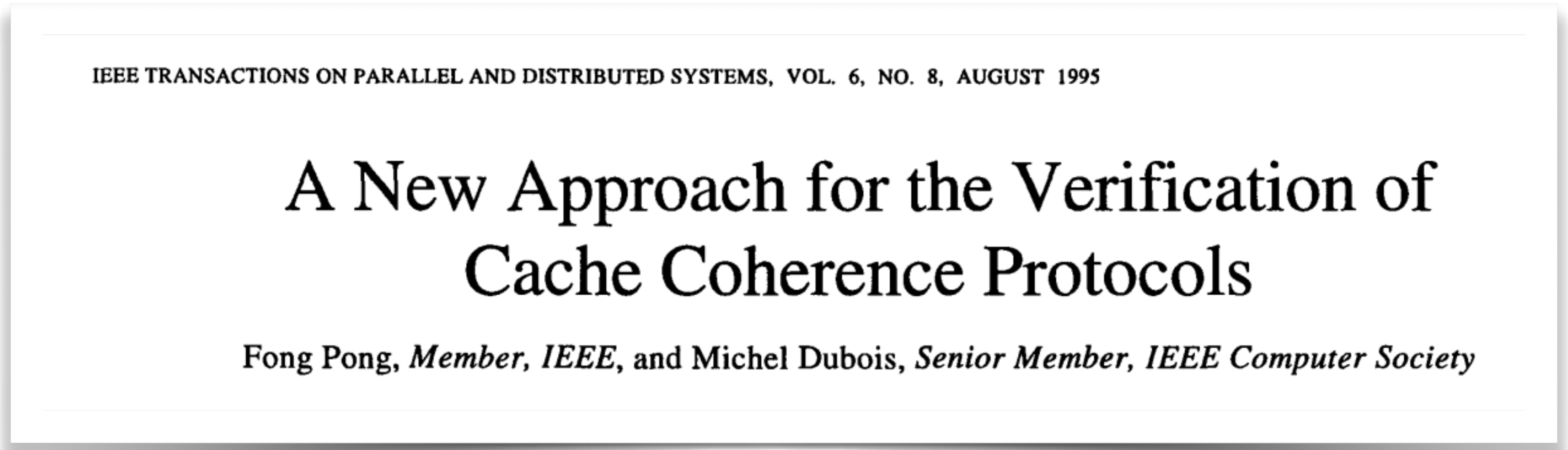
Shared Caches and Coherence


IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 6, NO. 8, AUGUST 1995

A New Approach for the Verification of Cache Coherence Protocols

Fong Pong, *Member, IEEE*, and Michel Dubois, *Senior Member, IEEE Computer Society*

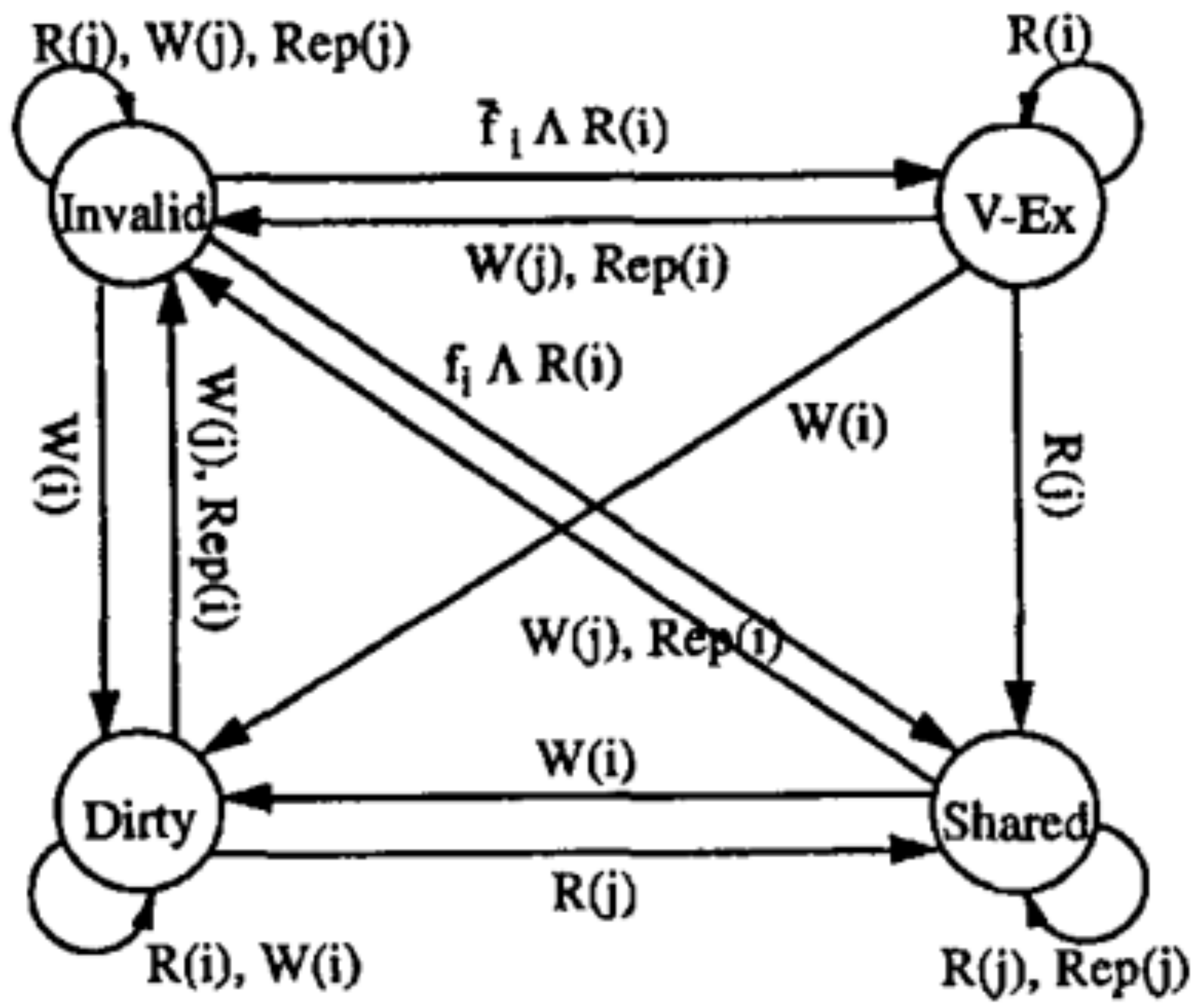


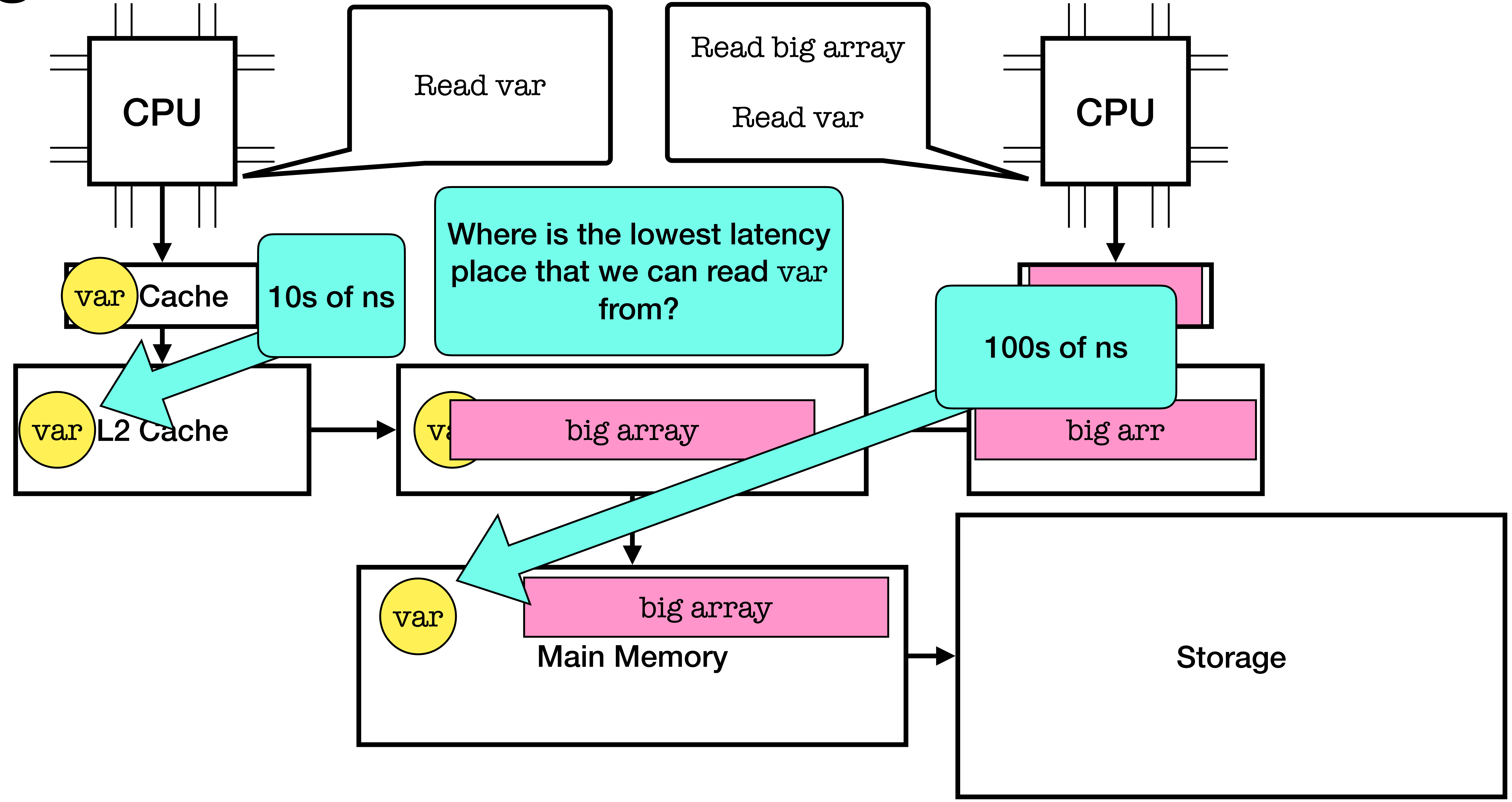
Fig. 1. The Illinois protocol transition diagram from the perspective of cache C_i.

Image Credit: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=406955

Outline

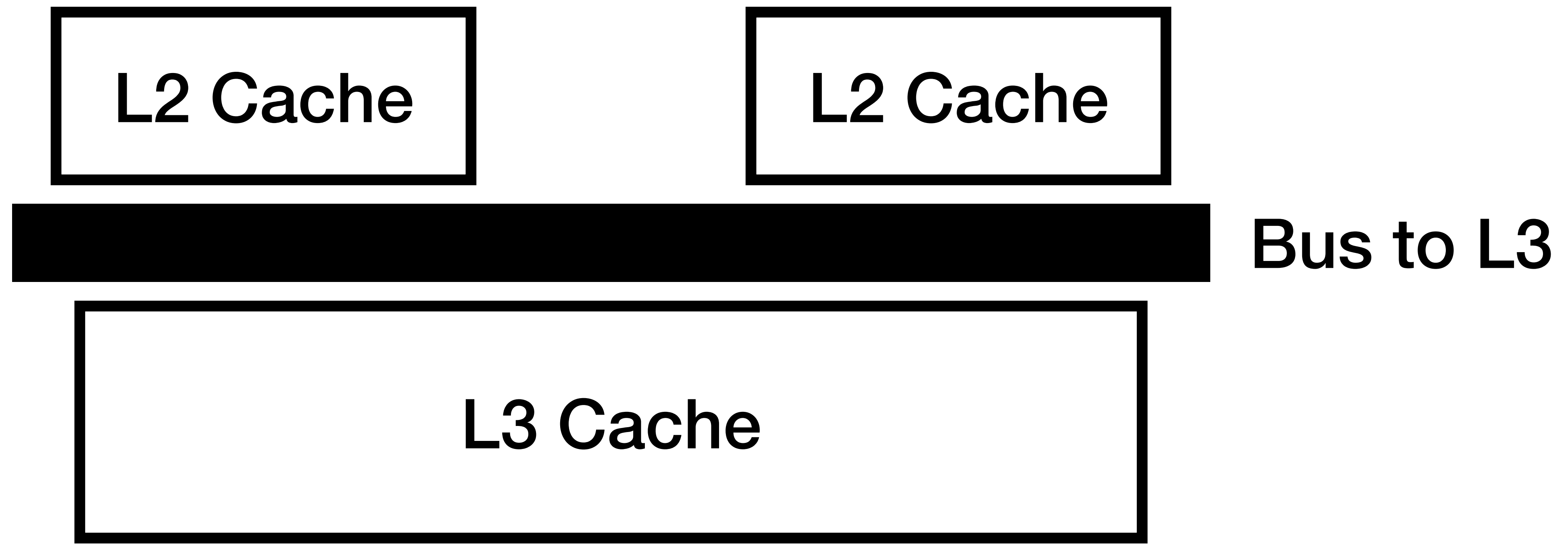
- Revisiting the “shared cache” setup
- Lookup optimization with snooping
- Consistency issues in a shared memory system
- Implementing cache coherence!

Using Shared Caches



Snooping to Accelerate Lookups

- Caches are connected to one another using collections of wires between ports (i.e., buses)



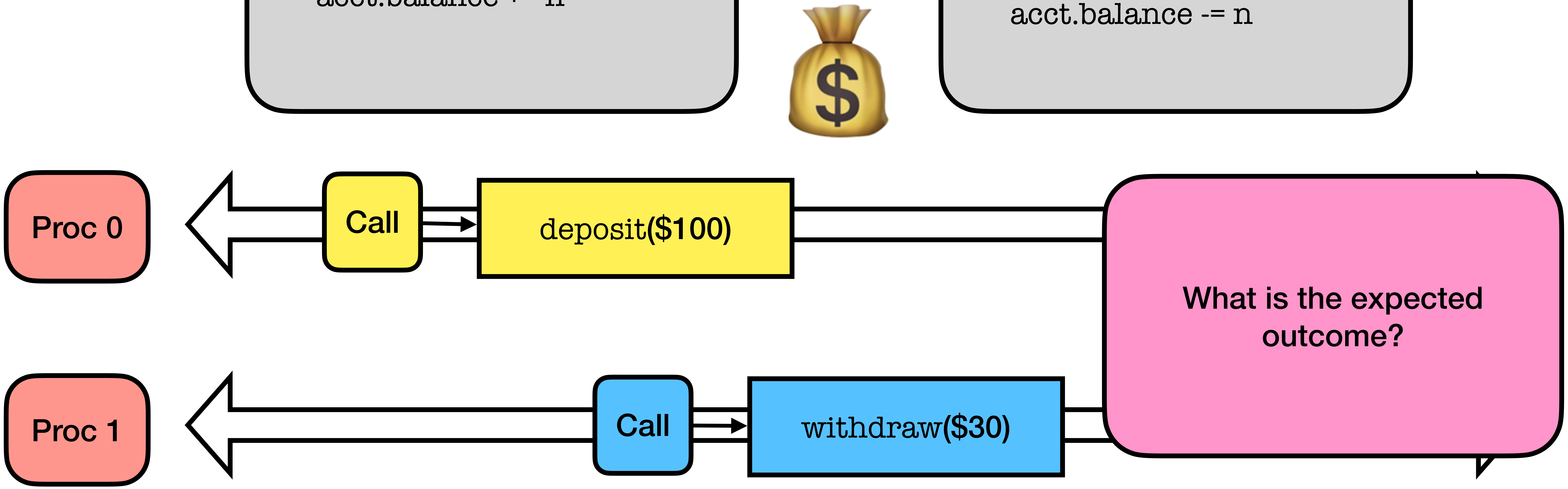
- Buses between caches can support different kinds of memory commands: caches can query the lower-level memory component directly via a *request* or to other peers on the processor-side of the bus via a *broadcast*

- Broadcasts can be used to implement optimistic *snooping* requests in which a cache asks a peer if they have a shared data value to avoid the longer latency lookup of the lower levels

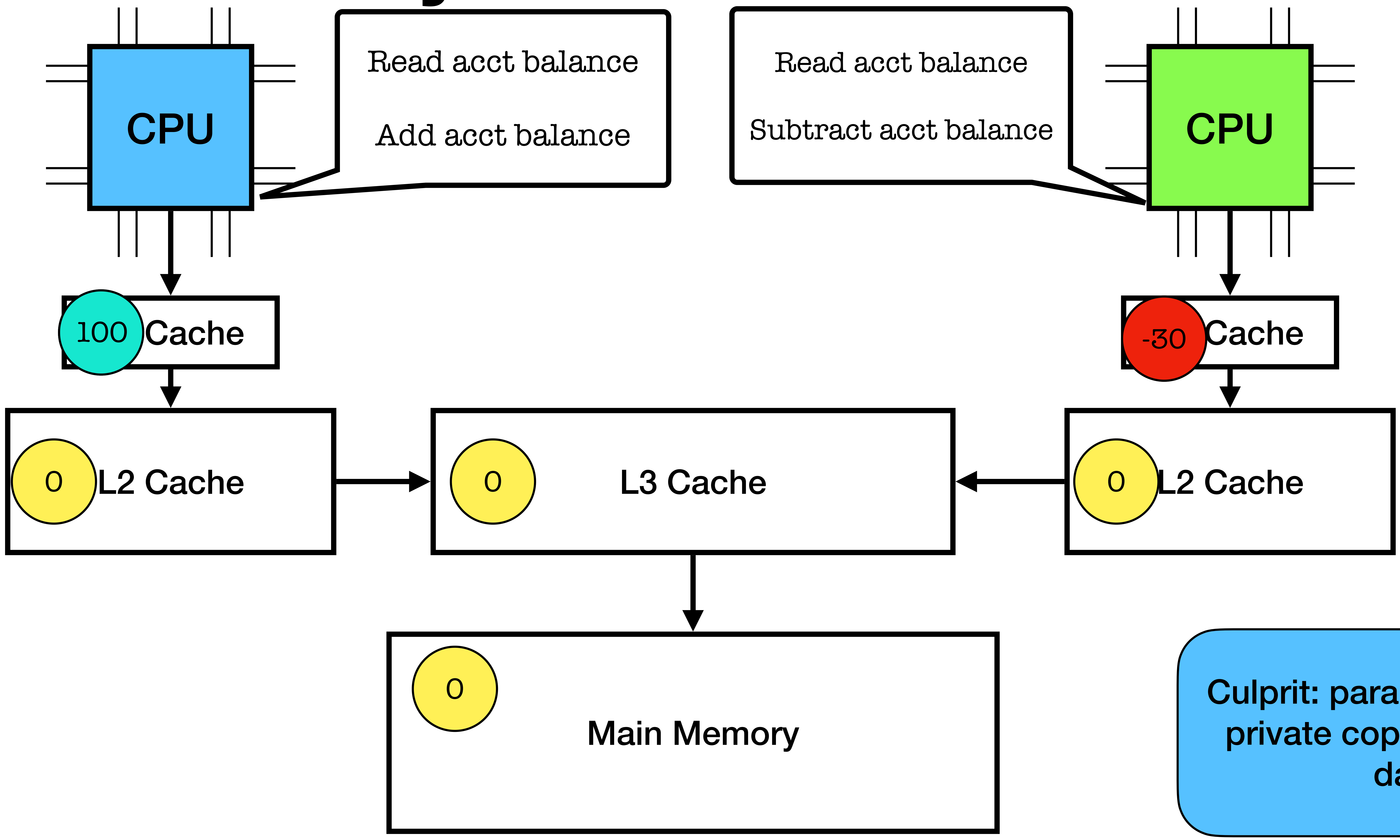
The consistency issue…

```
def deposit(n):
    acct.balance += n
```

```
def withdraw(n):
    assert(acct.balance >= n)
    acct.balance -= n
```

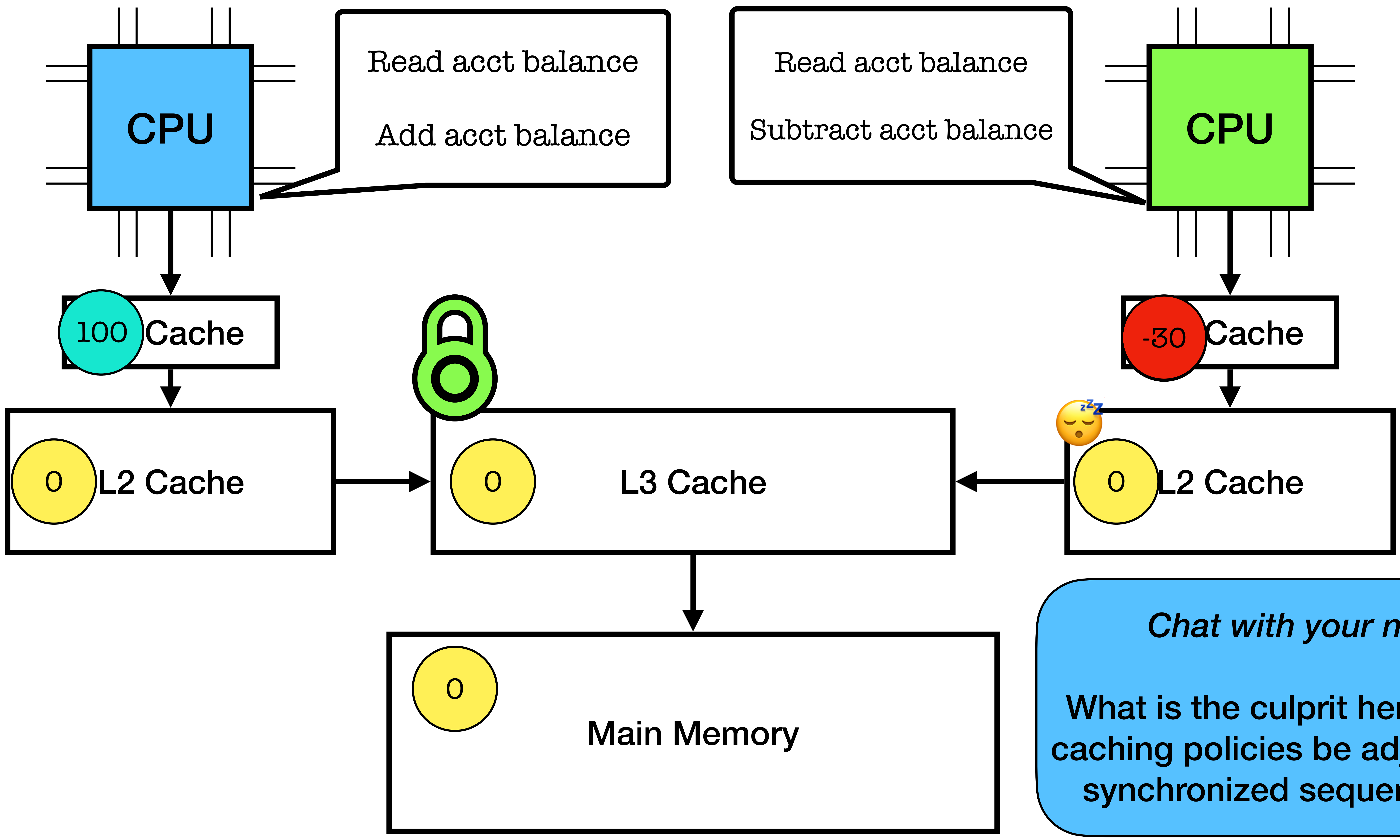


The consistency issue…



Culprit: parallel updates to private copies of shared data!

Potential Solution: Sequential Accesses (attempt 1)



Chat with your neighbor(s)!

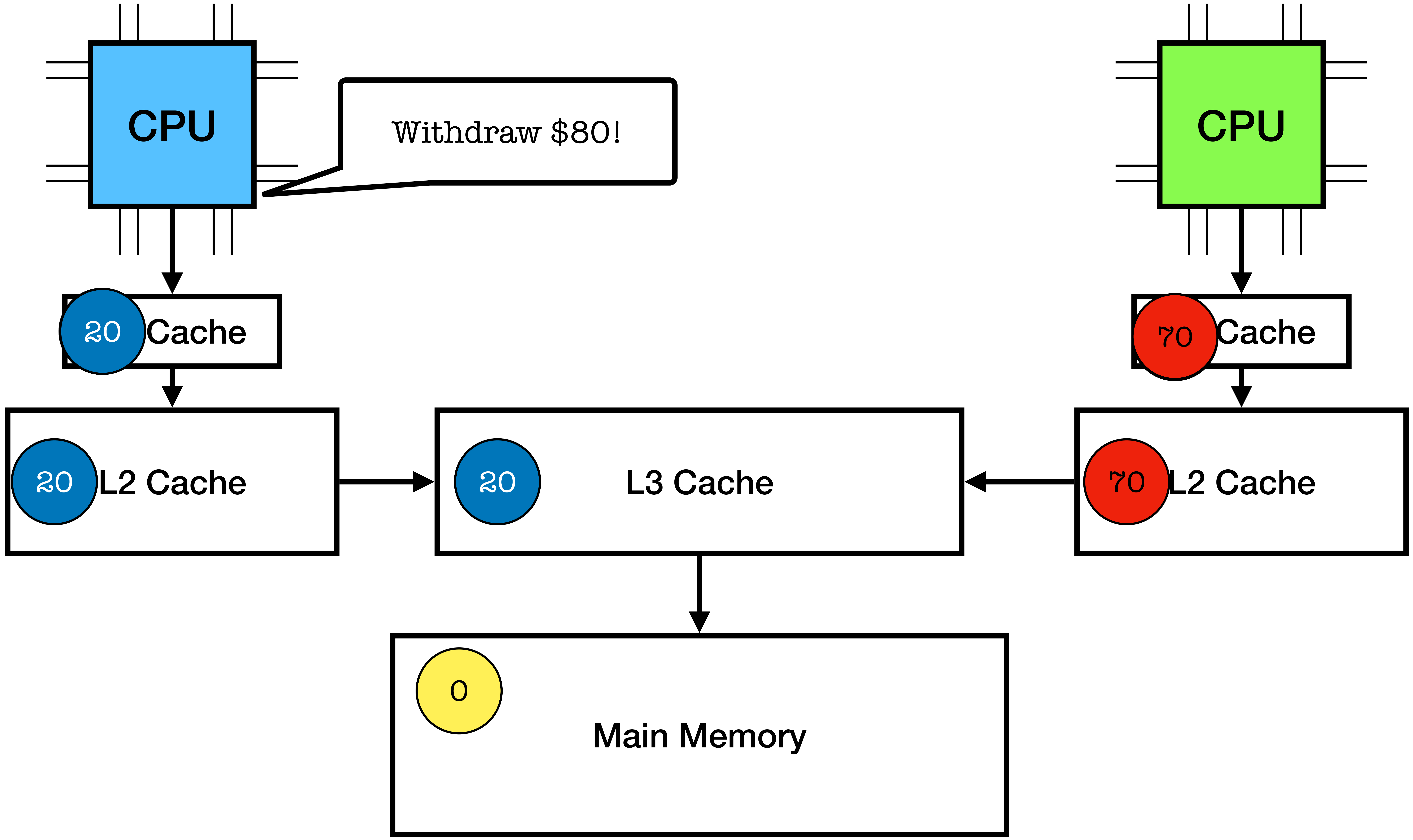
What is the culprit here? How can our caching policies be adjusted to achieve synchronized sequential accesses?

Solution 1: Sequential Accesses

- On each cache block, we will add a small number of *coherence bits* to track which processor currently “owns” access to the block!
- When accessing the block, first “acquire” the cache line by setting the coherence bits in the shared cache
- To release the lock on that cache line, update its state using a *write-through* update strategy!

Are we done??

Potential Solution: Sequential Accesses (attempt 2)



Potential Solution: Sequential Accesses

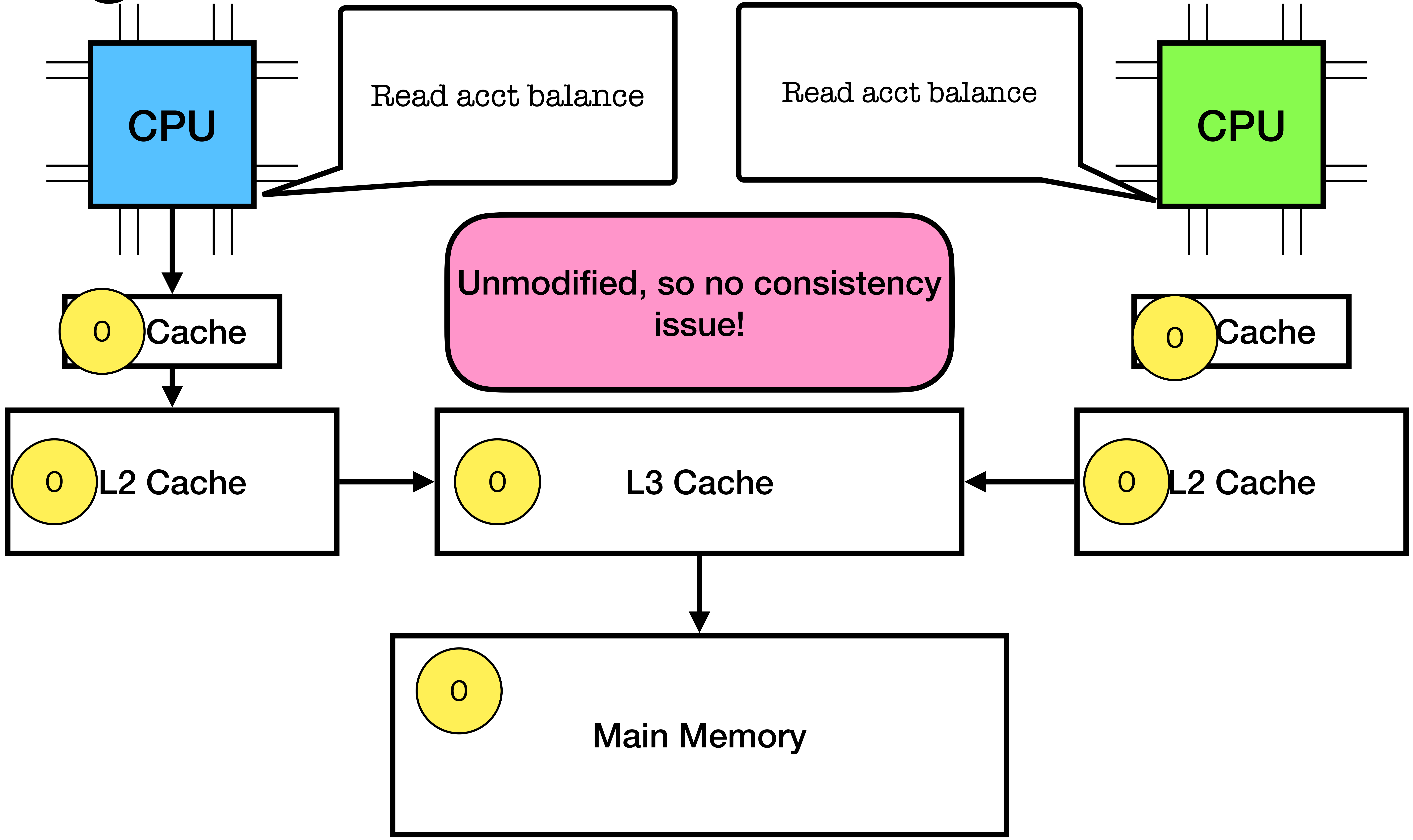
- On each cache block, we will add a small number of *coherence bits* to track which processor currently “owns” access to the block!
- When accessing the block, first “acquire” the cache line by setting the coherence bits in the shared cache
- To release the lock on that cache line, update its state using a *write-through* update strategy!
- We also need to *invalidate* or *update* any other private copies of the data elsewhere in the memory system ➡️ requires updating the *data* or *valid* bit associated with the cache block!

Benefits and Pitfalls of Sequential Accesses

Culprit: parallel updates to private copies of shared data!

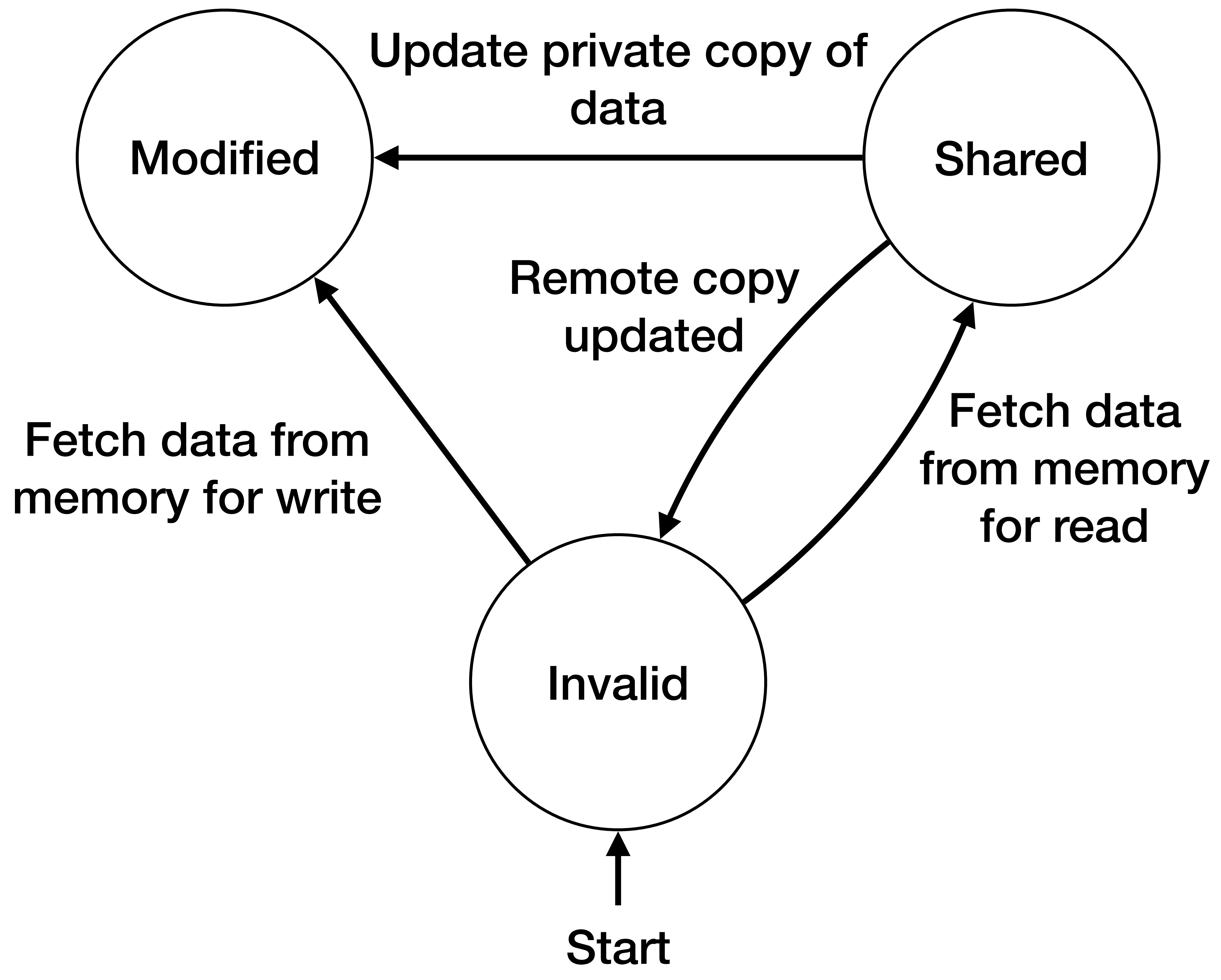
- Sequential accesses to shared data in the memory system provides a consistent view of data in all caches ✅
- Requires updating the cache block structure to track which processor "owns" the block, spinning and waiting is bad! ❌
  - ➡️ can be reduced to a single *blocked* bit so long as the operation is a `swap`… the current modifying processor is not relevant to consistency protocol
- Requires *write-through* update protocol to ensure that initial reads of shared data acquire the right value! This and invalidates means lots of traffic… ❌

Reading Shared Values

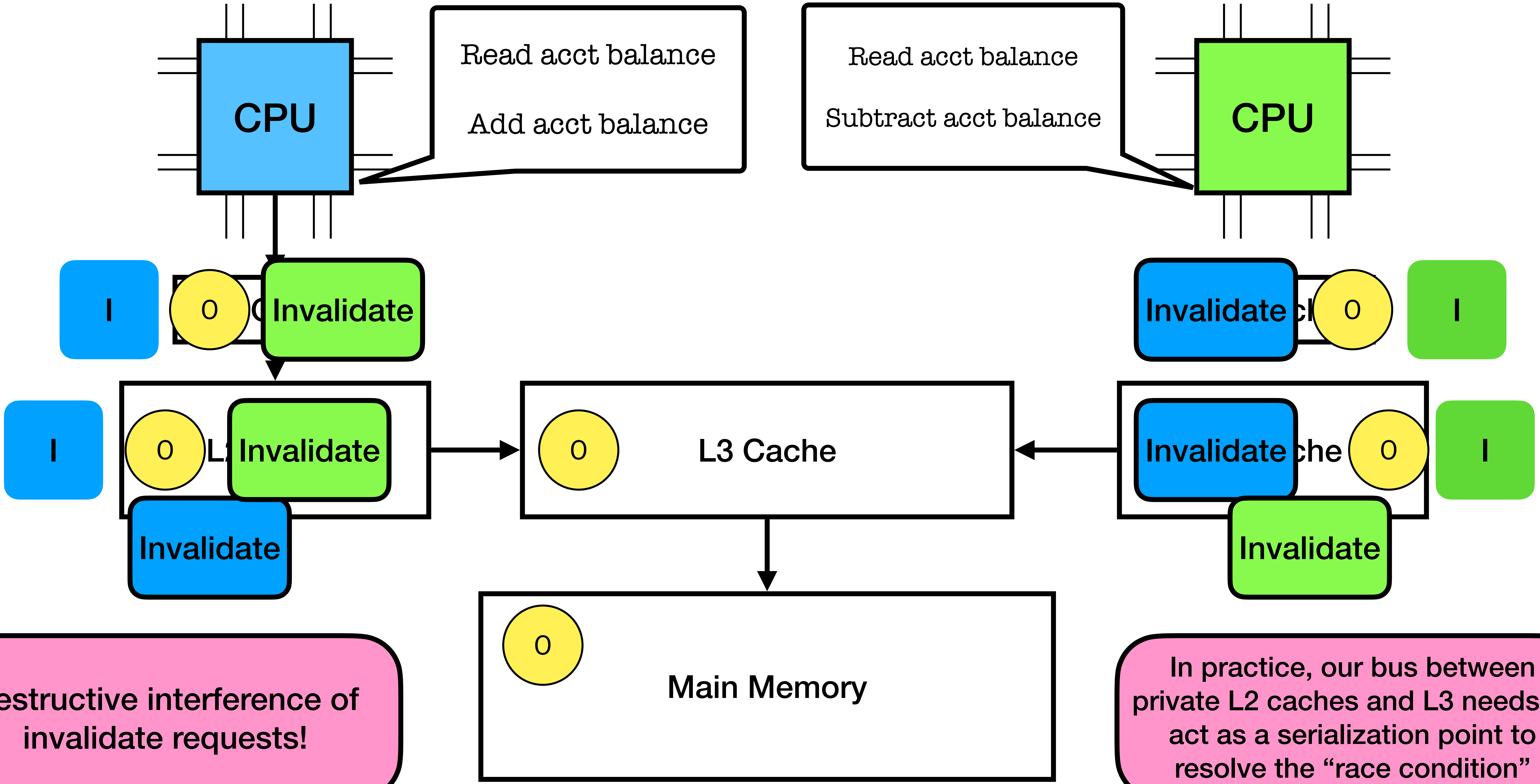


Implementing Simple Cache Coherence

- MSI coherence: at most one processor can own a cache block in *modified* state
- To update a block, first send an *invalidate* to other caches in the memory system for this address
- Once the invalidate has responded, the updating cache can safely set the cache block state from *shared* to *modified*


Modified
Update private copy of data
Shared
Remote copy updated
Fetch data from memory for write
Fetch data from memory for read
Invalid
Start

MSI Coherence



Destructive interference of invalidate requests!

In practice, our bus between private L2 caches and L3 needs to act as a serialization point to resolve the “race condition”

MOESI Coherence (Used in AMD64!)

- Two notable disadvantages of MSI are:
  - 1️⃣ every update requires notifying the rest of the memory hierarchy by sending an invalidate request… this is a lot of traffic!
  - 2️⃣ there is no conception of holding a cache line exclusively
- The *MOESI* coherence protocol extends MSI to also include an *owned* state and an *exclusive* state
- More context about what else is happening in the cache hierarchy means that more operations can safely be performed on caches across the memory system

Takeaways

- Shared caches present new challenges when thinking about the consistency of data
- Coherence protocols implement behaviors to reason about the consistency and private utilization of data across different components
- Coherence tends to add lots of traffic to the memory system, more context often means less traffic!