Transactional Memory

Parag Dixit Bruno Vavala

Computer Architecture Course, 2012

Overview

- Shared Memory Access by Multiple Threads
- Concurrency bugs
- Transactional Memory (TM)
- Fixing Concurrency bugs with TM
- Hardware support for TM (HTM)
- Hybrid Transactional Memories
- Hardware acceleration for STM
- Q & A

Shared Memory Accesses

- How to prevent shared data access by multiple threads?
  - Locks : allow only one thread to access.
  - Too conservative – performance ?
  - Programmer responsibility?
- Other idea ?
  - Transactional Memory : Let all threads access, make visible to others only if access is correct.

Concurrency bugs

- Writing correct parallel programs is *really* hard!
- Possible synchronization bugs :
  - Deadlock – **multiple locks** not managed well
  - Atomicity violation – no lock used
  - Others – priority inversion etc. not considered
- Possible solutions ?
  - Lock hierarchy; adding more locks!
  - Use Transactional Memory :

  Worry free atomic execution

Transactional Memory

- Transactions used in database systems since 1970s
- All or nothing – Atomicity
- No interference – Isolation
- Correctness – Consistency
- Transactional Memory : Make memory accesses transactional (atomic)
- Keywords : Commit, Abort, Spec access, Checkpoint

Fixing concurrency bug with Transactional Memory

- Procedure followed
  - Known bug database – Deadlock, AV
  - Try to apply TM fix instead of lock based
  - Come up with Recipes of fixes
- Ingredients :
  - Atomic regions
  - Preemptible resources
  - SW Rollback
  - Atomic/Lock serialization

Bug Fix Recipes

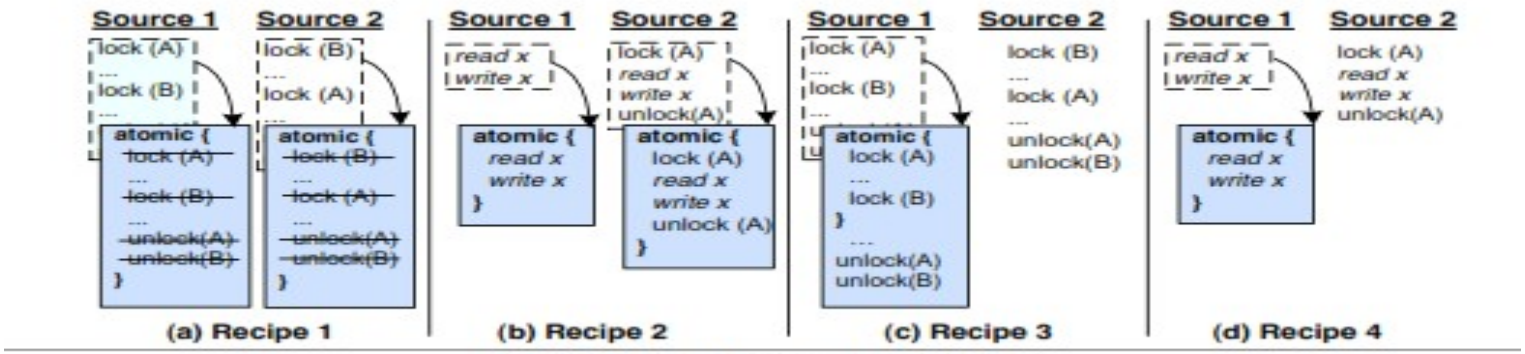


Figure 1. Example uses of our fix recipes.

- Recipes
  - Replace Deadlock-prone locks
  - Wrap all
  - Asymmetric Deadlock Preemption
  - Wrap Unprotected

Bug fix summary

Bug	App	All	Transactional Memory Fixes				
			Total	R1	R2	R3	R4
DL	Mozilla	13	9	8(2)	-	7(1)	-
	Apache	4	2	1(1)	-	1(1)	-
	MySQL	5	1	0	-	1(1)	-
AV	Mozilla	25	20	-	20(12)	-	8(0)
	Apache	7	5	-	5(3)	-	2(0)
	MySQL	6	6	-	6(2)	-	4(0)
	Total	60	43	9(2)	31(17)	9(3)	14(0)

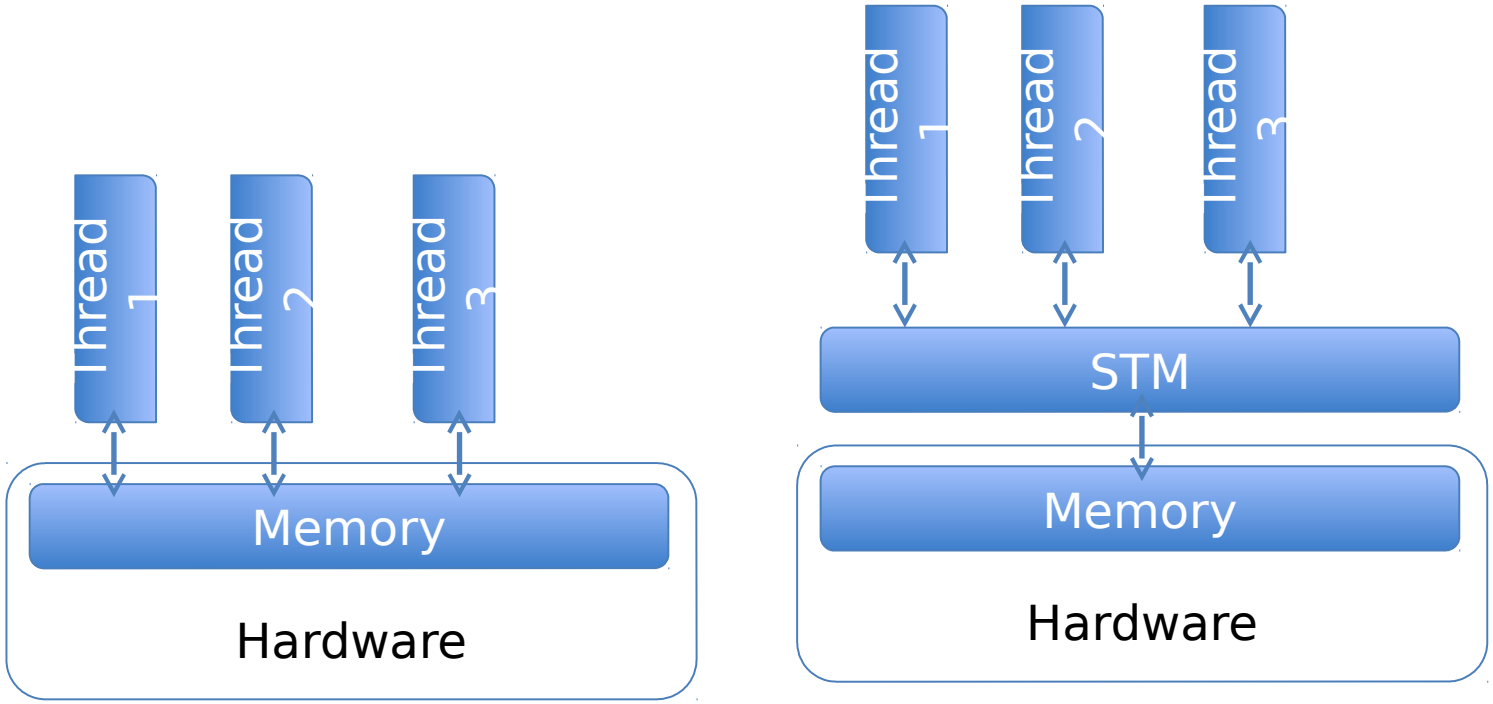
Bug ID	Cause	Characteristics	Fix	Perf.	Lines of Code	
					Dev	TM
Mozilla-I	DL	Involves locks only	1 3	21% 85%	23 23	1039 16
Apache-I	DL	Involves lock and wait	3	78%	32	14
Apache-II	AV	Complete missing synchronization	2	96.5%	20	5
MySQL-I	AV	Partial missing synchronization	4	50%	103	4

Table 4. Bugs and corresponding fix recipes applied for demonstration purposes. Performance is relative to that of developer's fix. Lines of code (LOC) includes both lines added and lines modified.

- TM fixes usually easier
- TM can't fix all bugs
- Locks better in some cases
- R3 and R4 more widely applicable

From Using to Implementing TM

- How is it implemented ?
  - Hardware (HTM)
  - Software (STM)



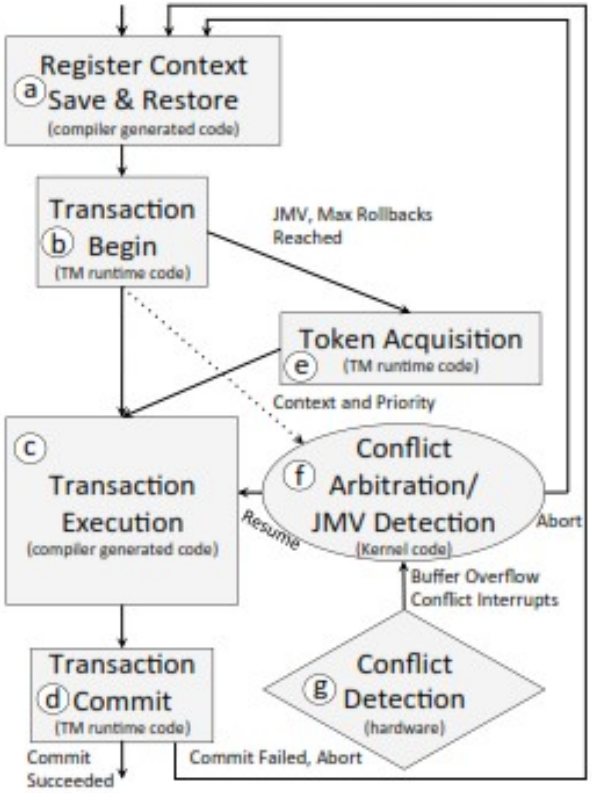
Hardware Transactional Memories

- Save architectural state to ‘checkpoint’
- Use caches to do versioning for memory
- Updates to coherency protocol
- Conflict detection in hardware
- ‘Commit’ transactions if no conflict
- ‘Abort’ transactions if conflict (or special cond)
- ‘Retry’ aborted transaction

BlueGene/Q : Hardware TM

- 16 core with 4 SMT with 32MB shared L2
- Multi-versioned L2 cache
- 128 speculative IDs for versioning
- L1 speculative writes invisible to other threads
  - Short running mode (L1-bypass)
  - Long running mode (TLB-Aliasing)
- Upto 10 speculative ways guaranteed in L2

Execution of a transaction



- Spcial cases :
  - Irrevocable mode – for forward progress
  - JMV example – MMIO
  - Actions on commit fail
  - Handling problematic transaction – single rollback

HTM vs. STM

Hardware	Software
Fast (due to hardware operations)	Slow (due to software validation/commit)
Light code instrumentation	Heavy code instrumentation
HW buffers keep amount of metadata low	Lots of metadata
No need of a middleware	Runtime library needed
Only short transactions allowed (why?)	Large transactions possible

How would you get the best of both?

Hybrid-TM

- Best-effort HTM (use STM for long trx)
- Possible conflicts between HW,SW and HW-SW Trx
  - What kind of conflicts do SW-Trx care about?
  - What kind of conflicts do HW-Trx care about?
- Some initial proposals:
  - HyTM: uses an ownership record per memory location (overhead?)
  - PhTM: HTM-only or (heavy) STM-only, low instrumentation

Hybrid NOrec

- Builds upon NOrec (no fine-grained shared metadata, only one global sequence lock)

- HW-Trx must wait SW-Trx writeback
- HW-Trx must notify SW-Trx of updates
- HW-Trx must be aborted by HW,SW-Trx

How to reduce conflicts?

```
SW_BEGIN
  snapshot = seqlock
  if (snapshot & 1)
    goto 16

SW_COMMIT
  if (writes.empty())
    return
  while (!CAS(&seqlock, snapshot,
      snapshot + 1))
    SW_VALIDATE
  foreach (addr, val) in writes
    *addr = val
  seqlock = seqlock + 1
  reads.reset(), writes.reset()
```

Instrumentation

- Subscription to SW commit notification
  - How about HW notification?
- Separation of subscribing and notifying
  - How about HW-Trx conflicts?
- Coordinate notification through HW-Trx
  - How about validation overhead?

```
HW_POST_BEGIN
  if (seqlock & 1)
    while (true) // await abort

HW_PRE_COMMIT
  seqlock = seqlock + 2
```

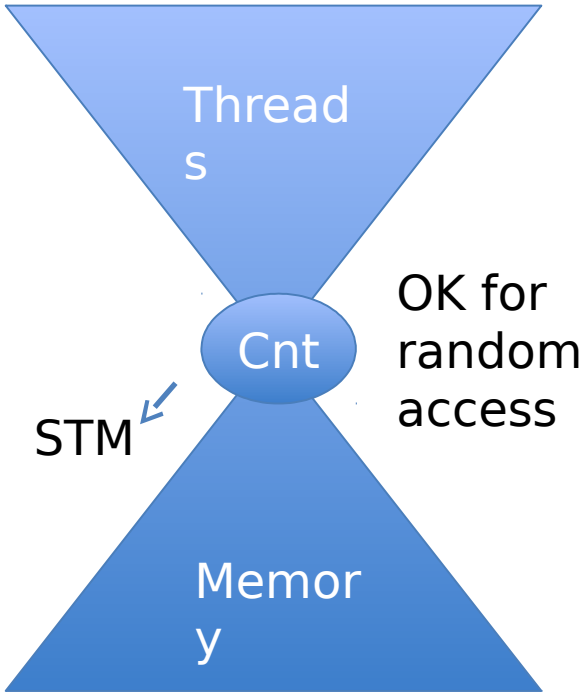
```
HW_POST_BEGIN
  if (seqlock & 1)
    while (true) // await abort

HW_PRE_COMMIT
  counter = counter + 1
```

```
HW_POST_BEGIN
  if (seqlock & 1)
    while (true) // await abort

HW_PRE_COMMIT
  counter[id] = counter[id] + 1
```

How to Avoid the Narrow Waist?

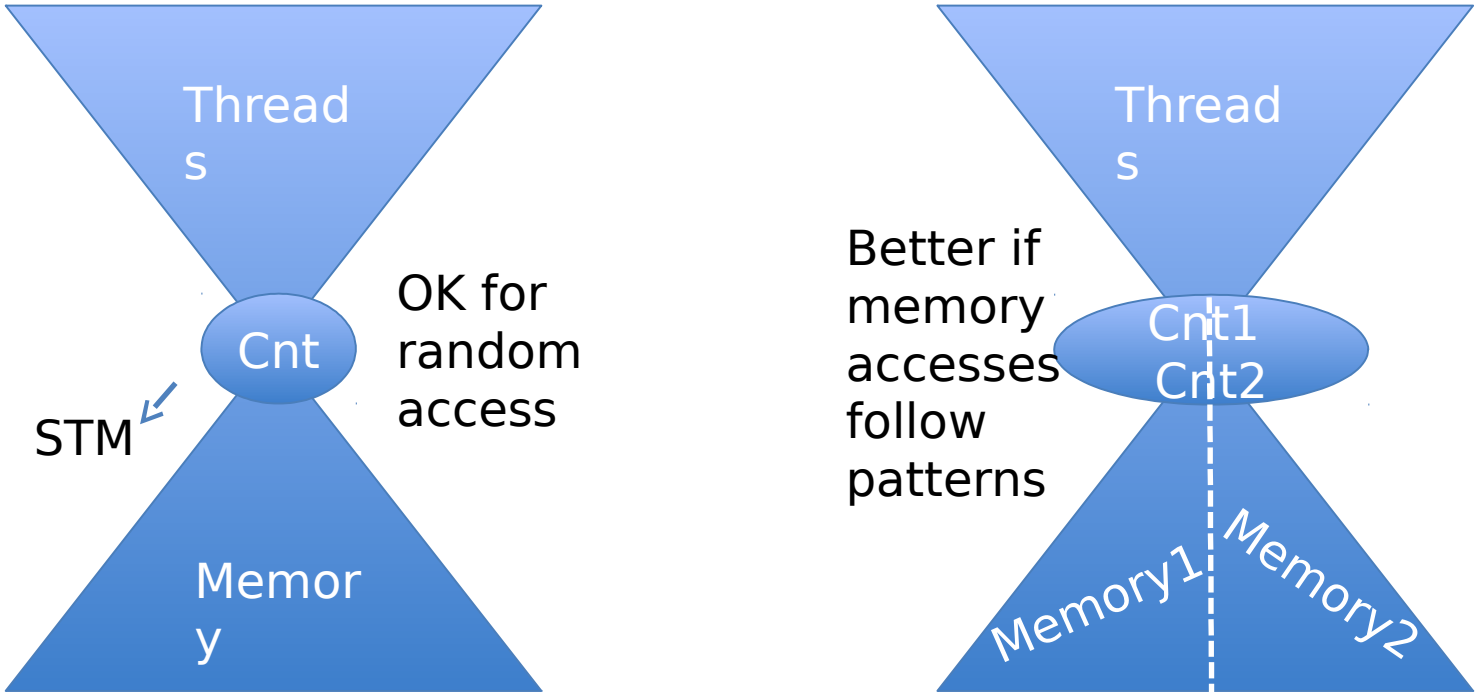



```
SW_BEGIN
  snapshot = seqlock
  if (snapshot & 1)
    goto 16
```

```
SW_COMMIT
  if (writes.empty())
    return
  while (!CAS(&seqlock, snapshot,
      snapshot + 1))
    SW_VALIDATE
  foreach (addr, val) in writes
    *addr = val
  seqlock = seqlock + 1
  reads.reset(), writes.reset()
```

- Update 1 variable atomically to access the whole memory
- Single counter, multiple threads/cores/processors
- Even worse in Norec, seqlock used for validation/lock

How to Avoid the Narrow Waist?



- Seqlock (or c-lock) used for serial order
- Update 1 variable atomically to access the whole memory
- Single counter, multiple threads/cores/processors

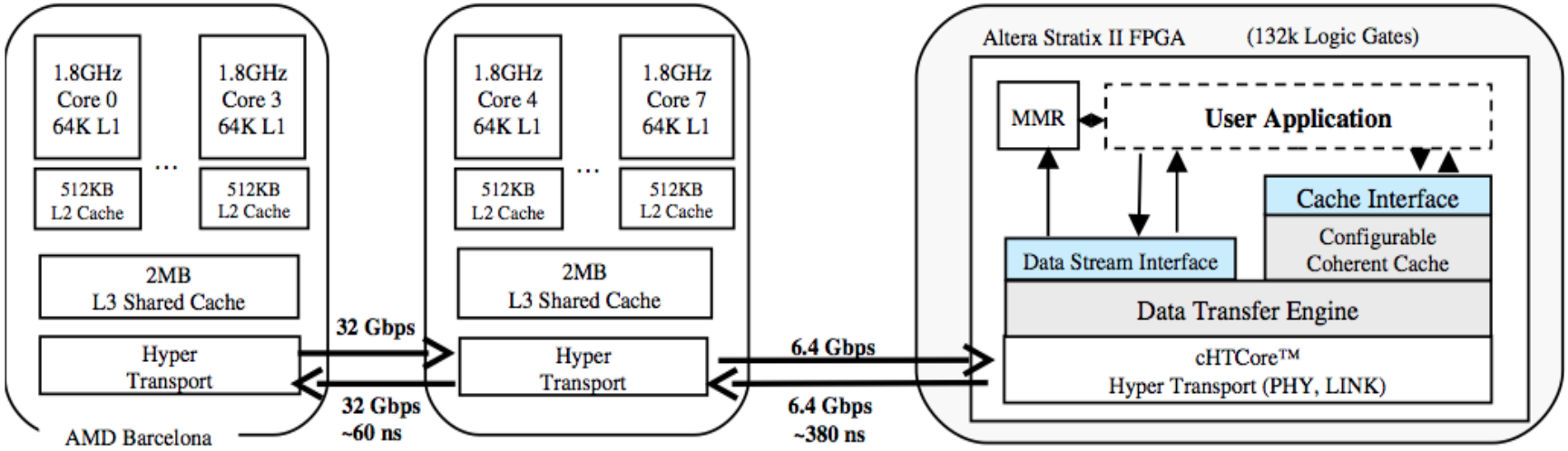
HTM vs. STM

Hardware	Software
Fast (due to hardware operations)	Slow (due to software validation/commit)
Light code instrumentation	Heavy code instrumentation
HW buffers keep amount of metadata low	Lots of metadata
No need of a middleware	Runtime library needed
Expensive to implement/change	Many versions currently available
Different support from different vendors	Flexible middleware
Only short transaction allowed	Large transactions possible

How would you get the best of both?

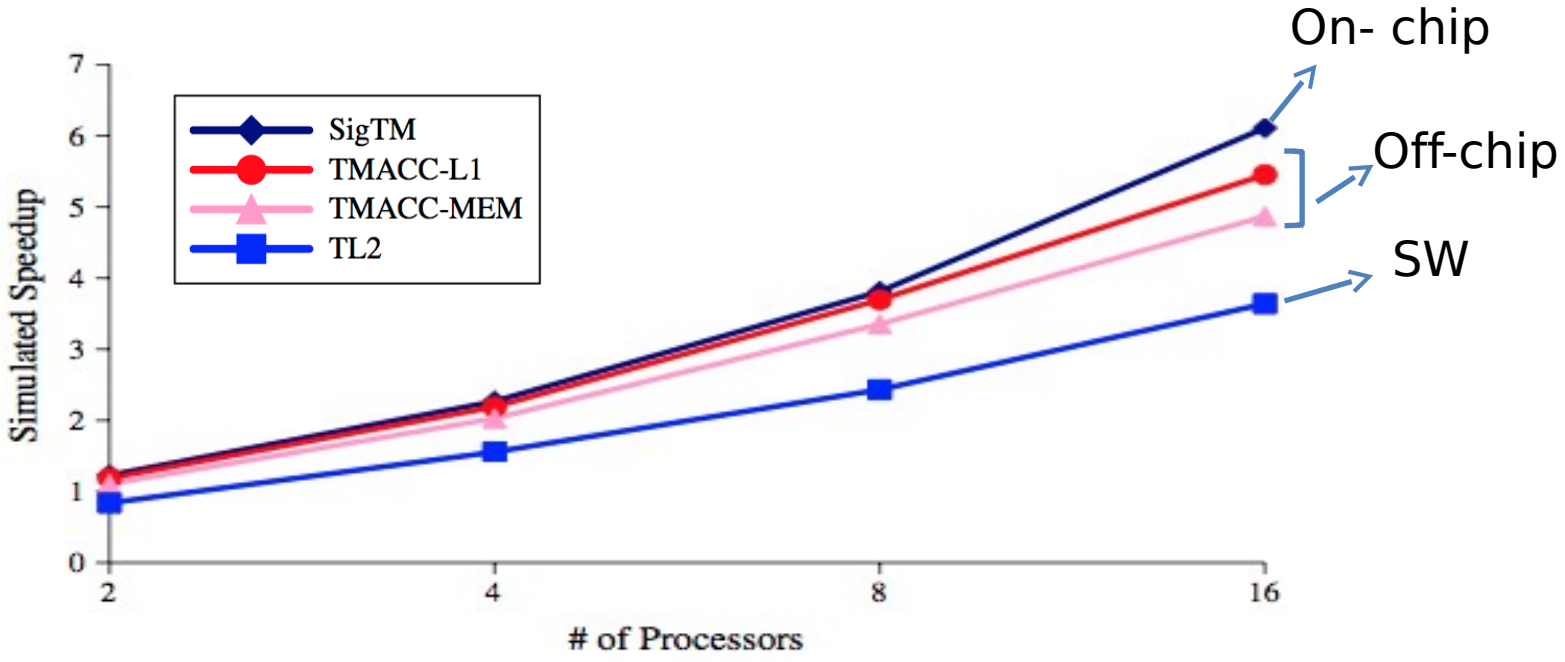
(HINT: current HW support implemented on processors, at the core of the platform, which means hard design)

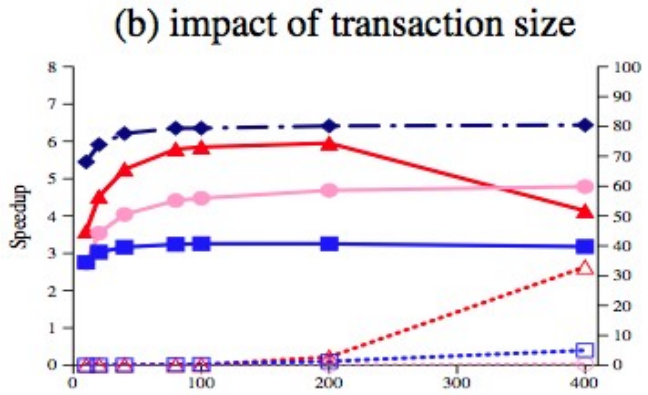
TMACC

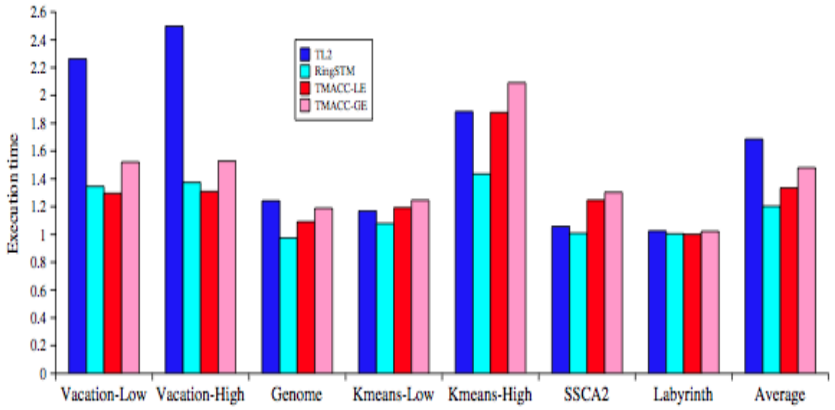


- *FARM*: FPGA coherently connected to 2 CPUs
- Mainly used for conflict detection
  (why not using it for operations on memory?)
- Asynch. Comm. with TMACC (possible? why is it good?)

TMACC Performance



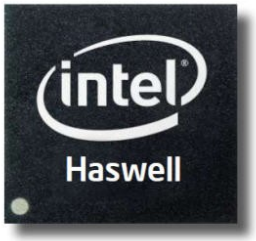




Thank you.

Hardware Transactional Memory

E.g., Intel Haswell Microarch., AMD Advanced Synch. Facility



- Natively support transactional operations
  - Versioning, conflict detection
- Use L1-cache to buffer read/write set
- Conflict detection through the existing coherency protocol
- Commit by checking the state of cache lines in read/write set

Hardware	Software
Fast (due to hardware operations)	Slow (due to software validation/commit)
Light code instrumentation	Heavy code instrumentation
HW buffers keep amount of metadata low	Lots of metadata (to keep consistent)
No need of a middleware	Runtime library needed