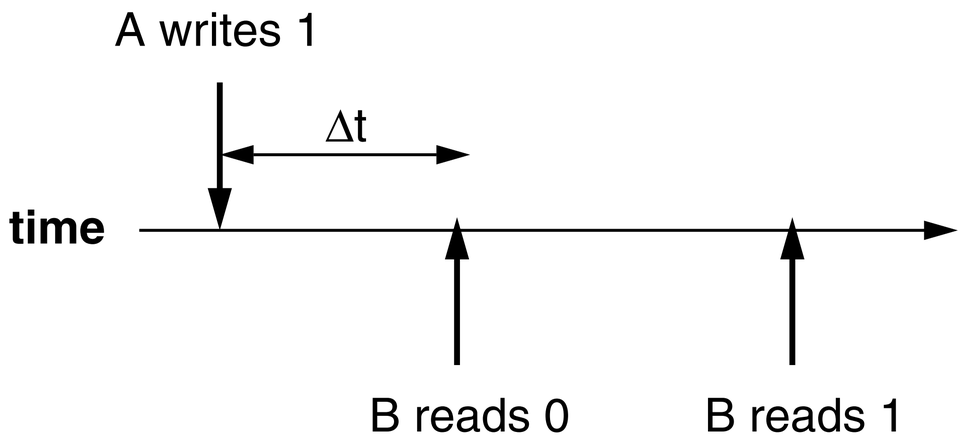
Strict Consistency: A read operation shall return the value written by the most recent store operation.

Sequential Consistency: The result of an execution is the same as a single interleaving of sequential, program-order accesses from different processors.

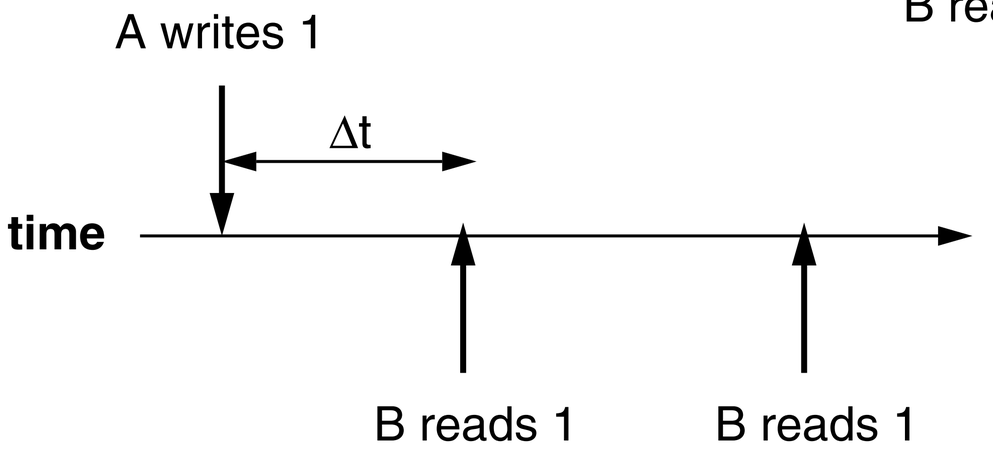
Processor Consistency: Writes from a process are observed by other clients to be in program order; all clients observe a single interleaving of writes from different processors.

Strict Consistency

Fails to satisfy strict consistency:

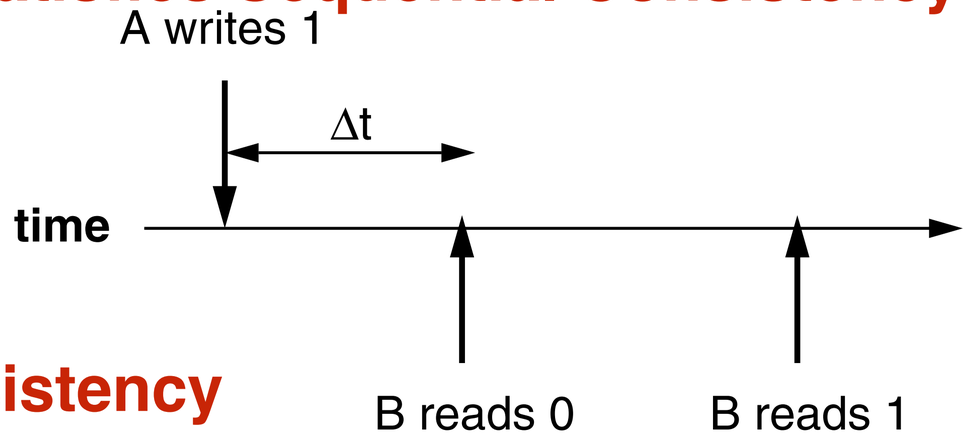


Satisfies strict consistency:

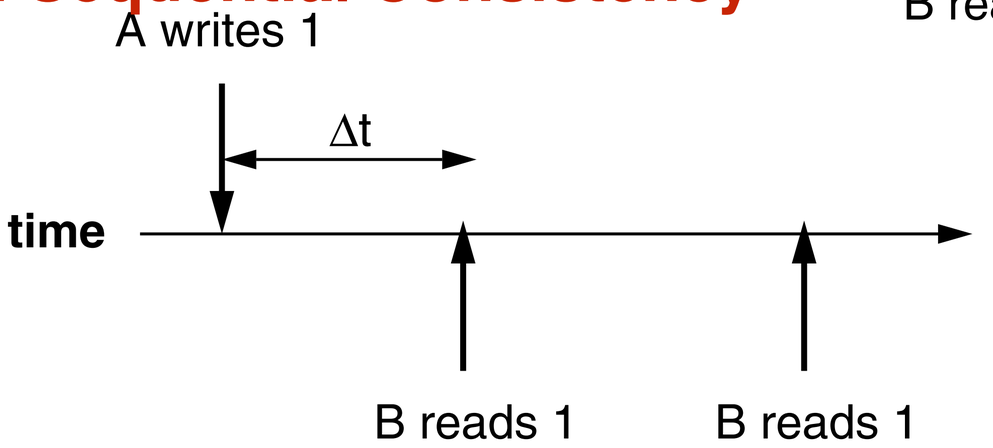


Sequential Consistency

Fails to satisfy strict consistency:
But satisfies Sequential Consistency

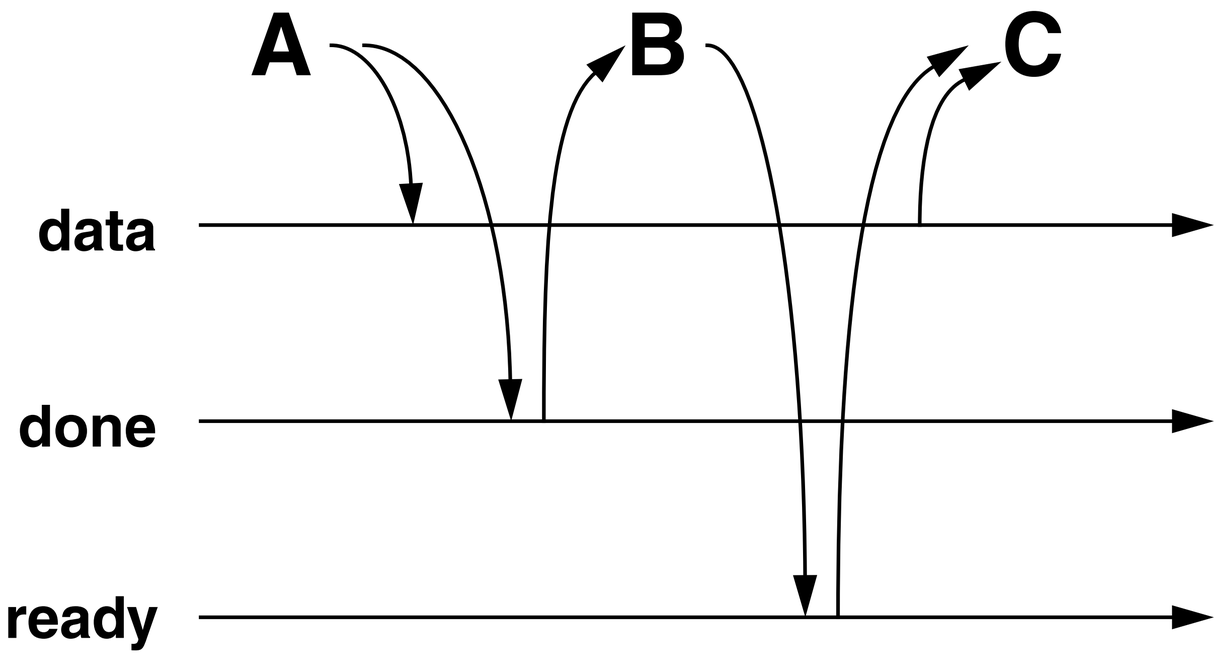


Satisfies strict consistency:
… and Sequential Consistency



Sequential Consistency

Handles our earlier problem:



Note: for this to work, memory controller may reorder internally, *but not externally*

Sequential Consistency

Requirements:

- **Everyone can reorder internally but not externally**
- **All I/O & memory references must go through the same sync point (e.g. memory-mapped I/O)**
- **Write of data and driver signal must be same client**
- **Write buffering presents significant problems**
- **Reads must be delayed by system latency**

… let's look at this last one more closely

Really Famous Example (Goodman 1989)

Process P1:

Initially, A=0

```
A=1;

if (B==0) {
   kill P2;
}
```

Process P2:

Initially, B=0

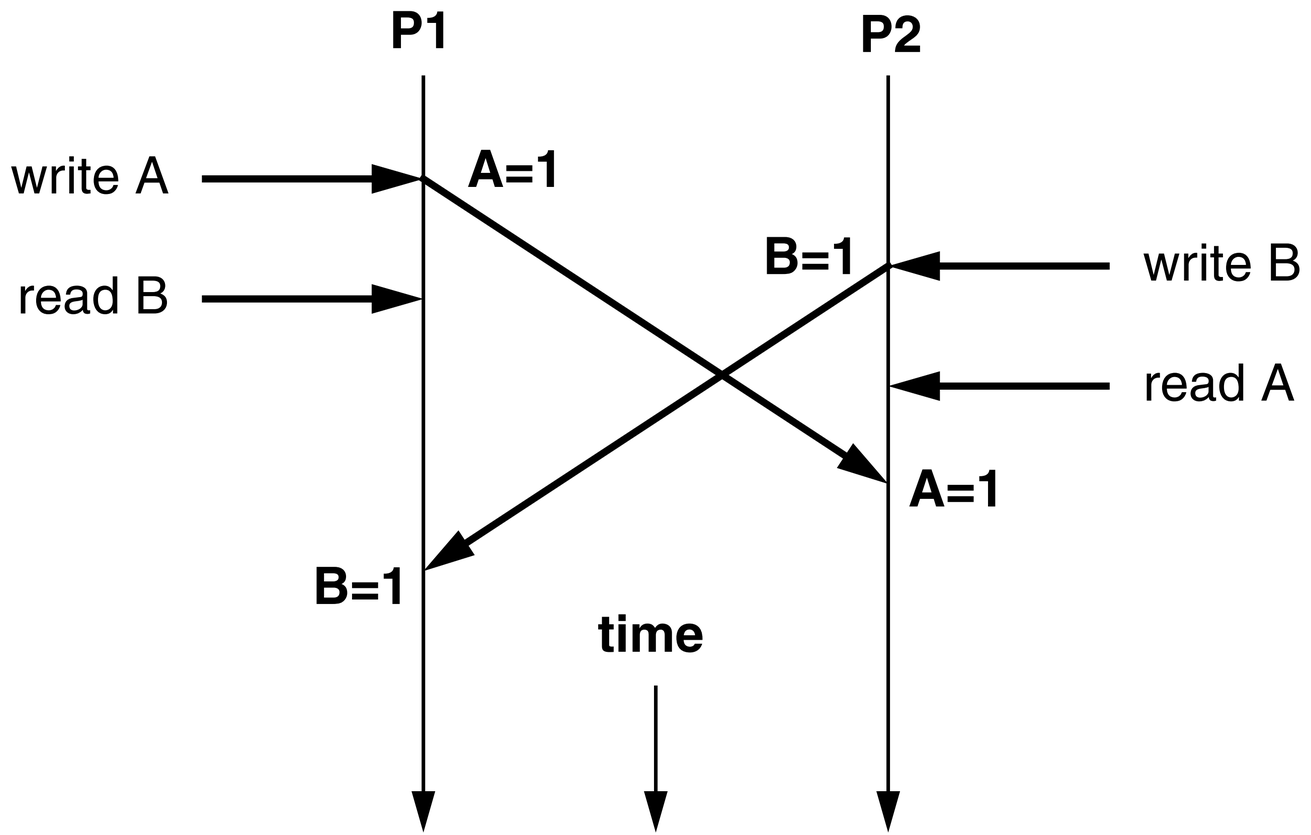
```
B=1;

if (A==0) {
   kill P1;
}
```

Sequential Consistency allows 0 or 1 processes to die

(not both)

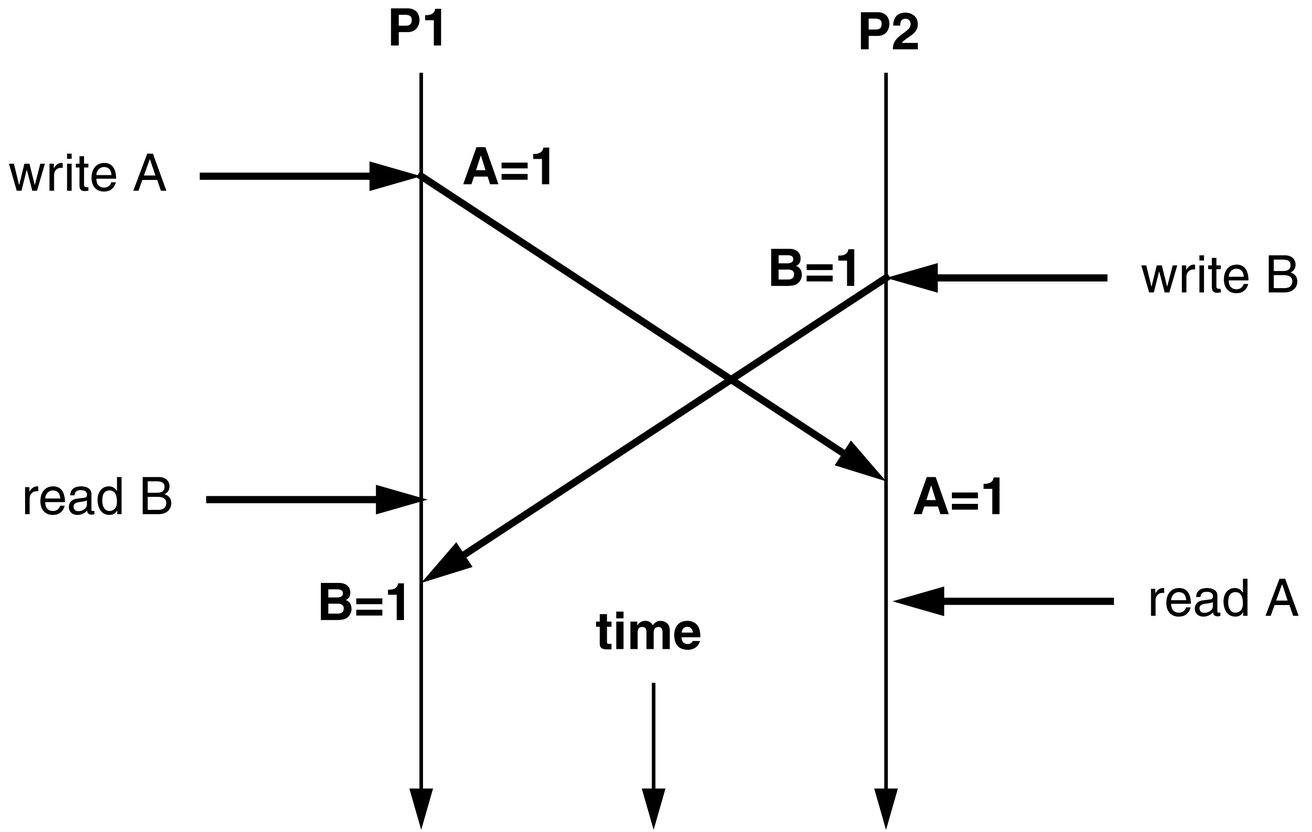
Race-Condition Example

Fails to satisfy sequential consistency:



Race-Condition Example

Satisfies sequential consistency:



Race-Condition Example

In practice:

- **speculate**
- **throw exception if problem occurs**

HOWEVER — from Jaleel & Jacob [HPCA 2005]:

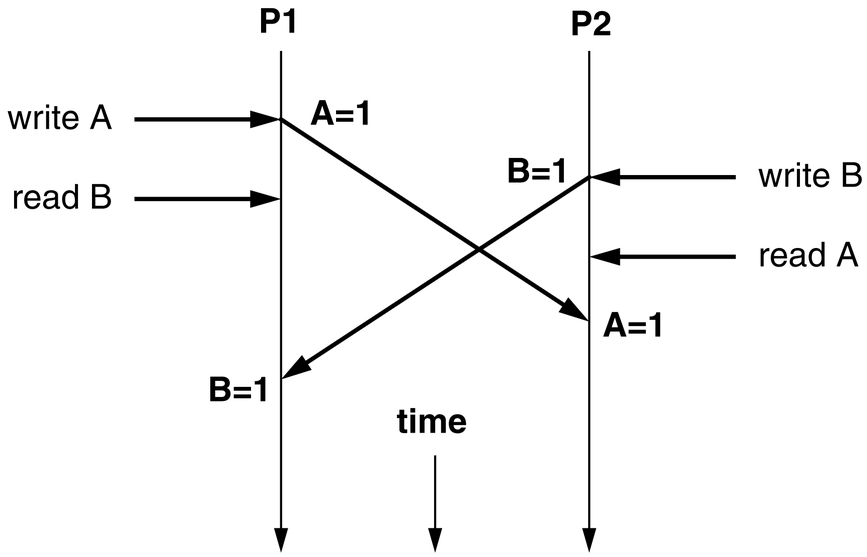
- increasing the reorder buffer from 80 to 512 entries results in an **increase in memory traps** by **6x** and an increase in total **execution** overhead by **10–40%**
- reordering memory instructions increases **L1** data cache **accesses** by **10–60%** and **L1** data cache **misses** by **10–20%**

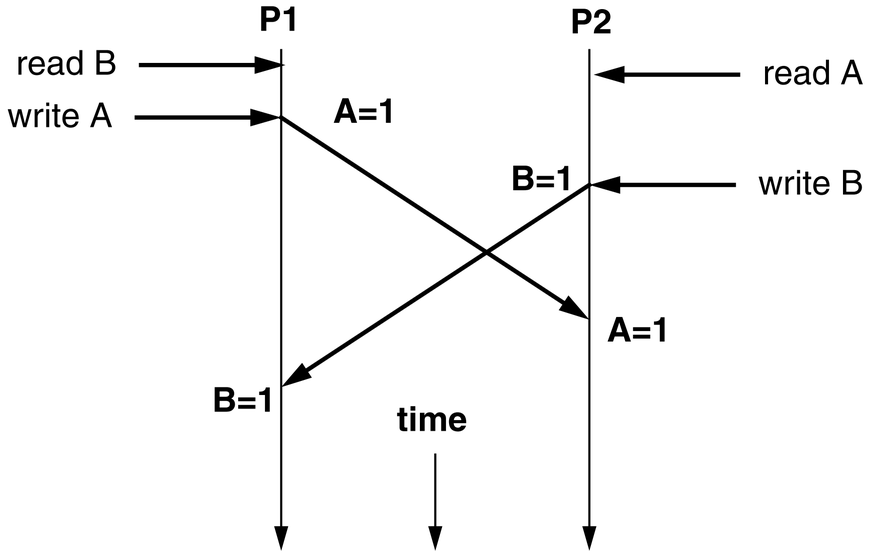
Processor Consistency

Also called *Total Store Order*

All writes in program order, reads freely reordered

Both of these scenarios are satisfied in this model:





Some Other Consistency Models

Partial store order — a processor can freely reorder local writes with respect to other local writes

Weak consistency — a processor can freely reorder local writes ahead of local reads

Release consistency — different classes of synchronization … enforces synchronization only w.r.t. *acquire/release* operations. On *acquire*, memory system updates all protected variables before continuing; on *release*, memory system propagates changes to the protected variables out to the rest of the system

Cache Coherence Schemes

Ways to implement a consistency model:

- **in software** (e.g. in virtual memory system, via page table)
- **in hardware**
- **combine hardware & software**

The hardware component is called “cache coherence”

Coherence Implementations

Cache-Block States:

I — Invalid

M — Modified — read-writable, forwardable, dirty

S — Shared — read-only (can be clean or dirty)

E — Exclusive — read-writable, clean

O — Owned — read-only, forwardable, dirty

Coherence Implementations: SI

Works with write-through caches

Block is either present (Shared) or not (Invalid)

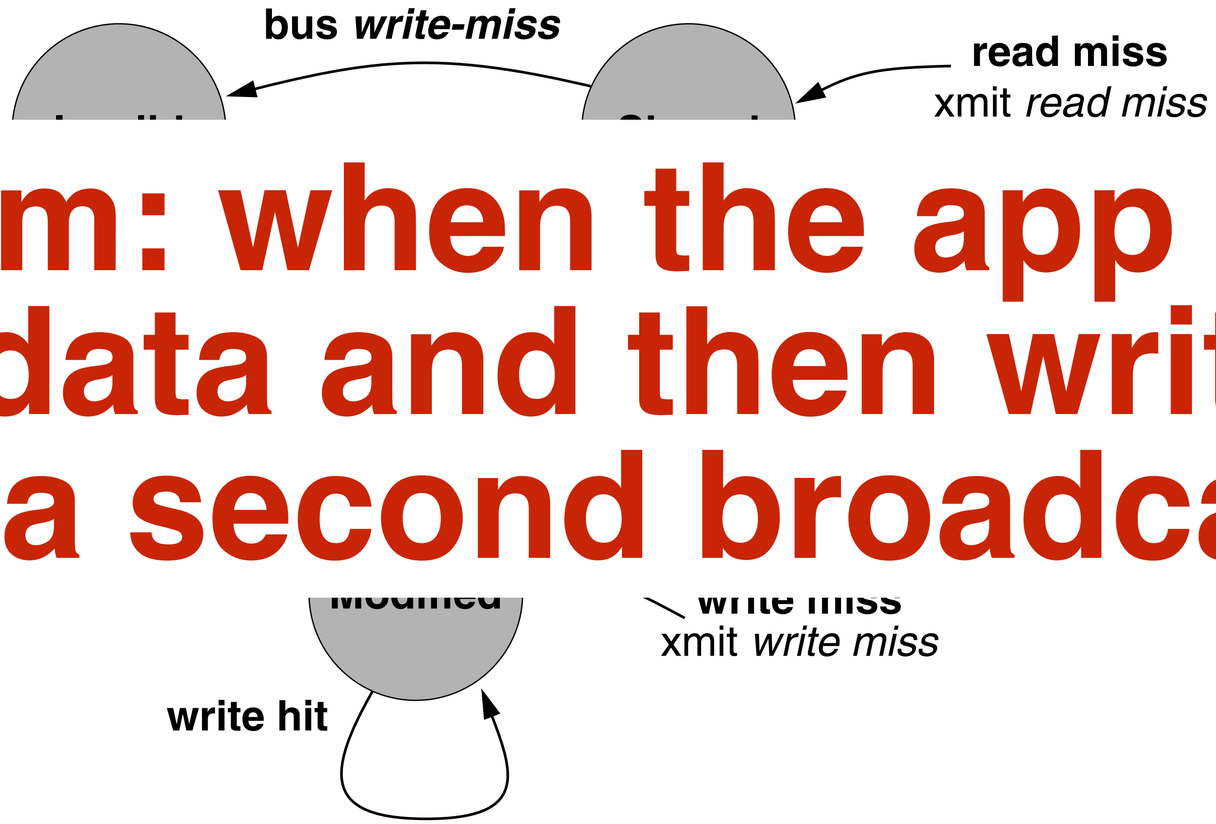
Problem: Nobody wants to use write-through caches

Both schemes require broadcast or multicast of coherence information and/or write data

Note: write-update and sequential consistency don't play nice together

Coherence Implementations: MSI

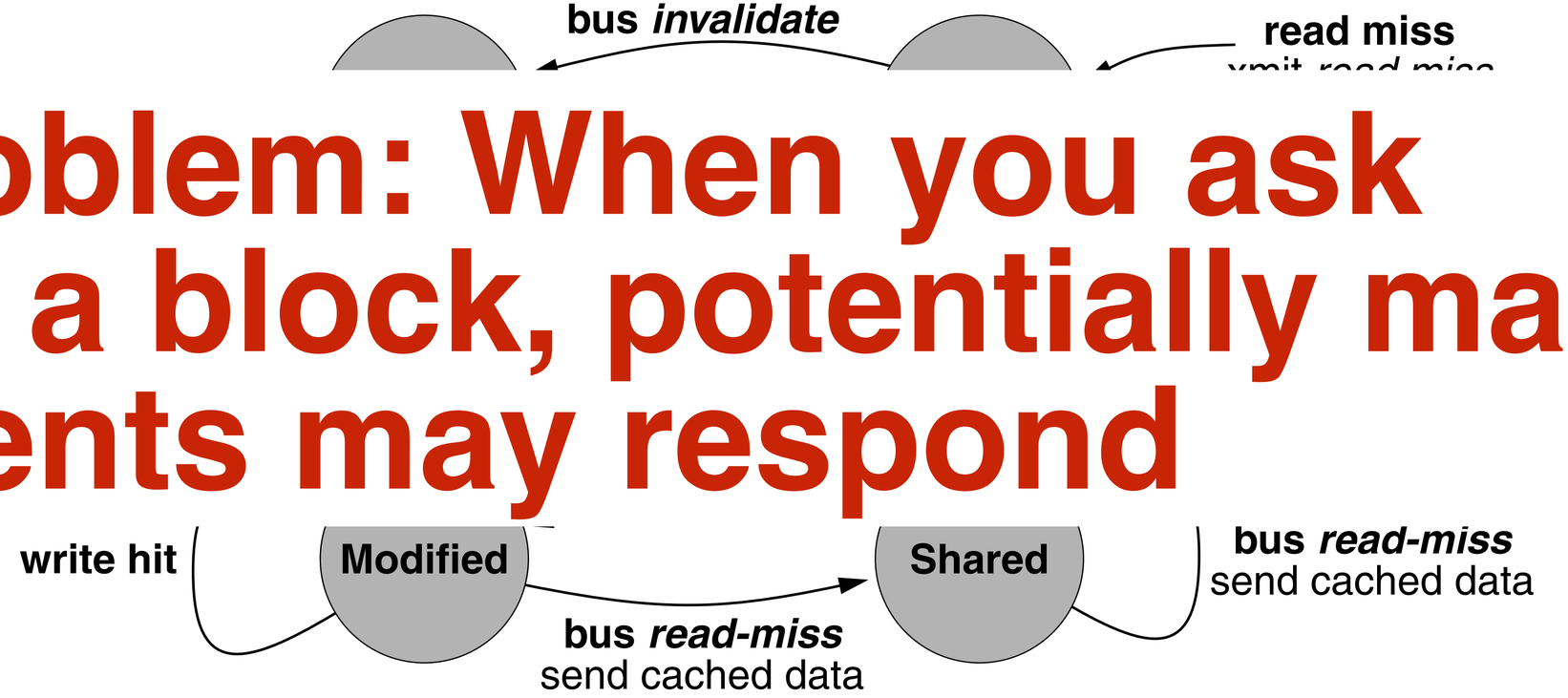
Write-back caches: dirty bit (Modified state)



Problem: when the app reads data and then writes, sends a second broadcast

Coherence Implementations: MESI

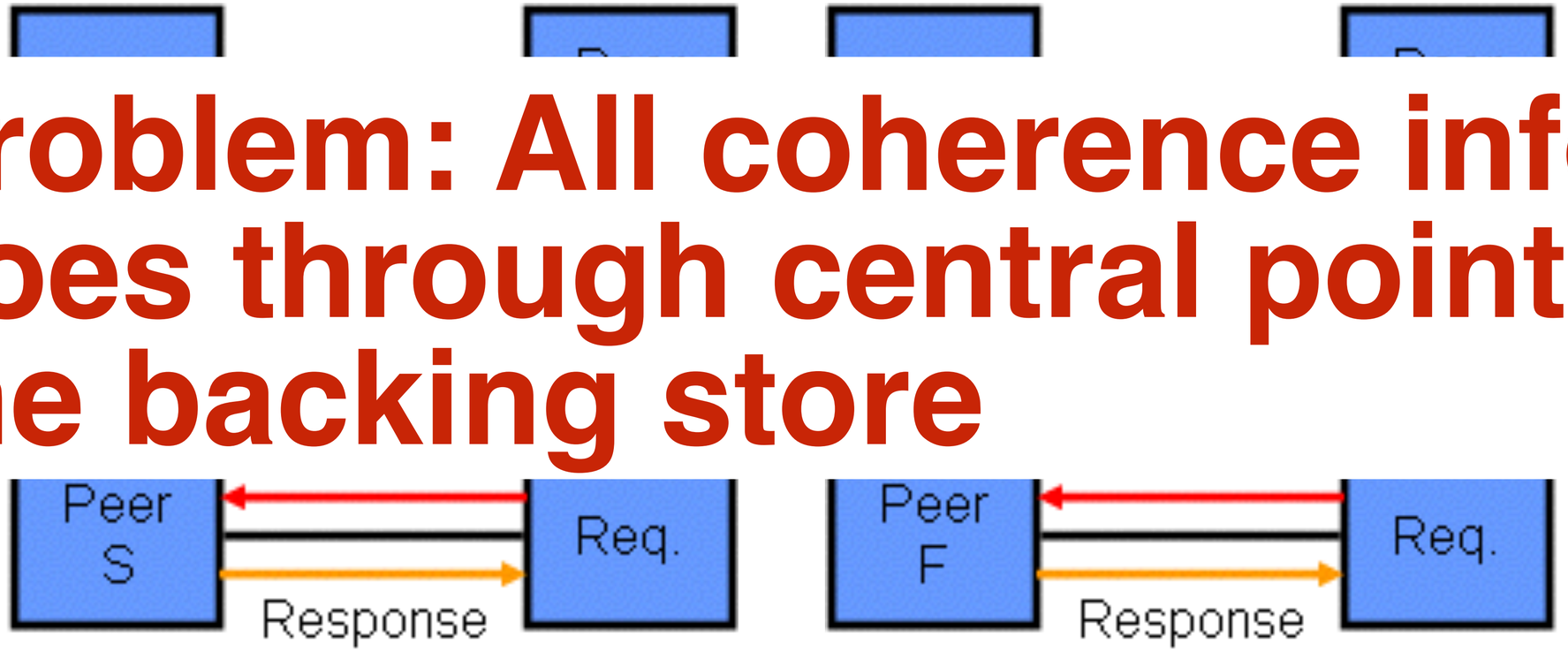
Reduces write broadcasts



Problem: When you ask for a block, potentially many clients may respond

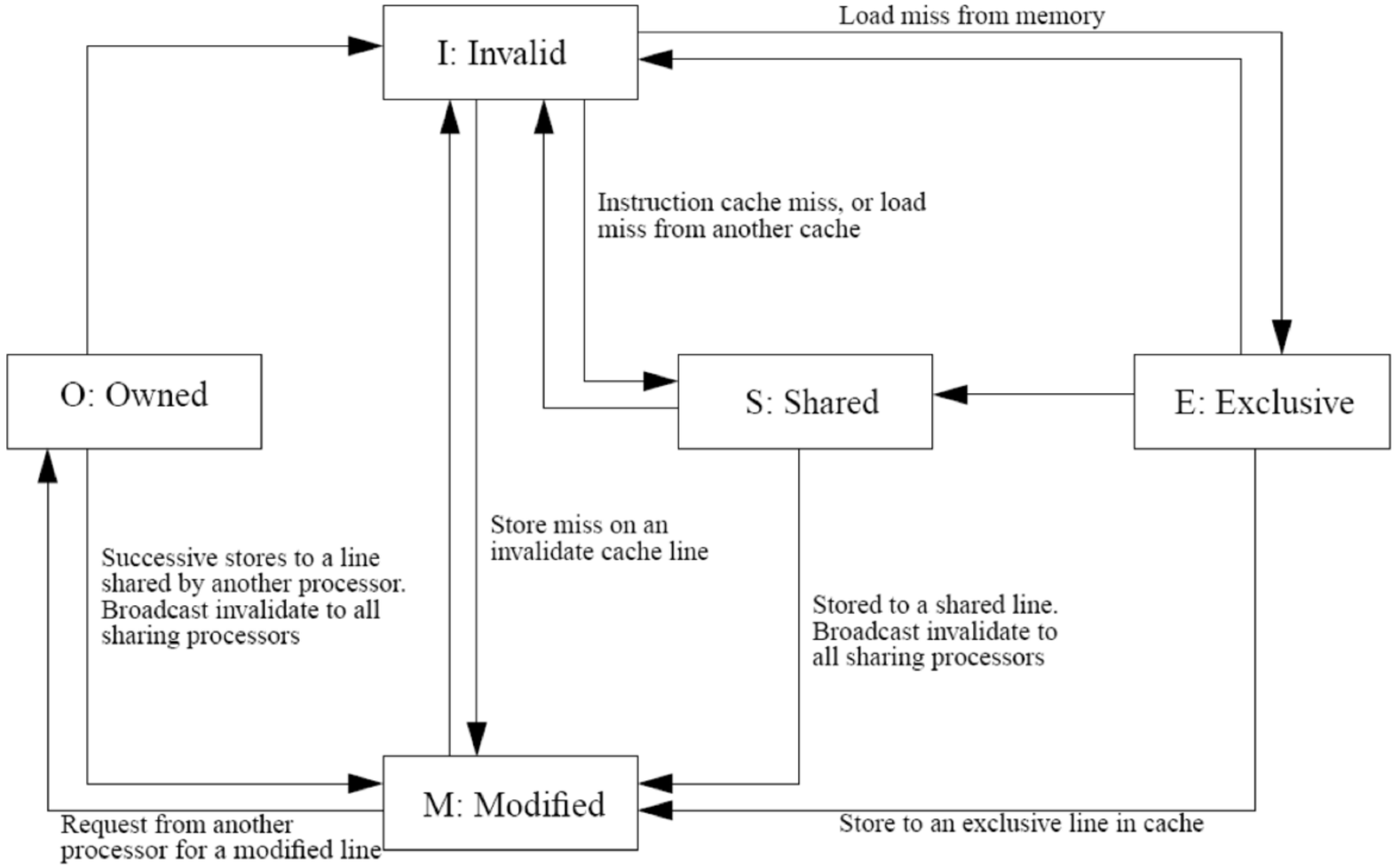
Coherence Implementations: MESIF

Shared broken into two: Shared (1+) and Forwardable (1)
Compare MESI (left) vs. MESIF (right):



Problem: All coherence info goes through central point: the backing store

Coherence Implementations: MOESI



MESI vs MOESI (AMD) vs MESIF (Intel)

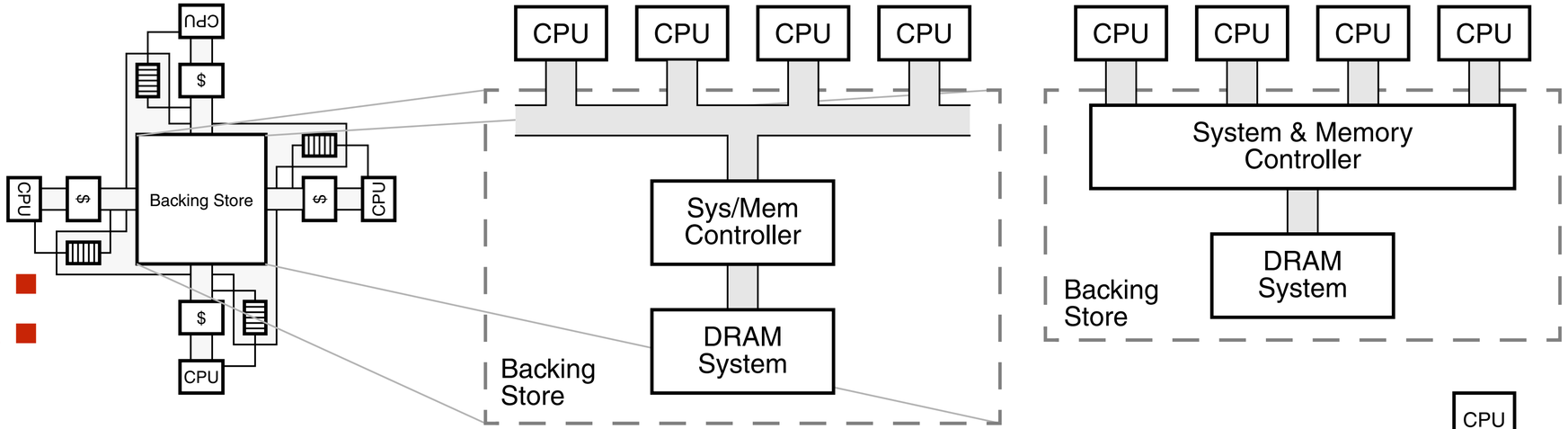
	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes		Must writeback to share or replace
Exclusive	Clean	Yes	Yes	Yes	MSIF	Transitions to M on write
Shared	Clean	No	No	No	I	Does not forward
Invalid	NA	NA	NA	NA		Cannot Read
Forwarding	Clean	Yes	No	Yes	SI	Must invalidate other copies to write

	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes	O	Can share without writeback
Owned	Dirty	Yes	Yes	Yes		Must writeback to transition
Exclusive	Clean	Yes	Yes	Yes	MSI	Transitions to M on write
Shared	Either	No	No	No	I	Shared can be dirty or clean
Invalid	NA	NA	NA	NA		Cannot Read

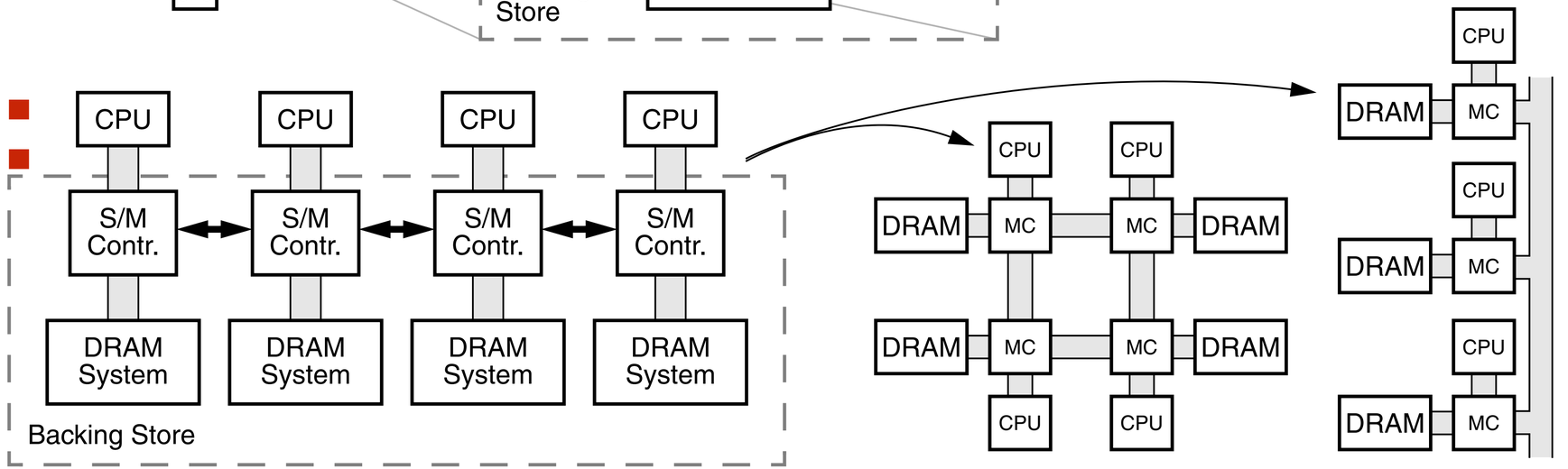
	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes		Must writeback to share or replace
Exclusive	Clean	Yes	Yes	Yes	MSI	Transitions to M on write
Shared	Clean	No	No	Yes	I	Shared implies clean, can forward
Invalid	NA	NA	NA	NA		Cannot Read

Some System Configurations

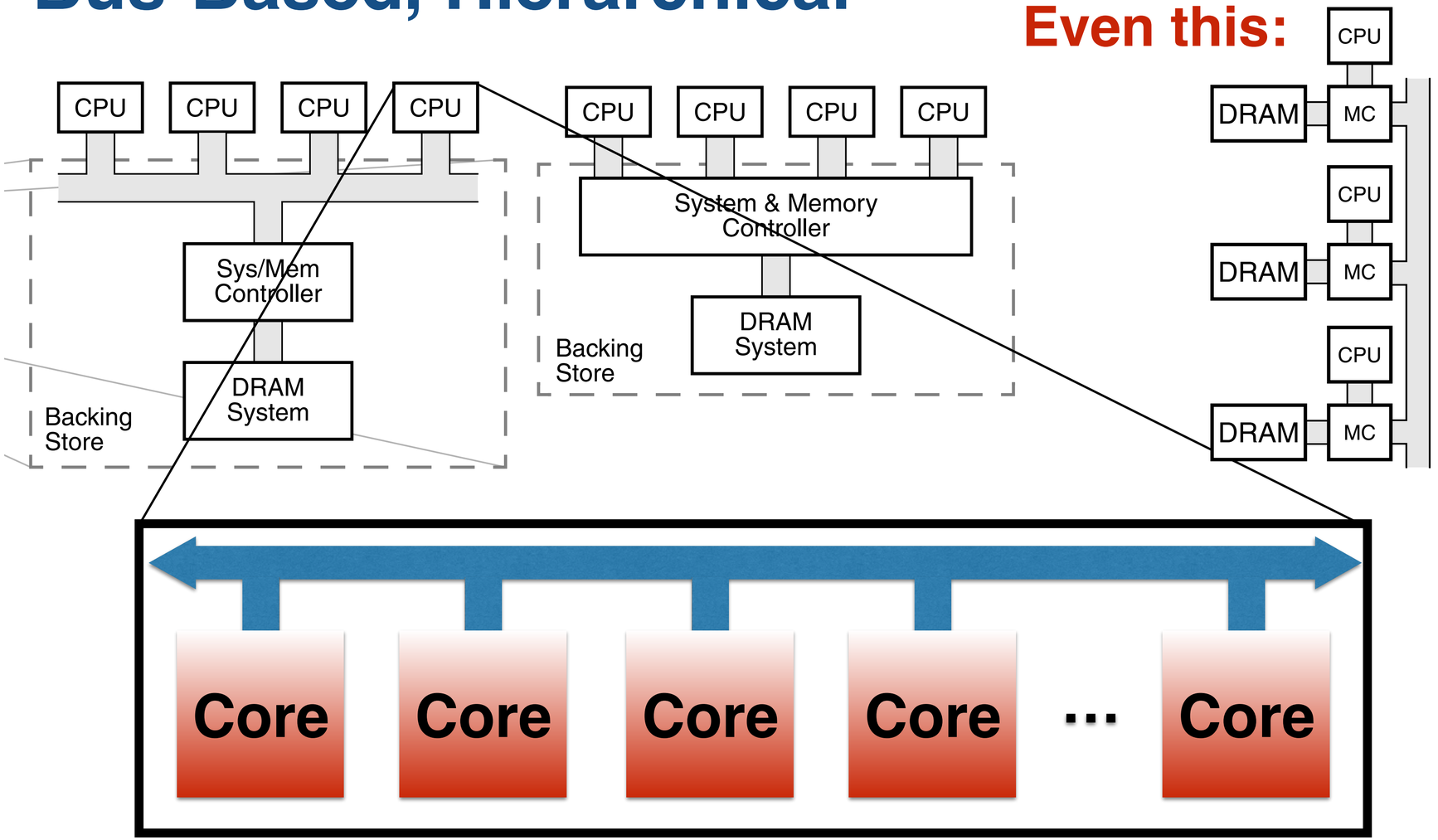
UMA:



NUMA:

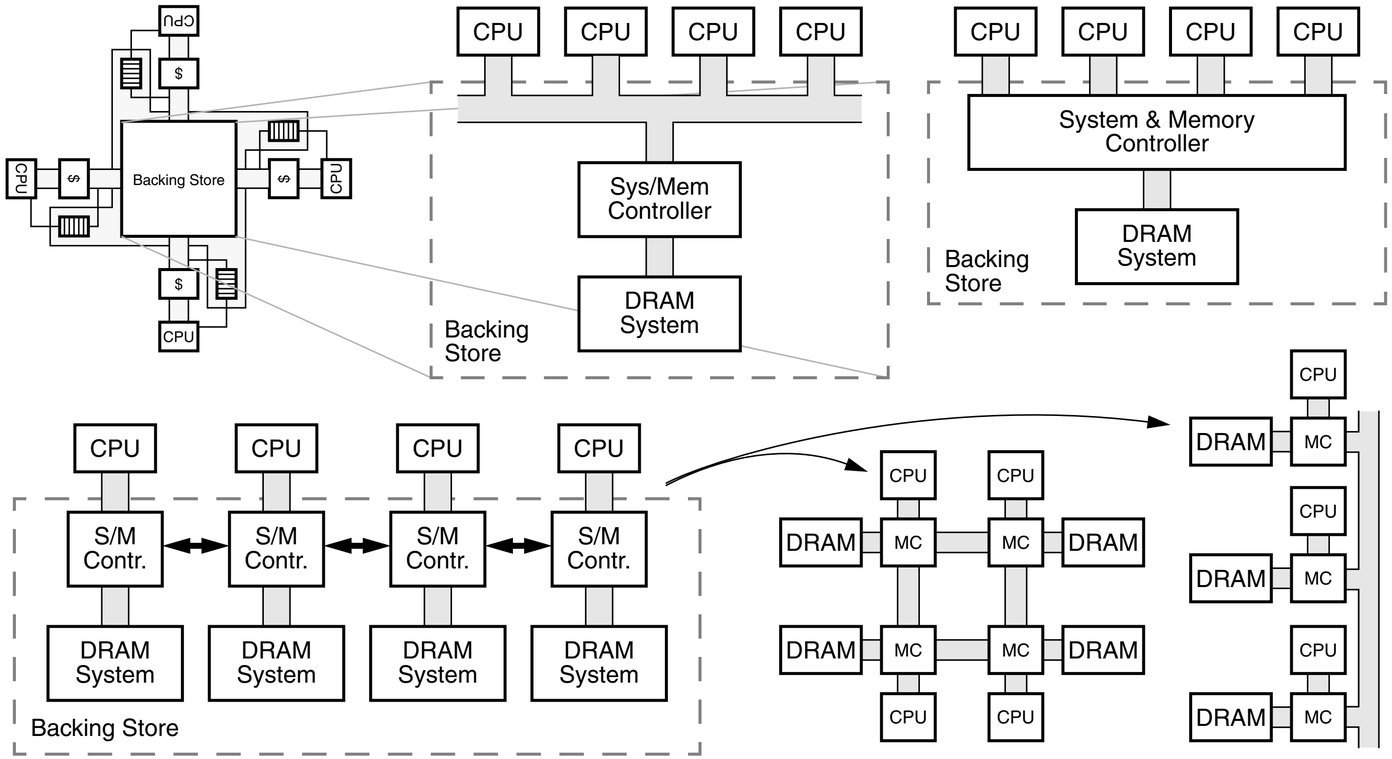


Bus-Based, Hierarchical



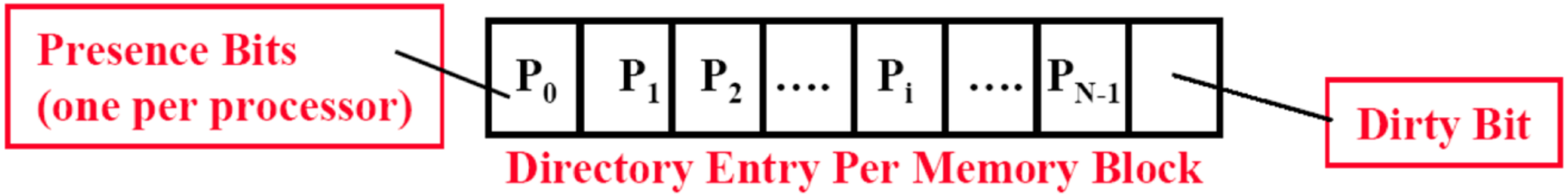
Directory-Based Protocols

Can run on any configuration—the main idea is to eliminate the need to broadcast every coherence event



Directory-Based Protocols

Each memory block has a directory entry



P+1 bits where P is the number of processors

One dirty bit per directory entry

If dirty bit is on then only one presence bit can be on

Nodes only communicate with other nodes that have the memory block

Directory-Based Protocols

Flavors:

- **Centralized Directory, Centralized Memory**
  Poor scalability, but better than bus-based …
- **Decentralized Directory, Distributed Memory**
  Each node stores small piece of entire directory corresponding to the memory resident at that node. Directory queries sent to the node that the address of the block corresponds to (i.e. its **Home Node**).
- **Clustered**
  Presence bits are coarse-grained

Problem with Scalability

What if latency to other procs > latency to local DRAM?