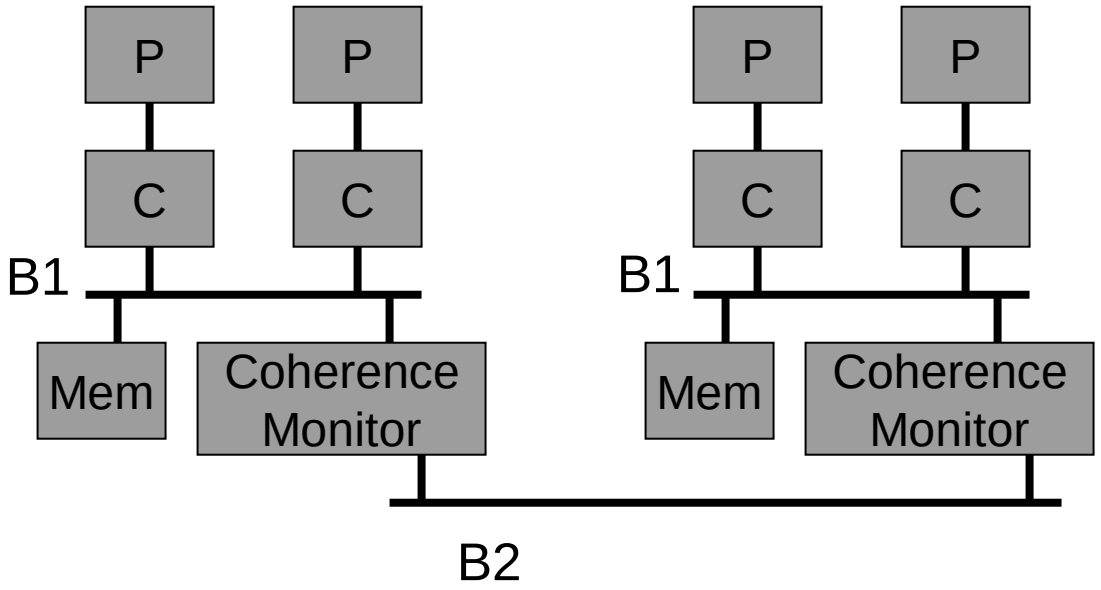
Lecture 12: Hardware/Software Trade-Offs

- Topics: COMA, Software Virtual Memory

Capacity Limitations



In a Sequent NUMA-Q design above,

- A remote access is involved if data cannot be found in the remote access cache
- The remote access cache and local memory are both DRAM

Can we expand cache and reduce local memory?

Cache-Only Memory Architectures

- COMA takes the extreme approach: no local memory and a very large remote access cache
- The cache is now known as an “attraction memory”
- Overheads/issues that must be addressed:
  - Need a much larger tag space
  - More care while evicting a block
  - Finding a clean copy of a block
- Easier to program – data need not be pre-allocated

COMA Performance

- Attraction memories reduce the frequency of remote accesses by reducing capacity/conflict misses
- Attraction memory access time is longer than local memory access time in the CC-NUMA case (since the latter does not involve tag comparison)
- COMA helps programs that have frequent capacity misses to remotely allocated data

COMA Implementation

- Even though the memory block has no fixed home, the directory can continue to remain fixed – on a miss or on a write, contact directory to identify valid cached copies

- In order to not evict the last block, one of the sharers has the block in “master” state – while replacing the master copy, a message must be sent to the directory – the directory attempts to find another node that can accommodate this block in master state

- For high performance, the physical memory allocated to an application must be smaller than attraction memory capacity, and attraction memory must be highly associative

Reducing Cost

- Hardware cache coherence involves specialized communication assists – cost can be reduced by using commodity hardware and software cache coherence

- Software cache coherence: each processor translates the application's virtual address space into its own physical memory – if the local physical memory does not exist (page fault), a copy is made by contacting the home node – a software layer is responsible for tracking updates and propagating them to cached copies – also known as shared virtual memory (SVM)

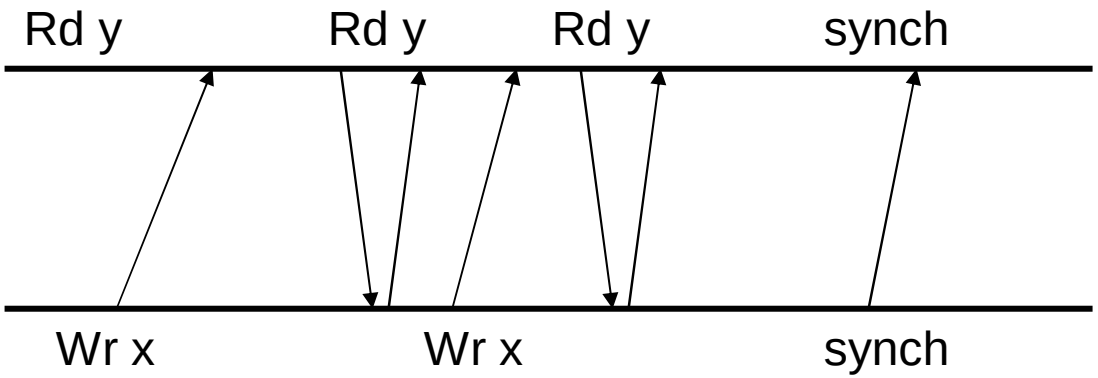
Shared Virtual Memory Performance

- Every communication is expensive – involves OS, message-passing over slower I/O interfaces, protocol processing happens at the processor
- Since the implementation is based on the processor's virtual memory support, granularity of sharing is a page → high degree of false sharing
- For a sequentially consistent execution, false sharing leads to a high degree of expensive communication

Relaxed Memory Models

- Relaxed models such as release consistency can reduce frequency of communication (while increasing programming effort)
- Writes are not immediately propagated, but have to wait until the next synchronization point
- In hardware CC, messages are sent immediately and relaxed models prevent the processor from stalling; in software CC, relaxed models allow us to defer message transfers to amortize their overheads

Hardware and Software CC



- Relaxed memory models in hardware cache coherence hide latency from processor → false sharing can result in significant network traffic

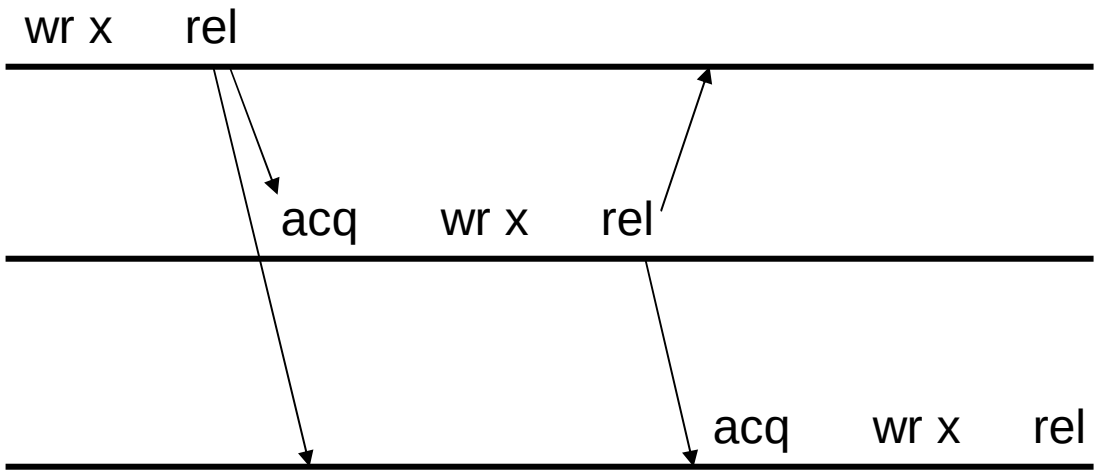
- In software cache coherence, the relaxed memory model sends messages only at synchronization points, reducing the traffic because of false sharing

Eager Release Consistency

- When a processor issues a release operation, all writes by that processor are propagated to other nodes (as updates or invalidates)
- When other processors issue reads, they encounter a cache miss (if we are using an invalidate protocol), and get a clean copy of the block from the last writer
- Does the read really have to see the latest value?

Eager Release Consistency

- Invalidates/Updates are sent out to the list of sharers when a processor executes a release

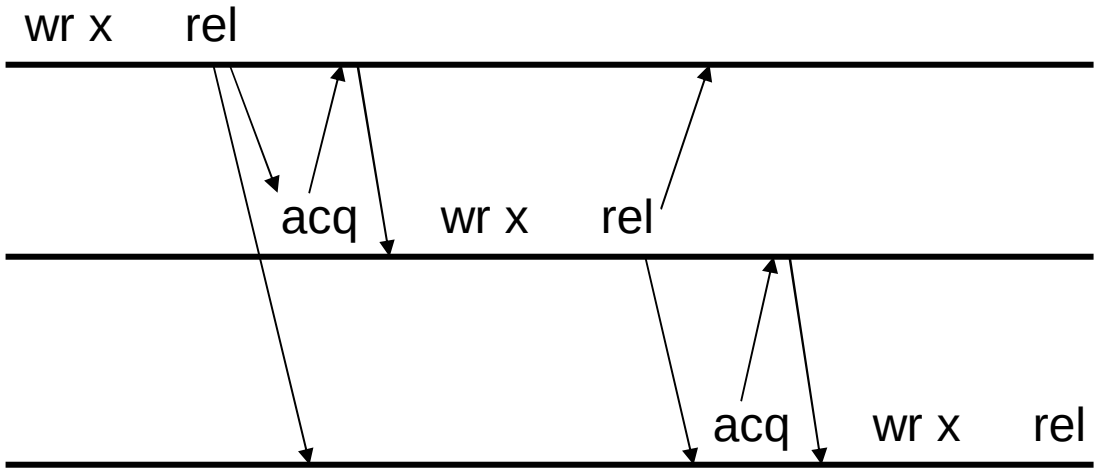


Lazy Release Consistency

- RCsc guarantees SC between special operations
- P2 must see updates by P1 only if P1 issued a release, followed by an acquire by P2
- In LRC, updates/invalidates are visible to a processor only after it does an acquire – it is possible that some processors will never see the update (not true cache coherence)
- LRC reduces the amount of traffic, but increases the latency and complexity of an acquire

Lazy Release Consistency

- Invalidates/Updates are sought when a processor executes an acquire – fewer messages, higher implementation complexity



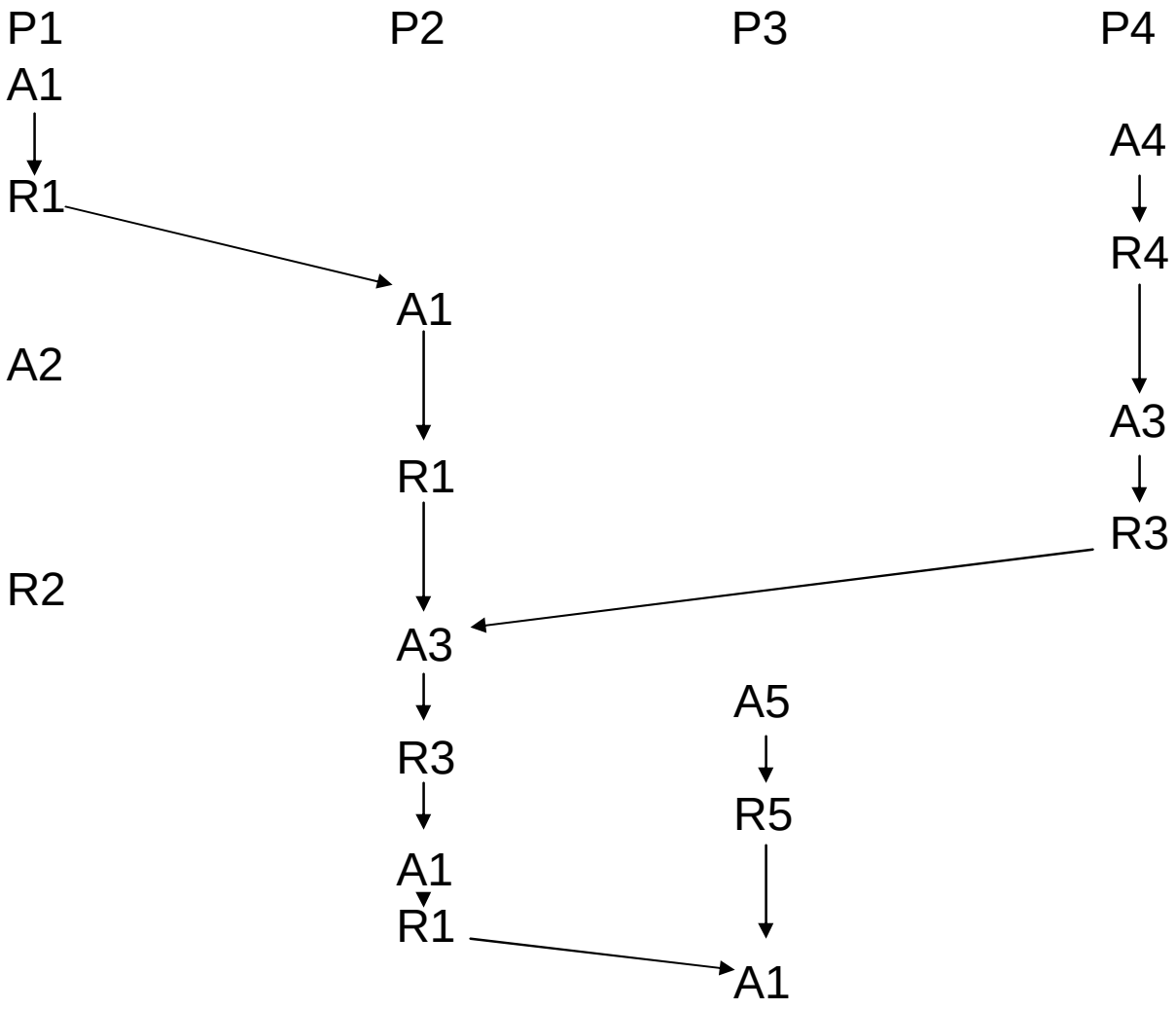
Causality

- Acquires and releases pertain to specific lock variables
- When a process executes an acquire, it should receive all updates that were seen before the corresponding release by the releasing processor
- Therefore, each process must keep track of all write notices (modifications to each shared page) that were applied at every synchronization point

Example

P1	P2	P3	P4
A1			
			A4
R1			
			R4
	A1		
A2			
			A3
	R1		
			R3
R2			
	A3		
		A5	
	R3		
		R5	
	A1		
	R1		
		A1	

Example



LRC Vs. ERC Vs. Hardware-RC

P1

```
lock L1;
ptr  = non_null_value;
unlock L1;
```

P2

```
while (ptr == null) { };
 lock L1;
 a = ptr;
 unlock L1;
```

Implementation

- Each pair of synch operations in a process defines an *interval*
- A partial order is defined on intervals based on release-acquire pairs
- For each interval, a process maintains a vector timestamp of “preceding” intervals: the vector stores the last preceding interval for each process
- On an acquire, the acquiring process sends its vector timestamp to the releasing process – the releasing process sends all write notices that have not been seen by acquirer

LRC Performance

- LRC can reduce traffic by more than a factor of two for many applications (compared to ERC)
- Programmers have to think harder (causality!)
- High memory overheads at each node (keep track of vector timestamps, write notices) – garbage collection helps significantly
- Memory overheads can be reduced by eagerly propagating write notices to processors or a home node – will change the memory model again!

Multiple Writer Protocols

- It is important to support two concurrent writes to different words within a page and to merge the writes at a later point
- Each process makes a twin copy of the page before it starts writing – updates are sent as a diff between the old and new copies – after an acquire, a process must get diffs from all releasing processes and apply them to its own copy of the page
- If twins are kept around for a long time, storage overhead increases – it helps to have a home location of the page that is periodically updated with diffs

Simple COMA

- SVM takes advantage of virtual memory to provide easy implementations of address translation, replication, and replacement
- These can be applied to the COMA architecture
- Simple COMA: if virtual address translation fails, the OS generates a local copy of the page; when the page is replaced, the OS ensures that the data is not lost; if data is not found in attraction memory, hardware is responsible for fetching the relevant cache block from a remote node (note that physical address must be translated back to virtual address)

Title

- Bullet