Hardware Transactional Memory Implementation

This handout introduces two hardware transactional memory (HTM) designs: *(1)* eager & pessimistic and *(2)* lazy & optimistic. It also details a specific implementation of the lazy & optimistic design. These details will be assumed in some questions in Quiz 3.

- **Eager & Pessimistic HTM** uses eager version management and pessimistic conflict detection. For every transactional load, the memory system checks whether this load reads an address in the write set of any other transaction, and declares a conflict if so. For every transactional store, the memory system checks whether this store writes an address in the read set or write set of any other transaction, and declares a conflict if so. Upon a conflict, the transaction receiving an invalidation or downgrade aborts, i.e. the *requester wins*.

- **Lazy & Optimistic HTM** uses lazy version management and optimistic conflict detection. Conflicts are detected when a transaction attempts to commit. The finished transaction validates its write-set with coherence actions. If any of its writes appear in the read- or write-set of other transactions in the system, a conflict is declared. Analogous to pessimistic requester-wins, the *committer wins*.

Implementation details of lazy & optimistic HTM

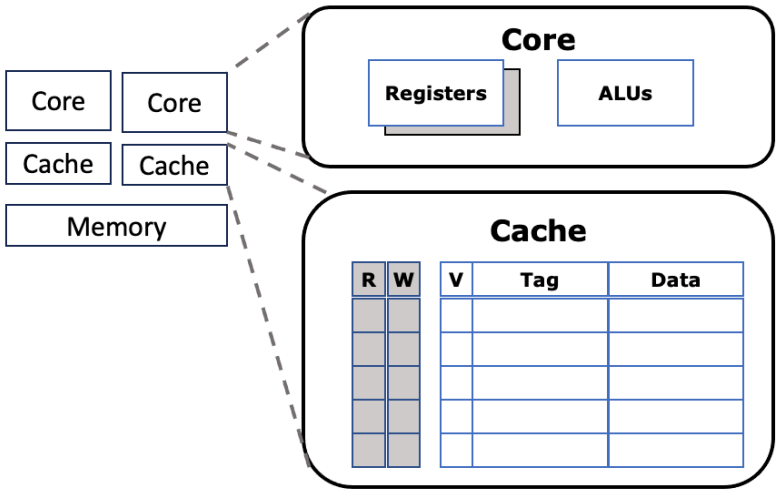

Figure 1. Illustration of our lazy & optimistic HTM implementation. HTM-specific state is shown in gray: the checkpoint register file, and read (R) and write (W) bits.

Figure 1 illustrates our lazy & optimistic HTM implementation, which we describe below.

Structure: This design uses a simple memory hierarchy: each core has a single private cache; private caches communicate with each other and with memory through a shared bus. Cache coherency and conflict detection are implemented by snooping on bus requests. Caches implement an MSI protocol.

Transactional state: Caches and cores are extended to track transactional state. These additions are highlighted in grey in Figure 1:

- Each line in the private caches has a read and a write bit associated with it. These read and write bits are used to keep track of the transaction's read- and write-sets, i.e., the sets of lines that have been read or written during the transaction.
- Each core includes a checkpoint register file that records the thread context at the beginning of the transaction.

Transactional execution:

- *Start:* A `tx_begin` instruction initiates transactional execution. The core checkpoints the thread's registers in the checkpoint register file.
- *Loads:* Within a transaction, each load instruction sets its line's *read bit*. Misses are handled just like for non-transactional loads, i.e., through share requests.
- *Stores:* Within a transaction, each store instruction sets its line's *write bit*. On a cache hit, if the write bit was not set and the existing line is dirty, the line is written back to memory before performing the store. This ensures that memory has the latest non-speculative value of the line; speculatively updated data from the transaction is kept in the private cache. On a cache miss, the core issues a *share request* for the line, not and exclusive one. Thus, to other caches, transactional store misses are indistinguishable from load misses, and do not cause invalidations.
- *End:* A `tx_end` instruction denotes the end of a transaction. The core initiates the commit process and does not issue further instructions until the transaction has committed or aborted.
- *Transaction commits are serialized:* A commit arbiter is a simple hardware component that ensures that *at most one transaction commits at a time.* When a transaction completes execution, the core requests commit permission to the arbiter. The arbiter grants this permission to at most one core at a time.
- *Commit:* When a core is granted commit permission from the arbiter, the core *(1)* issues upgrade requests for all the cache lines with the write bit set that do not already have write permission (i.e., are not in M state), getting exclusive permission for them and invalidating all other copies; *(2)* clears all read and write bits in the cache; and *(3)* notifies the arbiter that it has completed commit.
- *Aborts:* A core running a transaction aborts it if it snoops (sees) an exclusive request on the bus from another core that conflicts with its read or write bits (i.e., it matches a line with the read or write bit set). Aborts can happen any time from the first transactional memory access until the core receives commit permission.
- *No evictions of transactional lines:* A line that has its read or write bit set cannot be evicted. For simplicity, if the cache chooses to evict a transactional line, the transaction is aborted.