Architecture and design of AlphaServer GS320

Kourosh Gharachorloo, Madhu Sharma, Simon C Steely, Stephen Van Doren

ASPLOS IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems; November 2000 Pages 13–24

Presented in class by:
Ben Reber and Abhishek Tyagi

Key Contributions

- Presents AlphaServer GS320, a cache-coherent NUMA multiprocessor developed at Compaq targeting medium-scale computing
- Rely on ordering properties of network to simplify coherence protocol
- Propose solutions to decrease network occupancy
- Design naturally lends itself to elegant solutions for deadlock, livelock, starvation, and fairness.
- Efficient memory ordering techniques

Review: Snooping vs Directory Protocols

- Snooping based protocols requires a common bus for broadcasting messages and requests
- Works well with small scale machines but does not scale well for larger systems

- Directory based protocols keeps track of what is being shared in one centralized place
- Sends point-to-point requests to processors
- Scales better than snoop

Review: Directory Based Cache Coherence

- Each block has an entry centralized “directory” that maintains the state of the block in different caches
- Directory is colocated with the memory and stores information about all memory lines
- A presence vector stores a bit for every processor, for every memory block – the overhead is a function of the number of memory blocks and processors

Ownership State

Add ownership state to MESI stable states

Similar to the modified state, but allows sharing

This is called "Dirty Sharing"

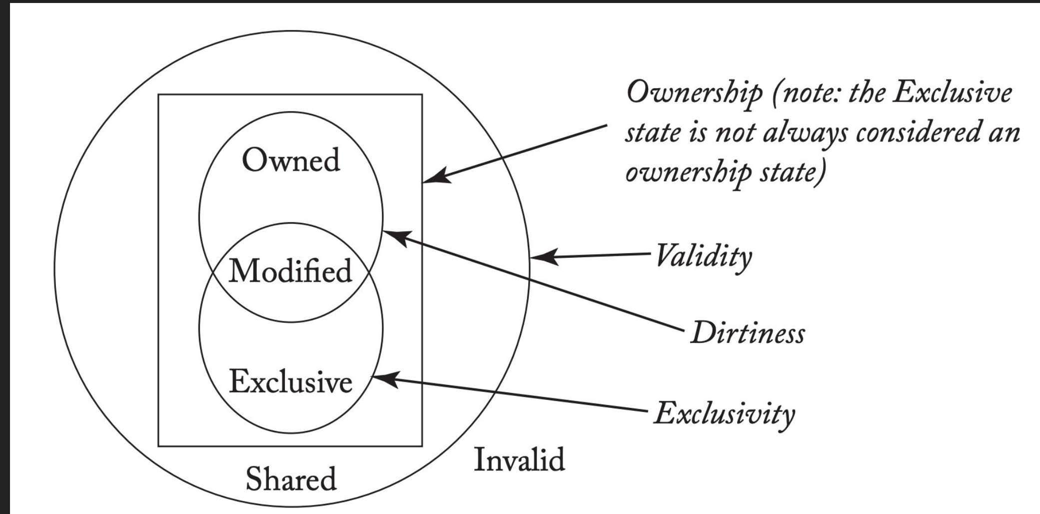


Figure from Nagarajan et al. "A Primer on Memory Consistency and Cache Coherence, 2nd Edition", pp 99.

Terminology

- Home node: the node that stores memory and directory state for the block in question
- Dirty node: the node that has a cache copy in modified state
- Owner node: the node responsible for supplying data (usually either the home or dirty node)

AlphaServer GS320 Architecture

-32 Alpha 21264 processors

-256 GB memory

-Hierarchical structure consisting of up to 8 nodes called QBBs

-QBB - Quad-processor building block

Architecture Overview: Block Diagram

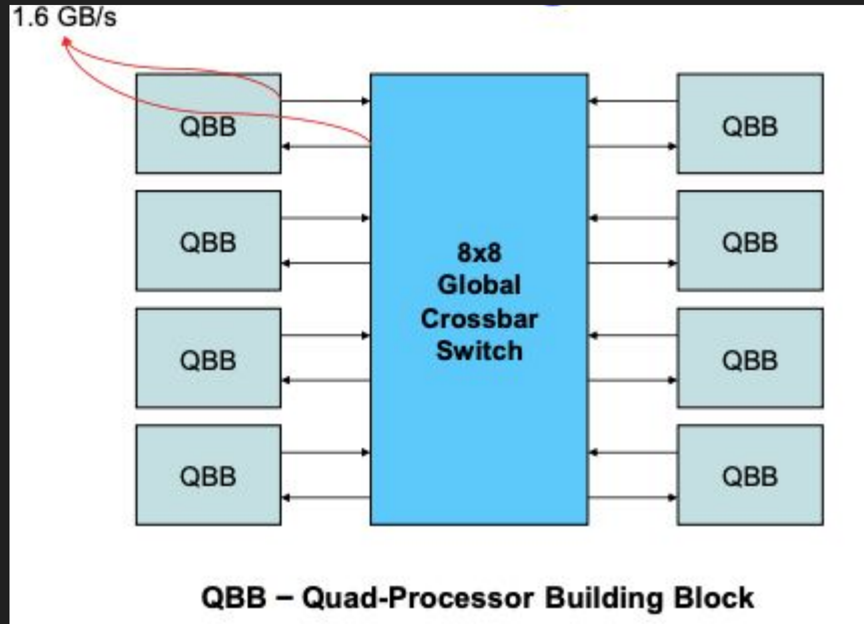


Figure by UR alum and former CS458 student Alok Garg

Quad-Processor Building Block (QBB)

- DTAG- Stores all cache tags of second level caches. Maintains coherence at QBB level
- DIR- Directory that maintains coherence across QBBs
- TTT- is a 48 entry associative storage to track data in pending transactions at a node.

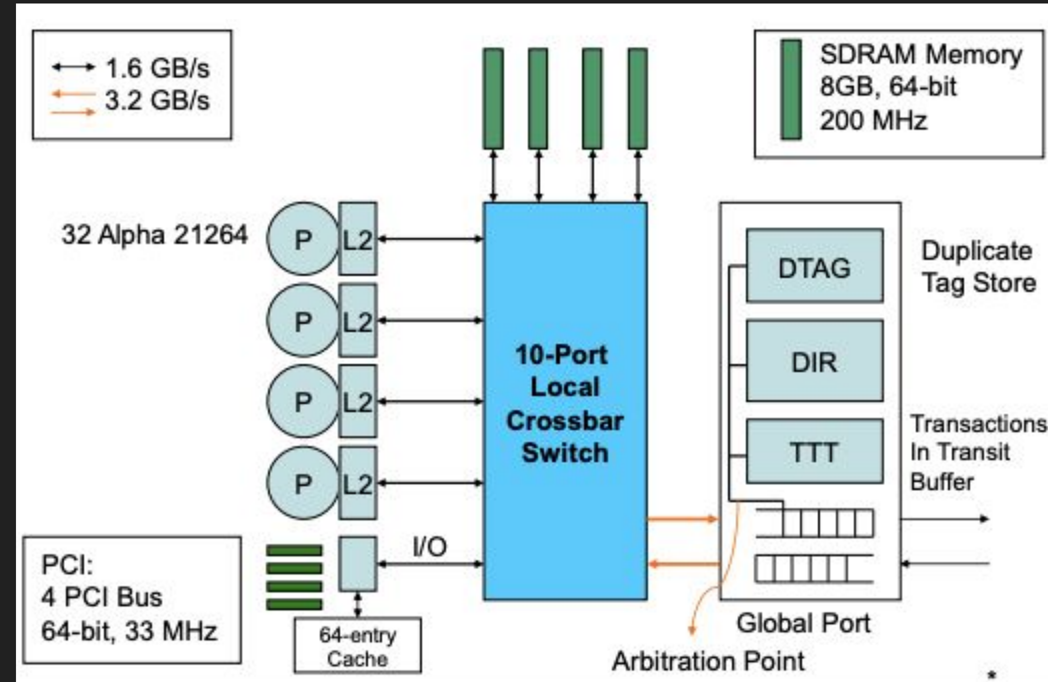


Figure by UR alum and former CS458 student Alok Garg

Directory entry in AlphaServer GS320

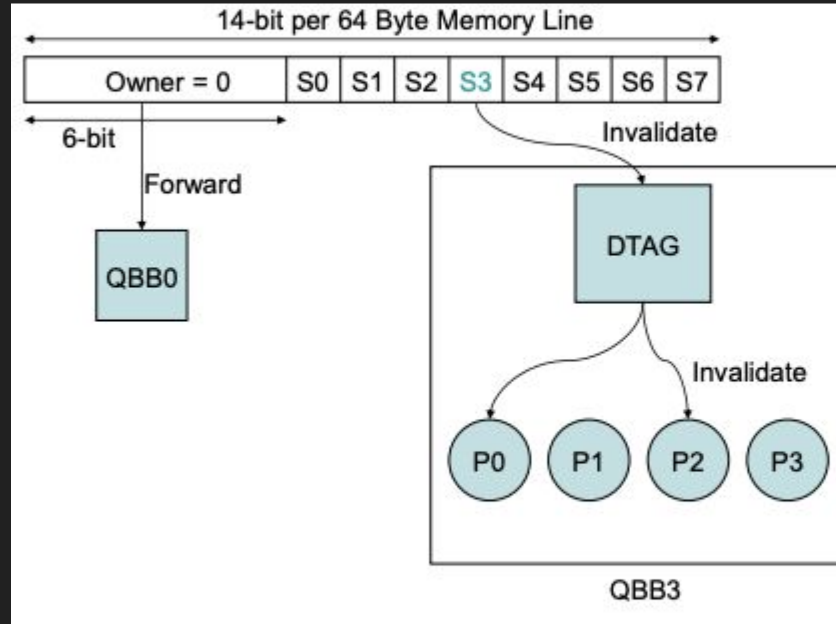


Figure by UR alum and former CS458 student Alok Garg

Crossbar Switch

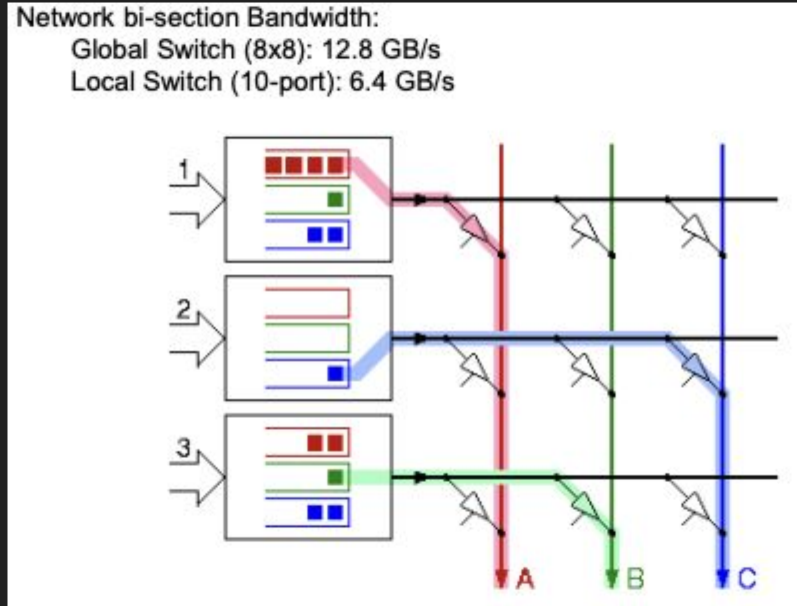


Figure by UR alum and former CS458 student Alok Garg

Benefits of the Hierarchical approach

Scalable Interconnects:

- Fully connected crossbar switch grows quadratically
- Only need to fully connect at QBB granularity
- Local switch within each QBB

Scalable Directory Metadata:

- Tracking sharers at coarse granularity saves bits
- 14 bits / memory line (6 for owner, 8 for QBB sharers)
- Tracking sharers at processor granularity would require 47 bits, ~10% space overhead

Distributes contention among multiple directory modules

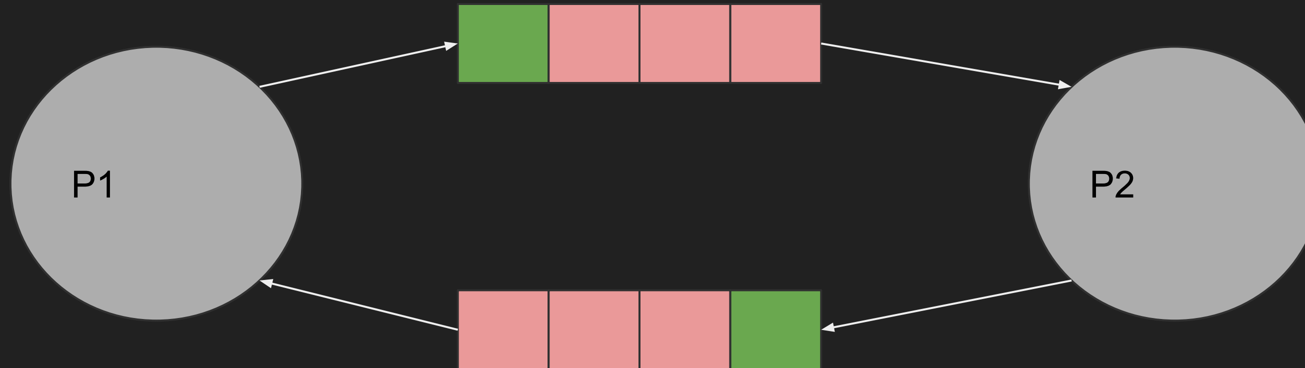
Proposed Coherence Protocol

- Designed with two goals in mind:
  - Reduce inefficiencies caused by slowing down common transactions with solutions to rare edge cases
  - Use ordering properties of the interconnect to reduce number of protocol messages
- Invalidation based protocol supporting *read, read-exclusive, exclusive (requesting processor has a shared copy) and exclusive without data (intent to write the entire cache line, thus avoiding a fetch of the line's current contents)*
- Supports dirty sharing without requiring negative acknowledgement messages (NAKs)
- Transactions require at most one visit to the home node, so directory state can be updated immediately upon message arrival
- DTAG serves as a centralized full-map directory and keeps track of sharing information for the four processors
- All remote accesses are sent to the home node

Removing Negative Acks (NAKs)

- NAKs are primarily used in scalable coherence protocols to
  - Resolve resource dependencies that may result in deadlocks
  - Resolve races where request fails to find data at the node or processor it is forwarded to
- Removing NAKs/retries and blocking has desirable characteristics
  - All directory state changes can occur immediately when home node is first visited as we guarantee that an owner node (or processor) can service the request for sure.
    - Results in fewer transaction messages for 3-hop read and write requests
  - Directory controller is a simple state machine which can be updated immediately avoiding blockages and extra occupancy
  - The early commit optimization, depends on the guarantee that an owner can always service a request
  - Avoid the livelock, starvation and fairness issues

Problem: Protocol Deadlock

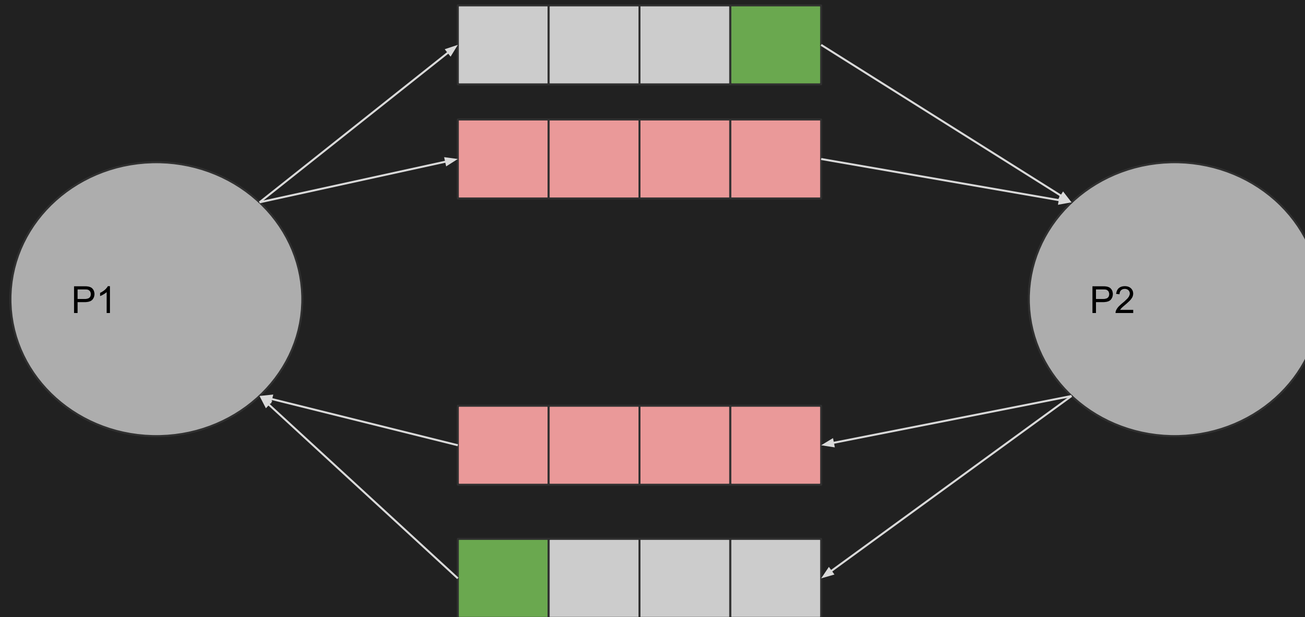


Solution: Virtual Lanes

-Uses three virtual lanes to eliminate the possibility of protocol deadlocks

- Q0: carries request from a processor to home (point-to-point ordering)
- Q1: carries messages from the home directory (requires total ordering)
- Q2: carries replies from third party node to the requester (no ordering required)

Solution: Virtual Lanes



Problem: Request Races

- *Late request race*: request arrives at the owner after it has written the block back to memory
- *Early request race:* request arrives at the owner before it has received it's copy of data

Solution: Late Request Race

-Each cache is required to maintain a copy of evicted cache lines until the directory acknowledges the change

-Total ordering on Q1 guarantees that a data request will not bypass this acknowledgement

Done in two steps:

- When a processor in QBB victimizes a line, it awaits a release signal before discarding data.
- This signal is delayed until all forwarded request in DTAG are satisfied
- Writeback is taken care by TTT and maintains a copy until home acknowledges writeback

Solution: Early Request Race

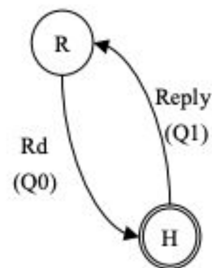
-Cache has already been declared the owner, but has not yet received data

-Delay forwarded request (on Q1) until the data (on Q2) has arrived in cache

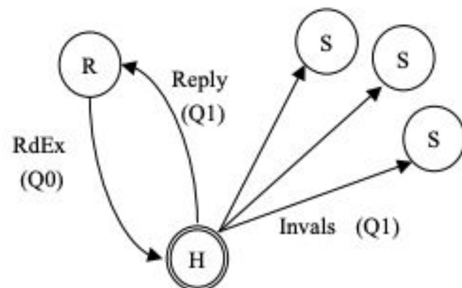
-Compare inbound request against addresses in the processor’s miss-address file

-Block queue locally on match

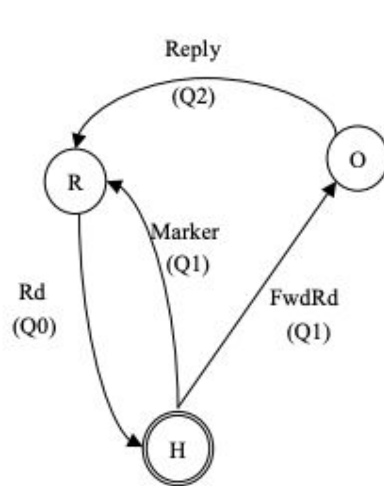
Example transaction flows



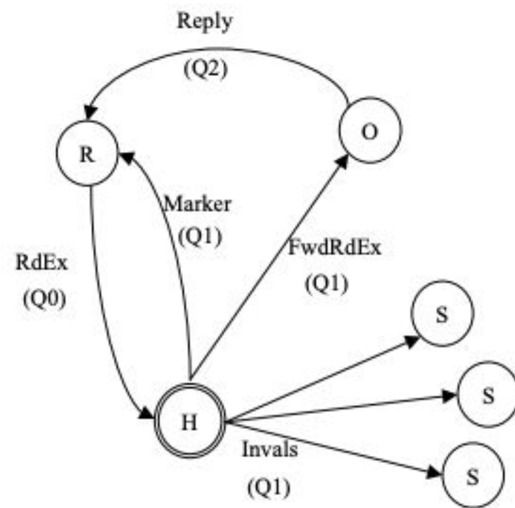
(a) 2-hop Read



(b) 2-hop Read-Exclusive



(c) 3-hop Read



(d) 3-hop Read-Exclusive

Consistency Review

-Recall that the strongest consistency model is sequential consistency, in which:

-All processors execute operations in some sequential order (i.e. program order)

-All operations are visible in this order to all other processors.

-This model disallows many valuable compiler optimizations

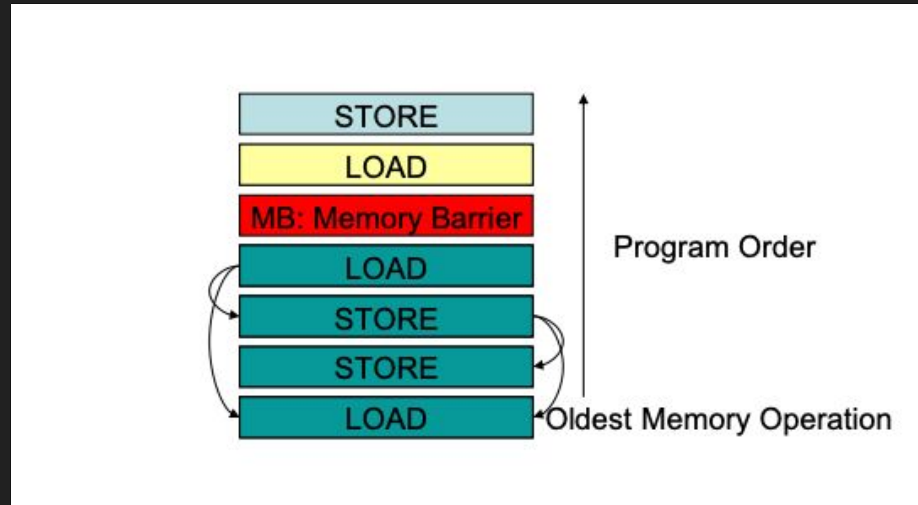
-Relaxing these requirements can result in performance benefits

Alpha Consistency Model

-Allow total read/write reordering

-Allow reading your own write “early”, i.e. before it is visible to all processors

-Memory dependences are enforced using memory barrier operations



Consistency Model Optimizations

-Separate replies into commit (used for ordering purposes) and data/reply

-Commit is placed on Q1 (totally ordered lane)

-Time sensitive data/reply may bypass other traffic by using another lane (Q2)

-Maintain count of total number of pending requests

-On memory barrier, wait until this count reaches zero.

Early Commit Optimization

-Early commit generated for any read or read exclusive request that is forwarded.

-Processor is allowed past a memory barrier if all outstanding requests have received their commit message, regardless of whether they have received data.

-Messages must be carefully ordered for correctness.

Results

Table 3: Effective latency for write operations.

Case	Pipelined Writes	Writes Separated by Memory Barriers
Local Home, No Sharers	58ns	387ns
Local Home, Remote Sharers	66ns	851ns
Remote Home, No Sharers	135ns	1192ns
Remote Home, Remote Sharers	148ns	1257ns

Table 4: Serialization latency for conflicting writes to the same line.

Case	Serialization Latency
1 QBB, 4 procs	138ns
8 QBBs, 1 proc/QBB	564ns