TSO-CC Specification

Marco Elver

marco.elver@ed.ac.uk

December 17, 2015

Contents

1 Introduction

This document provides a detailed specification of the TSO-CC protocol [EN14].

2 Storage Requirements

Table 1 is a summary of storage requirements and introduces parameter names and literals used in the protocol description.

3 Assumptions & Definitions

1. The protocol requires distinguishing valid and invalid timestamps (b.ts). In the following specification $\varnothing$ is used to denote an invalid entry. In our implementation, we use 0 to denote an invalid timestamp, which means the smallest valid timestamp is 1.

Table 1: Coherence specific storage requirements

	Per node	**Per line b**
L1	1. current timestamp, B_{ts} bits 2. write-group counter, $B_{write-group}$ bits 3. current epoch-id, $B_{epoch-id}$ bits 4. timestamp-table ts_L1$[n]$, $n \leq C_{L1}$ entries 5. epoch-ids epoch_ids_L1$[n]$, $n = C_{L1}$ entries 6. timestamp-table ts_L2$[n]$, $n \leq C_{L2-tiles}$ entries (for SharedRO) 7. epoch-ids epoch_ids_L2$[n]$, $n = C_{L2-tiles}$ entries (for SharedRO)	1. number of accesses b.acnt, B_{maxacc} bits 2. last-written timestamp b.ts, B_{ts} bits
L2	1. last-seen timestamp-table ts_L1$[n]$, $n = C_{L1}$ entries 2. epoch-ids epoch_ids_L1$[n]$, $n = C_{L1}$ entries 3. current timestamp, B_{ts} bits (for SharedRO) 4. current epoch-id, $B_{epoch-id}$ bits (for SharedRO) 5. increment-timestamp-flags, 2 bits (for SharedRO)	1. timestamp b.ts, B_{ts} bits 2. owner (Exclusive), last-writer (Shared), coarse vector (SharedRO) as b.owner, $\lceil log(C_{L1}) \rceil$ bits

2. DataS, DataX and Data messages are expected to carry data.
3. A receive message action is of the format: source?Message.
4. A send message action is of the format: destination!Message.
5. A batch transition of all lines in states State1, State2, ... to state NextState is abbreviated tr_all {State1, State2, . . . } NextState.

4 Protocol State Table

The state transition tables can be found in Tables 2 and 3. The following sections provide notes about events in the Tables marked by the respective raised number.

4.1 Private Cache Controller

1. We can't just set the state to Invalid, as the directory might have gotten a read and forwarded the request to us. So we must write back, and wait for Ack to ensure that the line propagated to the L2, and thus no more Fwd requests are outstanding.

2. If $B_{write-group} = 0$, in the presence of non-infinite timestamps, the comparison operator cannot be $<$, as it would violate correctness. This is due to how timestamp resets are dealt with in the L2 (see §5.3).

3. Must reset timestamp, in case the line has since been evicted from L2 and we obtain it in Exclusive. If this is the case, the line may have been modified by another node; now, if we get a FwdX request, the old timestamp must not be forwarded.

4.2 Directory Controller

1. Reuse the block's b.owner bits to maintain a superset of SharedRO sharers: each bit is a pointer to $\lceil \frac{C_{L1}}{\lceil log(C_{L1}) \rceil} \rceil$ sharers.

2. Checking if a line's timestamp in the L2 is **decayed**. In order to allow Shared blocks which have not been written to in a long time to transition to SharedRO, we can use the timestamp $b.ts$ and compare against the owner's entry in the last-seen table: check if a fixed period has passed between the last-seen timestamp and when the line was updated according to $b.ts$.

$$\mathsf{ts\_L1[b.owner]} > 2^{B_{ts}-n} \wedge \mathsf{b.ts} \leq \mathsf{ts\_L1[b.owner]} - 2^{B_{ts}-n}$$

Table 2: TSO Coherence private (L1) cache controller – table of states and events.

	Read	Write	Evict	src?DataS(state, owner, ts)	src?DataX(owner, ts, ackc)	src?FwdS(dst)	src?FwdX(dst)	src?Ack	src?InvRO
Invalid	dir!GetS; b.ts ← ∅; → WaitS;	dir!GetX; update b.ts; → WaitX;							dir!AckRO;

Exclusive	hit;	hit;
		update b.ts;
		→ Modified;
			dir!PutE;
			→ WaitEI;[1]
						dst!DataS(SharedRO, self, b.ts);
						dir!Ack(0);
						→ SharedRO;
							dst!DataX(self, b.ts, 1);
							→ Shared;
									dir!AckRO;

Modified	hit;	hit;
		update b.ts;
			dir!Data(b.ts);
			→ WaitMI;
						dst!DataS(Shared, self, b.ts);
						dir!Data(b.ts);
						→ Shared;
							dst!DataX(self, b.ts, 1);
							→ Shared;
									dir!AckRO;

Shared	**if** b.acnt < maxac **then**
	increment b.acnt;
	hit;
	else
	dir!GetS;
	b.ts ← ∅;[3]
	→ WaitS;
	endif
		dir!GetX;
		update b.ts;
		→ WaitX;
			→ Invalid;						dir!AckRO;

SharedRO	hit;	update b.ts; dir!GetX; → WaitX;	→ Invalid;						dir!AckRO; → Invalid;

WaitS	stall;	stall;	stall;	copy_data; hit;
				reset b.acnt;
				if state = Exclusive **then**
				dir!Ack(0);
				endif
				→ state;
									dir!AckRO;
									→ WaitSROI;

WaitSROI	stall;	stall;	stall;	copy_data; hit;
				reset b.acnt;
				if state = Exclusive **then**
				dir!Ack(0);
				elif state = SharedRO **then**
				→ Invalid;
				endif
				→ state;
									dir!AckRO;

WaitX	stall;	stall;	stall;		copy_data; hit; reset b.acnt; dir!Ack(ackc); → Modified;				dir!AckRO;
WaitEI	stall;	stall;	stall;			dst!DataS(SharedRO, self, b.ts); → Invalid;	dst!DataX(self, b.ts, 0); → Invalid;	→ Invalid;	dir!AckRO;
WaitMI	stall;	stall;	stall;			dst!DataS(Shared, self, b.ts); → Invalid;	dst!DataX(self, b.ts, 0); → Invalid;	→ Invalid;	dir!AckRO;

DataS
↳ @ WaitS, WaitSROI
DataX
↳ @ WaitX

if owner = ∅ ∧ ts ≠ ∅ **then**
  if ts_L2[src] < ts **then**
    ts_L2[src] ← ts;
  tr_all {Shared} Invalid;
  endif
...

...
elif owner ≠ self ∧ (ts = ∅ ∨ ts_L1[owner] ≤ ts) **then**[2]
  if ts ≠ ∅ **then**
    ts_L1[owner] ← ts;
  endif
  tr_all {Shared} Invalid;
endif

Table 3: Directory (L2) controller – table of states and events.

	p?GetS	p?GetX	p?Data(ts)	p?Ack(c)	p?PutE	p?AckRO
Invalid	p!DataS(Exclusive, ∅, ∅); b.owner ← p; b.ts ← ∅; → WaitE1;	p!DataX(∅, ∅, 0); b.owner ← p; b.ts ← ∅; → WaitE1;				
Uncached	p!DataS(Exclusive, b.owner, b.ts); b.owner ← p; b.ts ← ∅; → WaitE1;	p!DataX(b.owner, b.ts, 0); b.owner ← p; b.ts ← ∅; → WaitE1;				
Exclusive	b.owner!FwdS(p); tbe.sharers ← {p}; → WaitS;	b.owner!FwdX(p); b.owner ← p; b.ts ← ∅; → WaitE2;	copy_data; p!Ack; b.ts ← ts; → Uncached;		p!Ack; → Uncached;	

Shared	**if** expired b.ts ∨ decayed b.ts **then**[2]
	  b.owner ← {p};
	  update b.ts;
	  p!DataS(SharedRO, ∅, b.ts);
	  → SharedRO;
	else
	  p!DataS(Shared, b.owner, b.ts);
	endif
		p!DataX(b.owner, b.ts, 0);
		b.owner ← p;
		b.ts ← ∅;
		→ WaitE1;				

SharedRO	p!DataS(SharedRO, ∅, b.ts); b.owner ← b.owner ∪ {p};[1]	dst ← $\{q \mid q \in$ b.owner $\wedge\ q \neq$ p$\}$; dst!InvRO; tbe.need_acks ← |dst|; b.owner ← p; → WaitEn;				

WaitE1	stall;	stall;	**if** p ≠ b.owner **then**
			  → Exclusive;
			else
			  copy_data;
			  p!Ack;
			  b.ts ← ts;
			  → WaitU1;
			endif
				→ Exclusive;	**if** p ≠ b.owner **then**
					  → Exclusive;
					else
					  p!Ack;
					  → WaitU1;
					endif	

WaitE2	stall;	stall;	**if** p ≠ b.owner **then**
			  → WaitE1;
			else
			  copy_data;
			  p!Ack;
			  b.ts ← ts;
			  → WaitU2;
			endif
				if c = 1 **then**
				  → Exclusive;
				else
				  → WaitE1;
				endif
					if p ≠ b.owner **then**
					  → WaitE1;
					else
					  p!Ack;
					  → WaitU2;
					endif	

WaitU1	stall;	stall;	→ Uncached;	→ Uncached;	→ Uncached;	

WaitU2	stall;	stall;	→ WaitU1;	**if** c = 1 **then**
				  → Uncached;
				else
				  → WaitU1;
				endif
					→ WaitU1;	

WaitEn	stall;	stall;				tbe.need_acks − −;
						if tbe.need_acks = 0 **then**
						  b.owner!DataX(∅, b.ts, 0);
						  b.ts ← ∅;
						  → WaitE1;
						endif

WaitS	stall;	stall;	copy_data; b.ts ← ts; → Shared;	b.owner ← tbe.sharers ∪ {p}; update b.ts; → SharedRO;	b.owner ← tbe.sharers; update b.ts; → SharedRO;	

5 Additional Rules & Optimizations

The following is a list of additional rules and optimizations, which have an impact on both L1 and L2 controllers; this completes the full protocol description.

5.1 Cache inclusivity & evictions

Evictions from the L2 are omitted from the transition table; the following must hold: upon eviction of lines from the L2, inclusivity must be maintained for lines which are tracked by the L2 (Exclusive and SharedRO).

5.2 Timestamp table size relaxations

The L1's timestamp-tables ts_L1 and ts_L2 do not need to be able to hold as many entries as there are respective nodes. Applying an eviction policy to evict entries from the timestamp-tables allows to have a reduced-size timestamp-table.

5.3 Effect of L1 timestamp update

To **update** a timestamp in the L1 means assigning the locally maintained timestamp to the line, and also increment this timestamp based on either of the following policies:

1. Always (write-group = 1).
2. Write-groups: If constant number of writes falling under the same timestamp reached.

Timestamp overflows in the L1 are dealt with sending out a TimestampReset broadcast to L1s and L2 tiles:

1. Each L1 invalidates ts_L1[src] on receiving a TimestampReset.
2. Every L2 tile must also maintain a table ts_L1 of last-seen timestamps; ts_L1[src] is updated on every b.ts $\leftarrow$ ts, if ts is newer than the existing last-seen timestamp entry from an L1; on receiving a TimestampReset the respective entry is invalidated. The table of last-seen timestamps must be able to hold, unlike the L1's timestamp-tables, the full list of timestamps of every possible L1.
3. The L2 will assign a response message b.ts if the *last-seen timestamp from the owner is larger or equal to the line's timestamp* (not **expired**), *the smallest valid timestamp* ($\varnothing$ is valid, but degrades performance) otherwise. Similarly for **all** L1 data messages by comparing against L1's own timestamp.

5.4 Effect of L2 timestamp update

To **update** a timestamp in the L2 (for SharedRO) means assigning the L2-local timestamp to the line and incrementing the timestamp under the following conditions:

- *from*-Invalid, check against in WaitS to SharedRO transition: after a L2 eviction of a dirty line; after a GetS event in Uncached where b.ts $\neq \varnothing$ before resetting b.ts.
- *from*-Shared, check against in Shared to SharedRO transition: after a block transitions to Shared.

Maintain a bit for each from-condition to signify if the timestamp should be incremented on the next update or not, resetting *all* bits after the increment was performed.

In essence, the L2's timestamp should always be increment after a transition which can lead to a block ending up in the SharedRO state, but need not actually be incremented until the first block transitions to SharedRO.

It is possible to use only one bit for all conditions, but this would cause unnecessary timestamp increments when a cache line transitions to SharedRO based on the not-modified rule, as transitions to Shared are quite common, but Shared to SharedRO may not be, therefore it makes sense to maintain extra bits for each observed transition that may lead back to a SharedRO state, but only check the condition at the appropriate nearest transition to SharedRO.

To abstract the condition when to increment a L2-timestamp further (define $\rightarrow$ as the happens-before relation): if a set of writes $\mathbb{W} \rightarrow$ set of transitions $\mathbb{T}$ that can cause likely transitions $\mathbb{R}$ to SharedRO, but we can not keep track of which blocks are affected, the system should remember that $\mathbb{T}$ happened so that upon the first transition in $\mathbb{R}$ we can allow L1s to deduce $\mathbb{W} \rightarrow^{+} \mathbb{R}$. For two timestamps t and t', if $t < t'$ then $\mathbb{W} \rightarrow \mathbb{T} \rightarrow \mathbb{R} \rightarrow \mathbb{W}' \rightarrow \mathbb{T}' \rightarrow \mathbb{R}'$; in order to make visible all writes from $\mathbb{W}'$ the L1 needs to self-invalidate on $\mathbb{R}'$, if the largest timestamp value from the L2 it has seen is only t.

Dealing with timestamp overflows:

1. Timestamp overflows in the L2 are dealt with sending out a TimestampReset broadcast and each L1 resetting ts_L2[src]. To not send invalid timestamps, like in §5.3, the L2 will assign a response message b.ts if the *current L2-local timestamp is larger or equal to the line's timestamp*, *the smallest valid timestamp* ($\varnothing$ is valid, but degrades performance) otherwise; in case the smallest valid timestamp is used, the next timestamp assigned to a line after an overflow must always be larger than the smallest valid timestamp.
2. In a NUCA architecture, it will be necessary to either propagate all increments to the L2-local timestamp across all tiles, or each L2 tile maintains its own timestamp, and the L1s maintain a ts_L2 entry per tile or cluster of tiles in a separate table.

5.5 TimestampReset races

To resolve TimestampReset races, without requiring the sender of a TimestampReset to wait for acks, if we can assume a bounded time on message propagation delay:

1. Every node in the system (L1s and L2 tiles) maintains an epoch_id, which is set to a value different than the previous value on sending a TimestampReset; the TimestampReset message contains the new epoch_id. The number of bits required per epoch_id must be large enough to eliminate the probability of having more than one TimestampReset message with the same epoch_id in-flight, but small enough to satisfy storage requirements.
2. The L1s maintain a table of epoch_ids with entries for every L1 and L2 tiles in the system.
3. The L2 tiles each maintain a table of epoch_ids with entries for every L1.
4. On receiving a TimestampReset message, the sender's entry in the respective timestamp-table is invalidated but the epoch_ids entry for the sender is updated with the epoch_id that was received along with the TimestampReset message.
5. An epoch_id is sent with every Data, DataS and DataX message:
   - If the message originates from an L1, it is the L1s own epoch_id.
   - If the message originates from the L2, and owner $\neq \varnothing$, the entry in epoch_ids_L1[b.owner] is assigned.
   - If the message originates from the L2, and owner $= \varnothing \wedge$ ts $\neq \varnothing$, the L2's epoch_id is assigned.

6. The L2 updates epoch_ids_L1[p] along with every b.ts $\leftarrow ts$. If epoch_ids_L1[p] $\neq$ epoch_id, the last-seen entry in ts_L1 must be updated (timestamp reset).

7. On receiving a DataS or DataX message, before the check for potential acquires, the L1 must perform the following check:

```
if ts ≠ ∅ ∧ owner ≠ self then
  if owner ≠ ∅ then
    if epoch_ids_L1[owner] ≠ msg_epoch_id then
      invalidate ts_L1[owner];
      epoch_ids_L1[owner] ← msg_epoch_id;
    endif
  elif epoch_ids_L2[src] ≠ msg_epoch_id then
    invalidate ts_L2[src];
    epoch_ids_L2[src] ← msg_epoch_id;
  endif
endif
```

   If a self-invalidation is possible due to seeing a newer value than in the timestamp-tables *ts_L1* or *ts_L2* respectively, but not having done this check yet, check if the currently held epoch-id for the line's source is valid or not, if not, invalidate the entry in the timestamp-table, essentially performing the same action if a TimestampReset is received.

5.6 Out-of-Order Pipeline Interaction

The description thus far is compatible with a core with a FIFO write-buffer, but without load speculation (in-order). Introducing load speculation, e.g. via a load-queue (LQ), will require forwarding invalidations accordingly.

1. Similarly to other conventional eager protocols, any invalidation (self-invalidation, forced miss, or otherwise) must be forwarded to a LQ.

2. Upon self-invalidation, for all lines in WaitS state, an invalidation must also be forwarded to the LQ. To avoid a retry still hitting in a stale cache line, we propose either:
   a) Adding an additional bit of information to a request from a LQ to denote a retry. In case of a Read+retry, the protocol forces a miss in the Shared state only. This option offers potentially higher performance, as the LQ has more information about which instructions are potentially violated or not, and could still hit (more than once) in a stale cache line if no violation is detected.
   b) Sinking the self-invalidation and transitioning the line to Invalid if the response is Shared data. A retry will miss and fetch the correct data. This option is more conservative, as unnecessary invalidations take place; unlike LQ, the coherence protocol has no information about the order of in-flight instructions.

   The above is required to deal with the following case (and its variants): consider the example in Figure 1. Assume the LQ issues the Read request for 2b first (transition to WaitS), the L2 cache responds with the initial data but the response message remains in transit. Next, 1a and 1b are performed (and committed), and then 2a is issued and receives the value produced by 1b. The acquire at 2a causes self-invalidation. However, the response for 2b arrives with stale data, causing a TSO violation. The above options prevent this case from manifesting.

init: x = 0, y = 0	
Thread 1	**Thread 2**
1a. x ← 1	2a. r1 ← y
1b. y ← 1	2b. r2 ← x

Figure 1: Message passing pattern.

6 Changelog

Dec. 17, 2015: Add section §5.6 on out-of-order pipeline interaction. We would like to thank the authors of CCICheck [Man+15], who informed us that interaction with OOO was not clear; they further suggested that a corner case with speculative loads (§5.6, case 2.) may exist if not dealt with. We verified this using McVerSi [EN16] in Gem5: using the regular network model used, the bug does not manifest as the maximum bound on messages in transit prevents this; however, the problem manifests (if not dealt with as described) when adding large randomized interconnect delays.

References

[EN14] M. Elver and V. Nagarajan. "TSO-CC: Consistency directed cache coherence for TSO". In: *HPCA*. (Orlando, FL, USA). Feb. 2014. URL: http://ac.marcoelver.com/res/research/tsocc/hpca14_tso-cc.pdf.

[EN16] M. Elver and V. Nagarajan. "McVerSi: A Test Generation Framework for Fast Memory Consistency Verification in Simulation". In: *HPCA*. (Barcelona, Spain). Mar. 2016. URL: http://ac.marcoelver.com/res/hpca2016-mcversi.pdf.

[Man+15] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi. "CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface". In: *MICRO*. 2015. DOI: 10.1145/2830772.2830782.