Directory-Based Cache Coherence Protocols

*Joel Emer*

Computer Science and Artificial Intelligence Lab
M.I.T.

http://www.csg.csail.mit.edu/6.823
Maintaining Cache Coherence

It is sufficient to have hardware such that

- only one processor at a time has write permission for a location
- no processor can load a stale copy of the location after a write

⇒ A correct approach could be:

write request:

The address is invalidated in all other caches before the write is performed

read request:

If a dirty copy is found in some cache, a write-back is performed before the memory is read
Directory-Based Coherence (Censier and Feautrier, 1978)

Snoopy Protocols

- Snoopy schemes broadcast requests over memory bus
- Difficult to scale to large numbers of processors
- Requires additional bandwidth to cache tags for snoop requests

Directory Protocols

- Directory schemes send messages to only those caches that might have the line
- Can scale to large numbers of processors
- Requires extra directory storage to track possible sharers
A System with Multiple Caches

Assumptions: Caches are organized in a hierarchical manner

- Each cache has exactly one parent but can have zero or more children
- Only a parent and its children can communicate directly
- *Inclusion property* is maintained between a parent and its children, i.e.,
  \[ a \in L_i \implies a \in L_{i+1} \]
Directory State Encoding

Each address in a cache keeps two types of state info

- **Sibling info**: do my siblings have a copy of address \( a \)
  - \( \text{Ex} \) (means no), \( \text{Sh} \) (means maybe)
- **Children info**: has address \( a \) been passed on to any of my children
  - \( \text{W:} \{\text{id}\} \) means child id has a writable version
  - \( \text{R:dir} \) means only children named in the *directory* dir have copies
Cache State Invariants

Sh: Cache’s siblings and descendents can only have Sh copies

Ex: Each ancestor of the cache must be in Ex

⇒ either all children can have Sh copies or one child can have an Ex copy

• Once a parent gives an Ex copy to a child, the parent’s data is considered stale
• A processor cannot overwrite L1 data in Sh state
• By definition all addresses in the home memory are in the Ex state
Cache State Transitions

This state diagram is helpful as long as one remembers that each transition involves cooperation of other caches and the main memory.
Guarded Atomic Actions

• Rules specified using guarded atomic actions:
  \[ \text{<guard predicate>} \]
  \[ \rightarrow \{ \text{set of state updates that must occur atomically with respect to other rules}\} \]

• Example
  \[ m.\text{state}(a) \text{ is } R:\text{dir} \land \exists id \text{ s.t. } id \notin \text{dir} \]
  \[ \rightarrow m.\text{setState}(a, R:(\text{dir}+\{id\})); \]
  \[ c_{id}.\text{setState}(a, \text{Sh}); c_{id}.\text{setData}(a, \text{m.data}(a)); \]
Data Propagation Between Caches

Caching rules
- Read caching rule
- Write caching rule

De-caching rules
- Write-back rule
- Invalidate rule
Caching Rules: *Parent to Child*

- **Read caching rule**
  \[ \text{m.state}(a) \text{ is } R: \text{dir} \& \text{id} \notin \text{dir} \]
  \[ \rightarrow \text{m.setState}(a, R: (\text{dir}+\{\text{id}\})) \]
  \[ \quad \text{c}_{id}.\text{setState}(a, \text{Sh}); \text{c}_{id}.\text{setData}(a, \text{m.data}(a)); \]

- **Write caching rule**
  \[ \text{m.state}(a) \text{ is } R: \{\} \text{ (no cache has it)} \]
  \[ \rightarrow \text{m.setState}(a, W: \{\text{id}\}) \]
  \[ \quad \text{c}_{id}.\text{setState}(a, \text{Ex}); \quad \text{c}_{id}.\text{setData}(a, \text{m.data}(a)); \]
De-caching Rules: Child to Parent

- **Writeback rule**
  \[
  \text{m.state(a) is W:\{id\} \& c.state(a) is Ex} \implies \text{m.setState(a, R:\{id\})}
  \]
  \[
  \text{m.setData(a, c.data(a));}
  \]
  \[
  \text{c_{id}.setState(a, Sh);}
  \]

- **Invalidate rule**
  \[
  \text{m.state(a) is R:dir \& id \in dir \& c.state(a) is Sh} \implies \text{m.setState(a, R:(dir-\{id\}))}
  \]
  \[
  \text{c_{id}.invalidate(a);}\]
A simple CC Protocol: 6823s

• Assume only one processor can talk to the memory about an address at a time
  – a global lock (serving) per address. It remembers the processor and the type of request being serviced (id, (ShReq | ExReq))

• We can simultaneously examine the current cache state of one processor and the home directory, and atomically
  – begin a memory activity, and
  – set the directory state, and
  – set the cache state
6823s: State and Functions

Cache states: Sh, Ex, Pending, Nothing
Memory states: R:dir, W:{id}
Memory locks: False, (id, req-type)

Operations on cache:
  c.state(a) – returns state s
  c.data(a) - returns data v
  c.setState(a,s); c.setData(a,v); c.invalidate(a)

Operations on memory:
  m.data(a) - returns data v
  m.setData(a,v);

Operations on Directory:
  m.serving(a) – returns either (id, req-type) or False
  m.setServing(a, id, (ShReq | ExReq)) or m.setServing(False)
  m.state(a) - returns state either R:dir or W:{id}
  m.setState(a,s);
6823s CC protocol

Load Rules (at cache)

• **Load-hit rule**
  
  inst is (Load a) 
  & c_id.state(a) is Sh or Ex 
  \[\rightarrow\] p2m.deq; 
  m2p.enq(c_id.data(a))

• **Load-miss rule**
  
  inst is (Load a) 
  & c_id.state(a) is Nothing 
  & m.serving(a) is False 
  \[\rightarrow\] c_id.setState(a, Pen); m.setServing(a, id, ShReq)

\[\text{inst} = \text{p2m.first()}\]
6823s CC protocol

Store Rules (at cache)

• Store-hit rule
  inst is (Store a v)
  & \( c_{id}.state(a) \) is Ex
  \( \rightarrow \) p2m.deq;
  m2p.enq(Ack);
  \( c_{id}.setData(a, v) \)

• Store-miss rules
  inst is (Store a v)
  & \( c_{id}.state(a) \) is Nothing or Sh
  & m.serving(a) is False
  \( \rightarrow \) c_{id}.setState(a, Pen);
  m.setServing(a, id, ExReq)
Voluntary Rules (at cache)

• Purge rule
  "no space in cache"
  \[\text{\& } \text{id}.\text{state}(a) \text{ is } \text{Sh} \text{ \& } m.\text{state}(a) \text{ must be } \text{R:dir}\]
  \[\rightarrow \text{id}.\text{invalidate}(a); m.\text{setState}(a, \text{R:dir}-\{\text{id}\});\]

• Writeback rule
  \[\text{id}.\text{state}(a) \text{ is } \text{Ex} \text{ \& } m.\text{state}(a) \text{ must be } \text{W:dir}\]
  \[\rightarrow \text{c.setState}(a, \text{Sh}); m.\text{setState}(a, \text{R:dir});\]
  \[\text{m.setData}(a, \text{id}.\text{data}(a))\]
6823s CC protocol
Memory-side ShReq rules

• Serving Loads – Only Sh copies are out
  \( m.\text{serving}(a) \) is \((id, \text{ShReq})\) & \( m.\text{state}(a) \) is \( R:dir \)
  \[ \rightarrow \] \( m.\text{setState}(a, R: (\text{dir}+\{id\})); m.\text{setServing}(a, \text{False}) \)
  \( c_{id}.\text{setState}(a, \text{Sh}); c_{id}.\text{setData}(a, m.\text{data}(a)) \);

• Serving Loads – An Ex copy is out
  \( m.\text{serving}(a) \) is \((id, \text{ShReq})\) & \( m.\text{state}(a) \) is \( W: \{id’\} \)
  \[ \rightarrow \] \( m.\text{setState}(a, R: \{id’\}); m.\text{setData}(a, c_{id’}.\text{data}(a)); \)
  \( c_{id’}.\text{setState}(a, \text{Sh}); \)
6823s CC protocol
Memory-side ExReq rules

- **Serving Stores – No copies out**
  \[ m\text{.serving}(a) \text{ is } (id, \text{ExReq}) \& m\text{.state}(a) \text{ is } R:\{} \]
  \[ \rightarrow m\text{.setState}(a, W:\{}id\{}); m\text{.setServing}(a, \text{False}) \]
  \[ c_{id}\text{.setState}(a, \text{Ex}); c_{id}\text{.setData}(a, m\text{.data}(a)); \]

- **Serving Stores – Only the requesting cache has a copy**
  \[ m\text{.serving}(a) \text{ is } (id, \text{ExReq}) \& m\text{.state}(a) \text{ is } R:\{}id\{} \]
  \[ \rightarrow m\text{.setState}(a, W:\{}id\{}); m\text{.setServing}(a, \text{False}) \]
  \[ c_{id}\text{.setState}(a, \text{Ex}); c_{id}\text{.setData}(a, m\text{.data}(a)); \]

- **Serving Stores – Sh copies are out**
  \[ m\text{.serving}(a) \text{ is } (id, \text{ExReq}) \& m\text{.state}(a) \text{ is } R:\text{dir} \& \exists \text{id'} \text{ s.t. } ((\text{id'} \in \text{dir}) \& (\text{id'} \neq id)) \]
  \[ \rightarrow m\text{.setState}(a, R:\{}\text{dir}-\{}id'\{}\{}); c_{id'}\text{.invalidate}(a); \]

- **Serving Stores – A Ex copy is out**
  \[ m\text{.serving}(a) \text{ is } (id, \text{ExReq}) \& m\text{.state}(a) \text{ is } W:\{}id'\{} \]
  \[ \rightarrow m\text{.setState}(a, R:\{}\{}); m\text{.setData}(a, c_{id'}\text{.data}(a)); \]
  \[ c_{id'}\text{.invalidate}(a); \]
Making 6823s more realistic

- Rules require observing and changing the state of cache and memory simultaneously (atomically).
  - very difficult to implement, especially if caches are separated by a network

Split rules into multiple rules – “request for an action” followed by “an action and an ack”.
  - ultimately all actions are triggered by some processor
The 6823 CC Protocol *an abstract view*

- Each cache has 2 pairs of queues
  - one pair (c2m, m2c) to communicate with the memory
  - one pair (p2m, m2p) to communicate with the processor
- Message format:
  
  $$\text{Msg}(\text{idsrc, iddest, cmd-priority, a, v})$$

- FIFO message passing between each (src,dest) pair except a Low priority (L) msg cannot block a high priority (H) msg
H and L Priority Messages

- At the memory, unprocessed request messages cannot block reply messages. Hence all messages are classified as H or L priority.
  - All messages carrying replies are classified as high priority

- Accomplished by having separate paths for H and L priority
  - In Theory: separate networks
  - In Practice:
    - Separate Queues
    - Shared physical wires for both networks
6823: States and Functions

Cache states: Sh, Ex, Pending, Nothing
Memory states: R:dir, W:{id}, T_R:dir, T_W:{id}
   If dir is empty then R:dir and T_R:dir represent the same state

Messages:
   Cache to Memory requests: (ShReq a); (ExReq a)
   Memory to Cache requests: (WbReq a); (InvReq a); (FlushReq a)

   Cache to Memory replies: (WbRep a v); (InvRep a); (FlushRep a v)
   Memory to Cache replies: (ShRep a v); (ExRep a v)

Operations on cache:
   cache.state(a) – returns state s
   cache.data(a) – returns data v
   cache.setState(a,s); cache.setData(a,v); cache.invalidate(a)

inst = first(p2m); msg= first(m2c); mmsg = first(in)
6823 Protocol Animation
Protocol Diagram

Cache 1
Pen: $a$

ShReq $a$

Dir $a$: Sh $\{\}$

Main Memory

Cache 2

Cache N

...
Protocol Diagram

```

Cache 1

PSH: a

ShRep
<a,v>

Dir a: Sh {1}

Main Memory

Cache 2

Cache N

...  
```
Protocol Diagram

Cache 1
Sh: a

Cache 2
Pen: a

ShReq
a

Dir a: Sh {1,2}

Main Memory

Cache N

L23-26
Protocol Diagram

Cache 1
Sh: a

Cache 2
PSh: a

Cache N

Main Memory

Dir a: Sh {1,2}

ShRep <a,v>
Protocol Diagram

Cache 1
Sh: a

Cache 2
Sh: a

Cache N
Pen: a

InvReq a

ExReq a

Dir a: Sh {1,2}

Main Memory
Protocol Diagram

Cache 1
Sh: a
Inv a
Dir a: Sh {N}_2

Cache 2
Sh: a
Inv a

Cache N
Pen: a
Ex: a
ExRep <a, v>

Main Memory
Protocol Diagram

Cache 1
Pen: a

Cache 2

Cache N
Ex: a

Main Memory

Dir a: Ex \{N\}

ShReq a

WbReq a
Protocol Diagram

Cache 1: \textit{Shen}: a

ShRep: \langle a, v' \rangle

Main Memory

Dir: a: \textit{Sh}: \{N\} \{N\}

Cache 2

Cache N: \textit{Sh}: a

WbRep: \langle a, v' \rangle

Sanchez & Emer
May 5, 2014

http://www.csg.csail.mit.edu/6.823
Thank you