Scalable Cache Coherence

[§8.1] All of the cache-coherent systems we have talked about until now have had a bus.

Not only does the bus guarantee serialization of transactions; it also serves as convenient broadcast mechanism to assure that each transaction is propagated to all other processors’ caches.

Now we want to consider how cache coherence can be provided on a machine with physically distributed memory and no globally snoopable interconnect.

These machines provide a shared address space.

They must be able to satisfy a cache miss transparently from local or remote memory. This replicates data. How can it be kept coherent?

Scalable distributed memory machines consist of P-C-M nodes connected by a network.
The communication assist interprets network transactions, forms interface.

A coherent system must do these things.

- Provide set of states, state-transition diagram, and actions.
- Manage coherence protocol.
  - (0) Determine when to invoke the coherence protocol
  - (a) Find source of info about state of this block in other caches.
  - (b) Find out where the other copies are
  - (c) Communicate with those copies (invalidate/update)

(0) is done the same way on all systems

- The state of the line is maintained in the cache
- The protocol is invoked if an “access fault” occurs on the line.

The different approaches to scalable cache coherence are distinguished by their approach to (a), (b), and (c).

**Bus-based coherence**

In a bus-based coherence scheme, all of (a), (b), and (c) are done through broadcast on bus.

- The faulting processor sends out a “search.”
- Other processors respond to the search probe and take necessary action.

We *could* do this in a scalable network too—broadcast to all processors, and let them respond. Why don’t we?

Why not?
Approach #1: Hierarchical snooping

Extend the snooping approach to a hierarchy of broadcast media.

- The interconnection network is a not a bus, but a tree of buses or rings (e.g., KSR-1).
- The processors are the bus- or ring-based multiprocessors at the leaves of the network.
- Parents and children are connected by two-way snoopy interfaces.

Functions (a) through (c) are performed by a hierarchical extension of the broadcast and snooping mechanism.

- A processor puts a search request on the bus.
- It is propagated up and down the hierarchy as needed, based on snoop results.

Problems:

- 

Hierarchical snooping schemes are much less common than directory based schemes. We won’t consider them further; for details, see §8.10.2.

Approach #2: Directories

The basic idea of a directory-based approach is this.

- Every memory block has associated directory information; it keeps track of copies of cached blocks and their states.
- On a miss, it finds the directory entry, looks it up, and communicates only with the nodes that have copies (if necessary).
In scalable networks, communication with directory and copies is through *network transactions*.

Let us start off by considering a simple directory-based approach due to Tang (1976). It uses—

- A central directory (called a *present table*) of which caches contain which main-memory blocks. Entry $P[i, c]$ tells whether the $i$th block is cached in the $c$th cache.

- A central *modified table* that tells, for each block of main memory, whether the block is up-to-date (or whether some cache holds a more recent copy).

- A local table for each cache that keeps track of whether each line in the cache is also cached somewhere else.

Until a block is modified, it can be cached in any number of places. Once it is modified, only one cached copy is allowed to exist until it is written back to main memory.

Here is a diagram of the three tables:
A cached block can be in one of three states:

- **RO** — read-only. Several cached copies of the block exist, but none has been modified since it was loaded from main memory.
- **EX** — exclusive read-only. This is the only cached copy of the block, but it hasn’t been modified since it was loaded.
- **RW** — read/write. The block has been written since it was loaded from main memory. This is the only cached copy of the block.

When a processor writes a block, it changes its status to RW, and invalidates the other cached copies if its previous status was RO.

Here is a flowchart of the coherence checks on a read reference:
Here is a flowchart of the coherence checks on a write reference:
Operation of a simple directory scheme

The scheme we have just discussed is called a centralized directory scheme, because the directory is kept together in one place.

Let us now assume that the directory is distributed, with each node holding directory information for the blocks it contains.

This node is called the home node for these blocks.
What happens on a read-miss?

The requesting node sends a request transaction over the network to the home node.

The home node responds with the identity of the owner—the node that currently holds a valid copy of the block.

The requesting node then gets the data from the owner, and revises the directory entry accordingly.
On a write-miss, the directory identifies copies of the block, and invalidation or update messages may be sent to the copies.

One major difference from bus-based schemes is

What information will be held in the directory?

- There will be a dirty bit telling if the block is dirty in some cache.
- Not all state information (MESI, etc.) needs to be kept in the cache, only enough to determine what actions to take.

Sometimes the state information in the directory will be out of date. Why?

So, sometimes a directory will send a message to the cache that is no longer correct when it is received.

Let us consider an example system.

- Three stable cache states (MSI).
- Single-level caches.
- One processor per node.
• In the directory entry for each block, a *dirty* bit and presence bits $p[i]$, one for each processor.

► On a read-miss from main memory by processor $i$:

• If $\neg \text{dirty}$, then { read from main memory; turn $p[i]$ on }

• If $\text{dirty}$, then {
  
  recall line from dirty processor (setting cache state to shared);
  update memory;
  turn dirty off;
  turn $p[i]$ on; supply recalled data to processor $i$ }

► On a write-miss to main memory by processor $i$:

• If $\neg \text{dirty}$, then {
  
  supply line to $i$, along with $p$ vector;
  send invalidations to all caches that have the block;
  turn dirty on;
  turn $p[i]$ on, other $p$ bits off in directory entry }

• If $\text{dirty}$, then {
  
  recall line from dirty processor $d$;
  change $d$'s cache state to invalid;
  supply line to $i$;

On the replacement of a dirty block by node $i$, the data is written back to memory and the directory is updated to turn off dirty bit and $p[i]$.

On the replacement of a shared block, the directory may or may not be updated.

How does a directory help? It keeps track of which nodes have copies of a block, eliminating the need for

Would directories be valuable if most data were shared by most of the nodes in the system?
Fortunately, the number of valid copies of data on most writes is small.

**Scaling with number of processors**

[§8.2.2]  Scaling of memory and directory bandwidth provided
- Centralized directory is bandwidth bottleneck, just like centralized memory.
- How to maintain directory information in distributed way?

- Scaling of performance characteristics
  - traffic: # of network transactions each time protocol is invoked
  - latency: # of network transactions in critical path each time

- Scaling of directory storage requirements
  - Number of presence bits needed grows as the number of processors.
    E.g., 32-byte block size and 1024 processors. How many bits in block, vs. # of bits in directory?

How a directory is organized affects all these, performance at a target scale, as well as coherence-management issues.

**Alternatives for organizing directories**

[§8.2.3]  When a miss occurs, how do we find the directory information? There are two main alternatives.

- A flat directory scheme. Directory information is in a fixed place, usually at the home (where the memory is located).
  On a miss, a transaction is sent to the home node.

- A hierarchical directory scheme. Directory information is organized as a tree, with the processing nodes at the leaves.
  Each node keeps track of which, if any, of its (immediate) children have a copy of the block.
When a miss occurs, the directory information is found by traversing up the hierarchy level until the block is found (in the “appropriate state”).

The state indicates, e.g., whether copies of the block exist outside the subtree of this directory.

- **Centralized**
- **Distributed**

How to find source of directory information
- **Flat**
- **Hierarchical**

How to locate copies
- **Memory-based**
- **Cache-based**

How do flat schemes store information about copies?

- **Memory-based schemes** store the information about all cached copies at the home node of the block.

The directory schemes we considered earlier are memory based.

- **Cache-based schemes** distribute information about copies among the copies themselves.
  
  - The home contains a pointer to one cached copy of the block.
  - Each copy contains the identity of the next node that has a copy of the block.

The location of the copies is therefore determined through network transactions.
When do hierarchical schemes outperform flat schemes?

Why might hierarchical schemes be slower than flat schemes?

**Organizing a memory-based directory scheme**

All info about copies is colocated with the block itself at the home.

They work just like centralized scheme, except for being distributed.

**Scaling of performance characteristics**

- Traffic on a write is proportional to number of sharers.

**Scaling of storage overhead?** Assume representation is a full bit-vector.

**Optimizations for full bit-vector schemes**

- Increase __________ size (reduces storage overhead proportionally).
• Use multiprocessor nodes (one bit per multiprocessor node, not per processor)
• still scales as $pm$, but only a problem for very large machines
  — 256 procs, 4 per cluster, 128B line: 6.25% ovhd.

► Reducing “width”: addressing the $p$ term
• observation: most blocks cached by only few nodes
• don’t have a bit per node, but make entry contain a few

  — $p = 1024$, 10-bit pointers can use 100 pointers and still save space.
• sharing patterns indicate a few pointers should suffice (five or so).
• need an overflow strategy when there are more sharers (later).

► Reducing “height”: addressing the $m$ term.
• observation: number of memory blocks >> number of cache lines.
• most blocks will not be cached at any particular time; therefore, most directory entries are useless at any given time
  • organize directory as a cache, rather than having one entry per memory block

*Organizing a cache-based directory scheme.*

In a cache-based scheme, home only holds a pointer to the rest of the directory information.

The copies are lined together via a distributed list that weaves through caches.

Each cache tag has a pointer that points to the next cache with a copy.

  • On a read, a processor adds itself to the head of the list (communication needed).
• On a write, it makes itself the head node on the list, then propagates a chain of invalidations down the list. Each invalidation must be acknowledged.

• On a write-back, the node must delete itself from the list (and therefore communicate with the nodes before and after it).

Disadvantages: All operations require communicating with at least three nodes (node that is being operated on, previous node, and next node).

Write latency is proportional to number of sharers.

Synchronization is needed

Advantages: Directory overhead is small. Linked list records order that accesses were made, making it easier to avoid starvation.

Work of performing invalidations can be distributed among sharers.

IEEE Scalable Coherent Interface has formalized protocols for handling cache-based directory schemes.

Summary

Flat Schemes:
• Issue (a): finding source of directory data
  • go to home, based on address
• Issue (b): finding out where the copies are
  • memory-based: all info is in directory at home
  • cache-based: home has pointer to first element of distributed linked list
• Issue (c): communicating with those copies
  • memory-based: point-to-point messages (perhaps coarser on overflow)
    – can be multicast or overlapped
• cache-based: part of point-to-point linked list traversal to find them
  – serialized

Hierarchical Schemes:
• all three issues through sending messages up and down tree
• no single explicit list of sharers
• only direct communication is between parents and children