Hardware Transactional Memory

Shao-Hung Chiu, Upasana Sridhar

Transactional Memory - Where did we come from?

- Common problems in conventional lock techniques
  - Priority Inversion: a low-priority process is **preempted** while holding a lock
  - Convoying: a process holding a lock is **descheduled**
  - Deadlock: processes lock the same set of objects in different orders
  - Software lock-free data structure does not perform as well as lock-based counterparts

Transactional Memory - Where do we go?

- Herlihy and Moss proposed transactional memory which makes **lock-free** synchronization efficient and easy to use for mutual exclusion
- New **instructions** are used to load, store, commit, abort and validate
- Transactional memory exploits and extends multiprocessor **cache-coherence protocol** so that transactions can be kept local
- Results show its competitive performance on simple benchmarks

Definition and properties of transaction

- A finite sequence of machine instructions executed by a single process
  - A transaction’s instructions cannot be interleaved with another’s
  - A transaction’s computation cannot be observed before commit
  - The change caused by a transaction is atomic and either commits or discards

ISA for accessing memory

- Load-transactional (LT): reads a shared-memory location
- Load-transactional-exclusive (LTX): same as LT, but hints this location is likely to be updated
- Store-transactional (ST): write to a shared-memory location, but new value is not visible until the transaction commits

- Read set: locations accessed by LT
- Write Set: locations accessed by LTX, ST
- Data set: Union of read set and write set

ISA for manipulating transaction state

- COMMIT: makes changes in write set visible and permanent. It succeeds only if no other transactions update the data set and have read the write set
  - Return: success or failure
- ABORT: discards all updates in the write set
- VALIDATE: tests the current transaction status.
  - True: the current transaction has NOT aborted
  - False: the current transaction has aborted and discards the transaction's tentative updates

Use the instructions

1. Use LT or LTX to read
2. Use VALIDATE to check if the values are consistent
3. Use ST to update
4. Use COMMIT to make changes permanent. If step 2 or step 4 fails, the process returns to step 1

- Transactions are small enough to complete in a single quantum and number of locations accessed does not exceed architectural limit

Proposed Architecture

- Committing or aborting a transaction is local to the cache
- Accessibility indicated by cache is good enough to detect transaction conflicts
- Snoopy Cache:
  - There are regular caches and transactional caches
  - Transactional caches can hold tentatives writes which can only be snooped or written back to memory after COMMIT
  - Transactional caches are small and fully-associated for parallel logics to handle abort or commit

Cache line states

Name	Access	Shared?	Modified?
INVALID	none	—	—
VALID	R	Yes	No
DIRTY	R, W	No	Yes
RESERVED	R, W	No	No

Table 1: Cache line states

Name	Meaning
EMPTY	contains no data
NORMAL	contains committed data
XCOMMIT	discard on commit
XABORT	discard on abort

Table 2: Transactional tags

Bus cycle types

Name	Kind	Meaning	New access
READ	regular	read value	shared
RFO	regular	read value	exclusive
WRITE	both	write back	exclusive
T_READ	trans	read value	shared
T_RFO	trans	read value	exclusive
BUSY	trans	refuse access	unchanged

Table 3: Bus cycles

Processor actions for transactions

- Flags
  - TACTIVE: indicates a transaction is in progress
  - TSTATUS: indicates a transaction is active or aborted
- Flag state transition
  - If processors receive BUSY signal from bus, they set TSTATUS to false.
  - VALIDATE: returns TSTATUS. If false, sets TACTIVE to false and TSTATUS to true
  - ABORT: sets TSTATUS to true and TACTIVE to false
  - COMMIT: returns TSTATUS. Sets TSTATUS to true and TACTIVE to false.

Simulations - Counting Benchmark

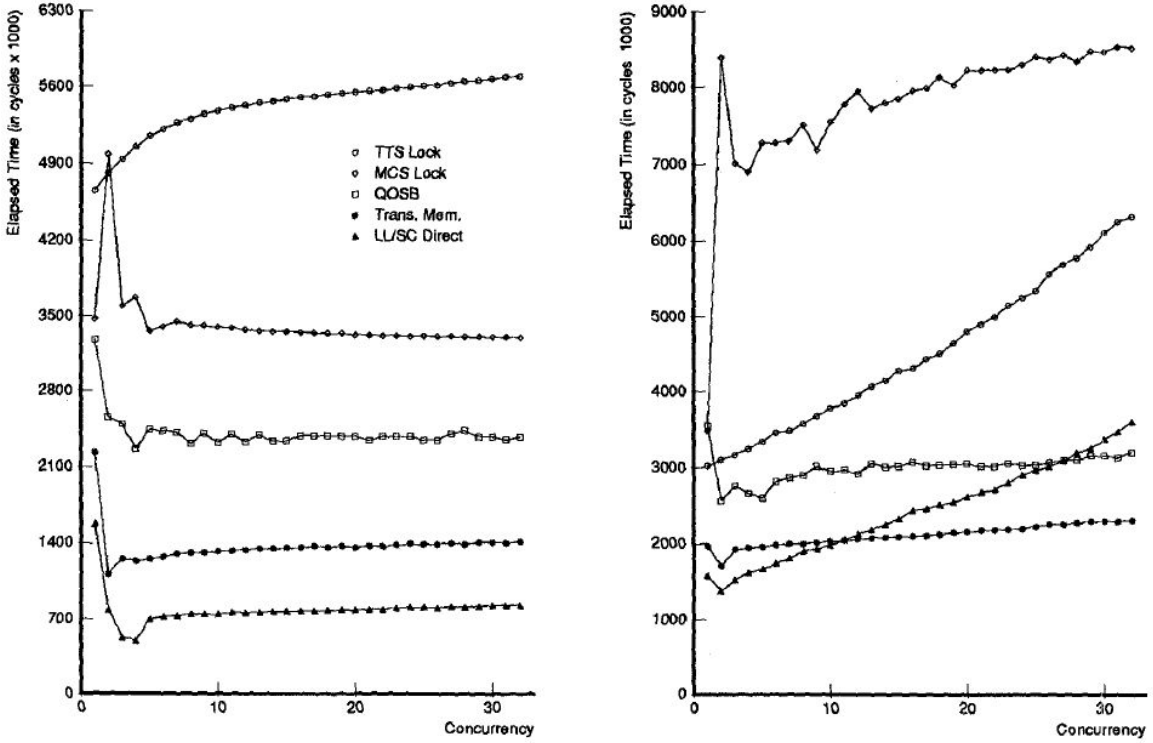


Figure 4: Counting Benchmark: Bus and Network

Explanation - Counting Benchmark

- Transactional memory performs better than TTS spin locks since it requires fewer memory access.
- LL/SC is the best for this task since it does not require COMMIT which operates on cache.
    - But LL/SC only has advantages for data not spanning over 1 word.

Simulations - Producer/Consumer Benchmark

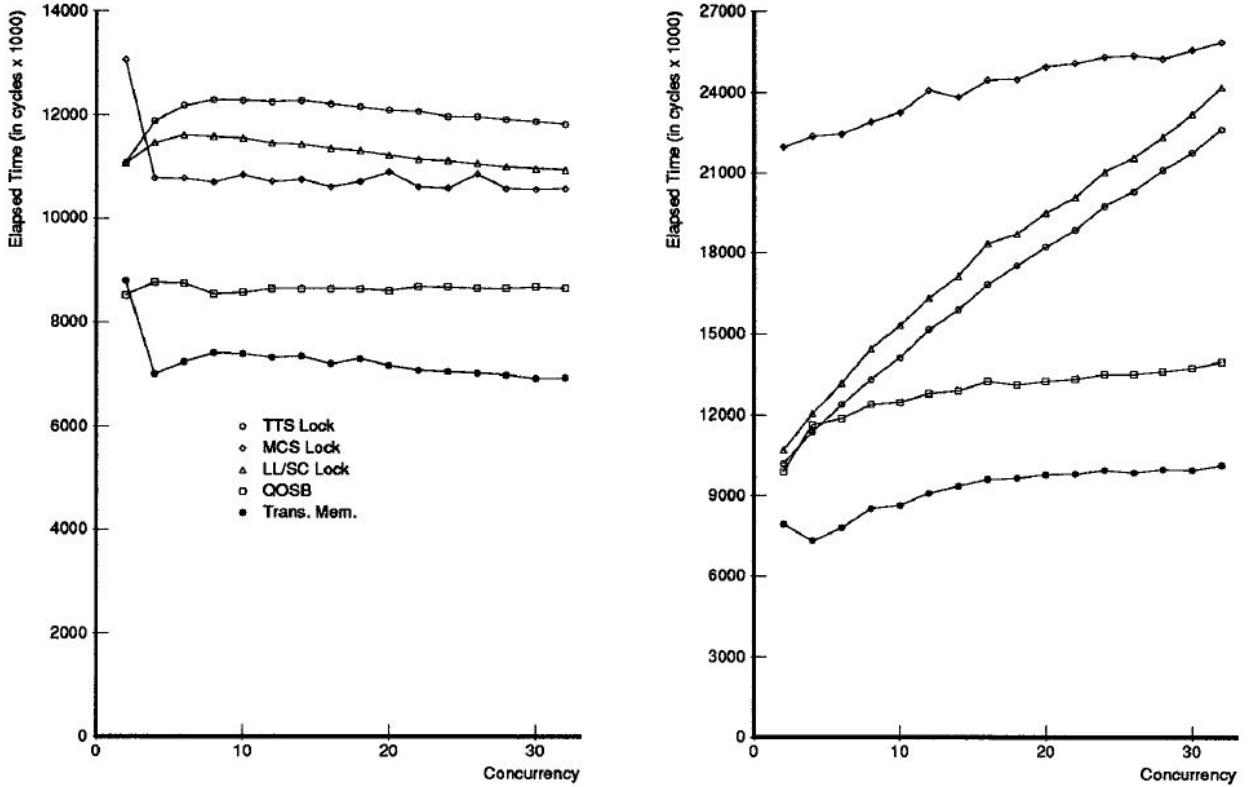


Figure 5: Producer/Consumer Benchmark: Bus and Network

Explanation - Producer/Consumer Benchmark

- For bus architecture, all throughputs are essentially flat
- For network architecture, throughputs suffer from contention increases, but transactional memory suffer the least.

Simulations - Doubly-Linked List Benchmark

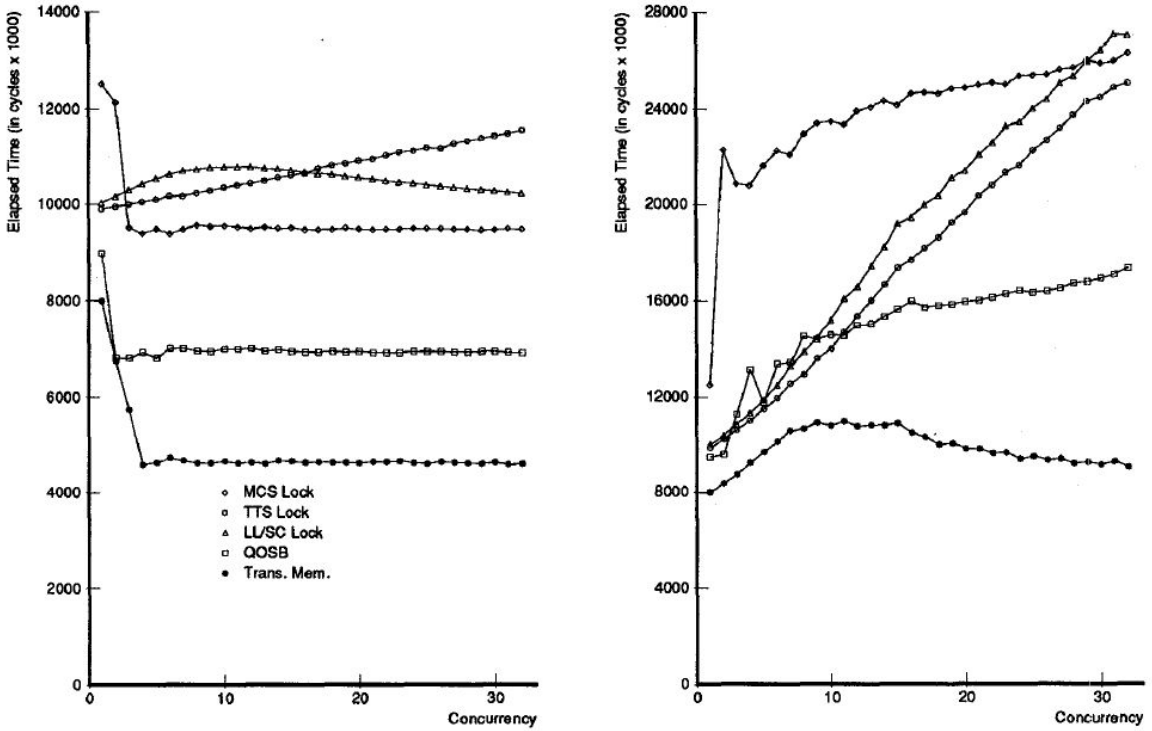


Figure 6: Doubly-Linked List Benchmark: Bus and Network

Explanation - Doubly-Linked List Benchmark

- This benchmark contains ambiguity
  - Empty list can cause enqueuers and dequeuers deadlock
- Transactional memory perform better capability of parallelism by using VALIDATE to check the validity of a pointer

Wrap-up for transactional memory

- Herlihy and Moss sketched how a lock-free synchronization mechanisms can be implemented
  - By adding new instructions
  - By adding a small transactional cache
  - By making minor changes to the cache coherence protocol
- Simulations show that transactional memory outperforms for fewer shared memory accesses
- Herlihy's and Moss' transactional memory assumes short durations and small data sets
  - A long transaction tends to be aborted by an interrupt or conflict
  - A large data needs larger transactional cache and leads to more synchronization conflicts

Making the fast case common and the uncommon case simple in unbounded transactional memory

Bounded Transactional Memory has Problems

- Herlihy and Moss - *‘Transactions are short and don’t access a lot of data.’*
- The cost of this assumption is large when
  - a transaction exceeds the time limit (interrupts/ context switches)
  - a transaction exceeds the data limit (size of the transactional cache).

Okay, then make Transactional Memory unbounded

- Allowing for multiple overflowed transactions to execute concurrently makes hardware complex.
- These implementations must keep track of
  - Each transaction’s dataset
  - Each memory block that is being accessed (grows with number of concurrently executing transactions)

Unbounded Transactional Memory, but slow

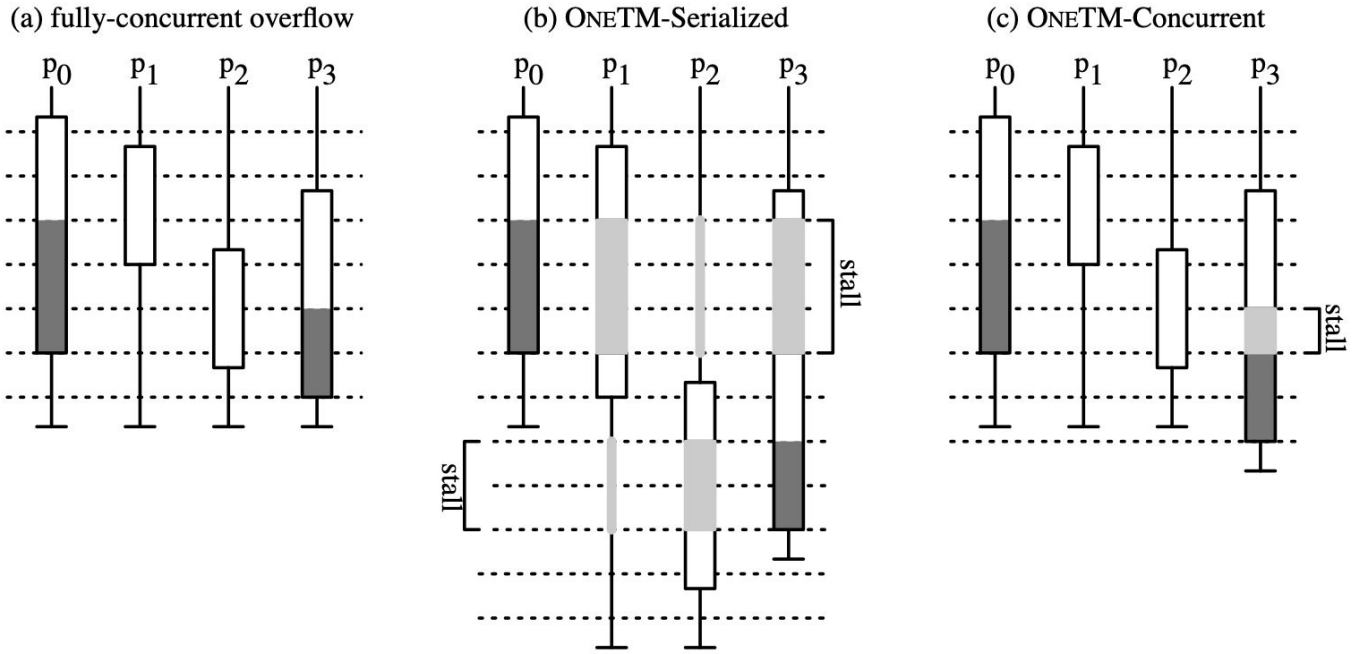
- No two overflowing transactions can execute concurrently
  - *This makes the logic to handle these overflows relatively simple*
  - Two proposals for overflow handlers: OneTM-Serialized and OneTM-Concurrent.
- Permissions-only cache tracks coherence state but contains no data
  - *This raises the threshold for a transaction to overflow.*

The Permissions Only Cache

- Data-less encoding of coherence information.
- No need to access it for processor-local memory ops.
- The cache is usually empty, can be turned off to save on power
- In the best case, permissions only cache can track 1MB of transactional data.

Backup Slides

Handling Overflowed Transactions



Gray blocks indicate overflowed transactions

OneTM-Serialized

- Abort the overflowed transaction and restart in “overflowed mode”
- Check STSW until no other thread is executing an overflowed transaction
  - Shared Transaction Status Word (STSW) resides in a location known to all threads.
  - STSW is hidden behind a mutex lock.
- Set the STSW.
- Execute the overflowed transaction.

OneTM-Serialized

- The PTSW stores state in case a thread is pre-empted while it is executing an overflowed transaction.
- This is saved across context switches, so that the thread can resume its transaction.

OneTM-Concurrent

- The system maintains metadata about the overflowed transaction
- All other threads check this metadata for conflicts

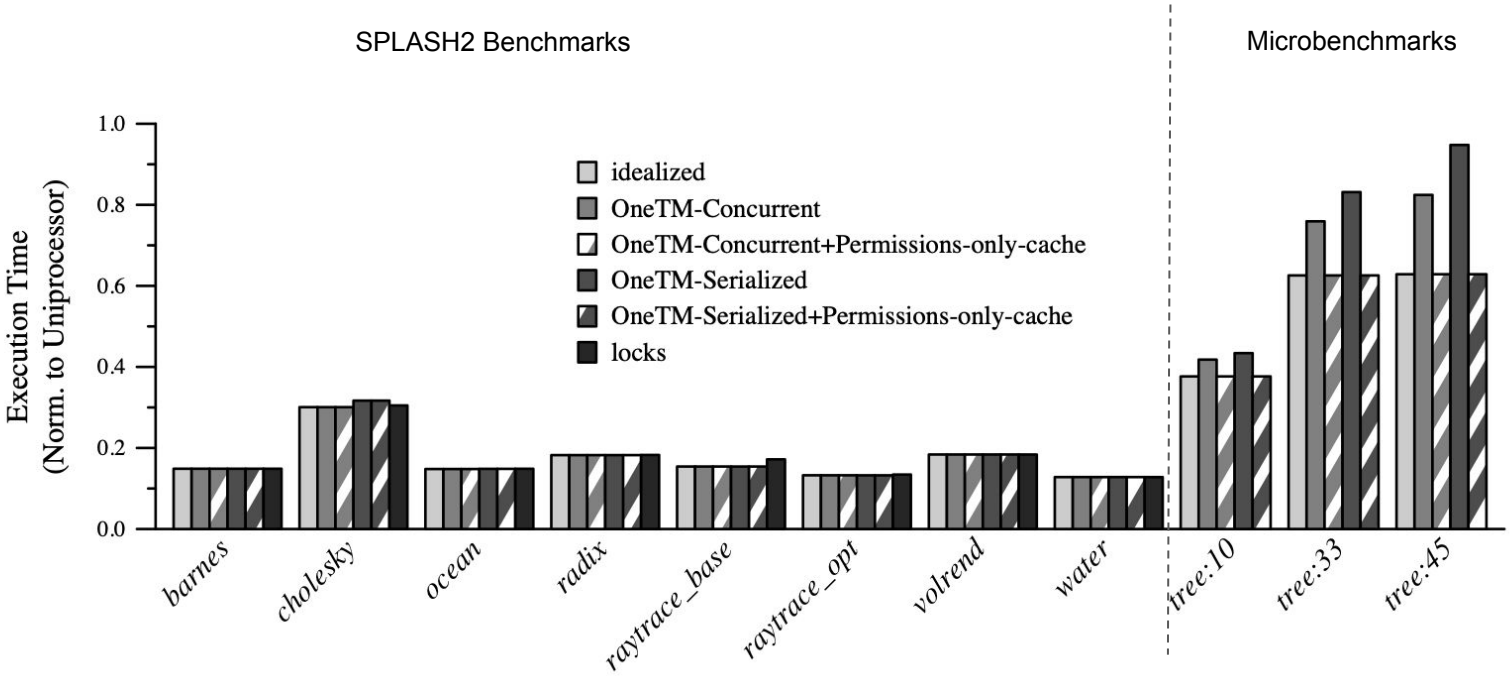
OneTM-Concurrent: Using Metadata

- Metadata is cleared lazily
- Use the concept of ownership to handle metadata coherence

Benefits of Simplicity

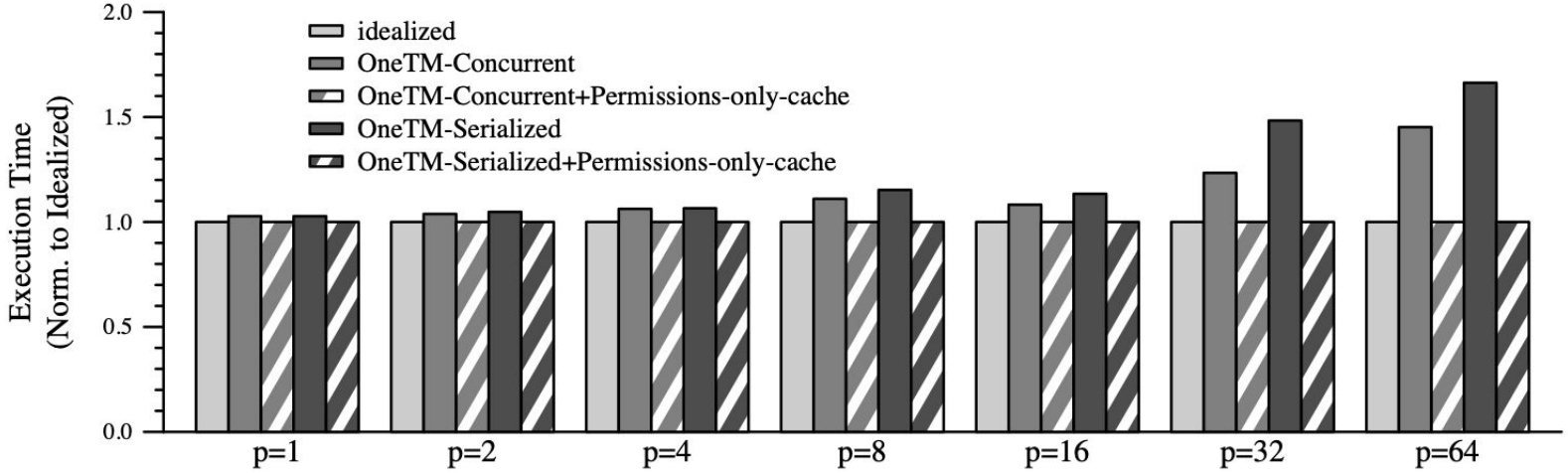
- Conflict Detection is cheap
- Committing an unbounded transaction is simple
- Aborts do not involve synchronization costs - walk down a thread local log

Results



Ideal Transactional Memory Vs Different Flavors of OneTM

Results



Scalability tests on the Microbenchmark

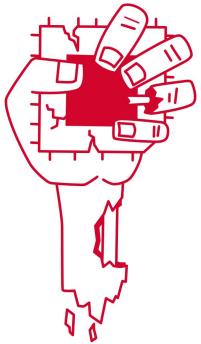
Critiques

- It would be interesting to see a comparison with a different implementation of unbounded transactional memory.
- This would quantify the difference between having multiple (concurrent) overflowing transactions and serializing them.
- Not clear how the permissions cache helps with overflows caused by interrupts.
- How does the metadata work work aligned data-types?

Ghosts of Transactions Past and Present

HSW136. Software Using Intel® TSX May Result in Unpredictable System Behavior

Problem:	Under a complex set of internal timing conditions and system events, software using the Intel® TSX instructions may result in unpredictable system behavior.
Implication:	This erratum may result in unpredictable system behavior.
Workaround:	It is possible for the BIOS to contain a workaround for this erratum.
Status:	For the steppings affected, see the *Summary Table of Changes*.



ZombieLoad Attack

Return of the Leaking Dead

Watch out! Your processor resurrects your private browsing-history and other sensitive data.

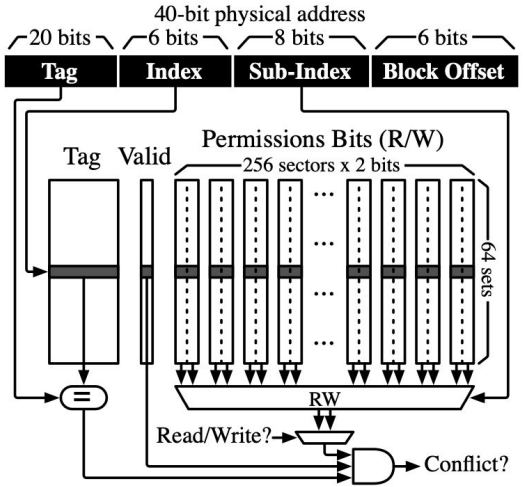
https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf
https://zombieloadattack.com

Usage

- Read by external coherence requests as part of conflict detection
- Updated when a transactional block is replaced from the data cache
- Invalidated on a commit or abort
- Read on transactional store misses to avoid redundantly logging the block

Implementing the Permissions Only Cache

- Optimizing logging - if a block's write bit has been set, it need not be logged again.
- Use second level cache frames instead of a dedicated structure
- Efficient Encoding a la sector caches



Metadata Implementations

- Cordon off a region of memory to store metadata
- Metadata is coupled with data.
- The OTID helps to defer clearing out metadata.
- But beware of false delays.