Transactional Memory

To read more...

This day's papers:

Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures"
McKenney et al, "Why The Grass May Not Be Greener On The Other Side: A Comparison of Locking vs. Transactional Memory"

Supplementary readings:

extended tech report version of Herlihy and Moss: `http://www.hpl.hp.com/techreports/Compaq-DEC/CRL-92-7.pdf` (includes more details generally, including extension to directory-based protocols)

Homework 2 questions?

From the paper reviews

Herlihy: benchmarks seemed very biased against locks

McKenney: where is quantitative data?

Can/How can locks and TM coexist?

Real-world implementations?

I/O, etc.

Herlihy benchmarks

very short critical sections

lots of contention

comparing against coarse-grained locking

didn't test priority inversion, etc. (motivations?)

Locks versus Transactions

	Locking		Transactional Memory	
Basic Idea	Allow only one thread at a time to access a given set of objects.		Cause a given operation over a set of objects to execute atomically.	
Scope	+	Idempotent and non-idempotent operations.	+	Idempotent and non-concurrent non-idempotent operations.
			⇓	Concurrent non-idempotent operations require hacks.
Composability	⇓	Limited by deadlock.	−	Limited by non-idempotent operations and by performance.
Scalability & Performance	−	Data must be partitionable to avoid lock contention.	−	Data must be partionable to avoid conflicts.
	⇓	Partioning must typically be fixed at design time.	+	Dynamic adjustment of partitioning carried out automatically for HTM.
			−	Static partitioning carried out automatically for STM.
	+	Contention effects are focused on acquisition and release, so that the critical section runs at full speed.	−	Contention effects can degrade the performance of processing within the transaction.
	+	Privatization operations are simple, intuitive, performant, and scalable.	−	Privatization either requires hardware support or incurs substantial performance and scalability penalties.
Hardware Support	+	Commodity hardware suffices.	−	New hardware required, otherwise performance is limited by STM.
	+	Performance is insensitive to cache-geometry details.	−	HTM performance depends critically on cache geometry.
Software Support	+	APIs exist, large body of code and experience, debuggers operate naturally.	−	APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.
Interaction With Other Synchronization Mechanism	+	Long experience of successful interaction.	⇓	Just beginning investigation of interaction.
Practical Applications	+	Yes.	+	Yes.
Wide Applicability	+	Yes.	−	Jury still out.

McKenney, Table 1

Locks versus Transactions [top]

	Locking		Transactional Memory	
Basic Idea	Allow only one thread at a time to access a given set of objects.		Cause a given operation over a set of objects to execute atomically.	
Scope	+	Idempotent and non-idempotent operations.	+	Idempotent and non-concurrent non-idempotent operations.
			⇓	Concurrent non-idempotent operations require hacks.
Composability	⇓	Limited by deadlock.	−	Limited by non-idempotent operations and by performance.
Scalability & Performance	−	Data must be partitionable to avoid lock contention.	−	Data must be partionable to avoid conflicts.
	⇓	Partioning must typically be fixed at design time.	+	Dynamic adjustment of partitioning carried out automatically for HTM.
			−	Static partitioning carried out automatically for STM.

Locks versus Transactions [bottom]

	+	Contention effects are focused on acquisition and release, so that the critical section runs at full speed.	−	Contention effects can degrade the performance of processing within the transaction.
	+	Privatization operations are simple, intuitive, performant, and scalable.	−	Privatization either requires hardware support or incurs substantial performance and scalability penalties.
Hardware Support	+	Commodity hardware suffices.	−	New hardware required, otherwise performance is limited by STM.
	+	Performance is insensitive to cache-geometry details.	−	HTM performance depends critically on cache geometry.
Software Support	+	APIs exist, large body of code and experience, debuggers operate naturally.	−	APIs emerging, little experience outside of DBMS, breakpoints mid-transaction can be problematic.
Interaction With Other Synchronization Mechanism	+	Long experience of successful interaction.	⇓	Just beginning investigation of interaction.

McKenney, Table 1 (bottom)

Transaction properties

serializable — apparently one at a time

atomic — commits or aborts, nothing in between

Basic Herlihey and Moss interface

`LT` — load value as part of transaction

`ST` — store value as part of transaction

`COMMIT` — try to make changes

Commit semantics:

caller must retry transaction if it fails

aborts instead if conflicting changes happened to read or written values

Weird Herlihey and Moss operation

`VALIDATE` — is transaction likely to commit?

Is this necessary?

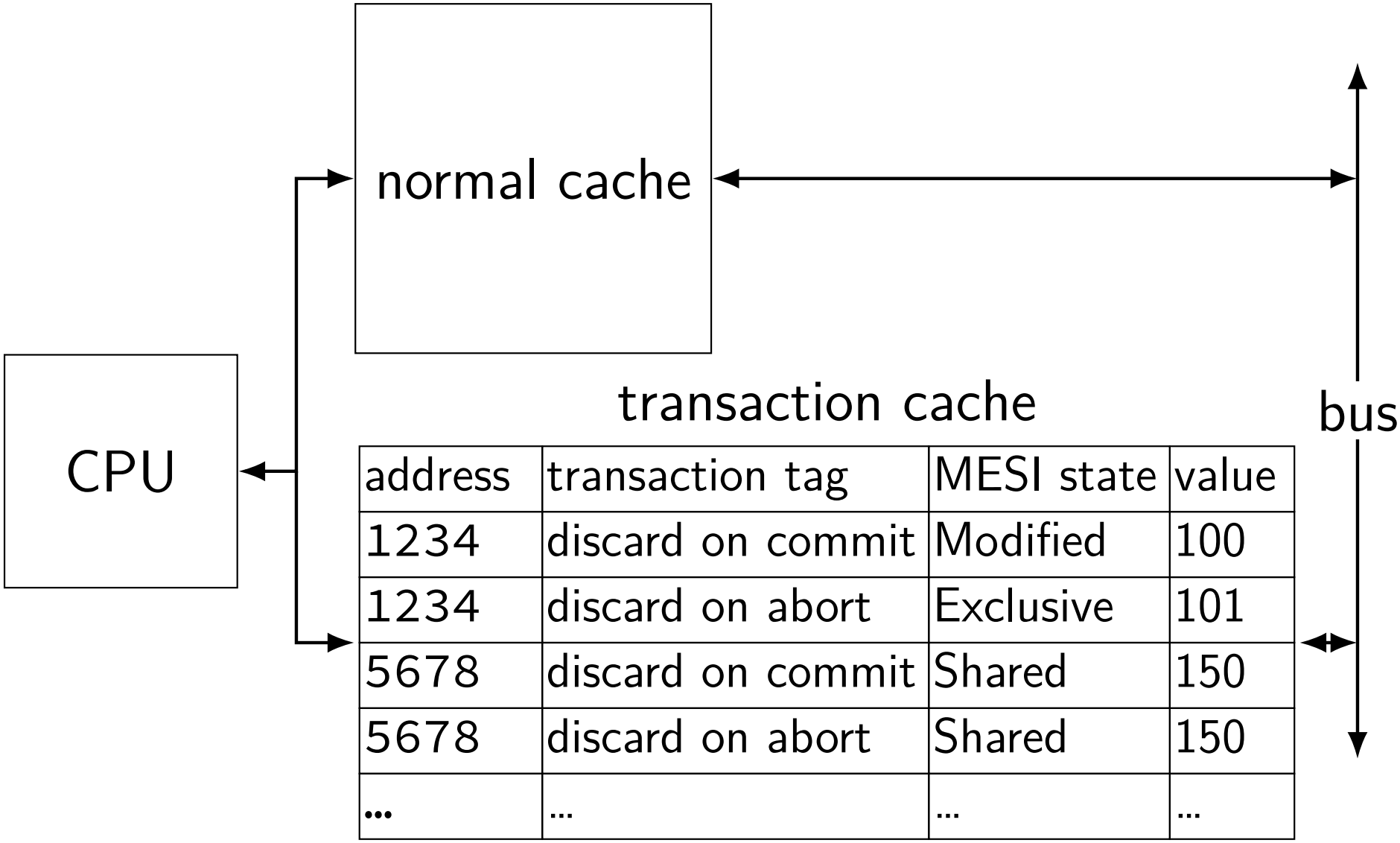
Extra Herlihey and Moss operations

I think these all just optimizations...

LTX — load with hint that we will write

ABORT — give up on transaction

the transaction cache



address	transaction tag	MESI state	value
1234	discard on commit	Modified	100
1234	discard on abort	Exclusive	101
5678	discard on commit	Shared	150
5678	discard on abort	Shared	150
...	...	...	...

the transcation cache

Extra cache — why?

- additional logic for transaction commit/abort
- fully-associativive — conflicts are worse than usual

Also acts as normal cache — analogy to Jouppi's victim cache

- ... but only stores things that were part of transactions

transcation cache tags

Normal	not part of pending transaction
Discard on Commit	pre-transaction version
Discard on Abort	transaction modified verison
Invalid	

transcation cache

has transaction tags *and* MESI states!

during transaction — two copies of values

- before and after transaction version
- might have the only copy of both!

after transaction — acts like normal cache

- "normal" tag represents normally cached values
- also "discard on commit" if transcation cannot commit

TSTATUS

flag: Can we commit?

If true, `COMMIT` will commit transaction

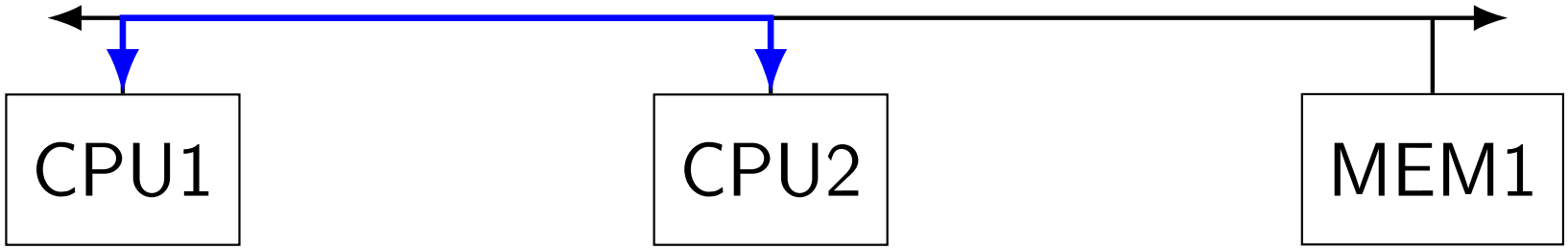
If false:

LT/LTX (reads) return "arbitrary value"

ST (writes) are discarded

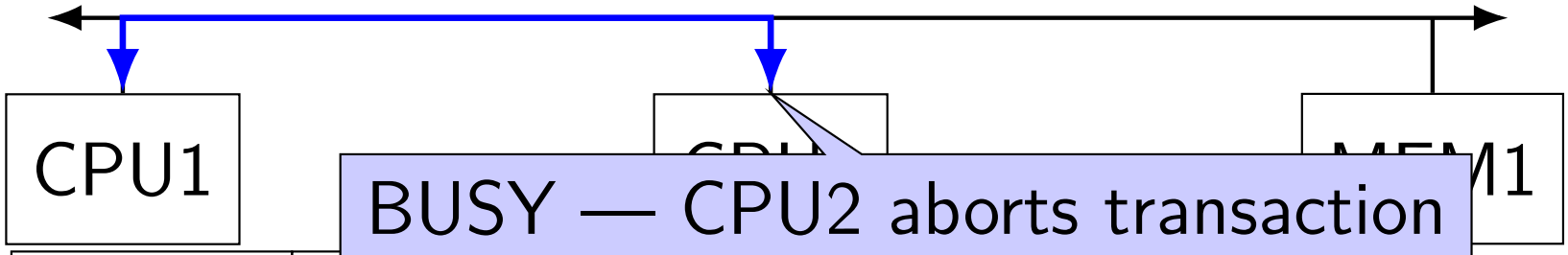
transaction can never commit

aborting a transaction



address	tag	state
0x100	Discard on Abort	Modified
0x100	Discard on Commit	Exclusive
0x101	Discard on Abort	Shared
0x101	Discard on Commit	Shared

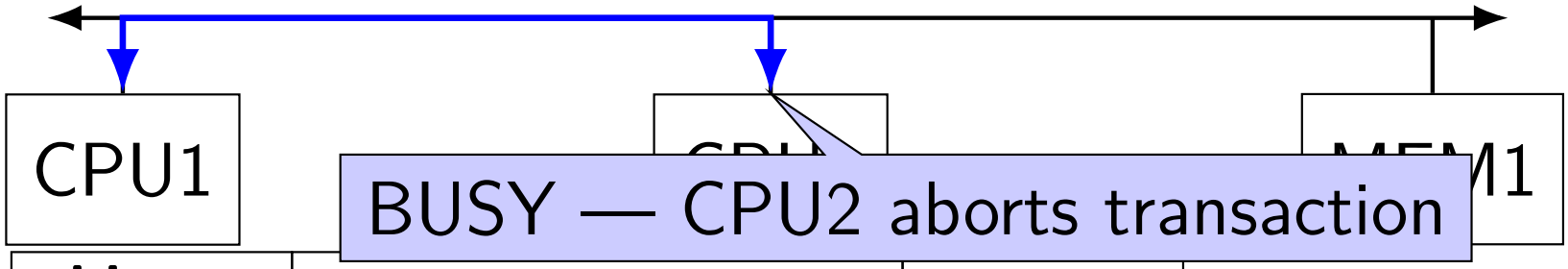
aborting a transaction



address	tag	state
0x100	Discard on Abort	Modified
0x100	Discard on Commit	Exclusive
0x101	Discard on Abort	Shared
0x101	Discard on Commit	Shared

CPU2: read for transaction 0x100
CPU1: it's busy!

aborting a transaction



address	tag	state
0x100	Discard on Abort	Modified
0x100	Discard on Commit	Exclusive
0x101	Discard on Abort	Shared
0x101	Discard on Commit	Shared

CPU2: read-to-own for transaction 0x101
CPU1: it's busy!

aborting a transaction (text)

bus read-for-ownership returns BUSY

- other transaction LT/LTX/ST same value
- other transaction might not commit

bus read (non-exclusive) returns BUSY

- other transaction LTX/ST same value
- other transactoin might not commit

VALIDATE

weird things happen during aborted transaction

VALIDATE tells us if this happened

needed to, e.g., not access invalid pointer:

```
while (TRUE) {
  old_tail = (entry*) LTX(&Tail);
  if (VALIDATE()) {
    ST(&new->prev, old_tail);
```

COMMIT and ABORT

local operations

cache checks “can I commit” flag

changes tags of transaction cache entries only

no gaurentee of progress

Thread 1	**Thread 2**	**Thread 3**
`t1 = LTX(a)`	`t2 = LTX(b)`	`t3 = LTX(c)`
`ST(b, t1)`		
aborts, restarts		
`t1 = LTX(a)`		
	`ST(c, t2)`	
	aborts, restarts	
		`ST(a, t3)`
		aborts, restarts
	`t2 = LTX(b)`	`t3 = LTX(c)`

transaction and non-transaction

"For brevity, we have chosen not to specify how transcational and non-transactional operations interact when applied concurrently to the same location"

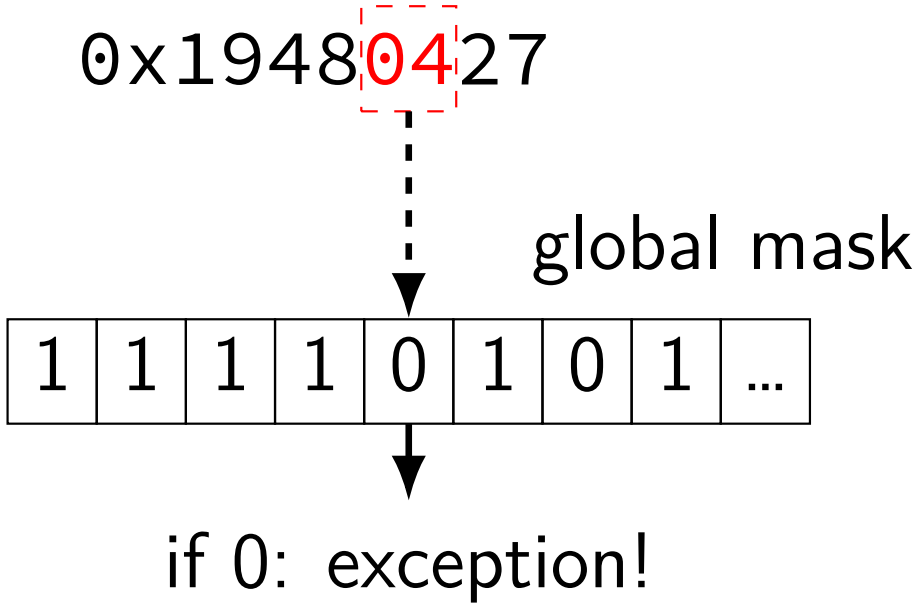
costs of transaction support

extra fully associative cache

- alternative: extra state bits on existing cache
- ... but what about conflicts?
- ... how much extra state??

larger transcations: bigger extra cache/state

transaction overflow: one idea



Exception handler:
Acquire lock for index `0x04` (or ABORT)
Record new/old value in local memory
Update value, release lock on COMMIT/ABORT
Return from exception

costs of transaction conflict

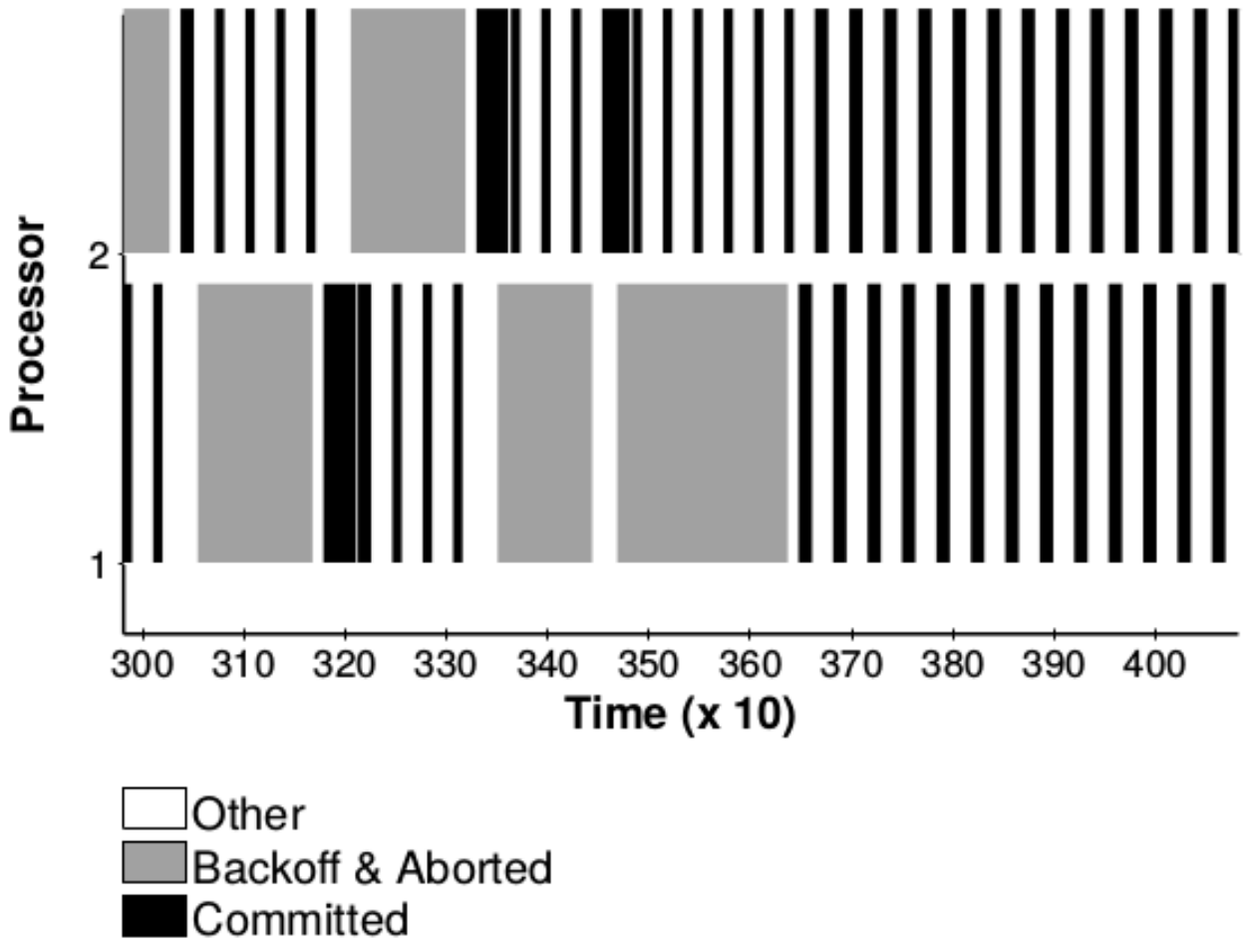


Figure 27: Two-Processor Time Line: Transactional Memory

costs of transaction conflict

extra work — bus traffic reading/invalidating

extra work — time to abort

locks would delay instead

transaction/lock iteraction option

non-transaction reads/writes abort transaction

... if transcation is also writing/reading it

... including to locks

real transcations

Intel TSX (recent Intel x86 chips):

- Restricted Transactional Memory (RTM)
- Hardware Lock Ellision (HLE)

IBM POWER8+

IBM System z (successor to S/370 — mainframes)

Restricted Transactional Memory

Intel real transactional memory suppport:

`XBEGIN abortDest`, `XEND` — mark transaction

`XABORT` — explicit abort

jump to `abortDest` if aborted (no validate)

abort discards all memory and register changes

size limits, I/O? transaction may always abort

Intel Hardware Lock Ellision

transactions for spin-locks only

`XACQUIRE`, `XRELEASE` — mark critical section

starts transaction reading lock only

ensure conflict with anything using lock normally

if aborted — run without transaction (modify lock)

backwards compatible!

Intel TSX Oops

Intel Disables TSX Instructions: Erratum Found in Haswell, Haswell-E/EP, Broadwell-Y

by Ian Cutress on August 12, 2014 8:20 PM EST

> **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 (Transactional Synchronization Extensions) 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*.

Other HTM implementations

generally require software fallback code using locks

common case — lock ellision

IBM POWER8 — transaction suspend/resume

- allow system calls/page faults/debugging during transaction
- context switch/etc.? transaction aborts on resume
- also assists software speculation

HTM limits

Intel Haswell

- 4 MB read set
- 22 KB write set

IBM POWER8

- 8 KB read set
- 8 KB write set

Nakaike et al, "Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8", ISCA'15

Next time: Cray-1 and GPUs

Cray-1 — vector processor

very wide registers

designed to optimize loops

programmable GPUs

prereq. to CUDA/etc. (next week)

designed to produce graphics

Graphics pipeline

part 1: list of triangles (vertices)

- figure out color/lighting
- adjust screen coordinates
- compute depth (to hide if object is in front)

part 2: fill triangles (fragment)

- compute pixels of triangle
- track depth of each pixel, replace only if closer
- based on settings of vertices (corners)

A User-Programmable Vertex Engine

Programmable vertex manipulation only

Seperate, very limited functionality fills in pixels

called fragment operations

... but based on colors, coordinates, etc. set by code

On Cray-1

paper spends a time on exchange registers, etc.

old alternative to virtual memory

not important for us

Logistics: Homework 3 Accounts?