Parallel Arch. Review

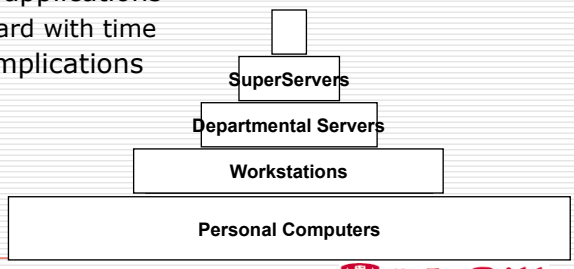
Zeljko Zilic
McConnell Engineering Building
Room 536

Main Points

- Understanding of the design and engineering of modern parallel computers
  - Technology forces
  - Fundamental architectural issues
    - Naming, replication, communication, synchronization
  - Basic design techniques
    - Cache coherence, protocols, networks, pipelining, …
  - Methods of evaluation
  - Underlying engineering trade-offs
- From moderate to large scale
- Across the hardware/software boundary

Useful?

- Absolutely! Not only for PP designers
- The fundamental issues and solutions translate across a wide spectrum of systems.
  - Crisp solutions in the context of parallel machines.
- Pioneered at the thin-end of the platform pyramid on the most-demanding applications
  - Migrate downward with time
- Understand SW implications



What is Parallel Architecture?

- A *parallel computer* is a collection of processing elements that cooperate to solve large problems fast
- Some broad issues:
  - Resource Allocation:
    - How large a collection?
    - How powerful are the elements?
    - How much memory?
  - Data access, Communication and Synchronization
    - How do the elements cooperate and communicate?
    - How are data transmitted between processors?
    - What are the abstractions and primitives for cooperation?
  - Performance and Scalability
    - How does it all translate into performance?
    - How does it scale?

Why Parallel Architecture?

Role of a computer/system architect:

To design and engineer various levels of a computer system to maximize *performance* and *programmability* within limits of *technology* and *cost*.

Parallelism:

- Provides alternative to faster clock for performance
- Applies at all levels of system design
- Is a fascinating perspective from which to view architecture
- Is increasingly central in information processing

Speedup

- Speedup (p processors) = $\frac{\textit{Performance (p processors)}}{\textit{Performance (1 processor)}}$
- For a fixed problem size (input data set), performance = 1/time
- $\text{Speedup}_{\text{fixed}}$ (p processors) = $\frac{\textit{Time (1 processor)}}{\textit{Time (p processors)}}$

Architectural Trends

- Architecture translates technology's gains into performance and capability
- Resolves the tradeoff between parallelism and locality
  - Change with scale and technology advances
    - Increasing role of memories (caches, …)
- Understanding microprocessor architectural trends

  => Helps build intuition about design issues or parallel machines

  => Shows fundamental role of parallelism even in "sequential" computers

Architectural Trends

- Greatest trend in VLSI generation is increase in parallelism
  - Up to 1985: bit level parallelism: 4-bit -> 8 bit -> 16-bit
    - Slows after 32 bit
    - Adoption of 64-bit now, 128-bit far (not performance issue)
    - Great inflection point when 32-bit micro and cache fit on a chip
  - Mid 80s to mid 90s: instruction level parallelism
    - Pipelining and simple instruction sets, + compiler advances (RISC)
    - On-chip caches and functional units => superscalar execution
    - Greater sophistication: out of order execution, speculation, prediction
      - To deal with control transfer and latency problems
  - Next step: thread level parallelism

Summary: Why Parallel?

- Increasingly attractive
  - Economics, technology, architecture, application demand
- Increasingly central and mainstream
- Parallelism exploited at many levels
  - Instruction-level parallelism
  - Multiprocessor servers
  - Large-scale multiprocessors ("MPPs")
- Focus of this class: multiprocessor level of parallelism
- Same story from memory system perspective
  - Increase bandwidth, reduce average latency with many local memories
- Spectrum of parallel architectures make sense
  - Different cost, performance and scalability

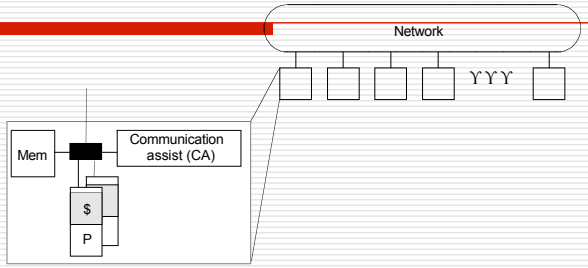
Programming Model

- *Conceptualization of the machine that programmer uses in coding applications*
  - How parts cooperate and coordinate their activities
  - Specifies communication and synchronization operations
- Multiprogramming
  - No communication or synch. at program level
- *Shared address space*
  - Like bulletin board
- *Message passing*
  - Like letters or phone calls, explicit point to point
- *Data parallel*:
  - More regimented, global actions on data
  - Implemented with shared address space or message passing

Toward Architectural Convergence

- Evolution and role of software have blurred boundary
  - Send/recv supported on SAS machines via buffers
  - Can construct global address space on MP (GA -> P | LA)
  - Page-based (or finer-grained) shared virtual memory
- Hardware organization converging too
  - Tighter NI integration even for MP (low-latency, high-bandwidth)
  - Hardware SAS passes messages
- Even clusters of workstations/SMPs are parallel systems
  - Emergence of fast system area networks (SAN)
- Programming models distinct, but organizations converging
  - Nodes connected by general network and communication assists
  - Implementations also converging, at least in high-end machines

Convergence: Generic Parallel Arch.



- Node: processor(s), memory system, *communication assist*
  - Network interface and communication controller
- Scalable network
- Convergence allows lots of innovation, within framework
  - Integration of assist with node, what operations, how efficiently...

Architecture

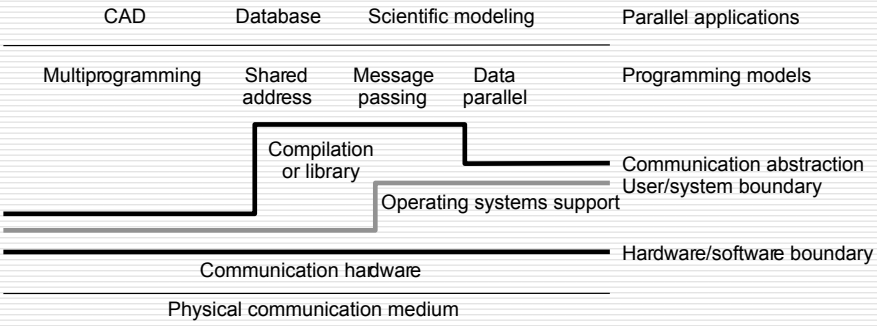
- Two facets of Computer Architecture:
  - Defines Critical Abstractions
    - Especially at HW/SW boundary
    - Set of operations and data types these operate on
  - Organizational structure that realizes these abstraction
- Parallel Computer Arch. =
  Comp. Arch + Communication Arch.
- Comm. Architecture has same two facets
  - Communication abstraction
  - Primitives at user/system and hw/sw boundary

Communication Architecture

User/System Interface + Organization

- User/System Interface:
  - Comm. primitives exposed to user-level by hw and system-level sw
- Implementation:
  - Organizational structures that implement the primitives: HW or OS
  - How optimized are they? How integrated into processing node?
  - Structure of network
- Goals:
  - Performance
  - Broad applicability
  - Programmability
  - Scalability
  - Low Cost

Modern Layered Framework



Communication Abstraction

- User level communication primitives provided
  - Realizes the programming model
  - Mapping exists between language primitives of programming model and these primitives
- Supported directly by hw, or via OS, or via user sw
- Lot of debate about what to support in sw and gap between layers
- Today:
  - Hw/sw interface tends to be flat, i.e. complexity roughly uniform
  - Compilers and software play important roles as bridges today
  - Technology trends exert strong influence
- Result is convergence in organizational structure
  - Relatively simple, general purpose communication primitives

Understanding Parallel Architecture

- Traditional taxonomies not very useful
- Programming models not enough, nor hardware structures
  - Same one can be supported by radically different architectures

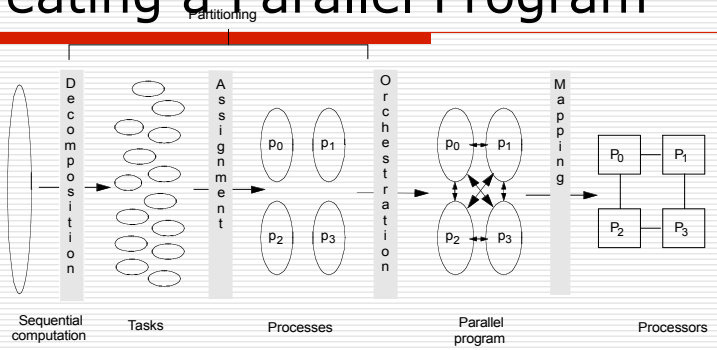
=> Architectural distinctions that affect software

  - Compilers, libraries, programs
- Design of user/system and hardware/software interface
  - Constrained from above by progr. models and below by technology
- Guiding principles provided by layers
  - What primitives are provided at communication abstraction
  - How programming models map to these
  - How they are mapped to hardware

Fundamental Design Issues

- At any layer, interface (contract) aspect and performance aspects
  - *Naming*: How are logically shared data and/or processes referenced?
  - *Operations*: What operations are provided on these data
  - *Ordering*: How are accesses to data ordered and coordinated?
  - *Replication*: How are data replicated to reduce communication?
  - *Communication Cost*: Latency, bandwidth, overhead, occupancy

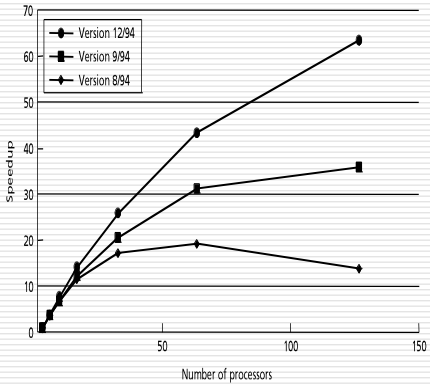
Creating a Parallel Program



- Decomposition of computation in tasks
- Assignment of tasks to processes
- Orchestration of data access, comm, synch.
- Mapping processes to processors

Performance Goal => Speedup

- Architect Goal
  - observe how program uses machine and improve the design to enhance performance
- Programmer Goal
  - observe how the program uses the machine and improve the implementation to enhance performance
- What do you observe?
- Who fixes what?



Recap: Performance Trade-offs

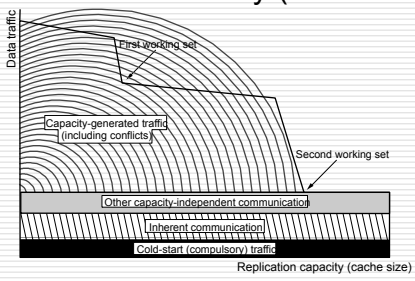
- Programmer's View of Performance

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\textit{Max}\ (\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})}$$

- Different goals often have conflicting demands
  - Load Balance
    - fine-grain tasks, random or dynamic assignment
  - Communication
    - coarse grain tasks, decompose to obtain locality
  - Extra Work
    - coarse grain tasks, simple assignment
  - Communication & synchronization Cost:
    - big transfers: amortize overhead and latency
    - small transfers: reduce contention

Working Set Perspective

- At a given level of the hierarchy (to the next further one)



- Hierarchy of working sets
- At first level cache (fully assoc, one-word block), inherent to algorithm
  - *Working set curve* for program
- Traffic from any type of miss can be local or nonlocal

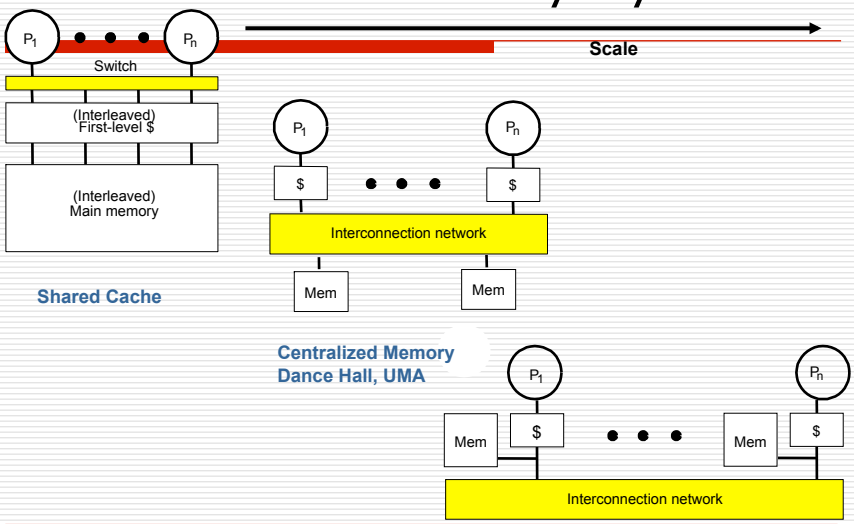
Relation Between Perspectives

Parallelization step(s)	Performance issue	Processor time component
Decomposition/ assignment/ orchestration	Load imbalance and synchronization	Synch wait
Decomposition/ assignment	Extra work	Busy-overhead
Decomposition/ assignment	Inherent communication volume	Data-remote
Orchestration	Artifactual communication and data locality	Data-local
Orchestration/ mapping	Communication structure	

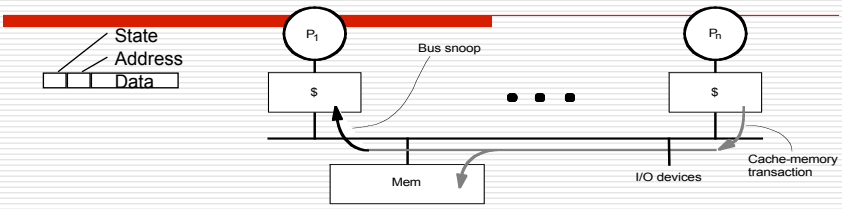
Speedup ≤

$$\frac{Busy(1) + Data(1)}{Busy_{useful}(p)+Data_{local}(p)+Synch(p)+Data_{remote}(p)+Busy_{overhead}(p)}$$

Extensions of Memory System

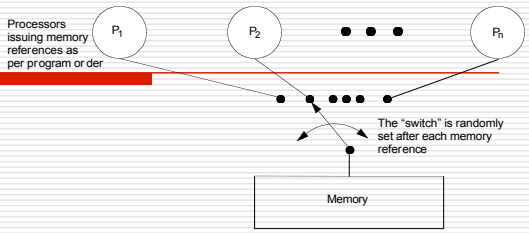


Snooping Cache-Coherence



- Bus is a broadcast medium & Caches know what they have
- Cache Controller "snoops" all transactions on the shared bus
  - Relevant transaction if for a block it contains
  - Take action to ensure coherence
    - Invalidate, update, or supply value
  - Depends on state of the block and the protocol

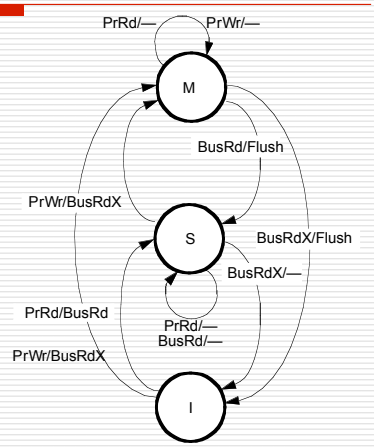
Sequential Consistency



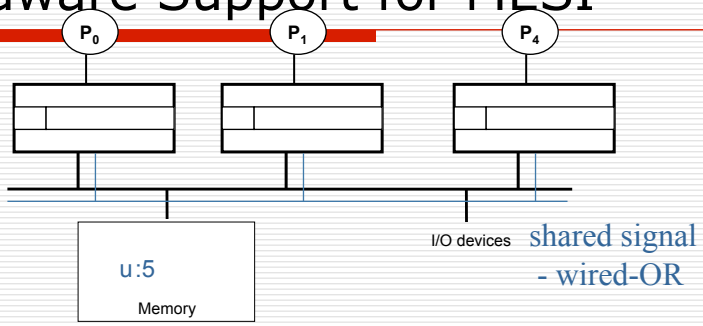
- Total order achieved by *interleaving* accesses from different processes
  - Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others
  - As if there were no caches, and a single memory
- **"A multiprocessor is sequentially 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]**

MSI Invalidate Protocol

- Read obtains block in "shared"
  - even if only cache copy
- Obtain exclusive ownership before writing
  - BusRdx causes others to invalidate (demote)
  - If M in another cache, will flush
  - BusRdx even if hit in S
    - promote to M (upgrade)
- What about replacement?
  - S->I, M->I as before

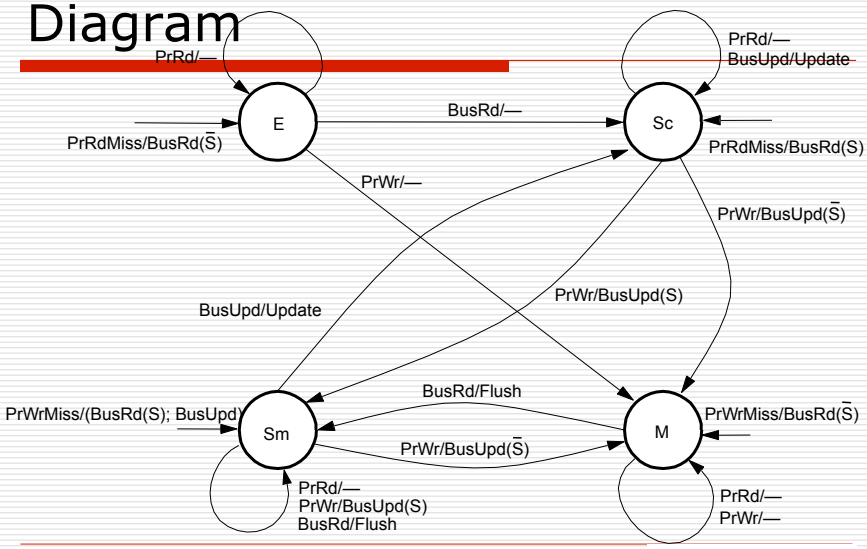


Hardware Support for MESI



- All cache controllers snoop on BusRd
- Assert 'shared' if present (S? E? M?)
- Issuer chooses between S and E
  - how does it know when all have voted?

Dragon State Transition Diagram



Workload-Driven Evaluation

- Evaluating real machines
- Evaluating an architectural idea or trade-offs

=> need good metrics of performance

=> need to pick good workloads

=> need to pay attention to scaling

  - many factors involved

- Today: narrow architectural comparison
- Set in wider context

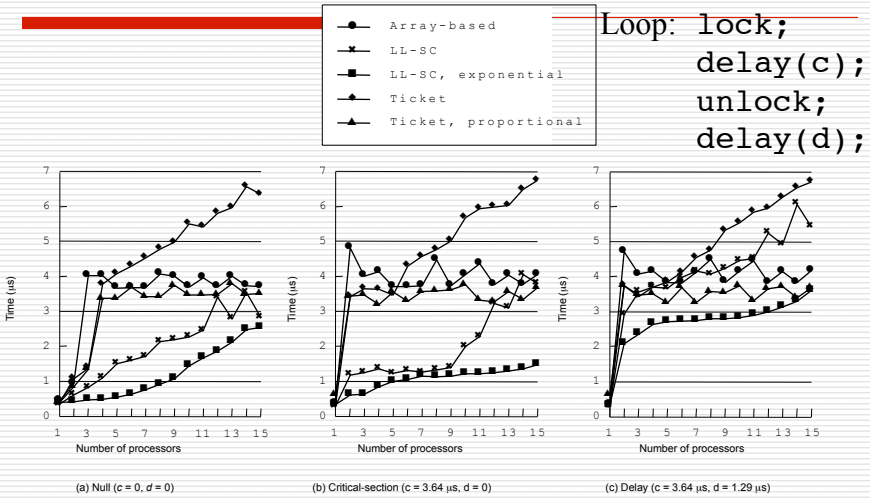
Evaluation Summary

- FSMs describe Cache Coherence Algorithm
  - Many underlying design choices
  - Prove coherence, consistency
- Evaluation must be based on sound understanding of workloads
  - Drive the factors you want to study
  - Representative
  - Scaling factors
- Use of workload driven evaluation to resolve architectural questions

Components of Synchronization Event

- Acquire method
  - Acquire right to the synch
    - Enter critical section, go past event
- Waiting algorithm
  - Wait for synch to become available
  - busy-waiting, blocking, or hybrid
- Release method
  - Enable other processors to acquire right to synch
- Waiting algorithm is independent of type of synchronization
  - Makes no sense to put in hardware

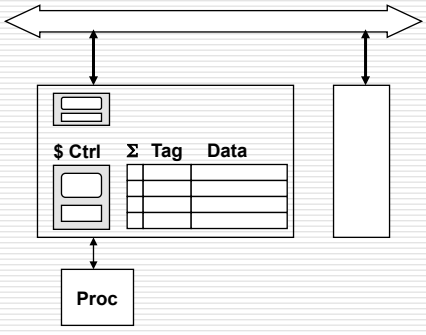
Lock Perf. on SGI Challenge




```
Loop: lock;
      delay(c);
      unlock;
      delay(d);
```


Reality

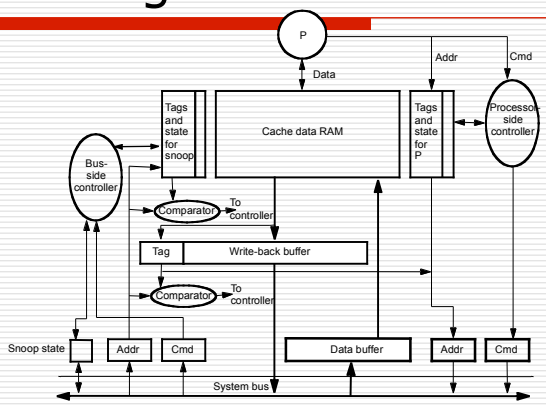
- Protocol defines logical FSM for each block
- Cache controller FSM
  - multiple states per miss
- Bus controller FSM
- Other $Ctrls Get bus
- Multiple Bus trnxs
  - write-back
- Multi-Level Caches
- Split-Transaction Busses



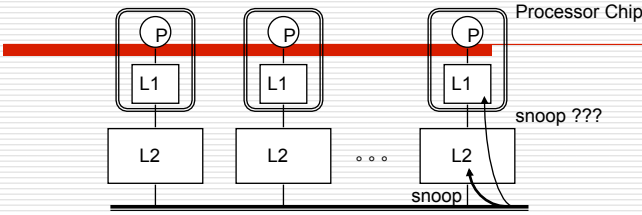
CC Design Issues

- Design of cache controller and tags
  - Both processor and bus need to look up
- How and when to present snoop results on bus
- Dealing with write-backs
- Overall set of actions for memory operation not atomic
  - Can introduce race conditions
- Atomic operations
- New issues deadlock, livelock, starvation, serialization, etc.

Basic design



Multilevel Cache Hierarchies



- Independent snoop hardware for each level?
  - processor pins for shared bus
  - contention for processor cache access ?
- Snoop only at L2 and propagate relevant transactions
- Inclusion property

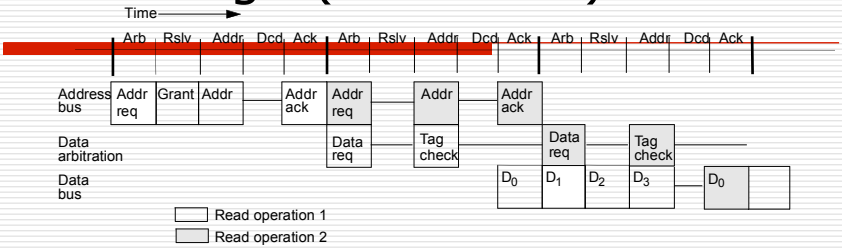
  (1) contents L1 is a subset of L

  (2) any block in modified state in L1 is in modified state in L2

  1 => all transactions relevant to L1 are relevant to L2

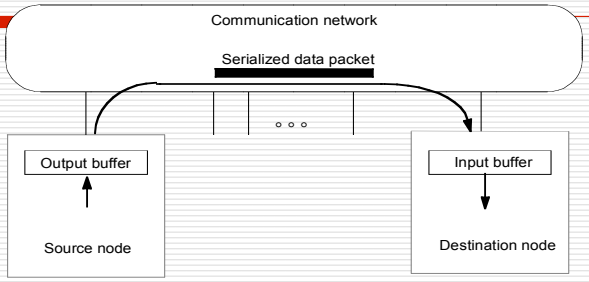
  2 => on BusRd L2 can wave off memory access and inform L1

Bus Design (continued)



- Each of request and response phase is 5 bus cycles
  - Response: 4 cycles for data (128 bytes, 256-bit bus), 1 turnaround
  - Request phase: arbitration, resolution, address, decode, ack
  - Request-response transaction takes 3 or more of these
- Cache tags looked up in decode; extend ack cycle if not possible
  - Determine who will respond (actual response comes later, with re-arbitration)
- Write-backs only request phase : arbitrate both data+addr buses
- Upgrades have only request part; ack'ed by bus on grant (commit)

Network Transaction Primitive



- One-way transfer of information from a source output buffer to a dest. input buffer
  - Causes some action at the destination
  - Occurrence is not directly visible at source
- Deposit data, state change, reply

Scalable Synchronization Ops

- Messages: point-to-point synchronization
- Build all-to-all as trees
- Recall: sophisticated locks reduced contention by spinning on separate locations
  - caching brought them local
  - test&test&set, ticket-lock, array lock
    - O(p) space
- Problem: with array lock location determined by arrival order => not likely to be local
- Solution: queue-lock
  - Build distributed linked-list, each spins on local node