TSO-CC: Consistency directed coherence for TSO

Marco Elver Vijay Nagarajan

University of Edinburgh

HPCA, February 2014

Cache Coherence Problem

Traditional approaches do not scale

- Snooping based protocols: *limited by broadcasts*.
- Directory based protocols: *sharing vector increases linearly*.

⇓

Many existing solutions attack *directory & cache organization*.

⇓

We can do even better, if we *consider consistency model and protocol*.

Consistency & Coherence

Consistency: contract between programmer and hardware

⇒ Specifies when memory operations become visible to other processors.

Cache Coherence: propagate memory operations

⇒ Make memory operations visible according to ordering rules of target *consistency model*.

Eager Coherence for SC

- SC enforces $w \rightarrow r$.
- Write must be globally visible before following reads.
- Enforce coherence *eagerly*, by requiring invalidation of other copies before write!

Typically requires sharing vector!

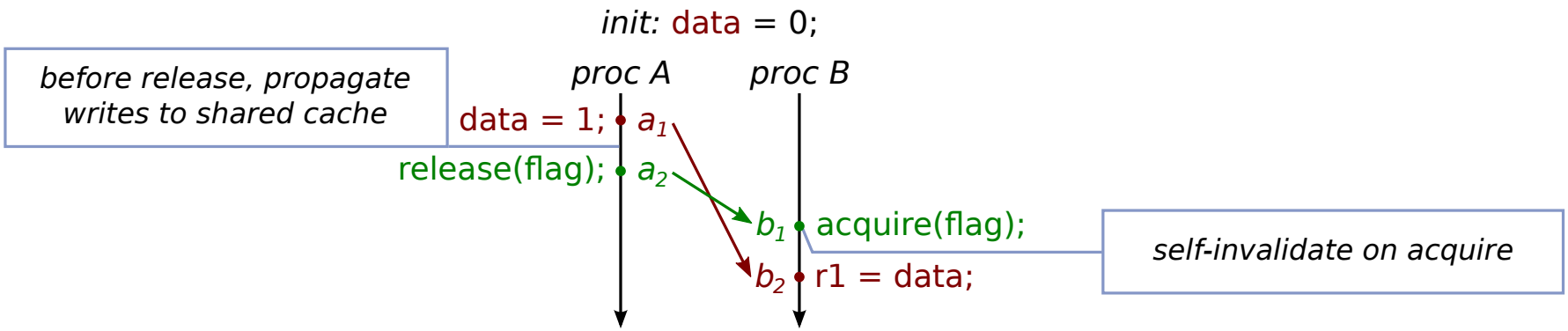
Lazy Coherence for RC

If consistency model is relaxed, why should coherence propagate writes eagerly?

This has been explored for RC:

- Earliest works (Lazy RC): [Keleher et al., 1994] [Kontothanassis et al., 1995]
- Recent works: [Choi et al., 2011] [Ros et al., 2012] [Sung et al., 2013]

Lazy Coherence for RC



In relaxed consistency models (e.g. RC), a write need not immediately become visible.

⇓

No sharing vector!

Research question

Coherence protocols for SC (eager) and RC (lazy) exist, *but* none for relaxed models in-between.

Can we implement any relaxed consistency model with a lazy coherence protocol (with same benefits)?

Consistency-directed coherence for TSO

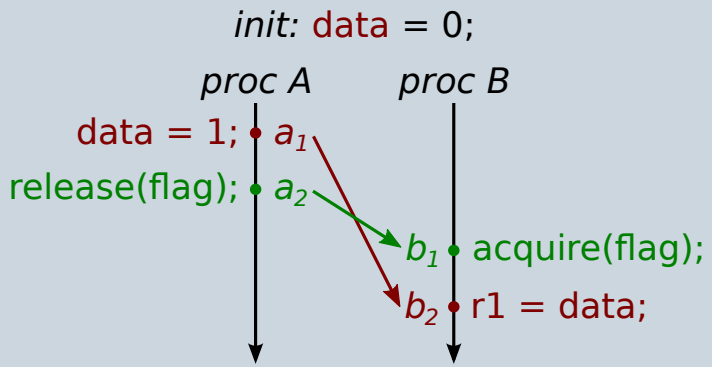
- TSO: Prevalent in x86 and SPARC architectures.
- TSO relaxes $w \rightarrow r$ ordering.
- RC lazy coherence approaches do not work for TSO.

Challenge

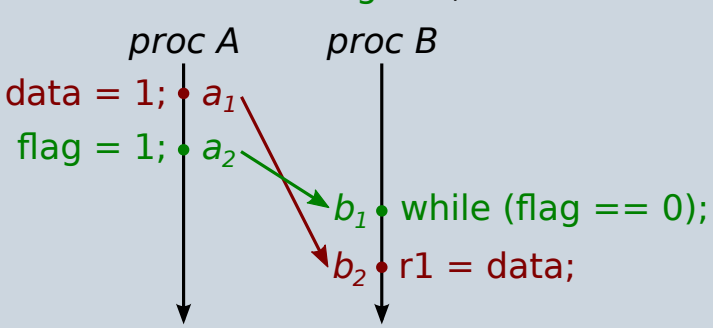
Absence of explicit synchronization in TSO.

RC vs. TSO

RC

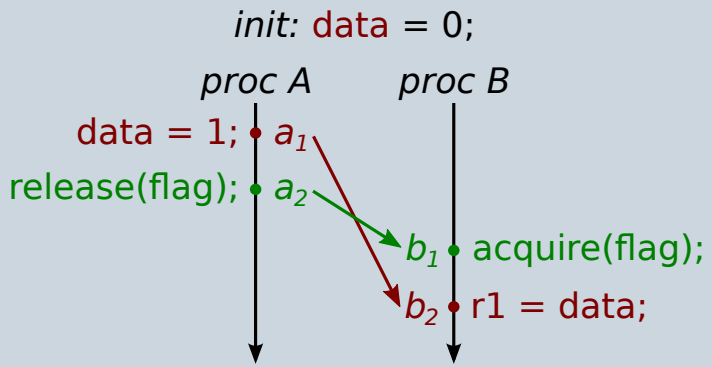


TSO

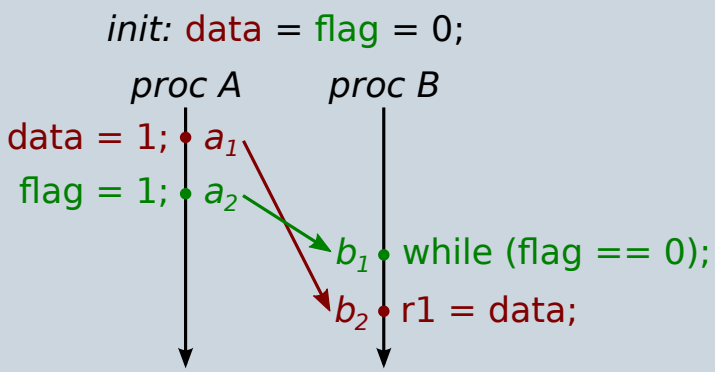


RC vs. TSO

RC



TSO



Requirements

- Ensure write-propagation.
- Ensure $r \rightarrow r$ ordering.
- Ensure $w \rightarrow w$ ordering.
- (Ensure $r \rightarrow w$ ordering.)

TSO-CC basic protocol (preliminaries)

Coherence state

- Shared L2 directory maintains pointer to last-writer/owner.
- States: `Invalid`, `Exclusive`, `Modified`, `Shared`, `Uncached`

TSO-CC basic protocol

`Shared` reads hit in L1s, but miss after pre-determined maximum accesses.

$\Rightarrow$ Ensures write-propagation.

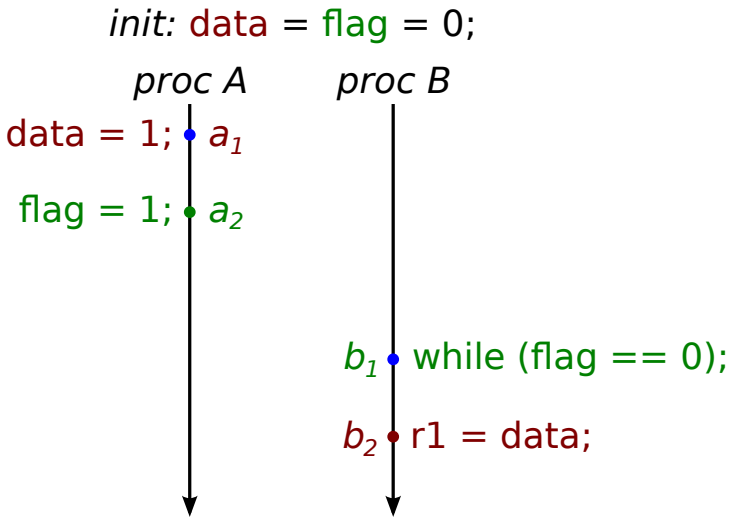
Upon a miss in L1, must self-invalidate all `Shared` lines.

$\Rightarrow$ Ensures $r \rightarrow r$ ordering.

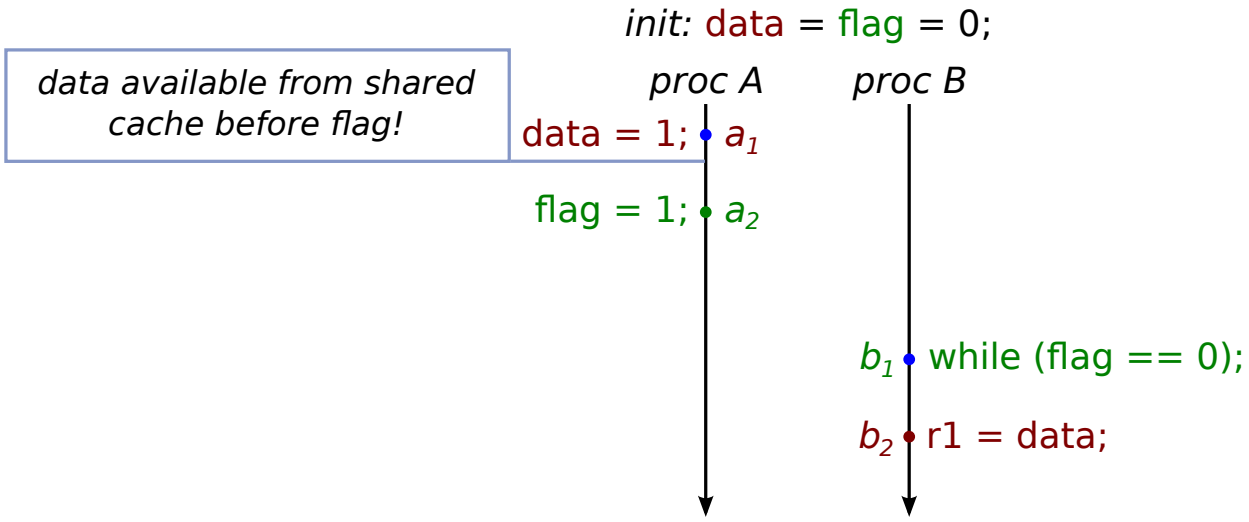
Writes in L1 are propagated to shared cache in program order.

$\Rightarrow$ Ensures $w \rightarrow w$ ordering.

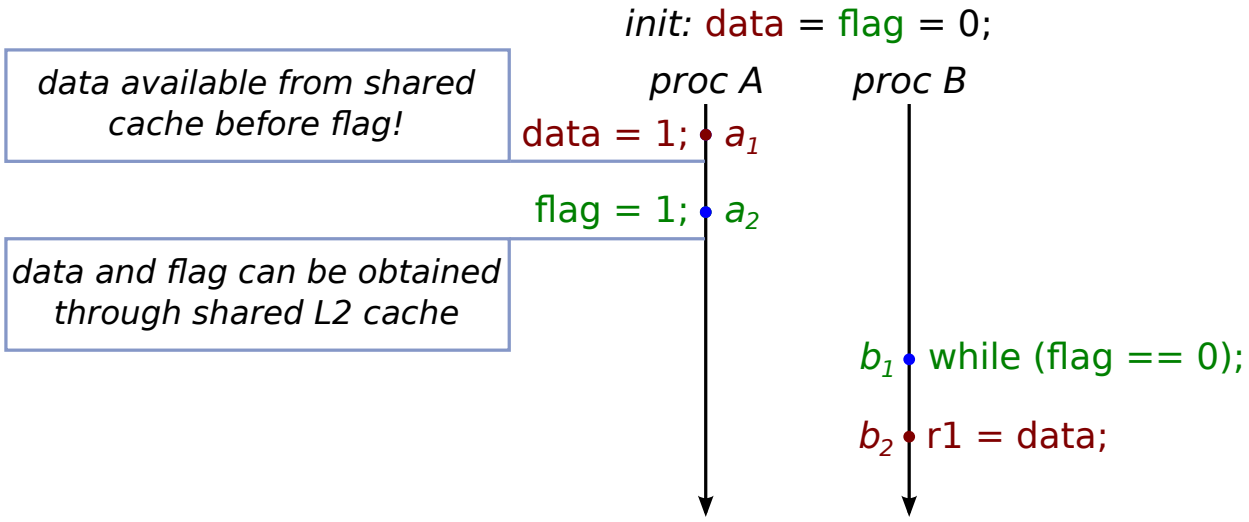
TSO-CC basic protocol



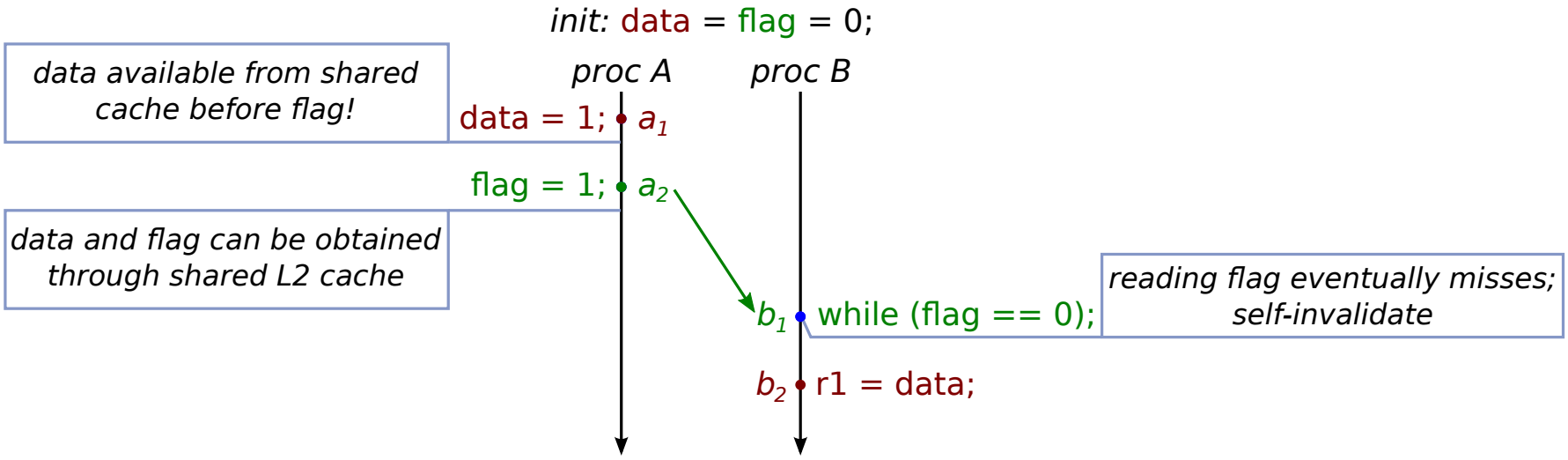
TSO-CC basic protocol



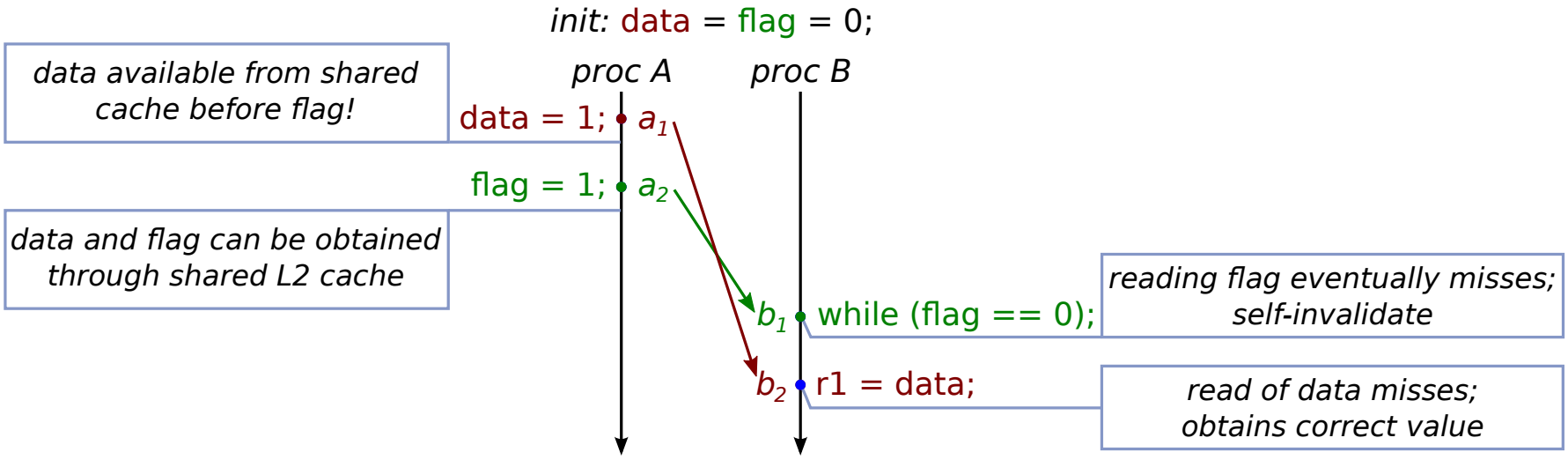
TSO-CC basic protocol



TSO-CC basic protocol



TSO-CC basic protocol



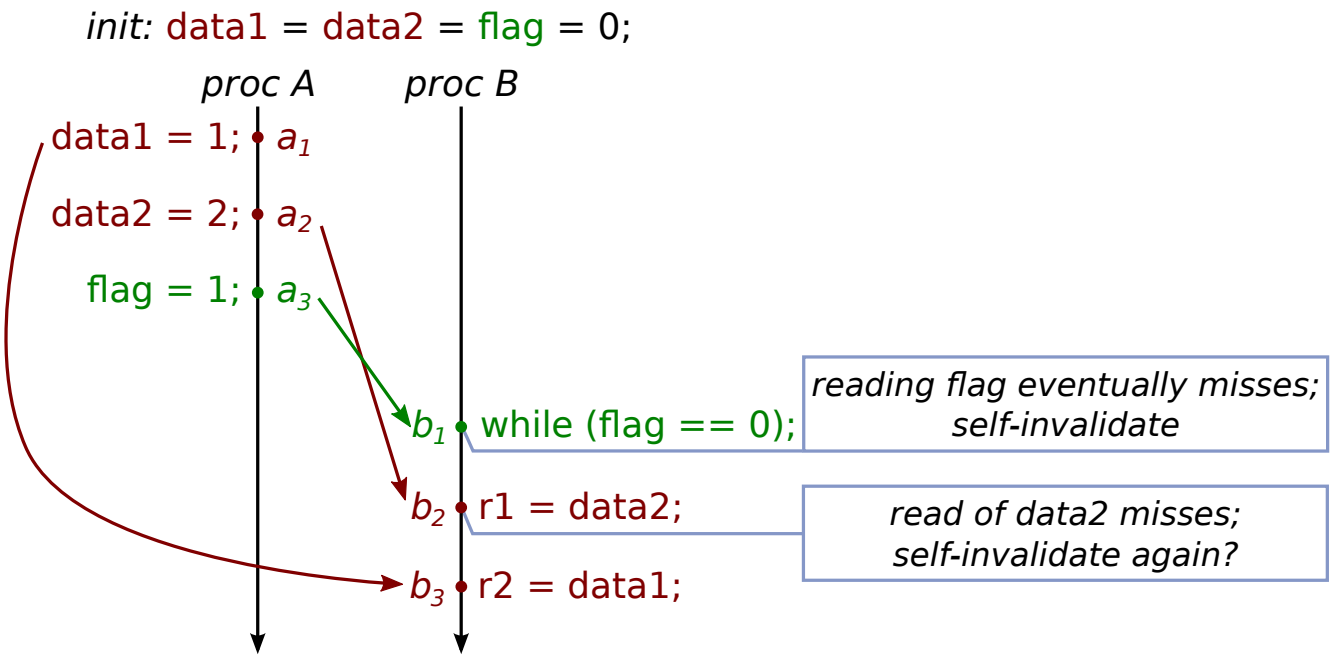
Guaranteed release/write propagation?

- No upper bound for release/write propagation delay.
- No consistency model guarantees this (including SC).
- Especially in TSO, where writes are propagated lazily.

Acquires require polling reads!

Correctness unaffected by maximum-accesses parameter.

How to reduce self-invalidations?

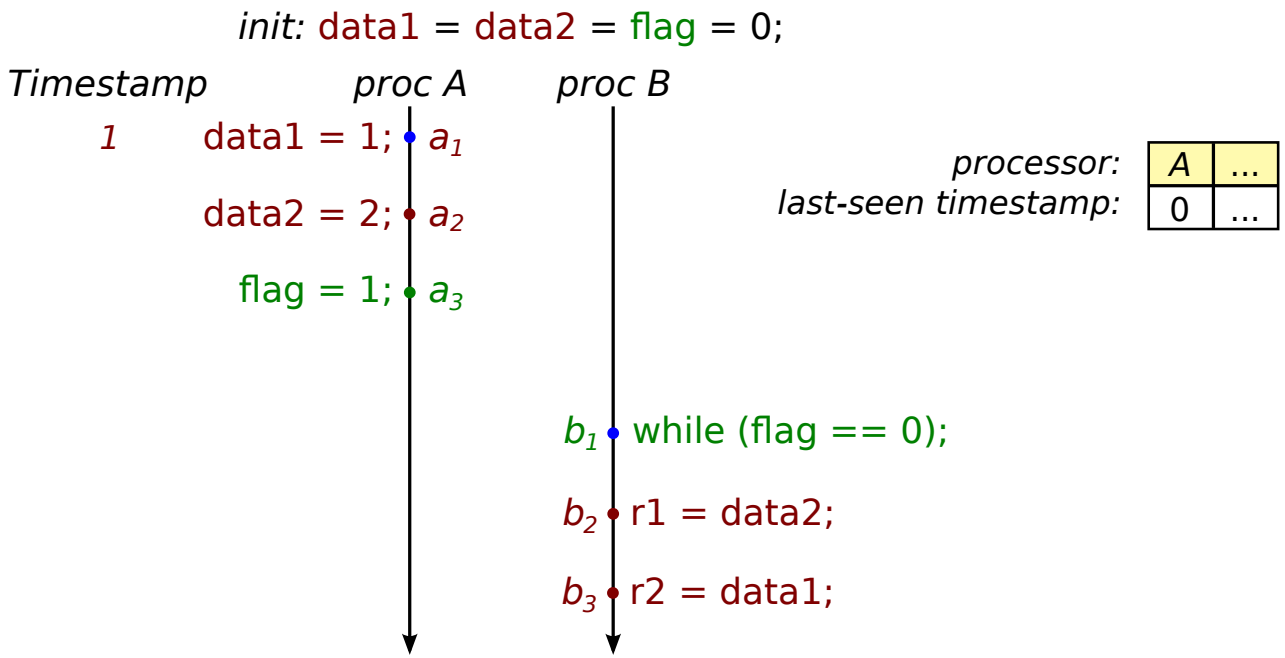


How to reduce self-invalidations?

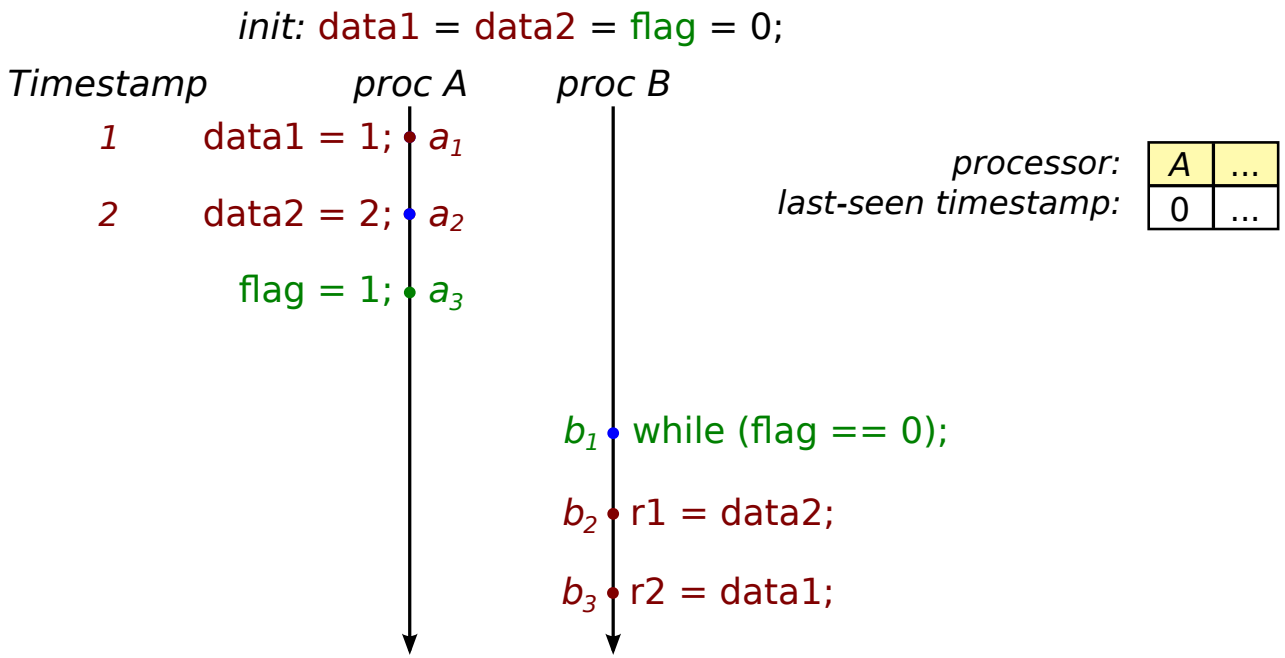
Reducing self-invalidations using timestamps

- L1s maintain monotonically increasing timestamp-source.
- Upon write, store current timestamp in cache line.
- L1s maintain a timestamp-table of last-seen timestamps.
- Upon miss, only self-invalidate if *response-timestamp* $>$ *last-seen timestamp*.

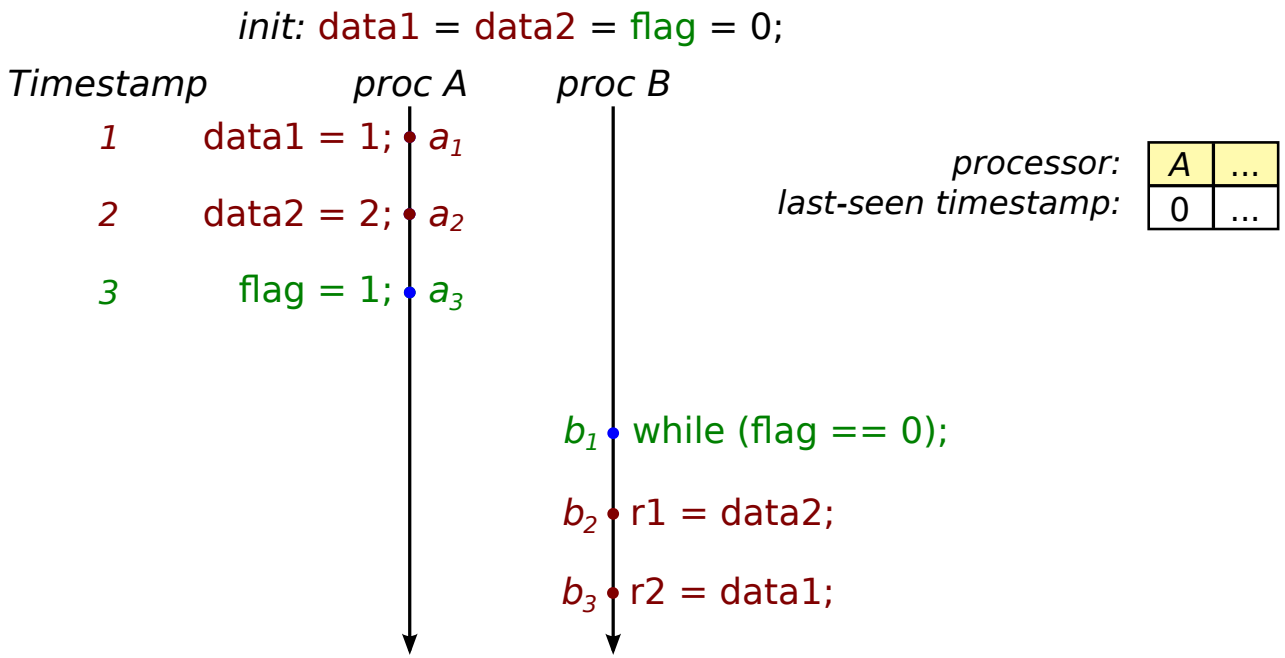
Reducing self-invalidations



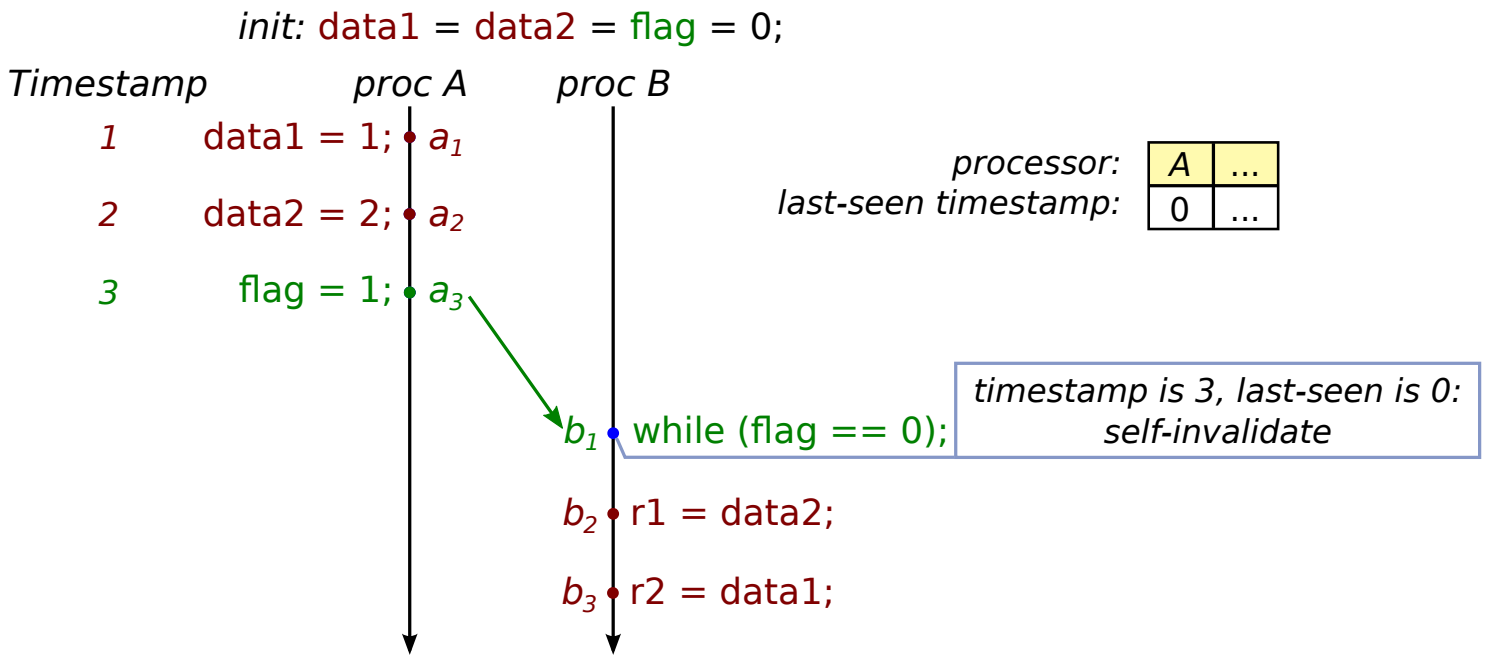
Reducing self-invalidations



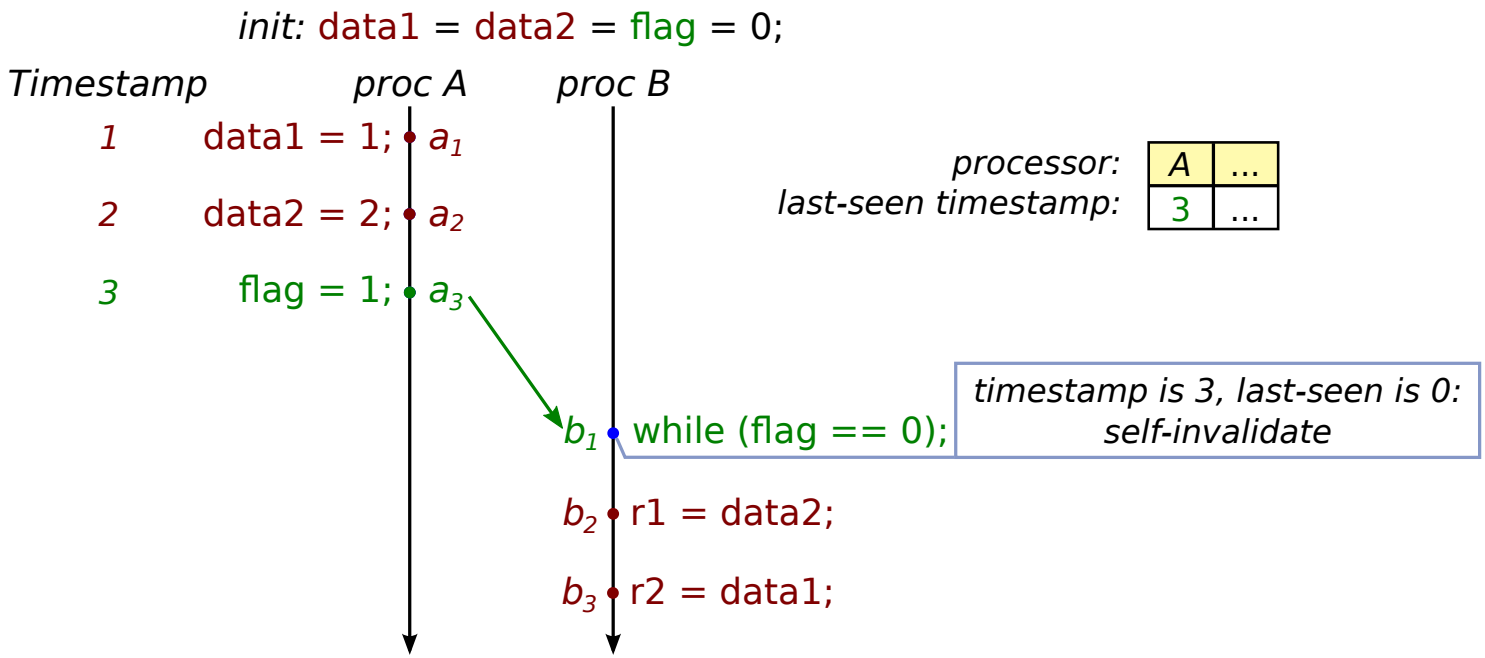
Reducing self-invalidations



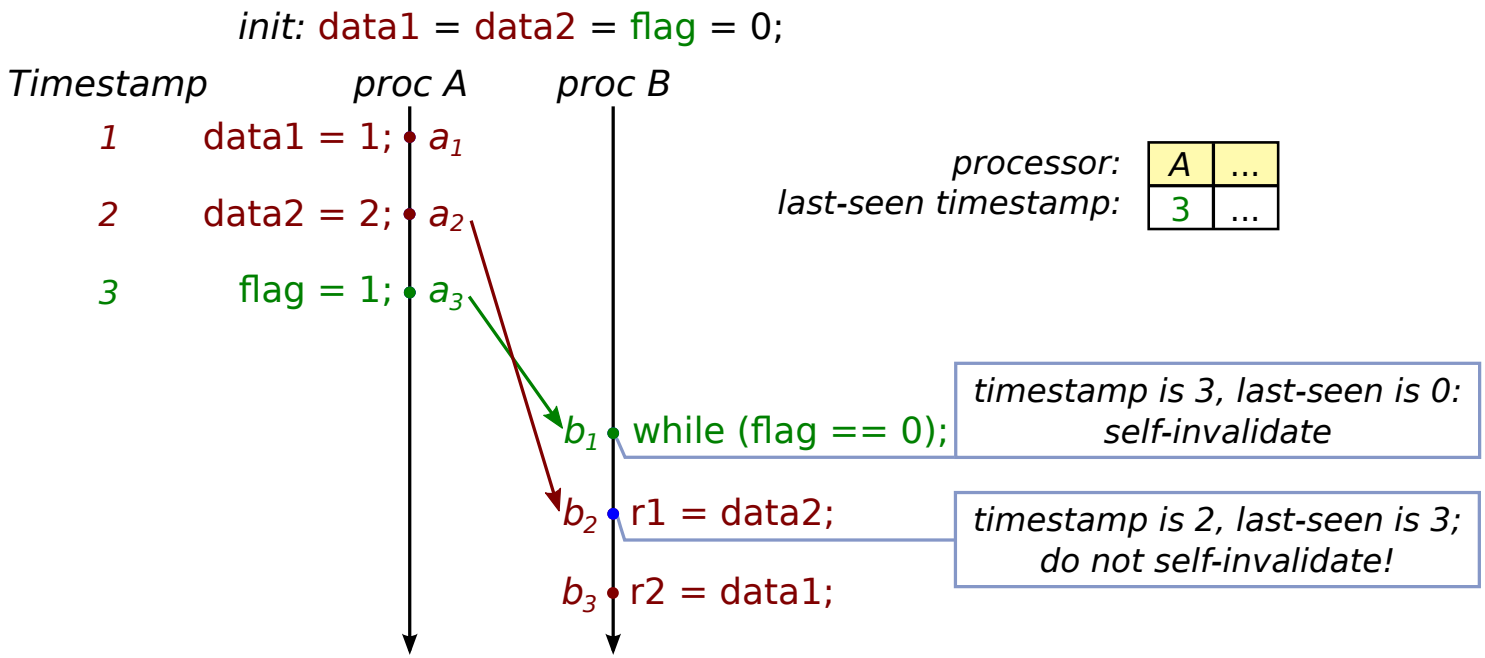
Reducing self-invalidations



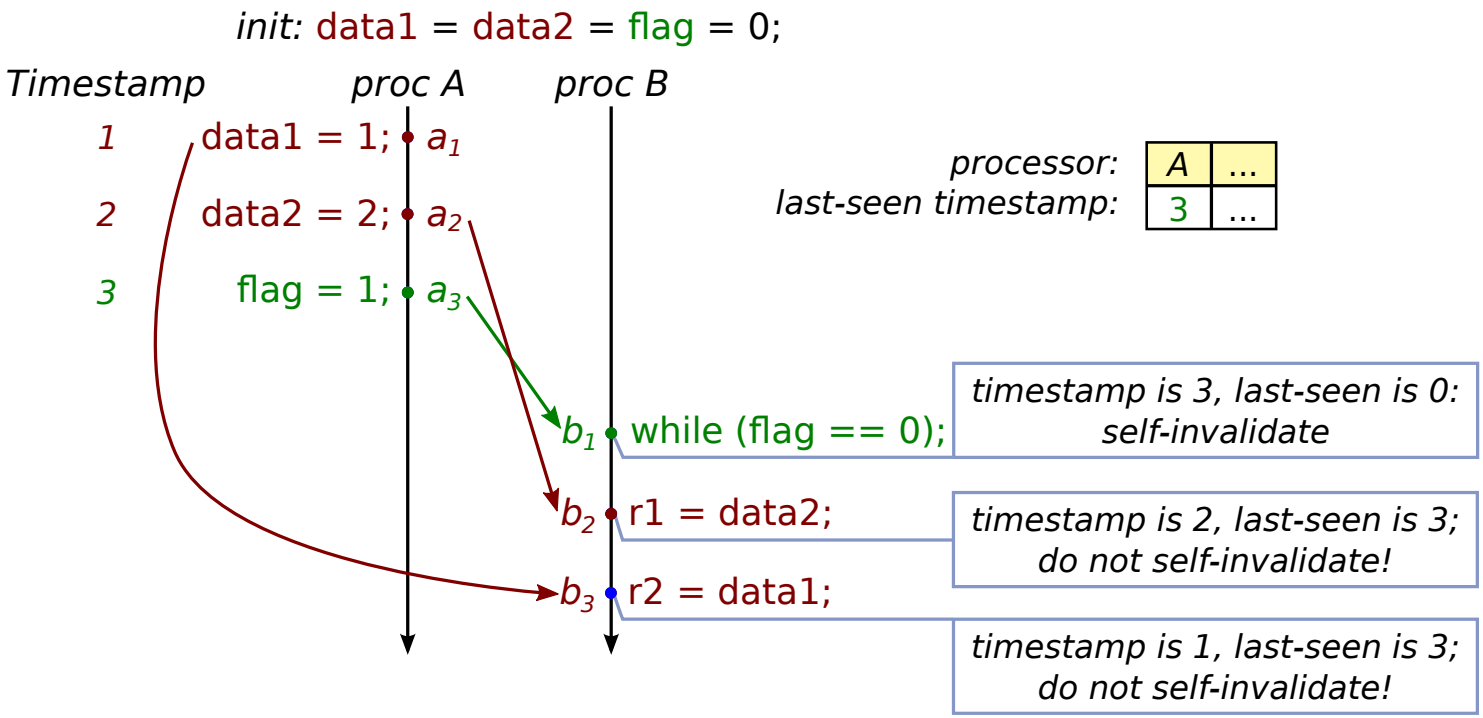
Reducing self-invalidations



Reducing self-invalidations



Reducing self-invalidations



More optimizations

Timestamp groups

Assign groups of writes the same timestamp (reduce resets).

Shared read-only

- Excluded from self-invalidation.
- Unmodified shared lines are classified as `SharedRO`.
- Modified shared lines “decay” (based on timestamp) to `SharedRO` state.
- Writes require invalidation broadcast (rare).

Evaluation Methodology

System setup

- Gem5 full-system simulator (x86_64).
- Ruby memory simulator with Garnet interconnect model.
- 32 out-of-order cores.
- Private L1s; NUCA organization for L2.

Benchmarks

- PARSEC, SPLASH-2 & STAMP.
- *Unmodified* program codes & Linux kernel.

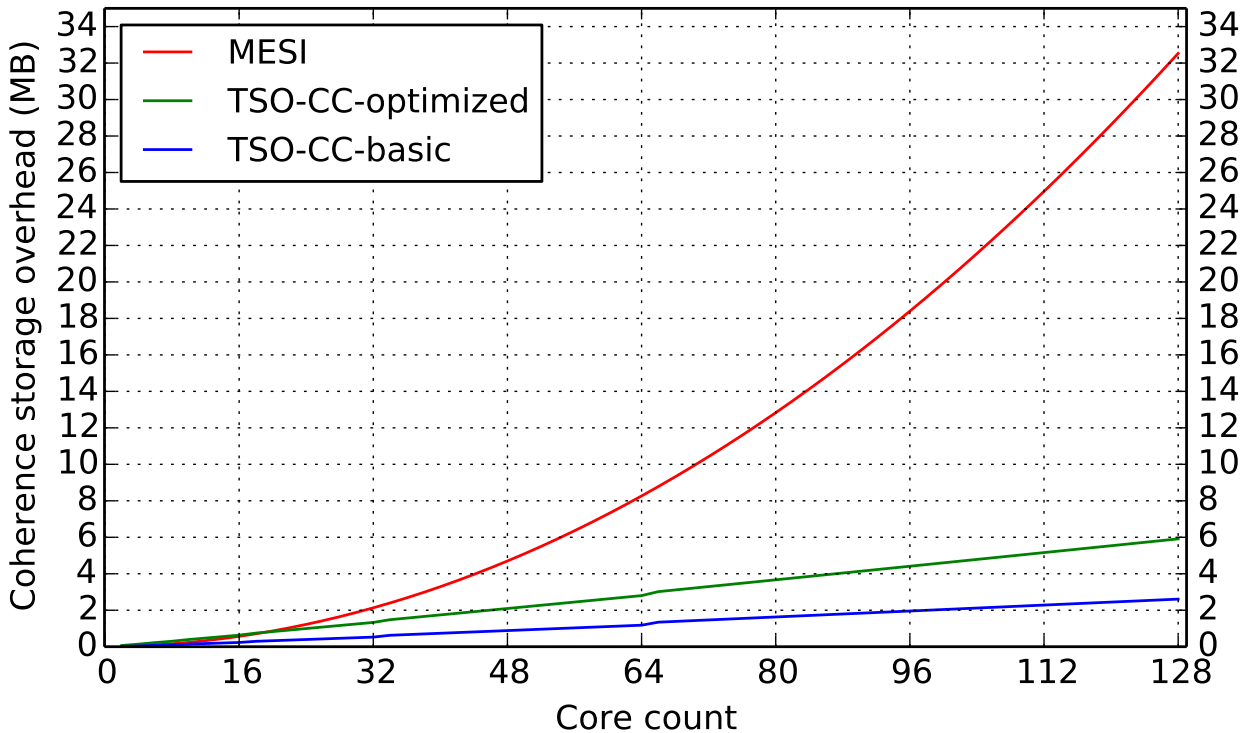
Verification

Experimental verification

- Generate Litmus tests for TSO using `diy`[1] tool.
- Litmus tests run in full-system simulator.
- But: does not provide full coverage.

[1] http://diy.inria.fr

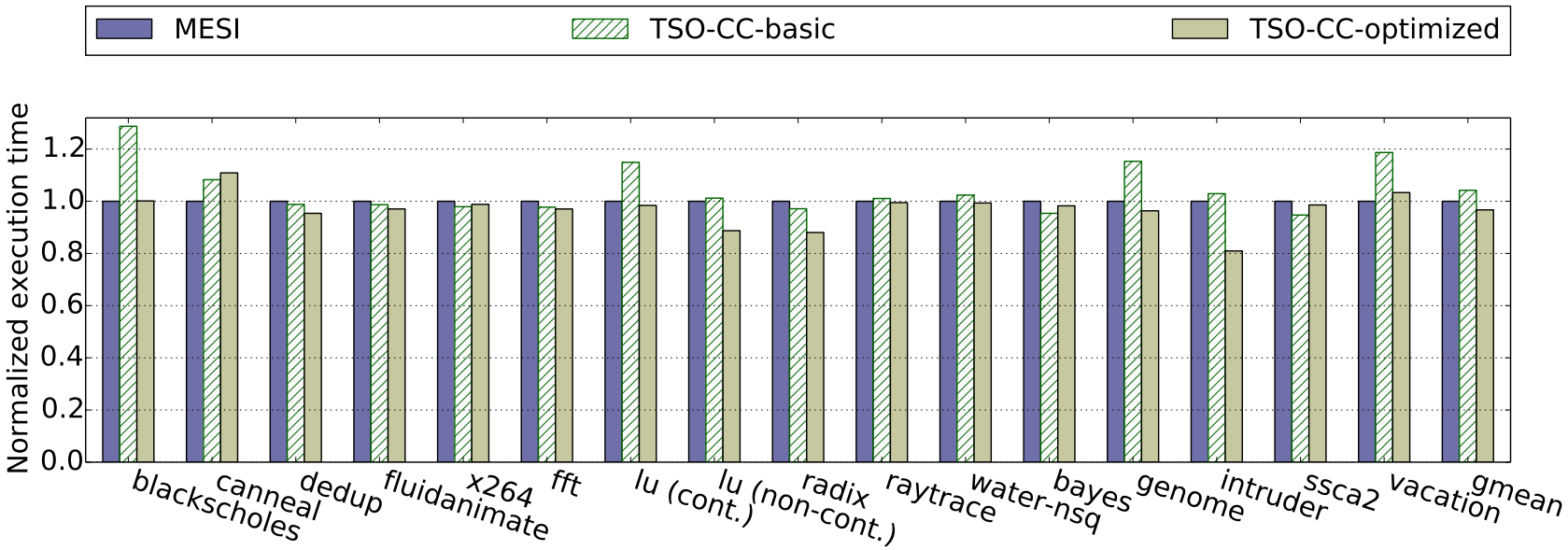
Storage overheads



At 32 cores = 40% reduction; 128 cores = 80% reduction.

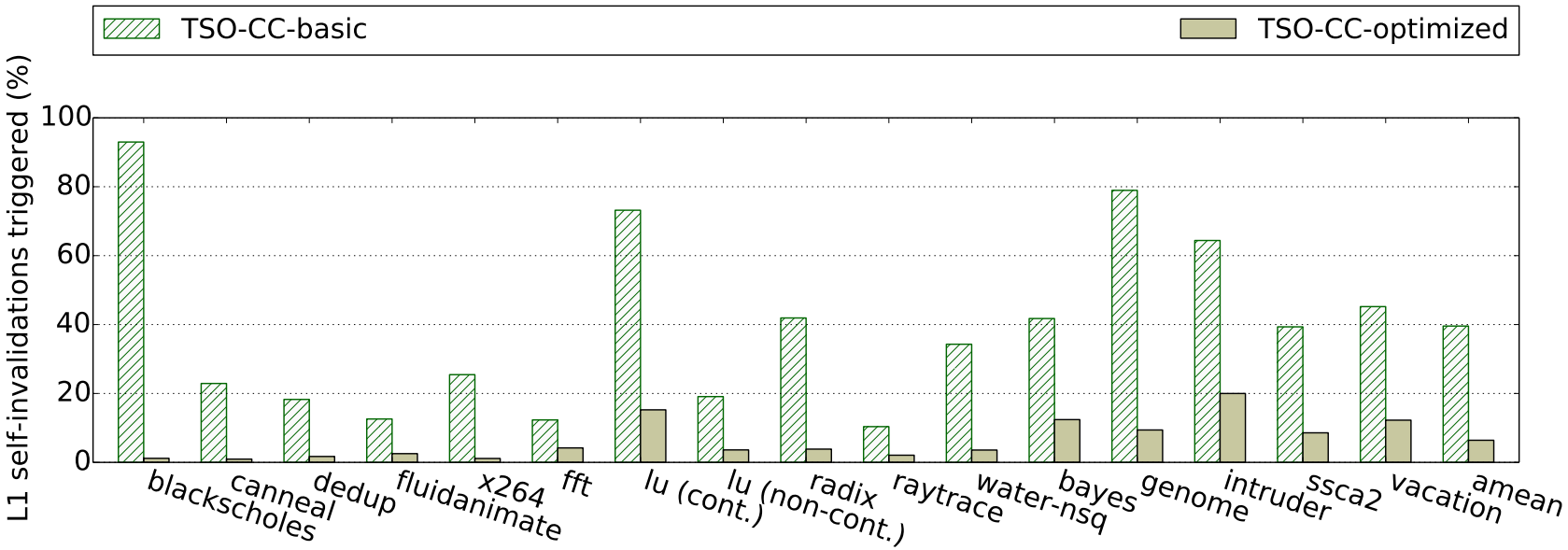
(4-bit accesses-counter, 12-bit timestamp; 1MB per L2 tile)

Results: Execution times



TSO-CC-optimized 3% (7%) faster than MESI (TSO-CC-basic).

Results: Self-invalidations



TSO-CC-optimized reduces self-invalidations by 87%.

Conclusion

Using eager coherence protocols for relaxed consistency processors is overkill.

TSO-CC: Lazy consistency-directed coherence for TSO

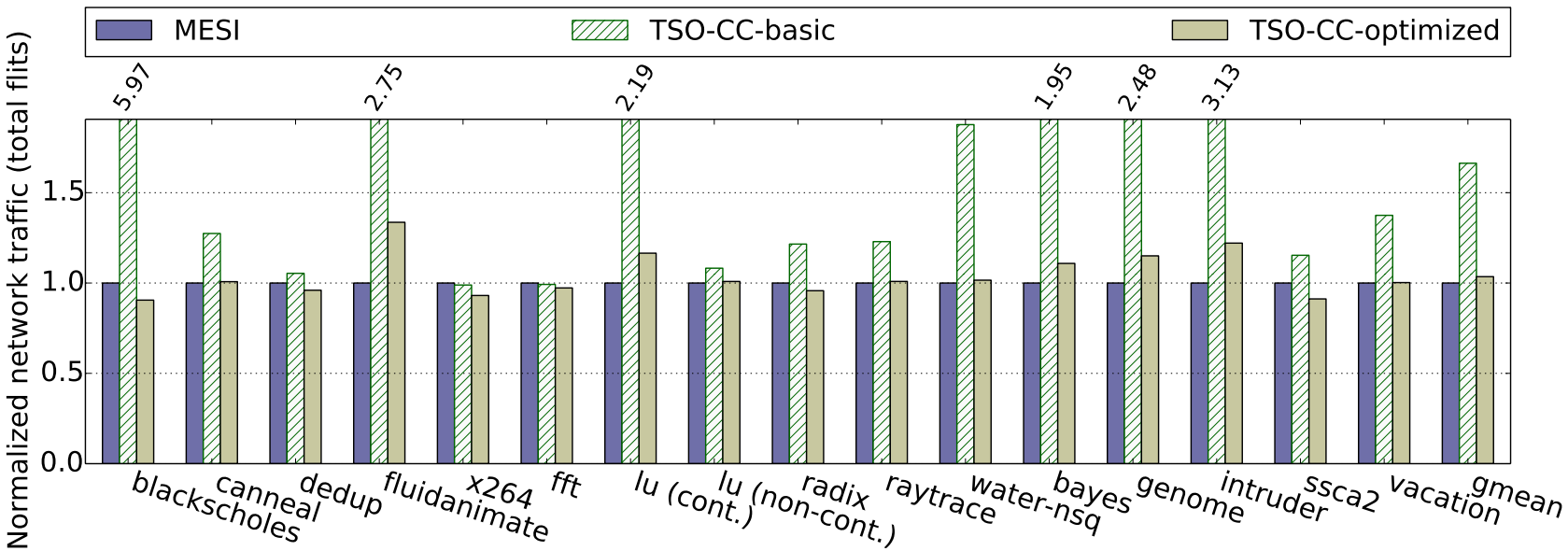
- Scalable in terms of storage overheads.
- Reduction in execution times.
- Executes unmodified program codes.

Future work

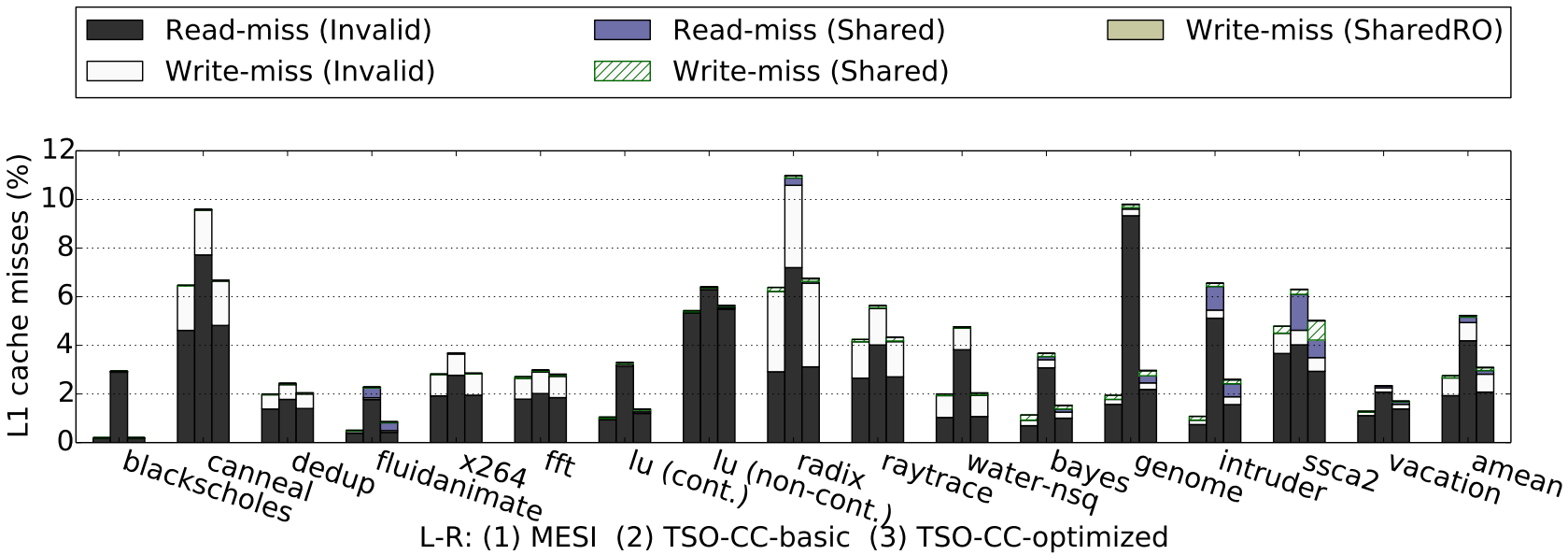
Further optimizations, verification.

[q&a]

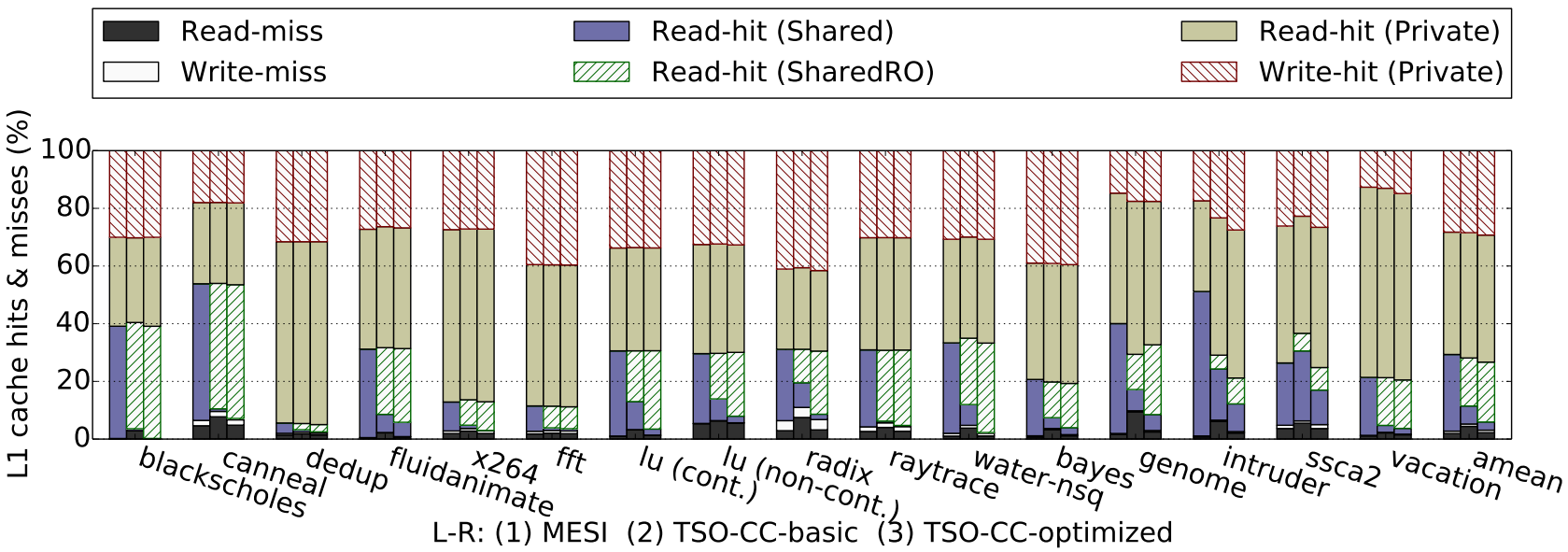
Results: Network traffic



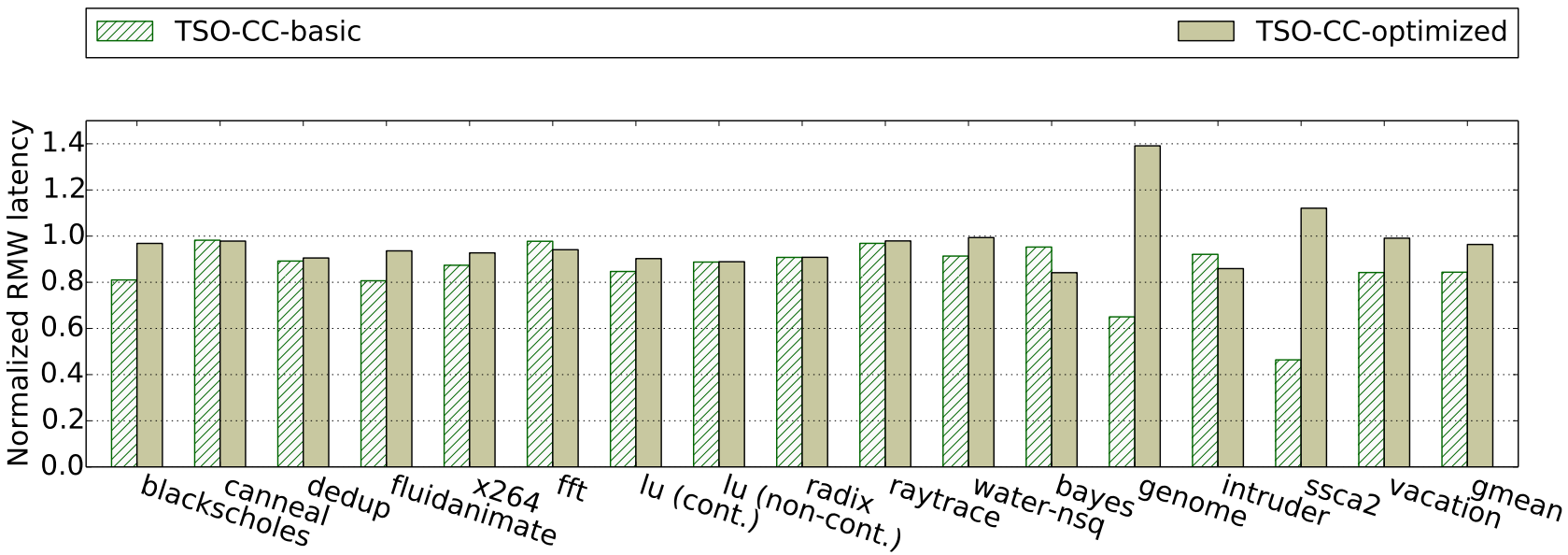
Results: L1 cache misses



Results: L1 cache hits & misses



Results: RMW latencies



TSO-CC basic protocol (preliminaries)

Transitions in L2 directory

read @ `Exclusive`: other L1 request forwards to owner, transition to `Shared`.

write @ `Exclusive`: other L1 request forwards to owner.

read @ `Shared`: L2 responds immediately.

write @ `Shared`: L2 responds immediately, transition to `Exclusive`.