Scalable Cache Coherence Methods and Foundations

Performance

\[ CPI = 1 + m_C T_M \]
\[ T_M = 100 + m_D T_P \]

Weather modeling app

- clocks per instruction
- memory miss handling time
- miss rate of cache: 1%
- Directory miss handling time (SW): 1000
- miss rate of directory: 2%
Some Variants

Directory caches:

Cache a few directory entries

No need to maintain directory entry for uncached blocks, or for private data (compiler/OS can tag private pages)

Chained directories: Simply different data structure for directory

- Link all cache entries
- On read, add new entry to end of list

What to do on write?
Chained directories: What to do on a write?
  • Invalidate all entries on write

Problems?
  But longer latencies
  Also more complex... why?
Chained directories:

- Must handle replacements of elements in chain due to misses!

Invalidate rest

Another idea...

Doubly linked chains

Even more complex
Hierarchical Protocols - E.g. KSR, Kendall Square Research Inc. (actually had rings...)

Multicores will probably initially scale this way as we look to do coherence across two chips

Network (ring, bus, other)

Maintain directory at each level

Hierarchical

First read

A?

A?

A?
Hierarchical cache

Subsequent read

Hierarchical cache

Write
Hierarchical
Subsequent writes

Remember: Meaning of Shared Memory

Sequential consistency: Final state (of memory) is as if all RDs and WRTs were executed in some fixed serial order (per processor order also maintained) → Lamport
Hardware perspective of sequential consistency

Examples
Assume all reads/writes could be to the same address a

Pa sees: r_{a1} r_{b1} r_{a2} w_{b2} w_{b3} w_{c1} w_{a3} w_{c2} r_{c3} r_{c4} ... OK

Pa sees: r_{a1} r_{b1} r_{a2} w_{b2} w_{b3} w_{c1} w_{a3} w_{c2} r_{c3} r_{c4} ... N-OK

Pb sees: r_{a1} r_{b1} r_{a2} w_{b2} w_{b3} w_{c1} w_{a3} w_{c2} r_{c3} r_{c4} ... OK

Pb sees: r_{a1} r_{b1} r_{a2} w_{b2} w_{b3} w_{c1} w_{a3} w_{c2} r_{c3} r_{c4} ... N-OK
Let us investigate a mechanism to optimize cache coherence

Twist 1

- On write to shared location A
  - Request for invalidations sent in background
  - Processor proceeds after write

\[ M_A = 0 \]
\[ Cache \]
\[ A = 1 \]
\[ \uparrow \text{request for invalidations} \]
\[ P \]
\[ A = 1 \]
\[ \text{Proceed} \]

Does Twist 1 caching violate sequential consistency?

Let's use this test scenario (remember this test!)

\[ M_A = 0 \]
\[ M_x = 0 \]
\[ C_A = 0 \]
\[ C_x = 0 \]
\[ P_1 \]
\[ P_2 \]
Does Twist 1 caching violate sequential consistency?

Let’s use this test scenario

A is shared var for producer-consumer comm. P1 → P2
x is a flag signalling A is ready to be read by consumer

If b is 0 at the end, sequential consistency is violated

What’s really going on in the test scenario

P2 should not see new value of x, but old value of A!
Now, let’s simulate execution of a few cycles

If \( b = 0 \) at the end, sequential consistency is violated

Continue simulating a few execution cycles

Assume worst case conditions...

If \( b = 0 \) at the end, sequential consistency is violated
Twist 2

On write to shared location A
Request for invalidations sent
Wait till invalidations are ack'd
Processor proceeds with writing flag x after that

\[ A=0 \]
\[ x=0 \]

\[ A=0 \]
\[ x=0 \]

\[ A=1 \]
\[ x=1 \]

fence

LOOP: If (x==0) GOTO LOOP;

b=A

Does twist 2 caching violate sequential consistency?

Wait till the effect of a write is globally visible before setting flag and proceeding
Twist 2 does not violate the model

Ensure that at time instant $t$, no two processors see different values of a given variable.

On a write (ownership protocol)
- Lock datum in memory
- Invalidate all copies of datum
- Update central copy of datum
- Release lock on datum

Proc does not proceed till write completes (ack got)

How do we implement an update protocol?

Hard!
- Lock central copy of datum
- Phase 1: Mark all copies as unreadable
- Phase 2: Update all copies --- release read lock on each copy after each update
- Unlock central copy
- Concept of “two-phase commit” protocol
Some writes are long-latency ops

Solutions -
1. May not be all that bad if basic latencies are low (not all writes are long)
2. Software protocol can select broadcast invalidate, and combine acks
3. Build latency tolerant processors
4. Change shared-memory semantics [solve a different problem!]
   - Notion of weaker memory semantics - in common use in multicores
     - Basic idea - Guarantee completion of writes only on "fence" operations
     - Typical fence is synchronization point
     - (or programmer puts fences in)
   - Usage model:
     - Modify shared data only within critical sections
     - Propagate changes at end of critical section, before releasing lock
   - Higher level locking protocols must guarantee that others do not try to read/write an object that has been modified and read by someone else.
   - For most parallel programs -- no problem