CMP SCI 635: Modern Computer Architecture

Transactional Memory: Architectural Support for Lock-Free Data Structures

Name: Kunjal Panchal Date: 26th Sept, 2019

Student ID: 32126469 Paper: 7 – Memory [Moss93]

Strengths:

1. It is flexible to add the transactional cache to the existing system. It doesn't upset the underlying memory architecture, just adds a new module to it. The changes made to cache coherence protocols are also straight forward.

2. Using no explicit locks and performing fewer shared memory accesses will result in outperforming other existing techniques for mutual exclusion.

Weaknesses:

1. If a transactional LTX [Load Transactional Exclusive] operation has been processed, then if a non-transactional Load on the same write item comes, the conflict will make the former transaction abort, in which case the extra overhead to use transactional instructions is not useful. And still allowing non-transactional instructions, may result in Deadlocks, Convoying or Priority Inversion as before.

2. There will be two entries for each transaction operations in the primary cache. It might incur a space overhead. Especially if a transaction is a big one with many operations. Also, multiple copies of a data item in different cache or different cache lines of the same cache might incur space overhead too.

3. "*Our implementation relies on the assumption that transactions have short durations and small data sets. The longer a transaction runs, the greater the likelihood it will be aborted by an interrupt or synchronization conflict.*"

Questions/Assertions:

1. For the timer interrupts for the stalled instructions; how to determine the balanced time interval to wait before releasing the resources held by a stalled transaction? If the interval is too small, most transactions can be flagged as stalled while it may be that they are false negatives, while a large interval may result in less throughput as many stalled transactions might hold resources for too long, making other transactions wait.

2. When a transaction reads BUSY bus cycle, how does it try to acquire resources again? Does it constantly poll or an interrupt is sent when a resource/ data item is released/ in shared mode again. {Answered in later sections: TTS Spin Locks and Software Queueing}

   {Follow up question} Are there any more sophisticated software queueing techniques considered besides FIFO? Although, I think only FIFO makes sense when we are talking about read and write dependencies. Cannot make a switch of order between R-W or W-R or W-W dependencies.

3. How/Why orphan transactions still continue the execution? Does aborting a transaction require more clean-up than just dropping the cache lines associated with it?

4. How does this transactional memory react to hardware failures? Even if a transaction has committed, a hardware failure (e.g. Power Outage, Component Failures) might occur before all the write data items have been written back to main memory? In what stage the transaction will be upon reboot?

5. It says that transactional memory doesn't have to deal with mutual lock variable overhead, but doesn't setting up the flags in cache require extra clock cycles too?

6. Are "CONCURRENCY" and "PARALLELISM" interchangeable terms when we are talking about performance? If not, which one should be the main focus of a software or hardware design if it pertains to performance or throughput?

7. How the granularity(atomicity) of Software Transactional Models are decided? Too minute granularity can result in race conditions and granted that STMs don't suffer from deadlocks, it can still suffer from live locks (?).