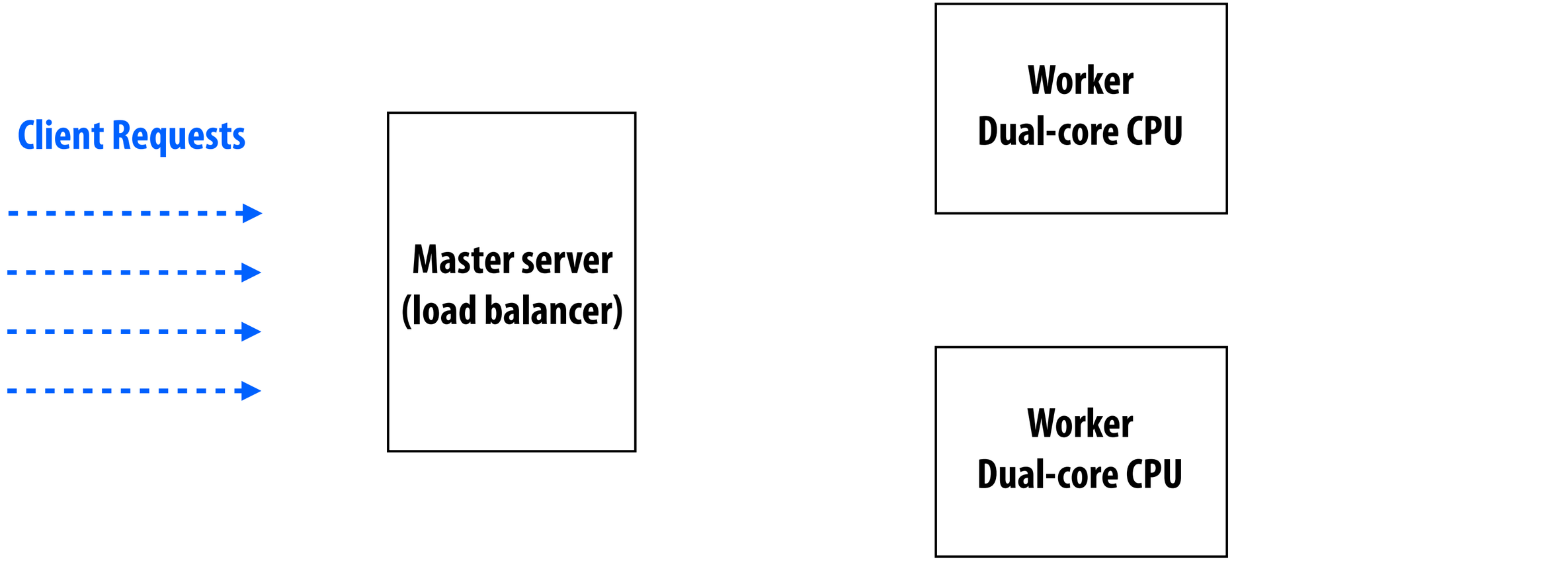
Lecture 16:

A More Sophisticated Snooping-Based Multi-Processor

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013

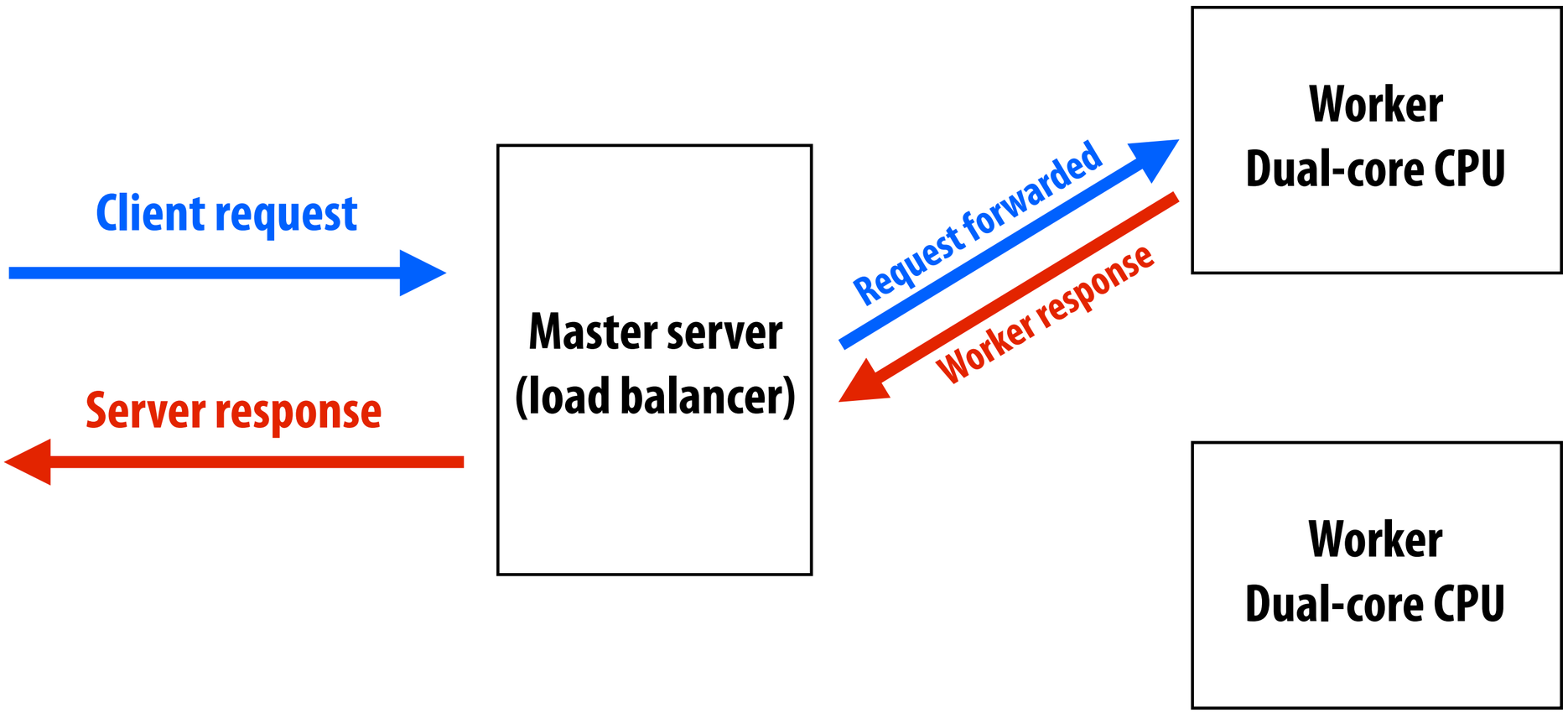
Assignment 4 (finally!)

- **Will be released late tonight or early Friday**
  (due at 11:59PM in two weeks from release date)
- **You will implement a simple web site that efficiently handles a request stream**



Assignment 4 (finally!)

- Will be released tonight (due in two weeks: April 2nd)
- You will implement a load balancer/scheduler to efficiently handle a request stream



Assignment 4: the master node

- The master is a work scheduler
- Structured as an event-driven system

Master server (load balancer)

You implement:

```
// take action when a request comes in
void handle_client_request(void* client_handle, const RequestMsg& req);

// take action when a worker provides a response
void handle_worker_response(void* worker_handle, const ResponseMsg& resp);
```

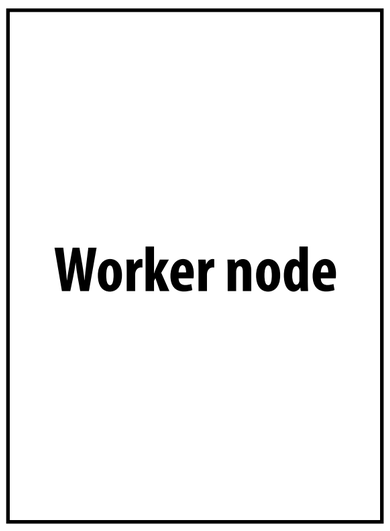
We give you:

```
// sends a request to a worker
void send_job_to_worker(void* worker_handle, const RequestMsg& req);

// sends a response to the client
void send_client_response(void* connection_handle, const ResponseMsg& resp);
```

Assignment 4: the worker nodes

- The worker nodes are responsible for the "heavy lifting" (executing the specified requests)



You implement:

```
// take action when a request comes in
void worker_handle_request(const RequestMsg& req);
```

We give you:

```
// send a response back to the master
void worker_send_response(const ResponseMsg& resp);


// black-box logic to actually do the work (and populate a response)
void execute_work(const RequestMsg& req, ResponseMsg& resp);
```

Assignment 4: challenge 1

- There a number of different types of requests with different workload characteristics
  - Compute intensive requests (both long and short)
  - Disk intensive requests

```
{"time": 0,  "work": "cmd=highcompute;x=5", "resp": "42"}
{"time": 10, "work": "cmd=highcompute;x=10", "resp": "59"}
{"time": 20, "work": "cmd=highcompute;x=15", "resp": "78"}
{"time": 21, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 22, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 23, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 24, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 30, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 40, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 50, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
```

Assignment 4: challenge 2

- The load varies over time!

```
{"time": 0,  "work": "cmd=highcompute;x=5", "resp": "42"}
{"time": 10, "work": "cmd=highcompute;x=10", "resp": "59"}
{"time": 20, "work": "cmd=highcompute;x=15", "resp": "78"}
{"time": 21, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 22, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 23, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 24, "work": "cmd=highcompute;x=20", "resp": "10"}
{"time": 30, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 40, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
{"time": 50, "work": "cmd=popular;start=2013-02-13;end=2013-03-23", "resp": "lecture/cachecoherence1 -- 856 views"}
```

We give you:

```
// ask for another worker node
void request_boot_worker(int tag);

// request a worker be shut down
void kill_worker(void* worker_handle);
```

You implement:

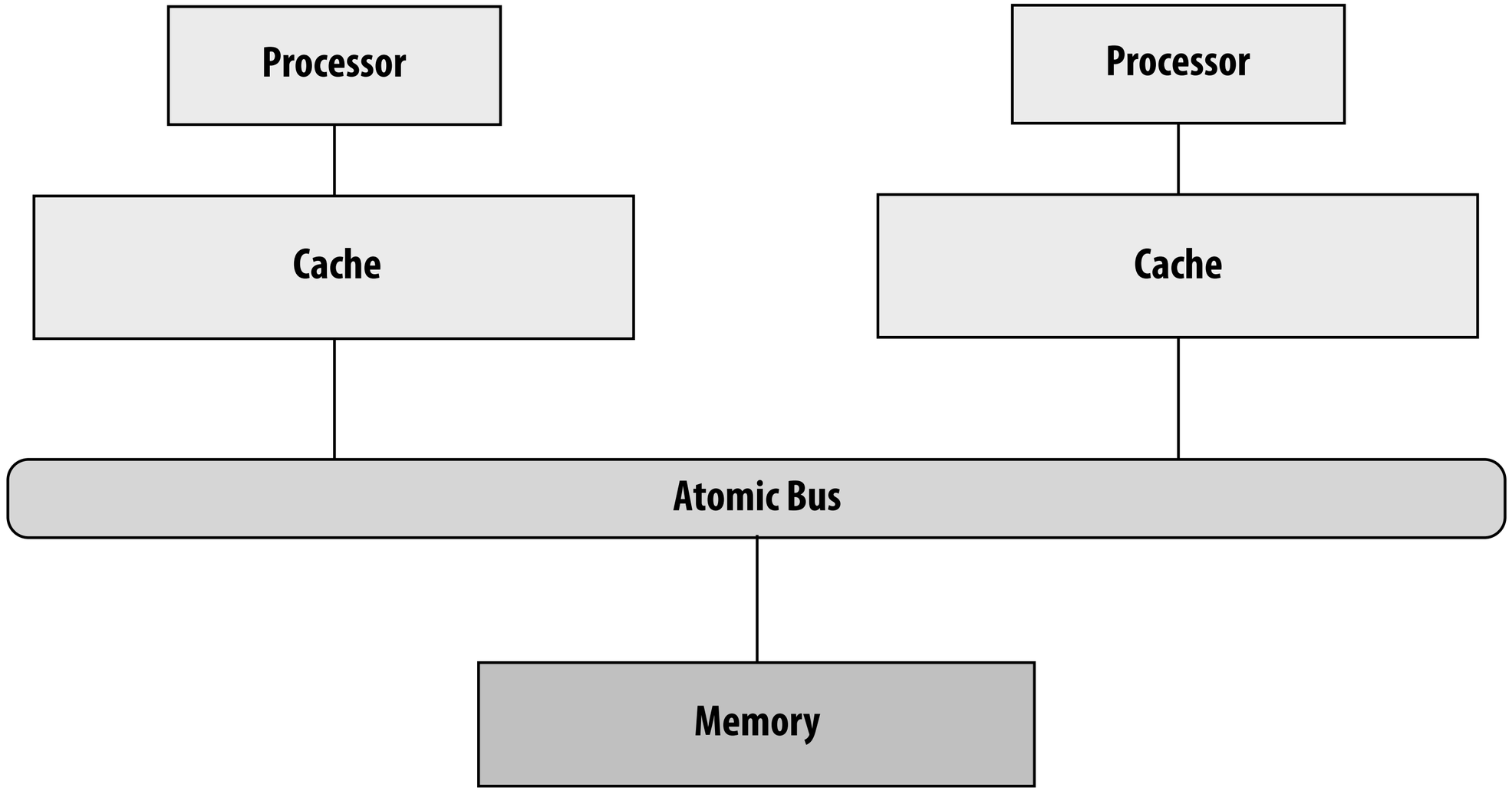
```
// notification that the worker is up and running
void handle_worker_boot(void* worker_handle, int tag);
```

Assignment 4

- Goal: service the request stream as efficiently as possible (low latency response time) using as few workers as possible (low website operation cost)

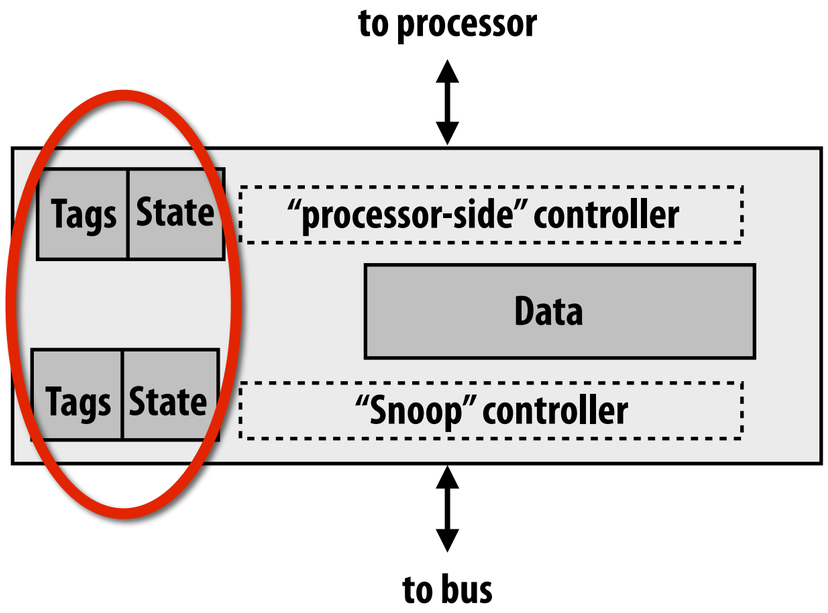
- Ideas you might want to consider:
  - What is a smart assignment of jobs to workers?
  - When to [request more/release idle] worker nodes?
  - Can costs be reduced by caching?

Last time we implemented a very simple cache-coherent multi-processor using an atomic bus as an interconnect



Key issues

We addressed the issue of contention for access to cache tags by duplicating tags.

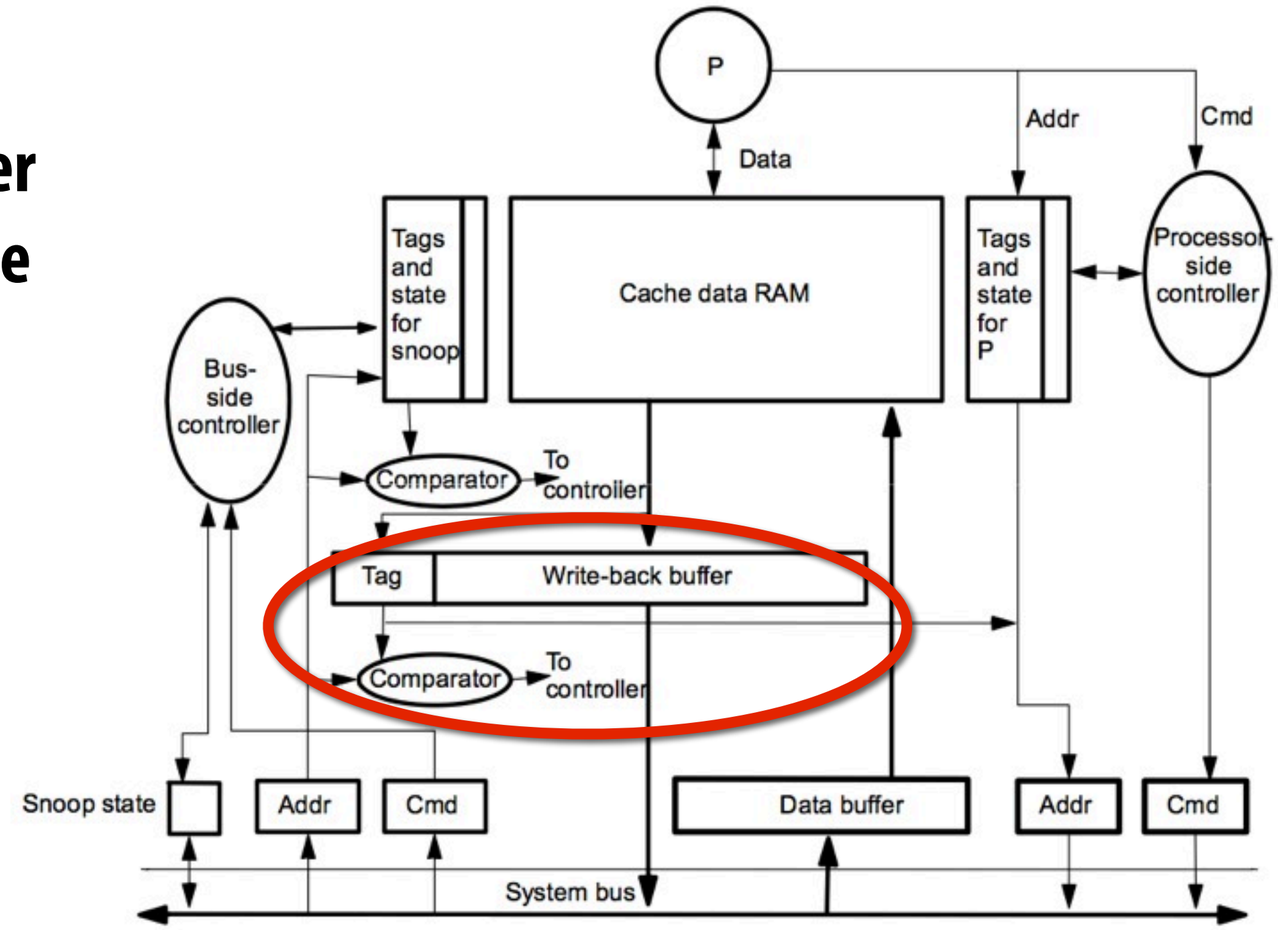


We described how snoop results can be collectively reported to a cache via shared, dirty, and valid lines on the bus.

Key issues

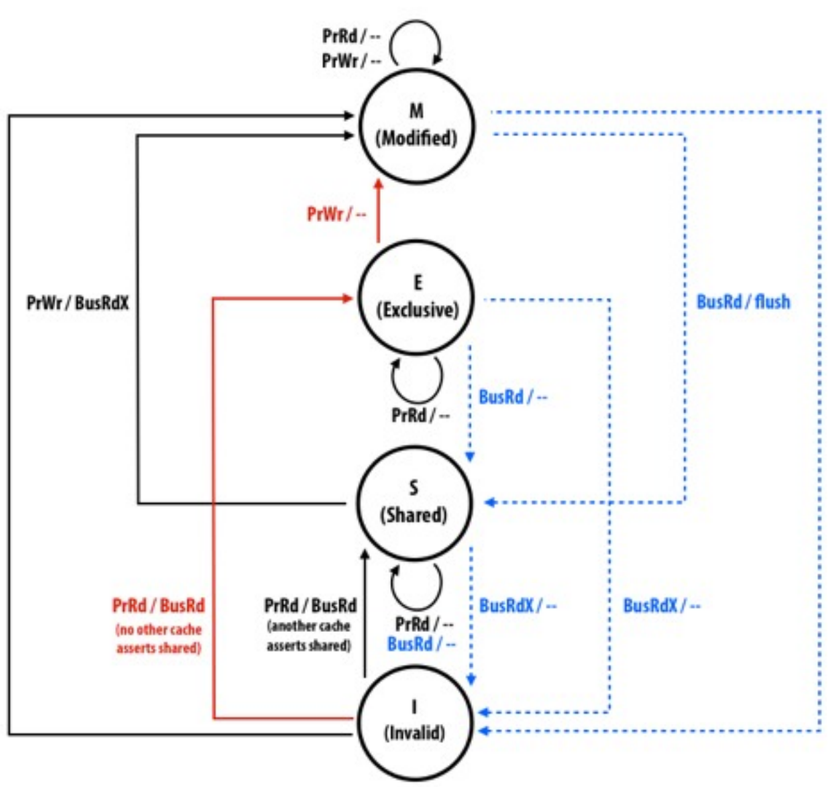
We addressed correctness issues that arise from the use of a write-back buffer by checking both the cache tags and the write-back buffer when snooping.

(and also added the ability to cancel pending bus transfer requests).



Key issues

We talked about ensuring write serialization: processor is held up by the cache until the "I want exclusive access" transaction appears on the bus (this is when the write "commits").



We talked about how coherence protocol state transitions are not really atomic operations in a real machine (even though the bus itself it atomic), leading to possible race conditions.

We discussed deadlock, livelock, and starvation

Situation 1:

P1 has a modified copy of cache line X

P1 is waiting for the bus to issue BusRdX on cache line Y

BusRd for X appears on bus while P1 is waiting

FETCH DEADLOCK!

To avoid deadlock, P1 must be able to service incoming transactions while waiting to issue its own requests

Livelock

Situation 2:

Two processors simultaneously write to cache line X (in S state)

P1 acquires bus access ("wins bus"), issues BusRdX

P2 invalidates its copy of the line in response to P1's BusRdX

Before P1 performs the write (updates block), P2 acquires bus and issues BusRdX

P1 invalidates in response to P2's BusRdX

LIVELOCK!

To avoid livelock, write that obtains exclusive ownership must be allowed to complete before exclusive ownership is relinquished.

Today's topics

Now we will build the system around a non-atomic bus transactions.

Optimize

Re-evaluate correctness

[and so on...]

Optimize

Re-evaluate correctness

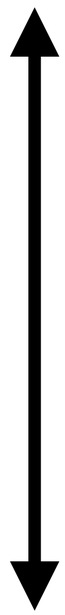
[and so on...]

What you should know

- How deadlock and livelock might occur in both atomic bus and non-atomic bus-based systems (what are possible solutions for avoiding it?)
- What is a major performance issue atomic bus transactions?
- The main components of a split-transaction bus, how transactions are split into requests and responses
- The role of queues in a parallel system (today is yet another example)

Transaction on an atomic bus

1. Client is granted bus access (result of arbitration)
2. Client places command on bus (may also place data on bus)



Problem: bus is idle while response is pending (decreases effective bus bandwidth)

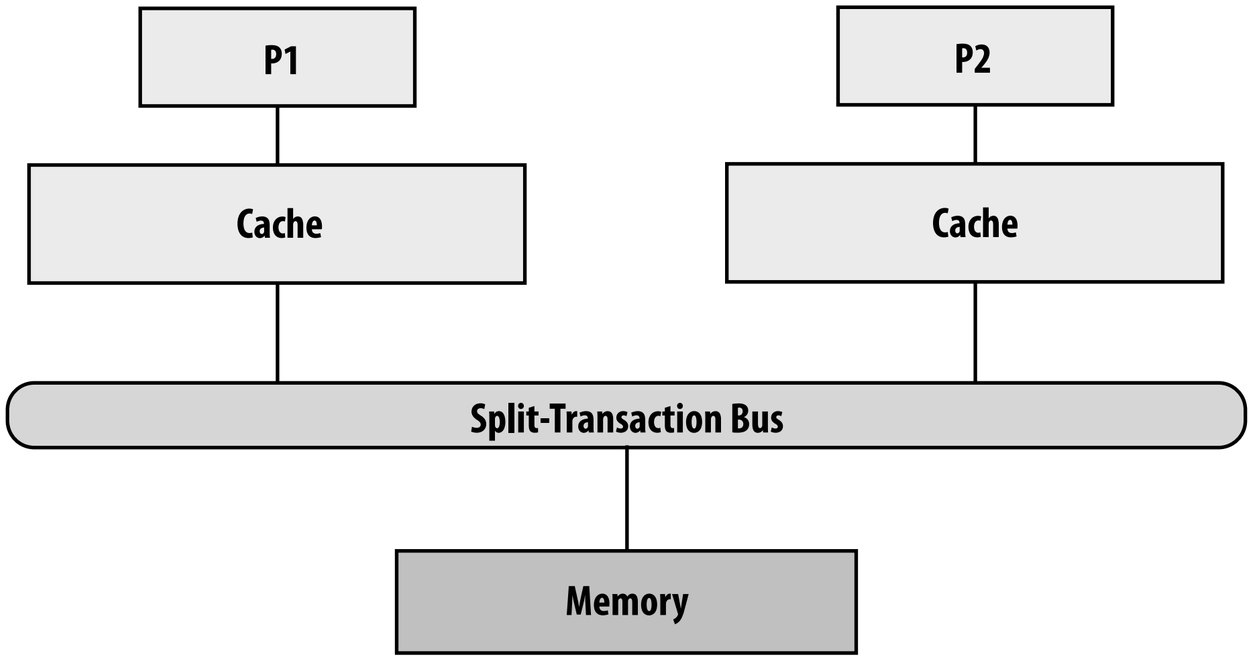
This is bad, because the interconnect is often a limited, shared resource in a multi-processor system. (It is important to use it as efficiently as possible)

3. Response to command by another bus client placed on bus
4. Next client obtains bus access (arbitration)

Split-transaction bus

Bus transactions are split into two transactions: a request and a response

Other transactions can intervene.



Consider:
Read miss to A by P1
Bus upgrade of B by P2

P1 gains access to bus

P1 sends BusRd command

[memory starts fetching data]

P2 gains access to bus

P2 sends BusUpg command

Memory gains access to bus

Memory places A on bus

New issues due to split transactions

1. How to match requests with responses?

2. Conflicting requests on bus

   Consider:

   - P1 has outstanding request for line A
   - Before response to P1 occurs, P2 makes request for line A

3. Flow control: how many requests can be outstanding at a time, and what should be done when buffers fill up?

4. When are snoop results reported? During the request? During the response?

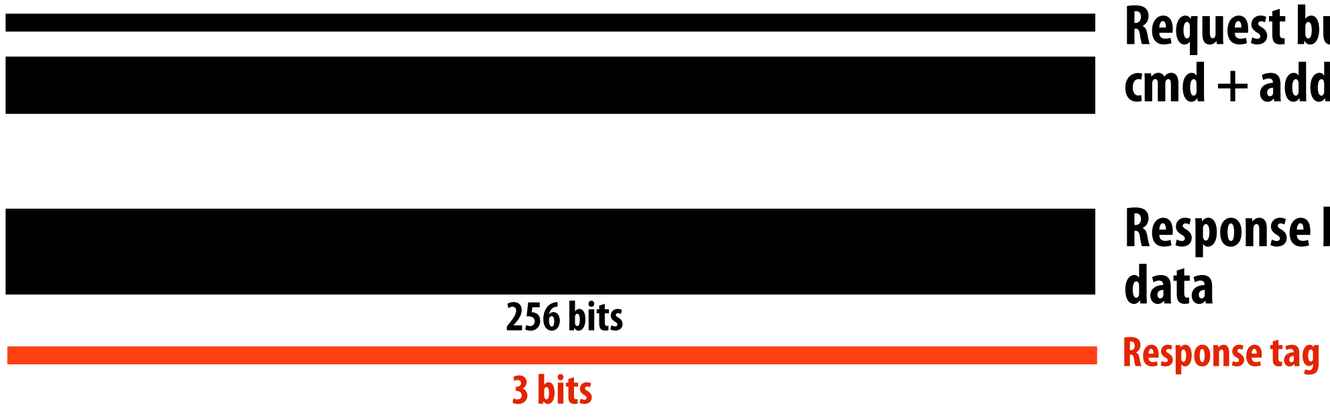
A basic design

- Up to eight outstanding requests at a time systemwide
- Responses need not be in the same order as requests
  - But request order establishes the total order for the system
- Flow control via negative acknowledgements (NACKs)
  - When a buffer is full, client can NACK a transaction, causing a retry

Initiating a request

Can think of a split-transaction bus as two separate buses:

a request bus and a response bus.



Request Table
(assume a copy of this table is maintained by each bus client: e.g., cache)

Transaction tag is index into table →

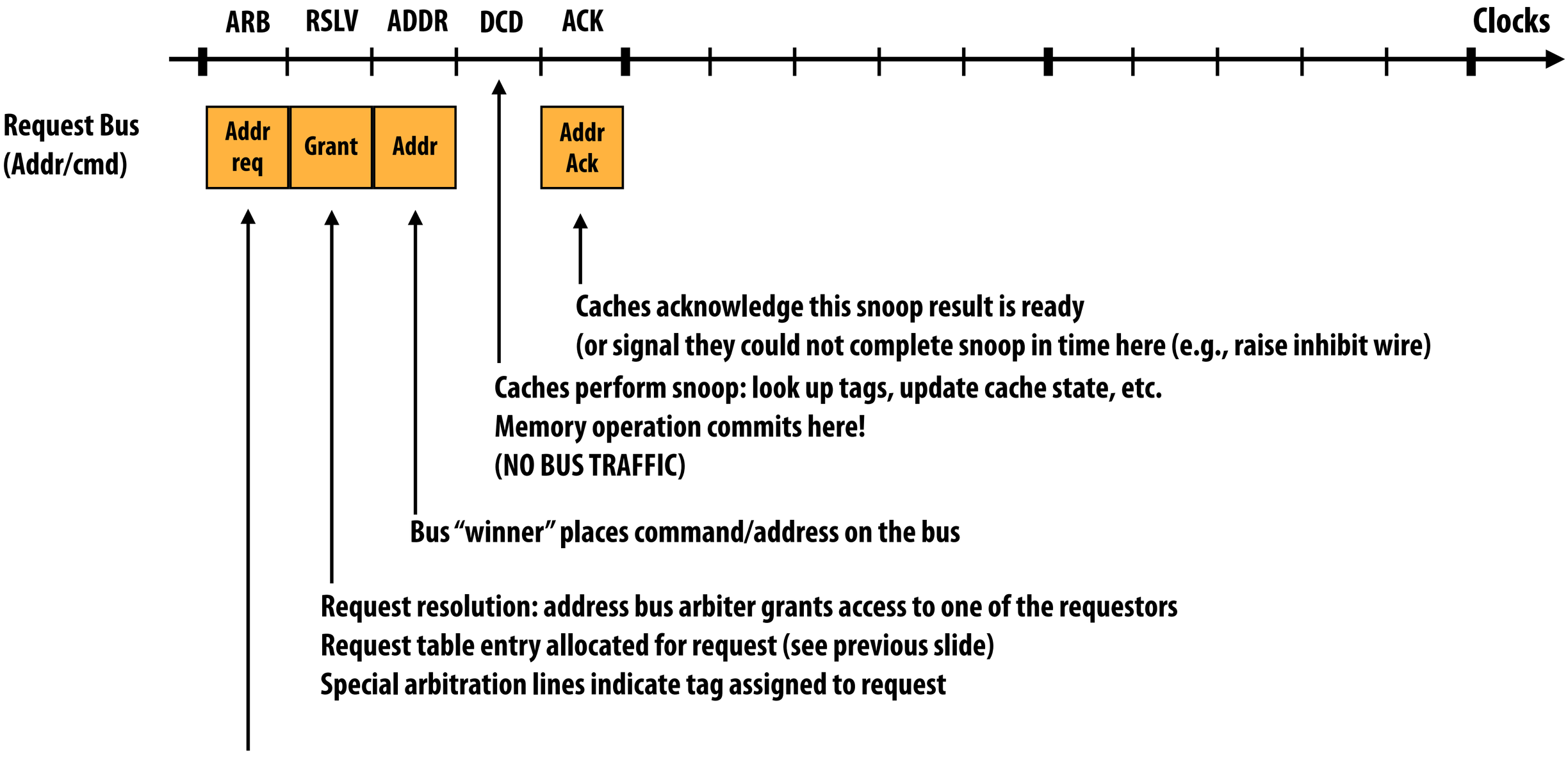
Requestor	Addr	State
P0	0xbeef	

Step 1: Requestor asks for request bus access

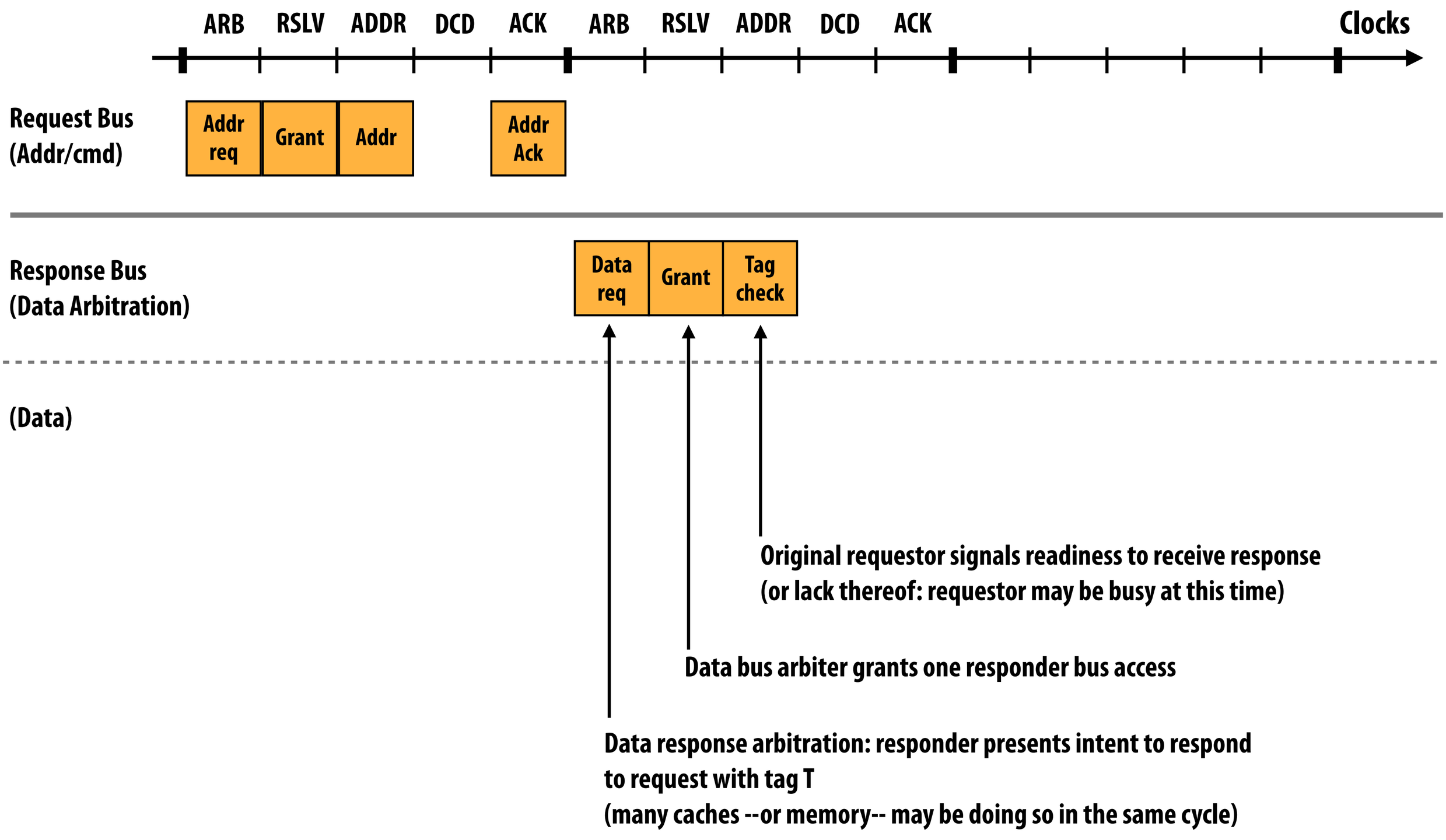
Step 2: Bus arbiter grants access, assigns transaction a tag

Step 3: Requestor places command + address on the request bus

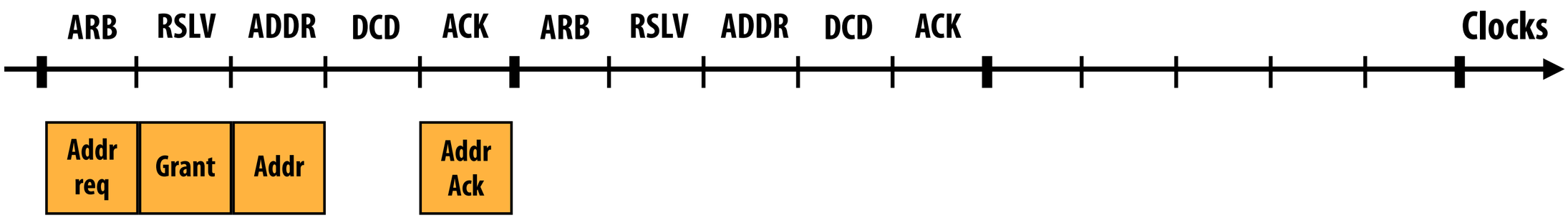
Read miss: cycle-by-cycle bus behavior (phase 1)



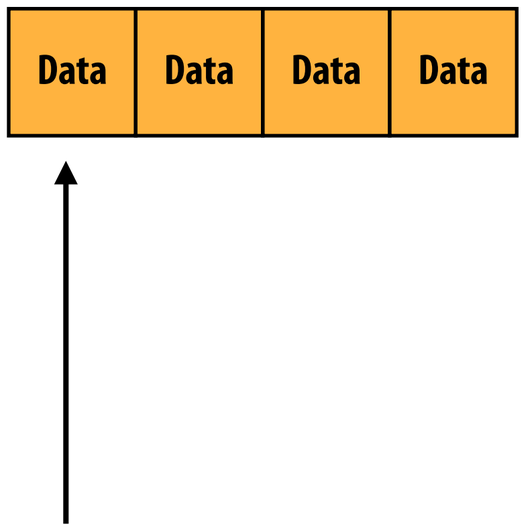
Read miss: cycle-by-cycle bus behavior (phase 2)



Read miss: cycle-by-cycle bus behavior (phase 3)



(Data)



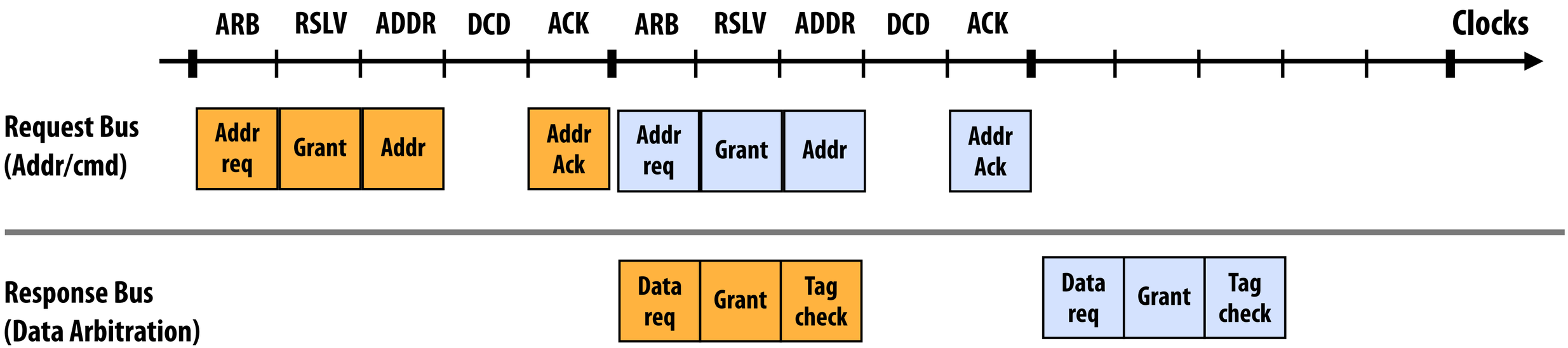
Responder places response data on data bus

Caches present snoop result for request with the data

Request table entry is freed

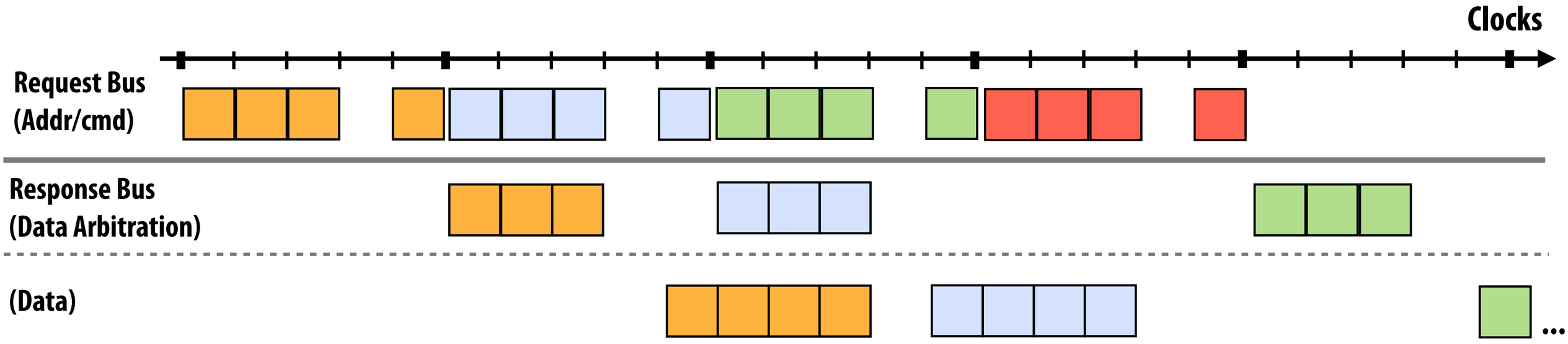
Here: assume 128 byte cache lines → 4 cycles on 256 bit bus

Pipelined transactions



Note: write backs and BusUpg transactions do not have a response component (write backs acquire access to both request address bus and data bus as part of "request" phase)

Pipelined transactions



= memory transaction 1

= memory transaction 2

= memory transaction 3

= memory transaction 4

Key issues to resolve

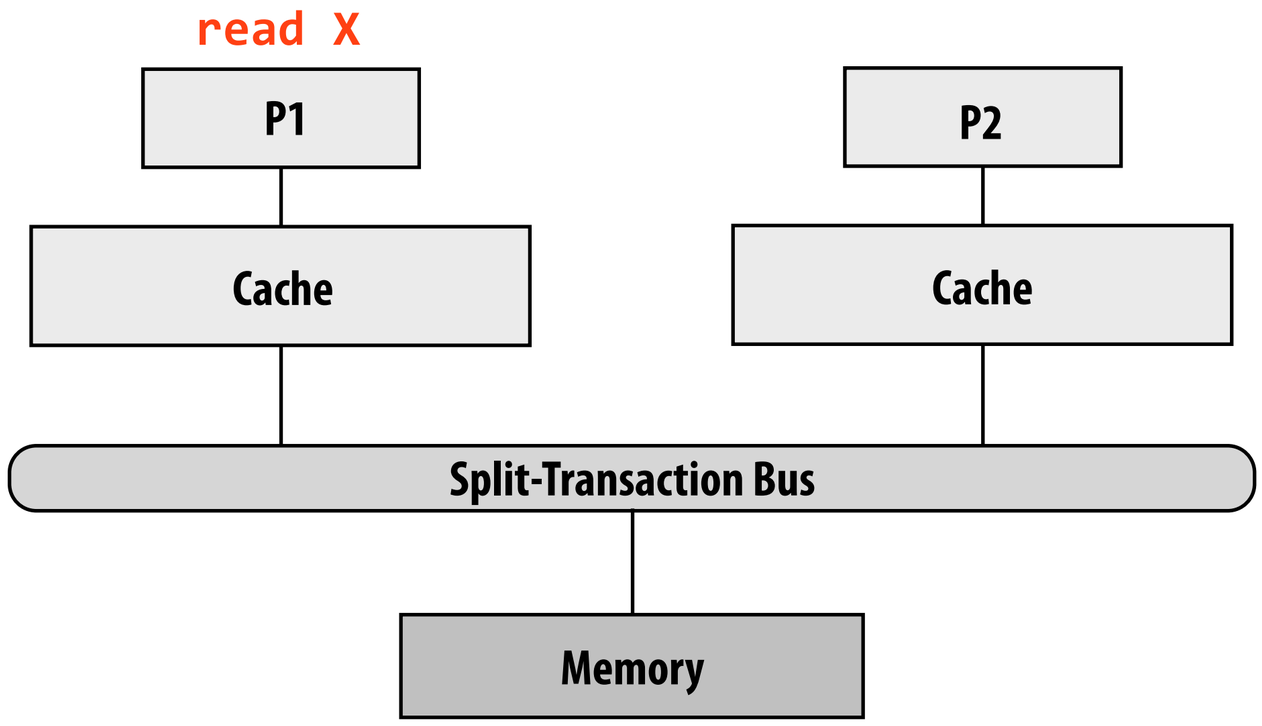
- Conflicting requests
  - Avoid conflicting requests by disallowing them
  - Each cache has a copy of the request table
  - Policy: caches do not make requests that conflict with requests in the request table

- Flow control:
  - Caches/memory have buffers for receiving data off the bus
  - If the buffer fills, client NACKs relevant requests or responses (NACK = negative acknowledgement)
  - Triggers a later retry

Situation 1: P1 read miss to X, transaction involving X is outstanding on bus

P1 Request Table

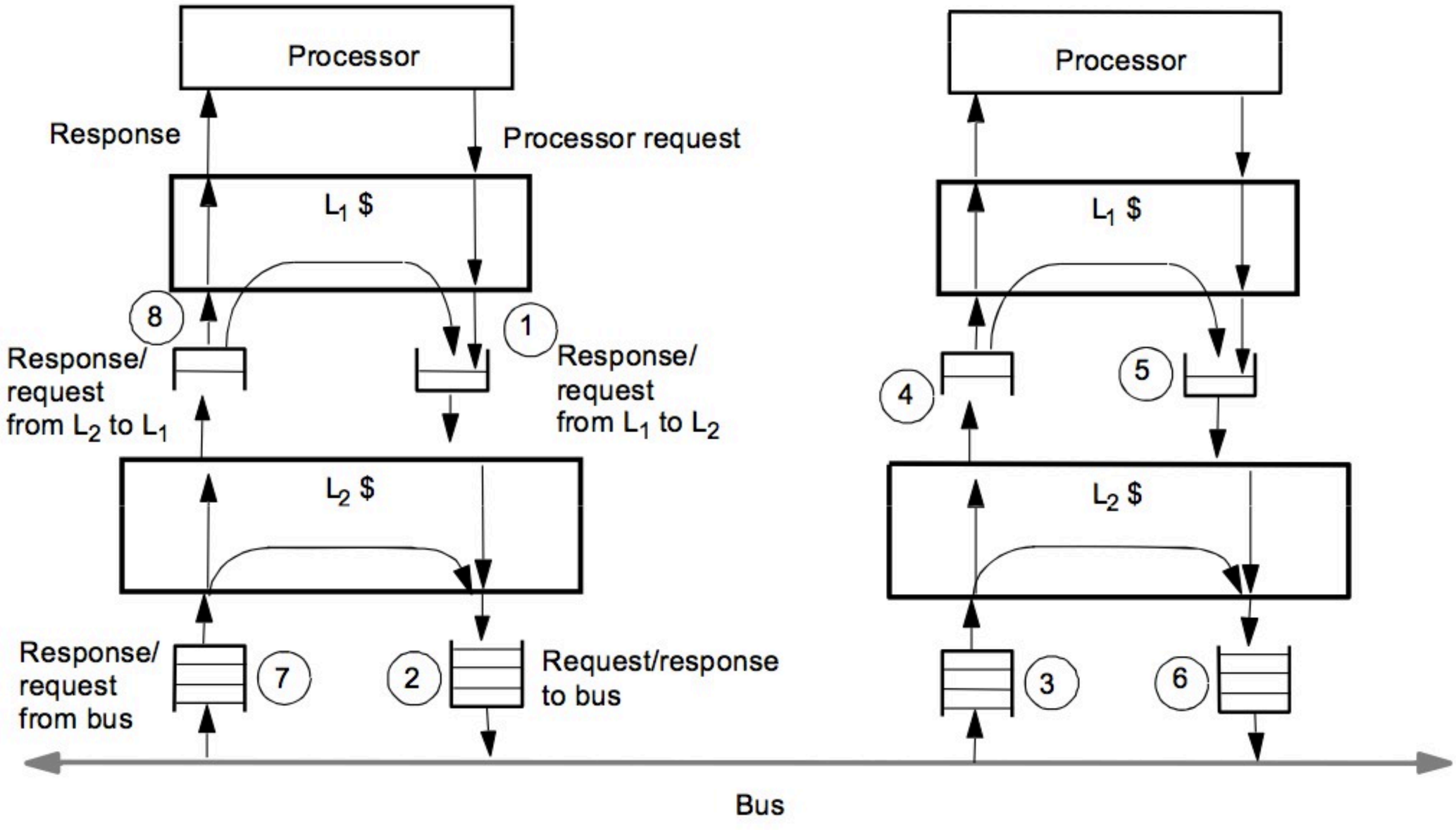
Requestor	Addr	State
P2	X	Op: BusRdX , share



If there is a conflicting outstanding request (as determined by checking the request table), cache must hold request until conflict clears

If outstanding request is a read: there is no conflict. No need to make a new bus request, just listen for the response to the outstanding one.

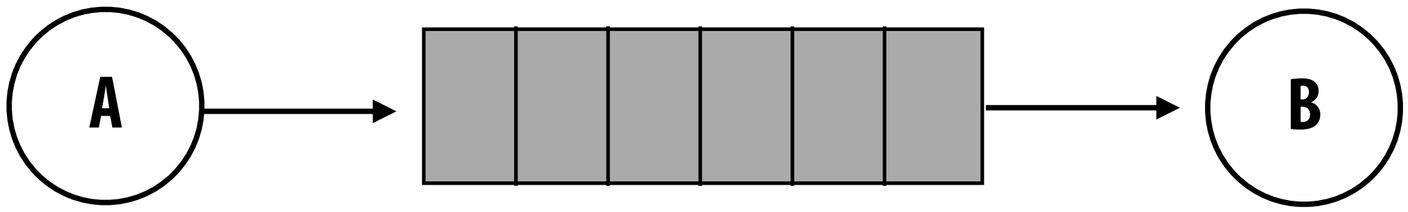
Multi-level cache hierarchies



Assume one outstanding memory request per processor.

Consider fetch deadlock problem: cache must be able to service requests while waiting on response to its own request (hierarchies increase response delay)

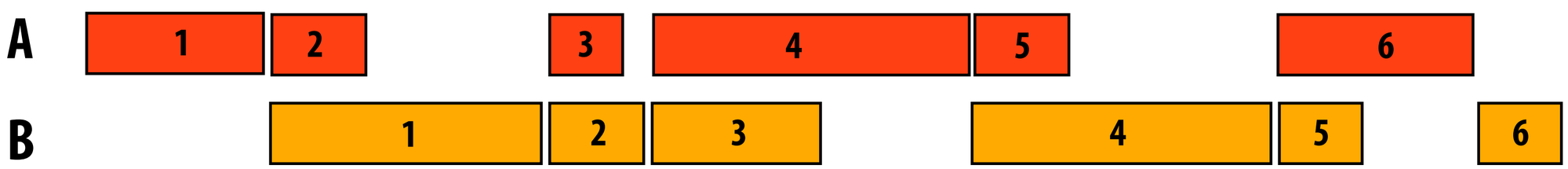
Why do we have queues?



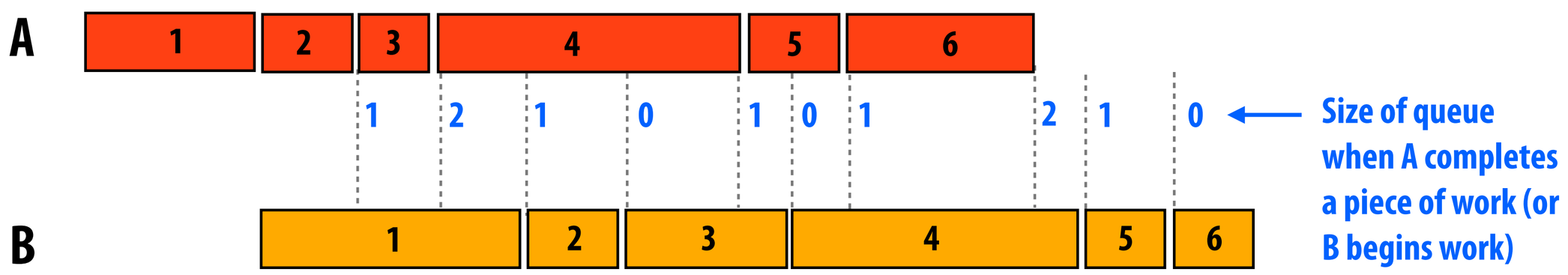
To accommodate variable (unpredictable) rates of production and consumption.

As long as A and B, on average, produce and consume at the same rate, both workers can run at full rate.

No queue: stalls exist

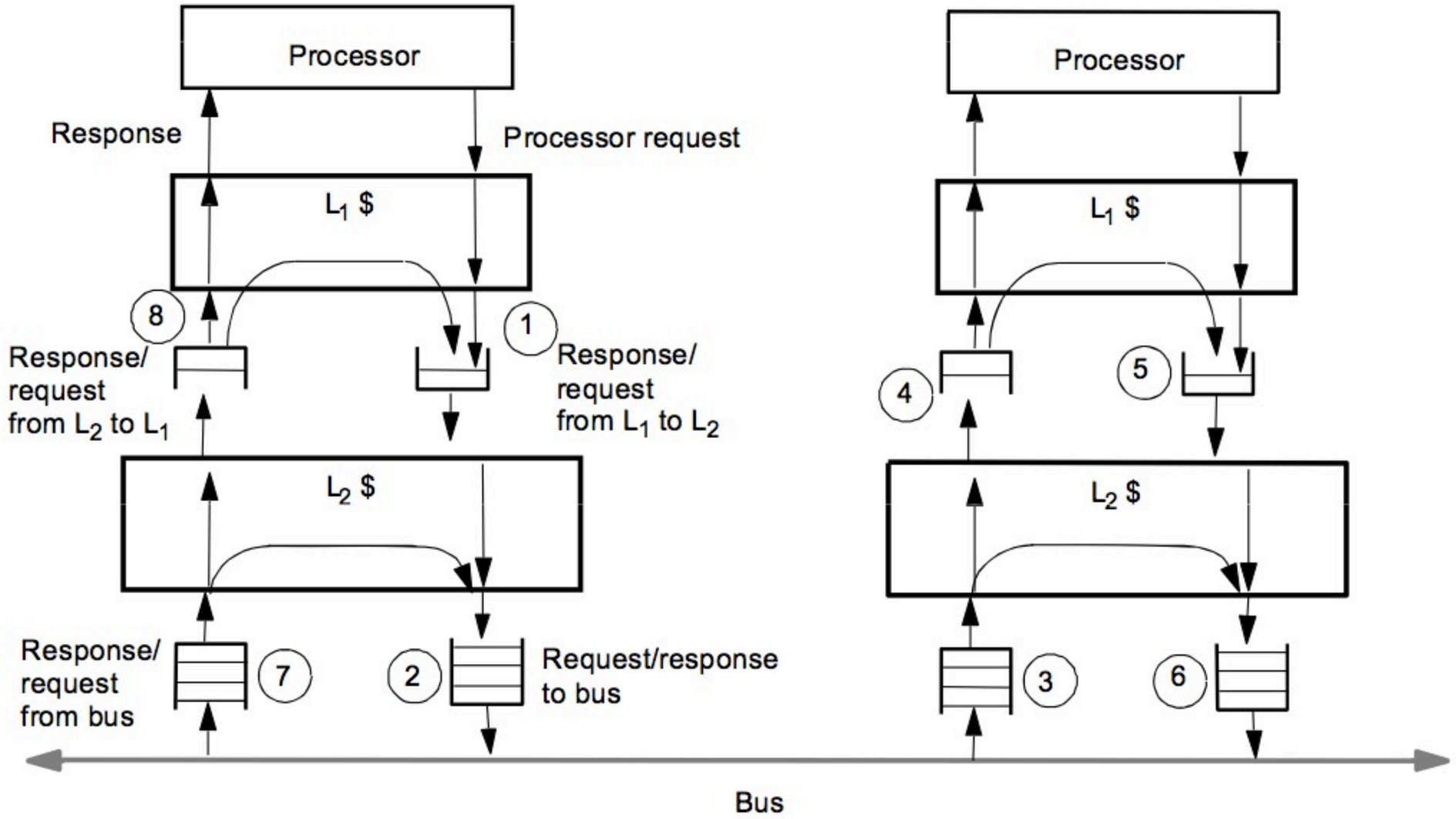


With queue of size 2: A and B never stall



Size of queue when A completes a piece of work (or B begins work)

Multi-level cache hierarchies

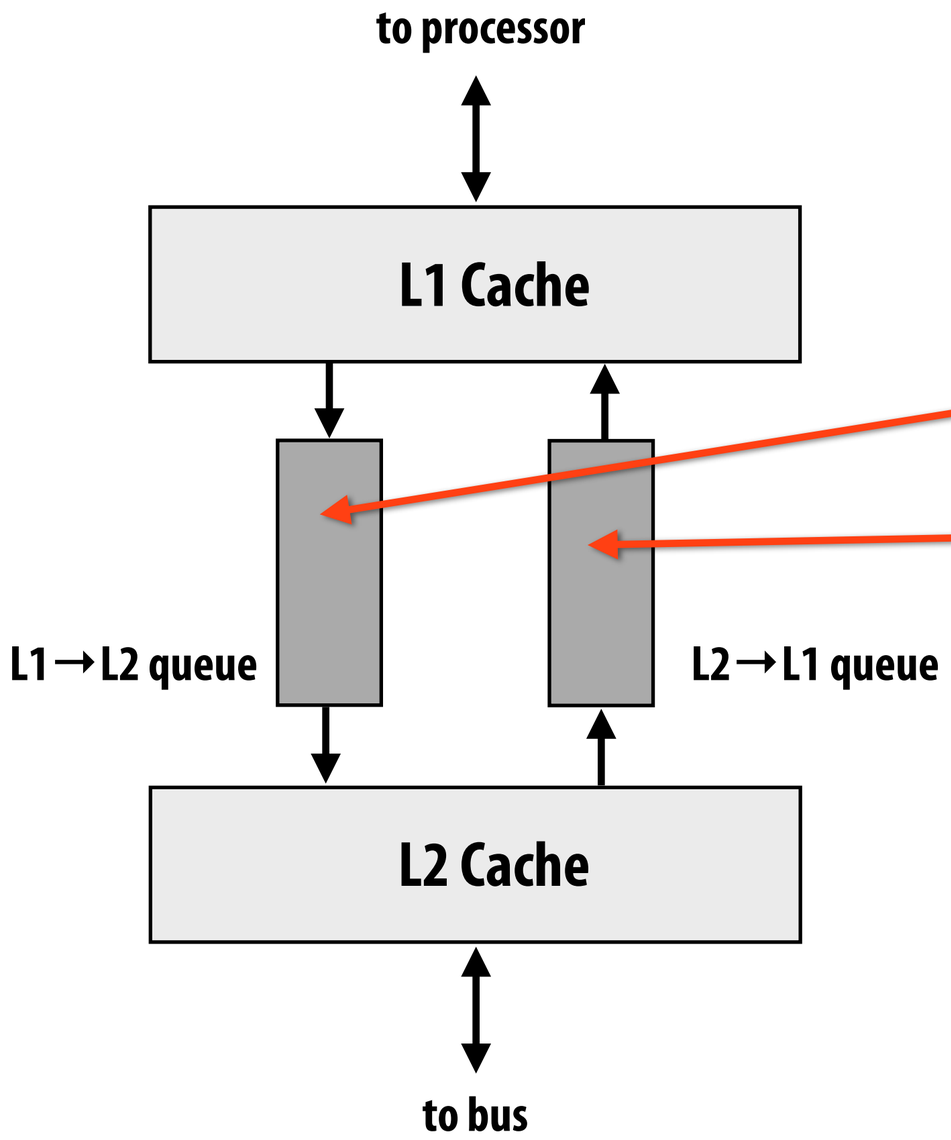


Assume one outstanding memory request per processor.

Consider fetch deadlock problem: cache must be able to service requests while waiting on response to its own request (hierarchies increase response delay)

Ideally, would like buffering at each cache for all requests that can be outstanding on bus.

Buffer deadlock



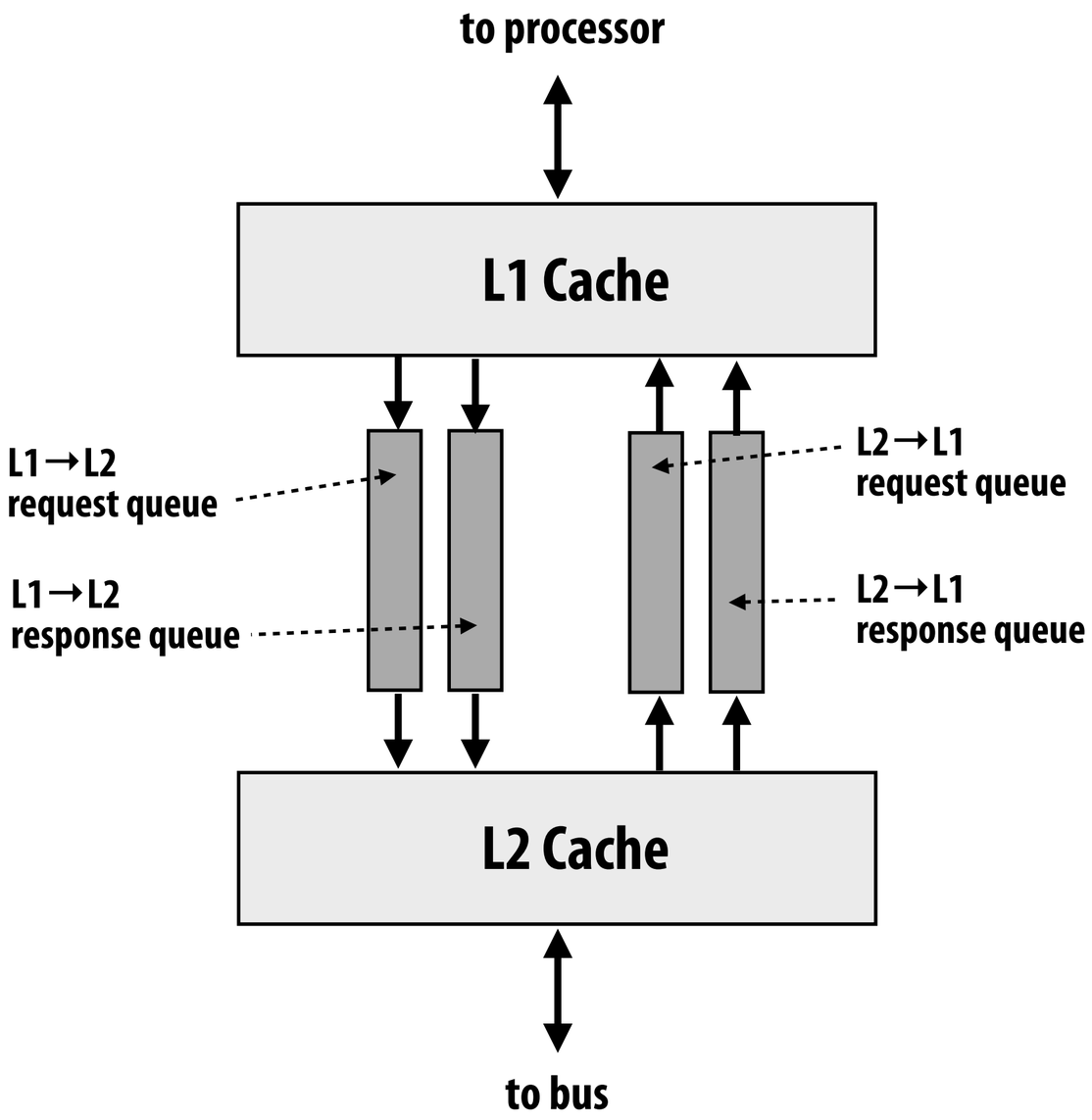
Outgoing read request (initiated by this processor)

Incoming read request (due to another cache) **

Both requests generate responses that require space in the other queue (circular dependency)

** will only occur if L1 is write back

Avoiding buffer deadlock



Classify all transactions as requests and responses

Responses can be completed without generating further transactions

While stalled attempting to send a request, cache must be able to service responses.

Responses will make progress (they generate no new work so there's no circular dependence), eventually freeing up resources for requests

** will only occur if L1 is write back

Putting it all together

Class exercise: describe everything that might occur during the execution of this statement

```
int x = 10;      // assume write to memory, not stored in register
```

Class exercise: describe everything that might occur during the execution of this statement

```
int x = 10;
```

Virtual address to physical (TLB lookup)
TLB miss
TLB update (might involve OS)
OS may need to swap in page (load from disk to physical address)
Cache lookup
Line not in cache (need to generate BusRdX)
Arbitrate for bus
Win bus, place address, command on bus
Another cache or memory decides it must respond (assume memory)
Memory request sent to memory controller
Memory controller is itself a scheduler
Memory checks active row in row buffer. May need to activate new row.
Values read from row buffer
Memory arbitrates for data bus
Memory wins bus
Memory puts data on bus
Cache grabs data, updates cache line and tags, moves line into Exclusive state
Processor notified data exists
Instruction proceeds