Sharing Patterns with Directory Protocols

§8.3.1 How is data shared between processes in a large parallel application? Four patterns of sharing appear.

- **Read-only.** Data is read, but never written, so no invalidations occur. *Examples:* Code, scene data in raytrace applications.

- **Producer-consumer.** One processor writes data and one or more processors read it. Then the producer writes it again, and the consumer(s) read it again, etc.

  The producer may or may not be the same process each time. *Example:* Branch-and-bound, where a process may produce an improved bound.

  The number of invalidations depends on how many consumers there have been.

- **Migratory.** Data moves around, being written and usually read by each processor it moves to. *Example:* Global sum.

  The number of invalidations is

- **Irregular read-write.** Any other pattern. *Example:* Distributed work queue for tasks. Tasks write it every time they add new work to the list, and read it when they are looking for work.

  Usually, number of invalidations is small.

When it is necessary to invalidate a cache block, how many invalidation messages need to be sent? Let’s look at some applications.
In the LU-factorization application, more than 90% of the time when an invalidation is sent, one block is invalidated. This is typical of many applications.

The Ocean application has slightly more blocks to invalidate for each invalidation message sent.

From what you know about Ocean, why do you think it invalidates one, two, or three sharers almost all of the time?
In the Barnes-Hut \((n\text{-body problem})\) application, a given body’s position is usually read by only a few processors.

However, the positions (centers of mass) of the cells are read by many cells, especially the root, which is read by all the cells.

So when one of these values changes, the number of processes affected varies widely.

**Summary of sharing patterns**

Generally, there are only a few sharers at a write, scales slowly with \(p\).

- Code and read-only objects (e.g., scene data in Raytrace)
  - no problems as rarely written
- Migratory objects (e.g., cost-array cells in LocusRoute)
  - even as \# of PEs scale, only 1-2 invalidations
- Mostly-read objects (e.g., root of tree in Barnes)
  - invalidations are large but infrequent, so little impact on performance
- Frequently read/written objects (e.g., task queues)
  - invalidations usually remain small, though frequent
- Synchronization objects
  - low-contention locks result in small invalidations
— high-contention locks need special support (software trees, queueing locks)

This implies that directories very useful in containing traffic.

**Issues for Directory Protocols**

[§8.4] A usable protocol must be—

- Correct; it must preserve coherence and consistency.
- Fast; it cannot perform so many transactions that the system grinds to a halt.

**Performance**

[§8.4.1] Several network transactions are generated each time a miss occurs in a directory-based protocol.

In our example from the last lecture, how many transactions were generated on a read-miss? How many were generated on a write-miss?

It is obvious that these transactions must be fast to avoid imposing a big overhead on the system.

In optimizing protocols, there are two main goals.

- Reducing the number of transactions generated per memory operation.
- Reducing the number of transactions that are on the critical path.

*Forwarding in memory-based protocols*

Consider a read-miss to a dirty block in a flat memory-based protocol.
The home node responds (2) to the requestor with a message containing the identity of the owner node.

The requestor sends a request (3) to the owner, which responds (4a) with the data.

The owner also updates (4b) memory with the data and sets the directory state to shared.

(An “intervention” here is just like a request, but is issued in reaction to a request, and is directed at a cache rather than main memory.)

How many of these transactions are on the critical path?

This number can be reduced by intervention forwarding.

Which step has been eliminated here?

The home forwards the request to the owner, asking it to retrieve the block from its cache.

How many transactions are now in the critical path?

We might also use reply forwarding.

The home forwards the intervention message to the requestor node, passing along the identity of the requestor.

This allows the owner to respond directly to the requestor.

How many transactions are now in the critical path?
**Problems raised by forwarding**

The protocol no longer uses a strict request-response protocol. This can complicate deadlock avoidance.

Also, intervention forwarding keeps track of outstanding intervention requests at the ______ rather than the ______.

Assume that a node is allowed to have up to $k$ requests outstanding. What is the worst case for how many requests the home might have to keep track of?

Does reply forwarding also force the home to keep track of outstanding requests?

Which type of forwarding appears more effective?

**Forwarding in cache-based protocols**

In a cache-based protocol, invalidations are sent from the home to the sharers $S_i$ when a write occurs.

In the strict request-response case, each node sends an ack back to the requestor, along with a pointer to the next sharer.

What is the total number of transactions?

How many are on the critical path?

With intervention forwarding, each invalidated node forward the invalidation to the next sharer.

What is the total number of transactions?

How many are on the critical path?
With reply forwarding, only the last sharer on the list sends a single ack.

What is the total number of transactions?

**Speculation**

When a request arrives at the home, the CA can read the data from memory in parallel with the directory lookup.

When does this help performance?

Another way to help performance would be to detect common sharing patterns and dynamically adapt the protocol.

**SMP nodes**

A high-level design decision in a scalable multiprocessor is to use shared-memory multiprocessors as the nodes. This has advantages and disadvantages.

**Advantages**

- Fixed per-node costs may be amortized among the processors within the nodes. SMP commodity parts may not cost $n\times$uniprocessor parts.
- The caches will be shared among nodes, producing the advantages we discussed in Lecture 16.
- Some applications will be able to run out of local-cluster memory, potentially speeding them up.

**Disadvantages**

- If bus supports snooping coherence,
Correctness

[§8.4.2] Correctness considerations can be divided into three sets.

- Ensure basics of coherence at state-transition level.
  - lines are updated/invalidated/fetched
  - correct state transitions and actions happen
- Ensure that ordering and serialization constraints are met.
  - for coherence (single location)
  - for consistency (multiple locations); we assume sequential consistency.
- Avoid deadlock, livelock, starvation

This poses several problems that didn’t exist on bus-based systems.

- There are now multiple copies and multiple paths through the network (distributed pathways).
- The large latency makes optimizations attractive.
- Increasing concurrency also complicates correctness.

Coherence

Coherence requires that the writes to a location be serialized, that is, seen in the same order by all processors.

This was less of a problem before.

- On a bus-based multiprocessor, there could be multiple copies, but the bus imposed a serialization order.
- On a “scalable” without coherent caches, the main-memory module determined the ordering of writes.
  The order that writes become visible to all processors is the order in which they reached memory.

Can we use the home memory to serialize accesses?

Well, all memory operations come first to the home.
If the home could satisfy all requests for that memory by itself, then our problem would be solved. Can it?

- Processors may see operations to a location in a different order than they reached the home memory.
- If two processors issue read-exclusive requests for a particular word,
  the home will provide the requestors with the location of the dirty node,
  but how do we know that the requests will reach the dirty node in the same order that they reached the home?

Let’s consider four solutions to these problems. All require an additional “busy” state in the directory. What does the busy state indicate?

- **Buffer at the home.** Hold a request at the home node until any previous request for that block has completed.
  (The home node may process requests for *other* blocks in the meantime.)
  This ensures that requests will be serviced everywhere in order of their arrival at the home.
  What are some disadvantages?

- **Buffer at the requestors.** Hold the requests at the requesting nodes, by constructing a linked list of such requests. (Like the cache-based protocol approach.)
  What is the advantage of this over the buffer-at-the-home approach?
• **NACK and retry.** The home “rejects” further requests for a particular block as long as the state of the block is busy. The request must then be retried later. Requests are serialized in the order that

• **Forward to the dirty node.** If the directory state is busy, subsequent requests are simply forwarded to the dirty node.

  The dirty node then determines the order of serialization.

  If the dirty node leaves the dirty state before a forwarded request reaches it, the request will be NACKed and retried.

*Consistency*

Consistency requires that the writes to different shared locations be seen in the same order by all processors.

In order to insure consistency, we must be able to—

• detect write completion, and
• insure write atomicity.

In a bus-based machine, the write can be considered to have completed when it acquires the bus.

Also, writes to a bus are atomic, since one must complete before the next one starts.

In the non-coherent scalable case,

• for write completion, we just need to wait for an explicit ack from memory.

• write atomicity is achieved trivially, due to the fact that there is only a single copy of each word.

Now, with multiple copies and distributed network pathways, how do we achieve consistency?
• For write completion, we need explicit acks

• Further, writes are not easily atomic.

Why isn’t it good enough to assume that a write has completed when the requesting processor receives an ack for it?

• Assume that processor $p$ issues a write to location $x$.
• Processor $p$ receives an ack that this write has made it to memory.
• Processor $p$ now issues a write to location $y$.

Is it possible for some other processor to see the write to location $y$ before the write to location $x$? Why or why not?

This is why we can only assume completion after getting acks from all the copies.

Of course, a processor can ack an invalidation as soon as it gets the message to do so, and before the invalidation is applied to the cache.

Also, it is easy to show how write atomicity can be violated. Consider this example.
Assume that each cache starts out with copies of \( A \) and \( B = 0 \).

We may assume the network preserves point-to-point order.

What value do we expect to be printed by \( P_3 \)?

What can happen to prevent this?

1.
2.
3.
4.

How did this violate atomicity?

With invalidation-based protocols, atomicity can be guaranteed in this manner:

After issuing a write, the current owner of a block waits until all invalidation acks have returned before allowing any processor to read the new value.

With an update-based protocol, it is much more costly to guarantee consistency.