Architectures for Transactional Memory

Austen McDonald

Our New MULTICORE Overlords

- The free lunch for software developers is over
  - No longer improving thread performance with new processors
- Chip Multiprocessors (CMP/Multicore) are here
  - Improve performance by exploiting thread parallelism

To make programs faster, **mortal** programmers will try parallel programming…

Parallel Programming is Hard

- Thread level parallelism is great until we want to share data
- Fundamentally, it's hard to work on shared data at the same time
  - so we don't—mutual exclusion via locks
- Locks have problems
  - performance/correctness, fine/coarse tradeoff
  - deadlocks and failure recovery

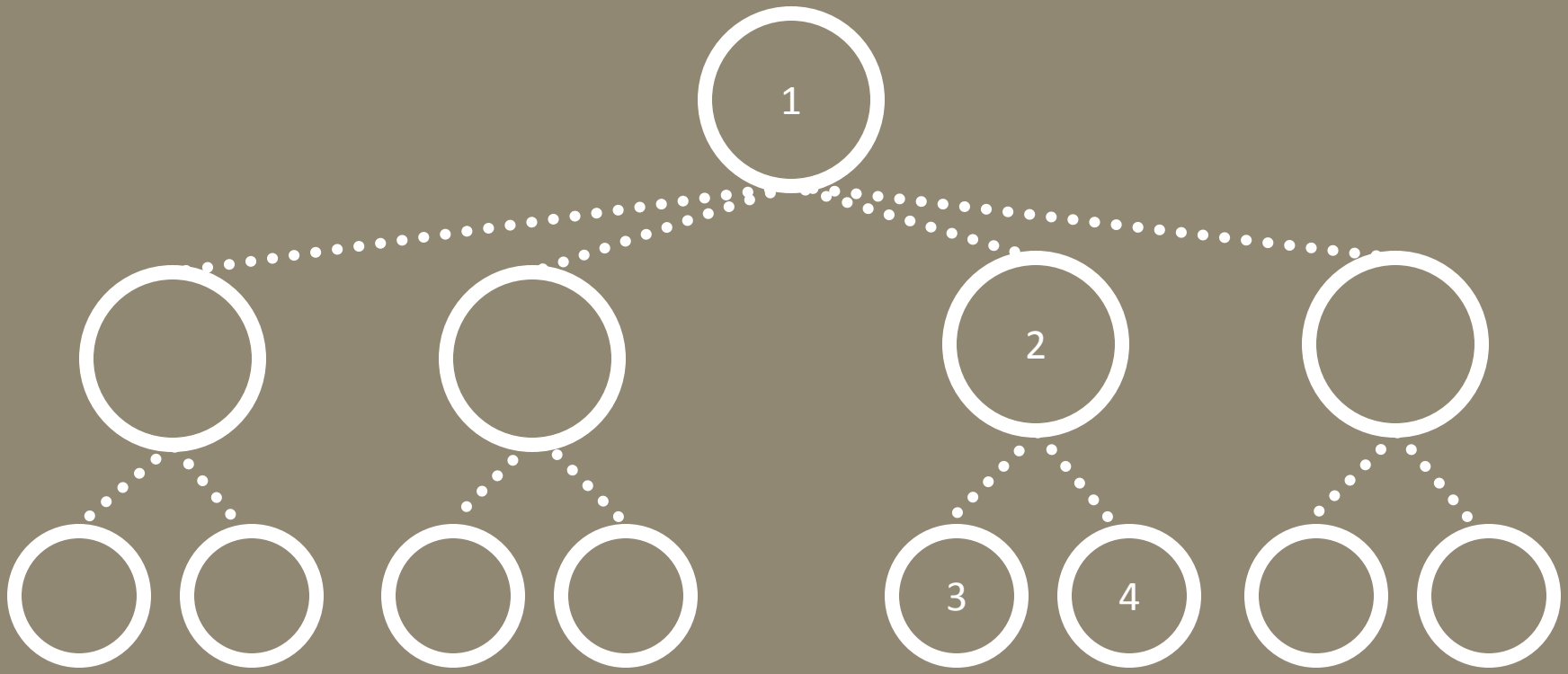
Transactional Memory (TM)

- Execute large, programmer-defined regions **atomically** and in **isolation** [Knight ’86, Herlihy & Moss ’93]

```
atomic {
    x = x + y;
}
```

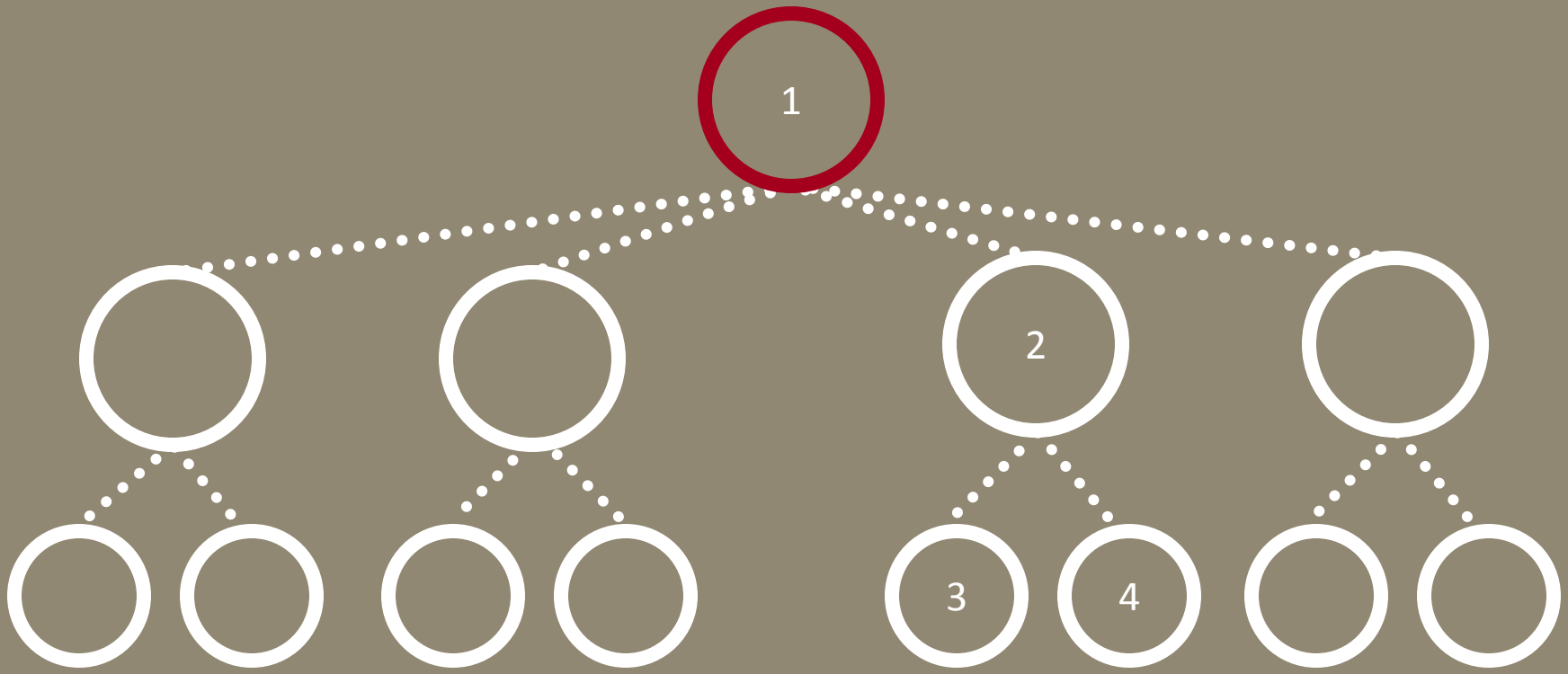
- Declarative
  - No management of locks
- Optimistically executing in parallel gains performance

TM Example

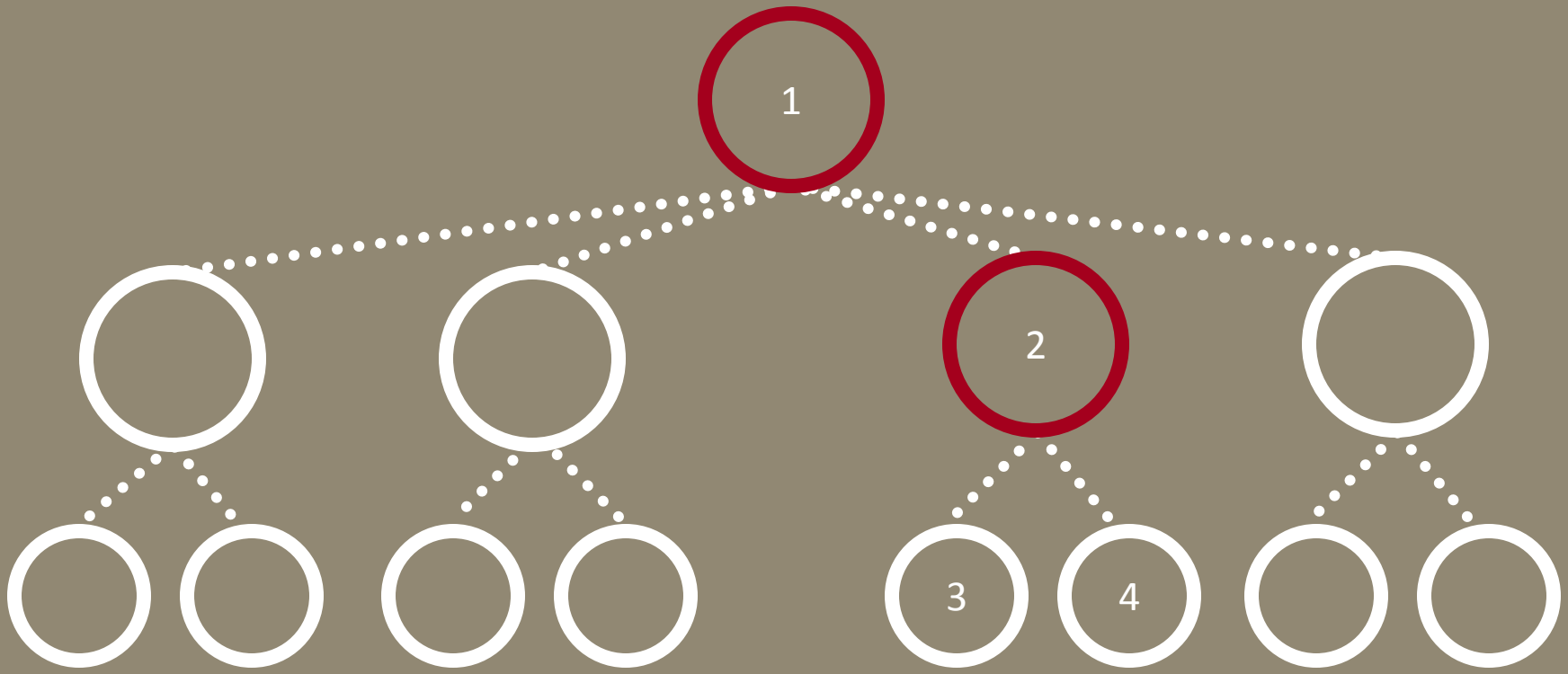


Goal: Modify node 3 in a thread-safe way.

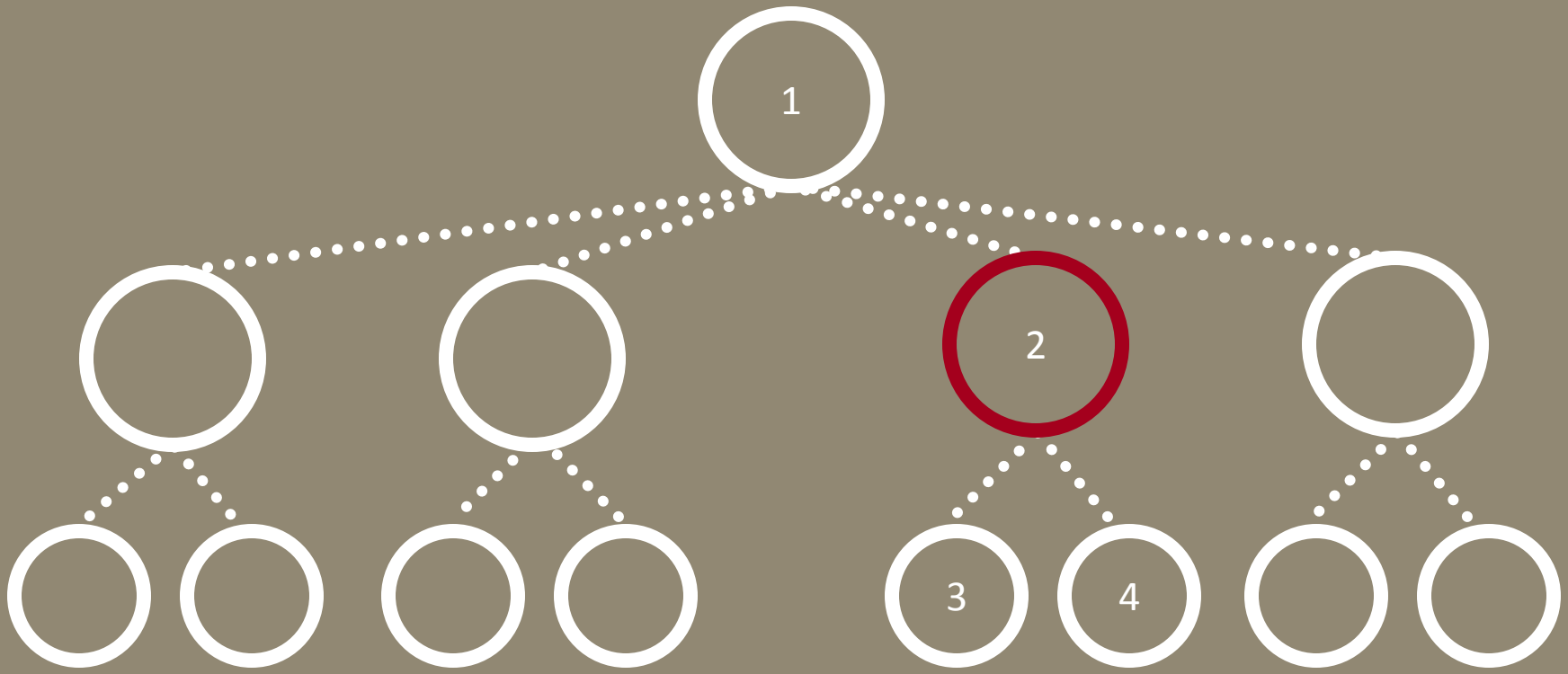
TM Example



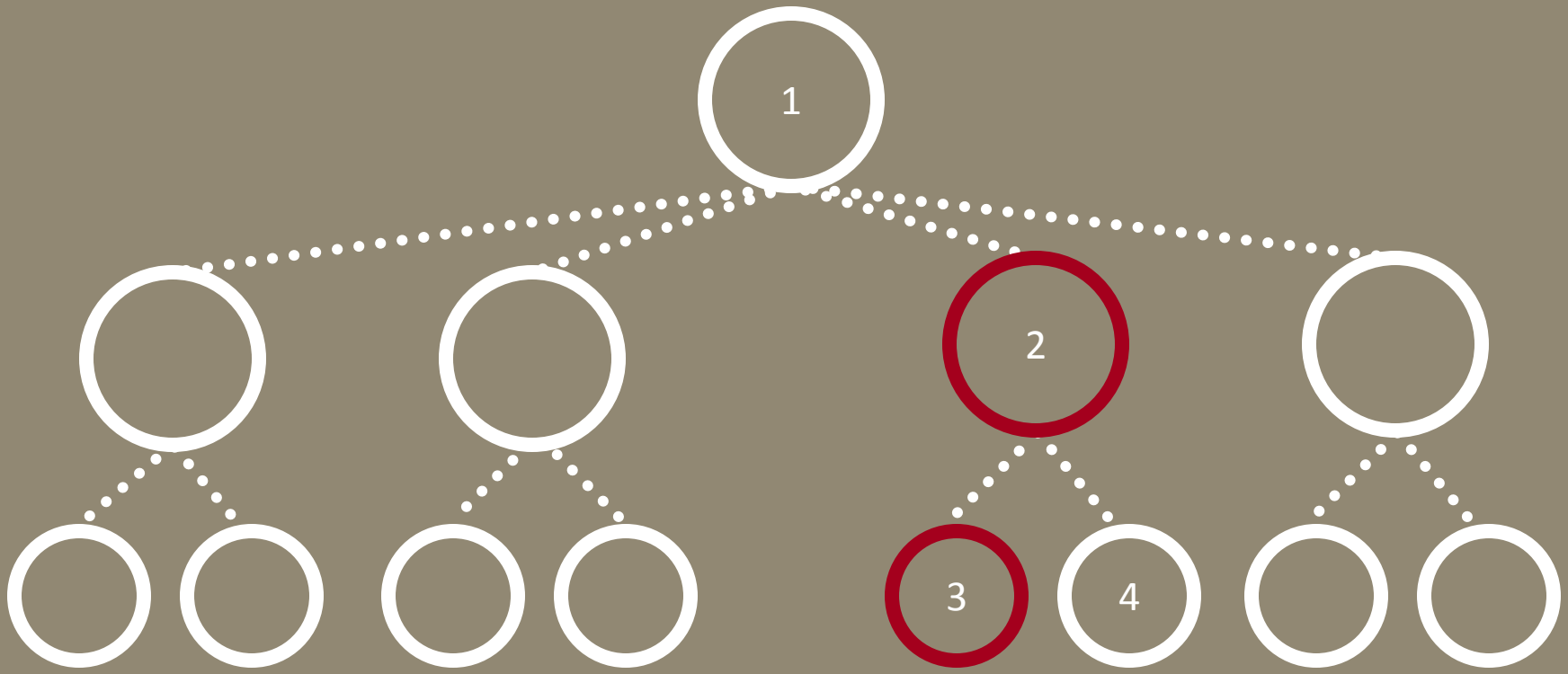
TM Example



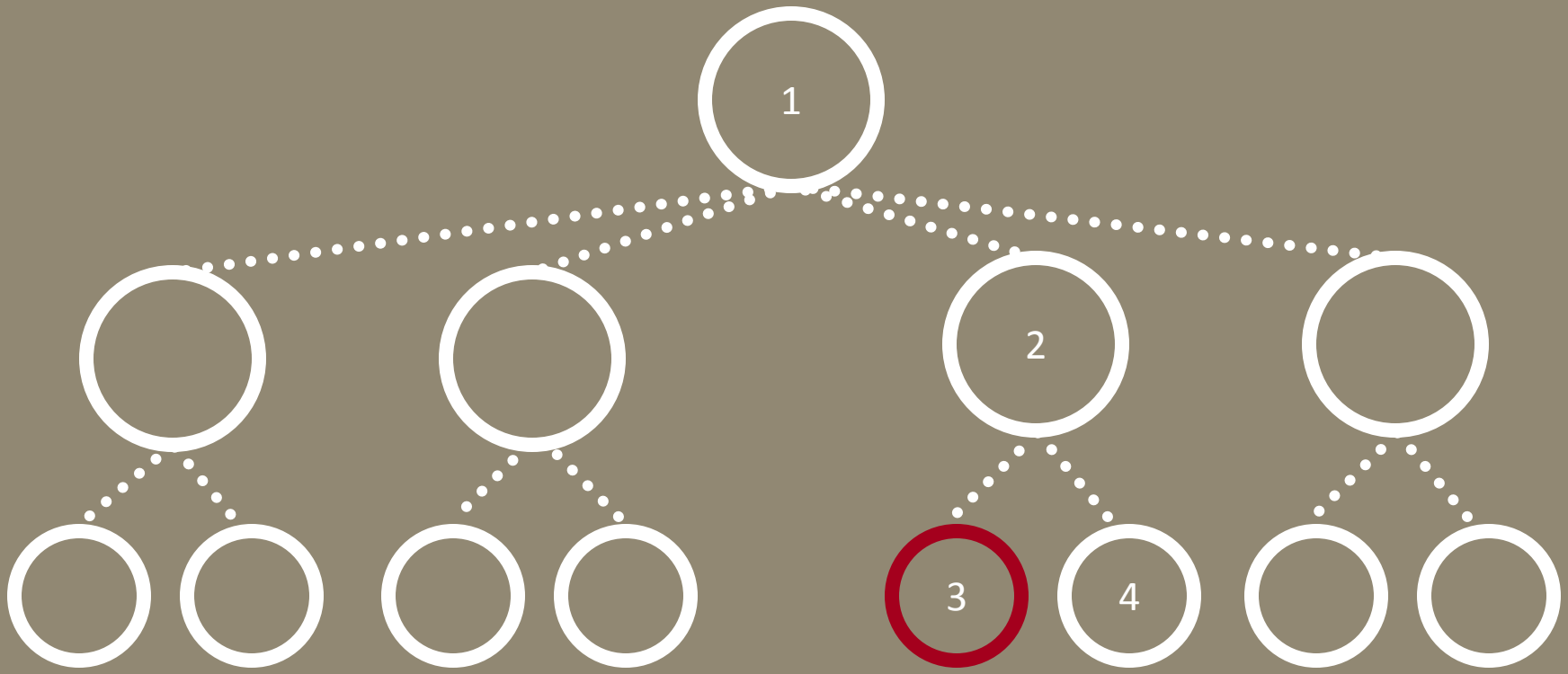
TM Example



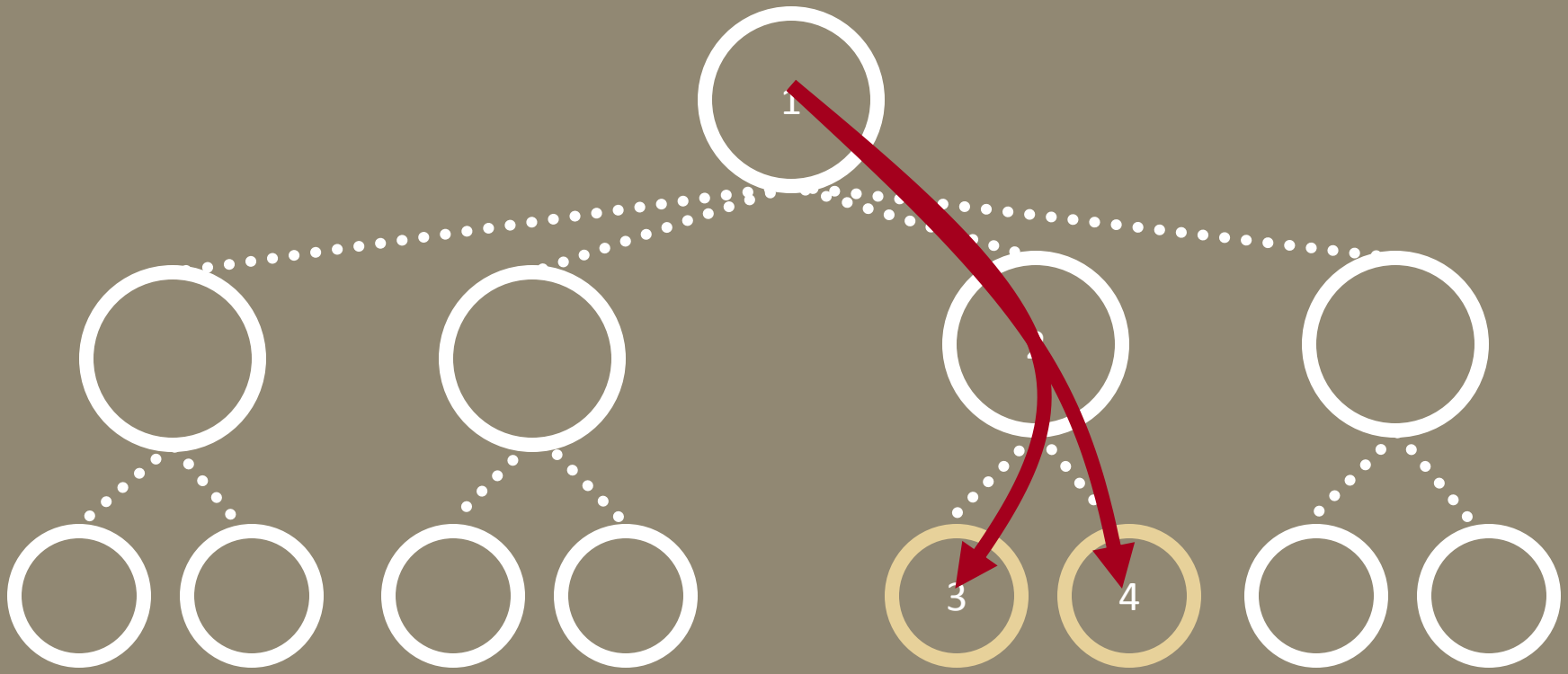
TM Example



TM Example



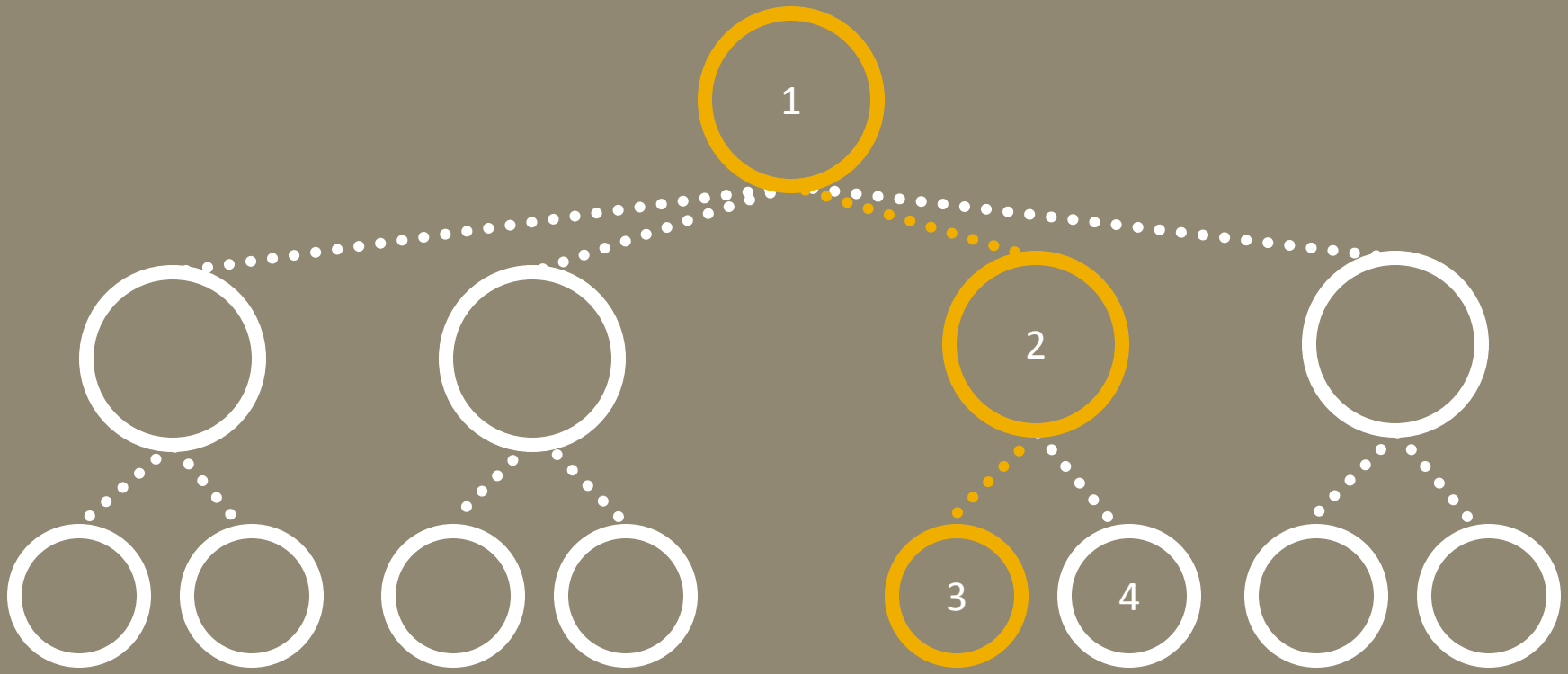
TM Example



Goals: Modify nodes **3 and 4** in a thread-safe way.

Locking prevents concurrency

TM Example



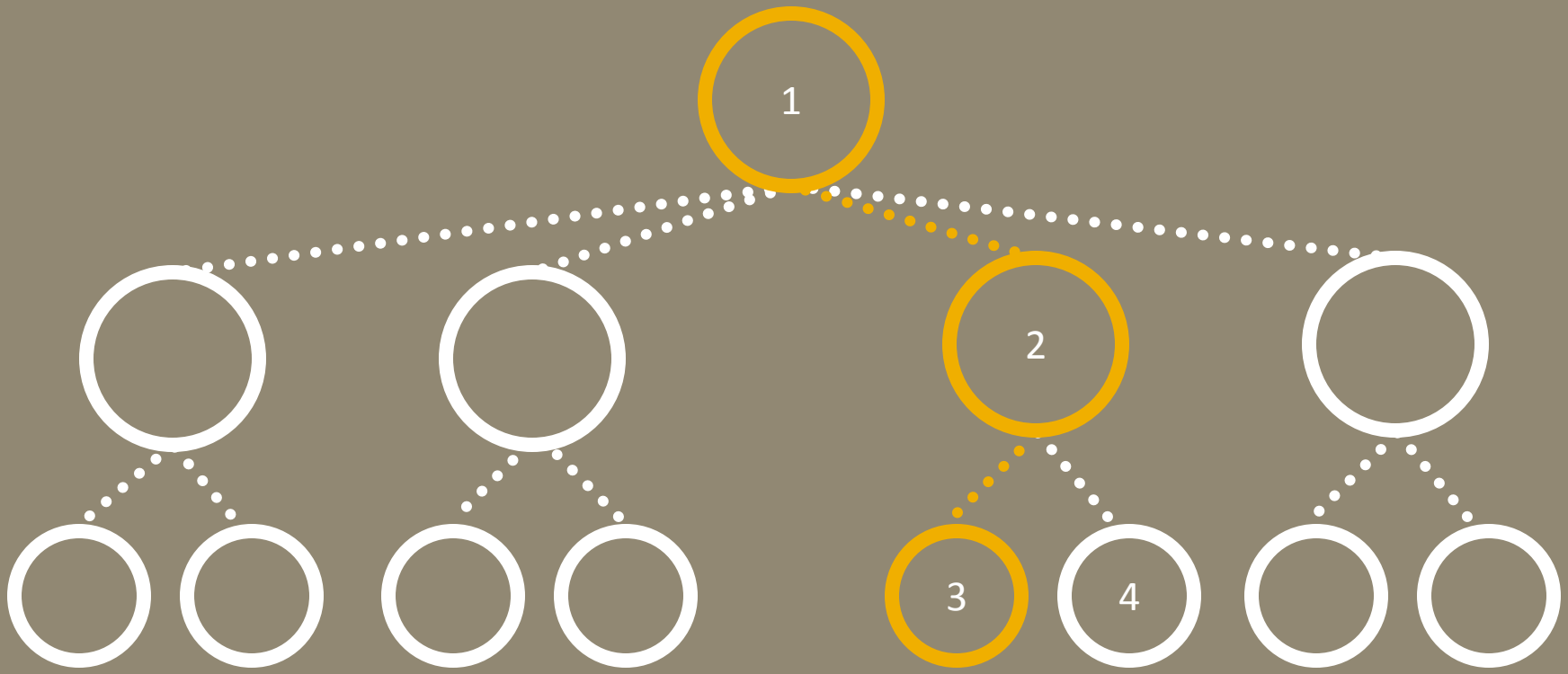
Transaction A

READ:

WRITE:

Goal: Modify node 3 in a thread-safe way.

TM Example

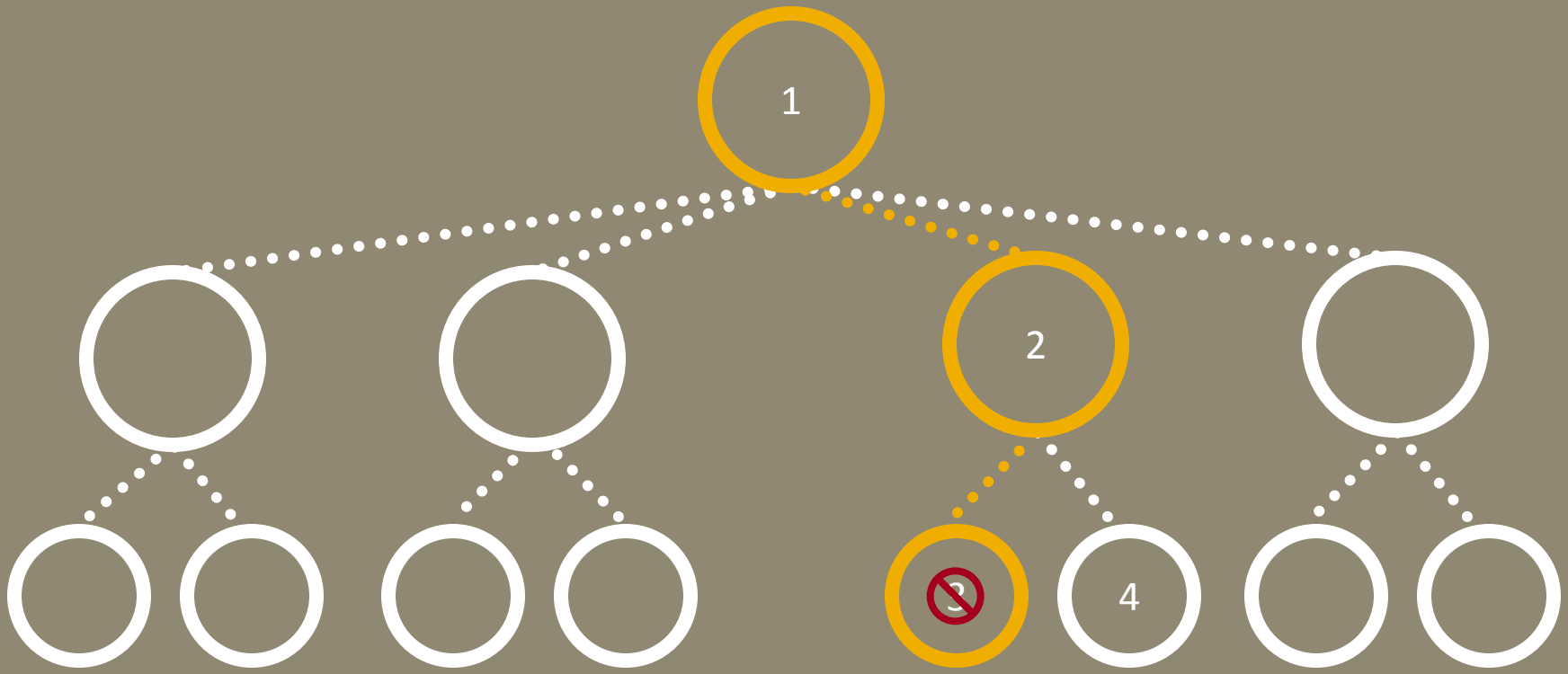


Transaction A

READ: 1, 2, 3

WRITE:

TM Example

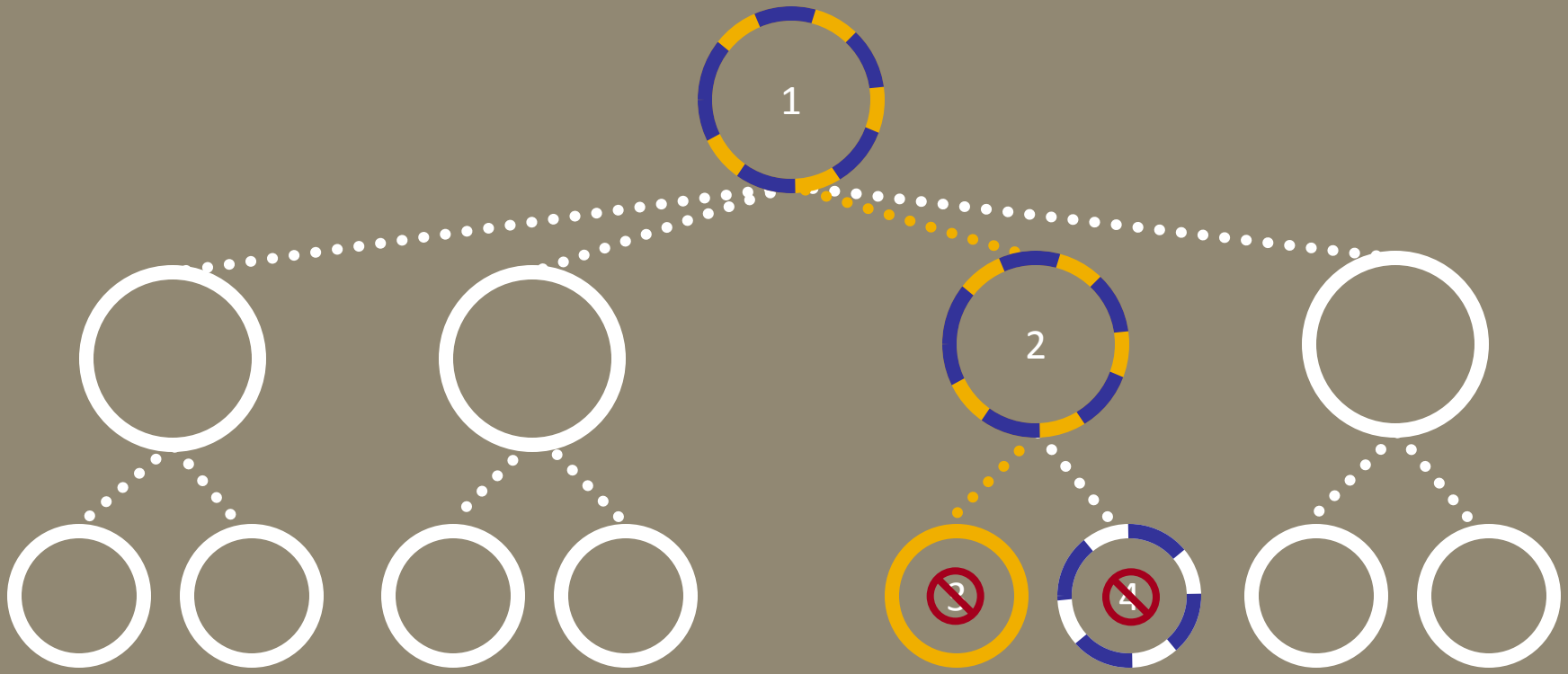


Transaction A

READ: 1, 2, 3

WRITE: 3

TM Example

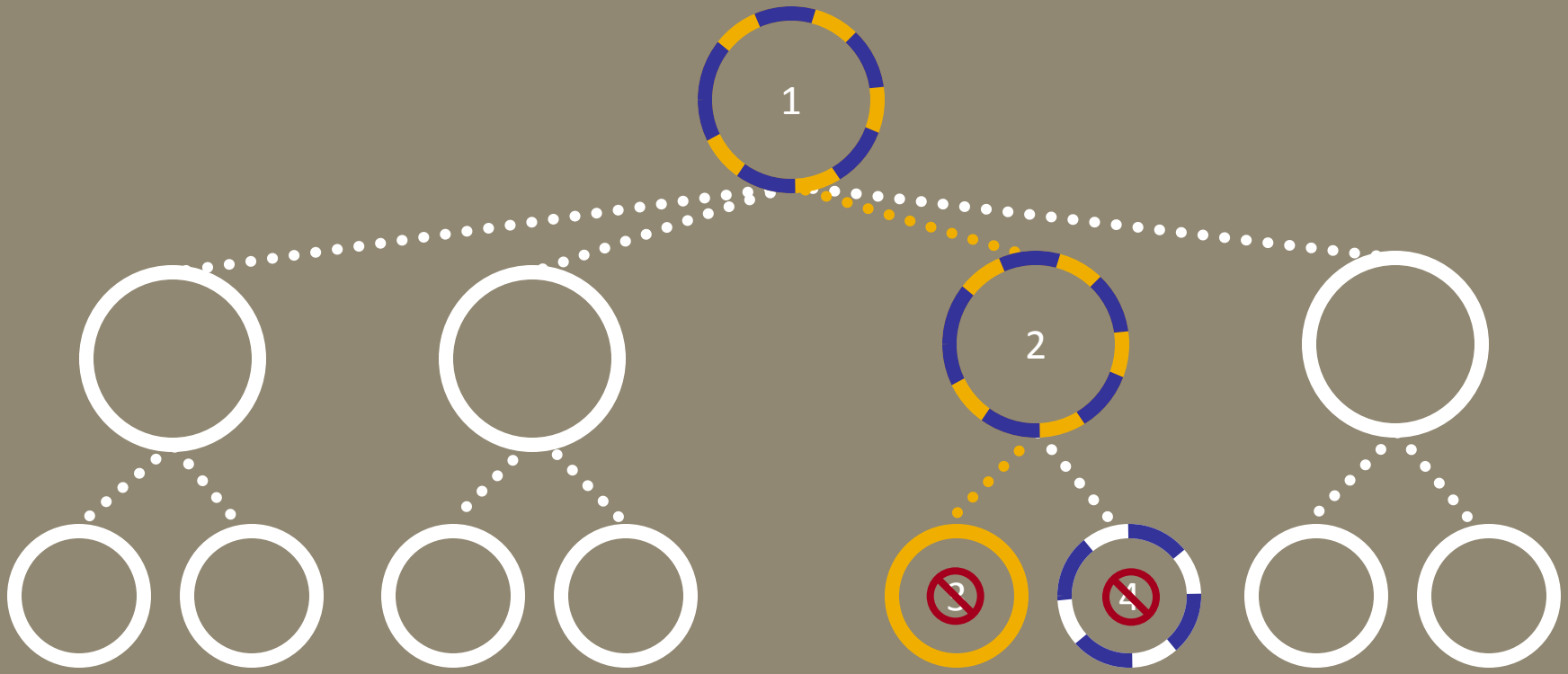


Transaction A
READ: 1, 2, 3
WRITE: 3

Transaction B
READ: 1, 2, 4
WRITE: 4

Goals: Modify nodes **3 and 4** in a thread-safe way.

TM Example



Transaction A

READ: 1, 2, 3

WRITE: 3

WW conflicts

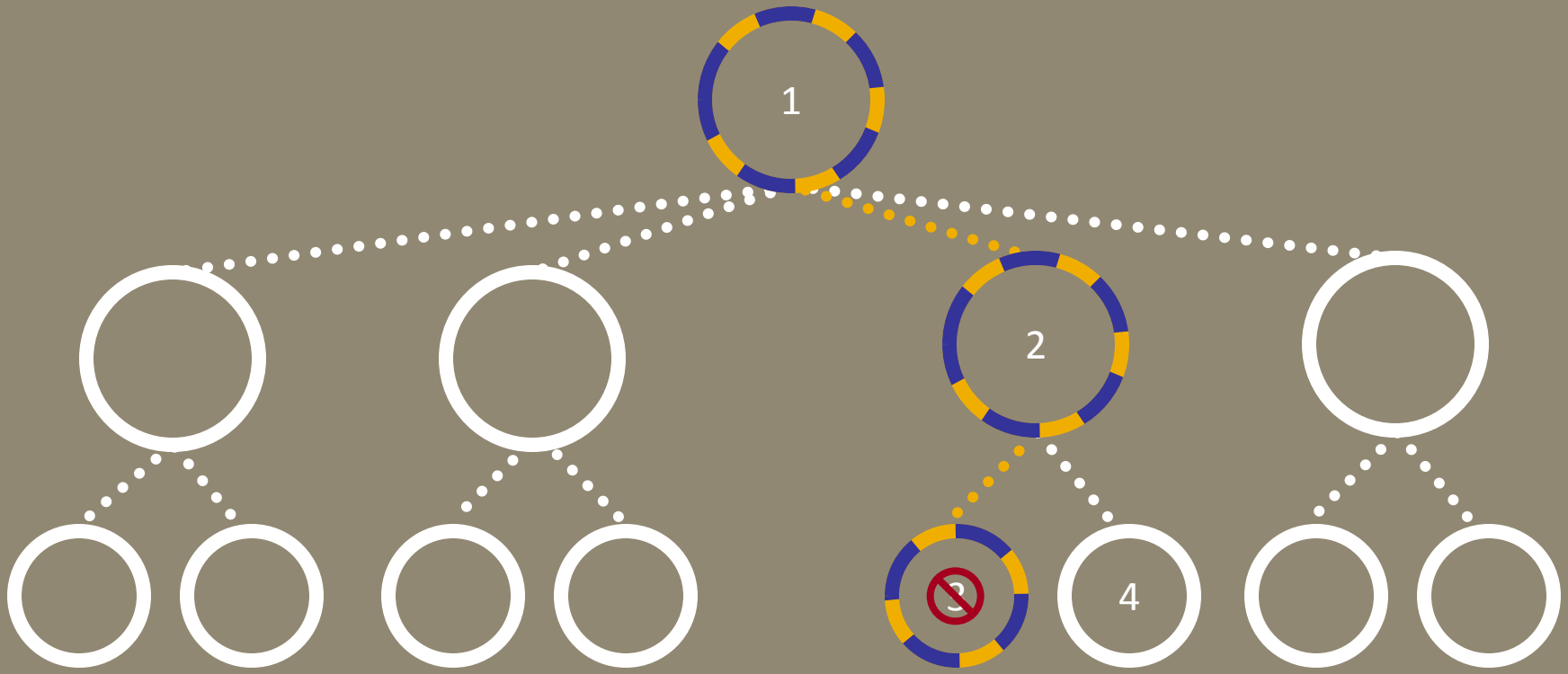
RW conflicts

Transaction B

READ: 1, 2, 4

WRITE: 4

TM Example



Transaction A

READ: 1, 2, 3

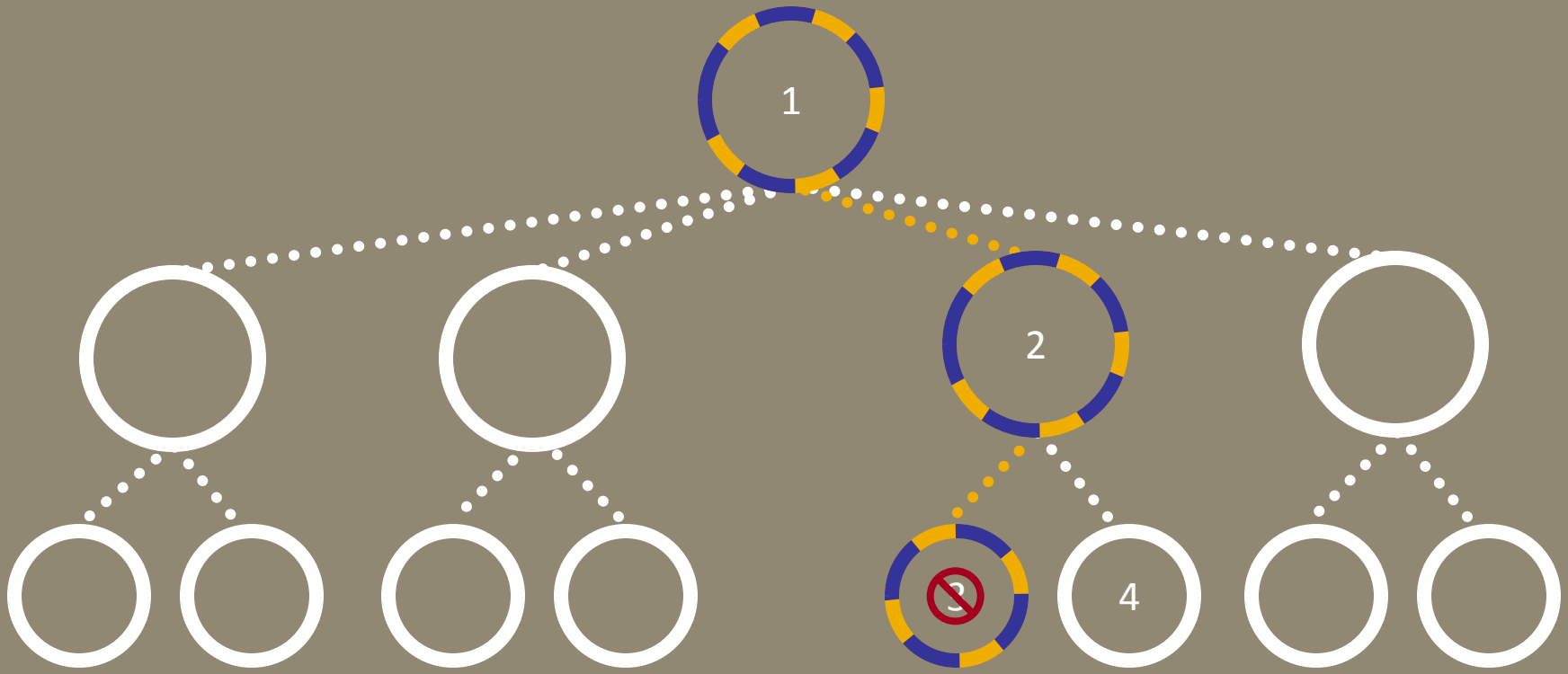
WRITE: 3

Transaction B

READ: 1, 2, 3

WRITE: 3

TM Example



Transaction A

READ: 1, 2, 3

WRITE: 3

WW conflicts

RW conflicts

Transaction B

READ: 1, 2, 3

WRITE: 3

Guts of TM

- To build TM, you need…

Versioning

```
atomic {
   x = x + y;
}
```

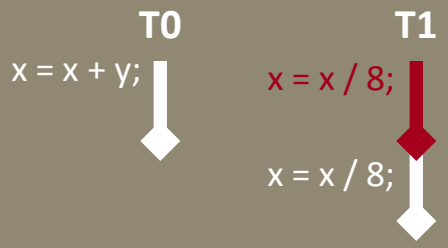
Where do you put the new x until commit?

Conflict Detection

```
T0                    T1
atomic {              atomic {
   x = x + y;            x = x / 8;
}                     }
```

How do you detect that reads/writes to x need to be serialized?

Conflict Resolution



How do you enforce serialization when required?

Hardware or Software TM?

- Can be implemented in HW or SW
- SW is slow
  - Bookkeeping is expensive: 2-8x slowdown
- SW has correctness pitfalls
  - Even for correctly synchronized code!

- Let's use hardware for TM

Challenges

1. What's the best implementation in hardware?
    - Many available options
2. What's the right HW/SW interface?
    - Changing software needs (OSs and Languages)
    - Changing parallel architectures

Contributions

- Designed and compared HTM systems
- Extended one system to replace coherence and consistency with only transactions
- Devised a sufficient software/hardware interface for current and future OS/PL on TM

5 Years of My Life on One Slide

1. Motivation & Contributions
2. Building a TM system in hardware
3. An architecture with only transactions
4. What about the interface to software?
5. Conclusions

Versioning

- Versioning: storing new values
- Eager: store new values in memory, old values in undo log
    - Commits fast, Aborts slow
- Lazy: store new values in writebuffer
    - Aborts fast, Commits slow

Conflict Detection

- Conflict Detection: detecting RW/WW conflicts
  - Pessimistic: detect conflicts on cache misses
    - Avoids useless work, but may cause deadlock/livelock and prevents some serializable schedules
  - Optimistic: wait until end of transaction
    - Forward progress can be guaranteed, but some wasted work [explain forward progress]

Versioning+Conflict Detection

- EP, LP, LO
  - Not Eager-Optimistic
- Note: conflict resolution depends on other two choices

Building a Lazy-Optimistic HTM

Lazy Versioning

- Need to keep new versions (and read-set tracking) until commit
- Already have a cache—let's put it there!

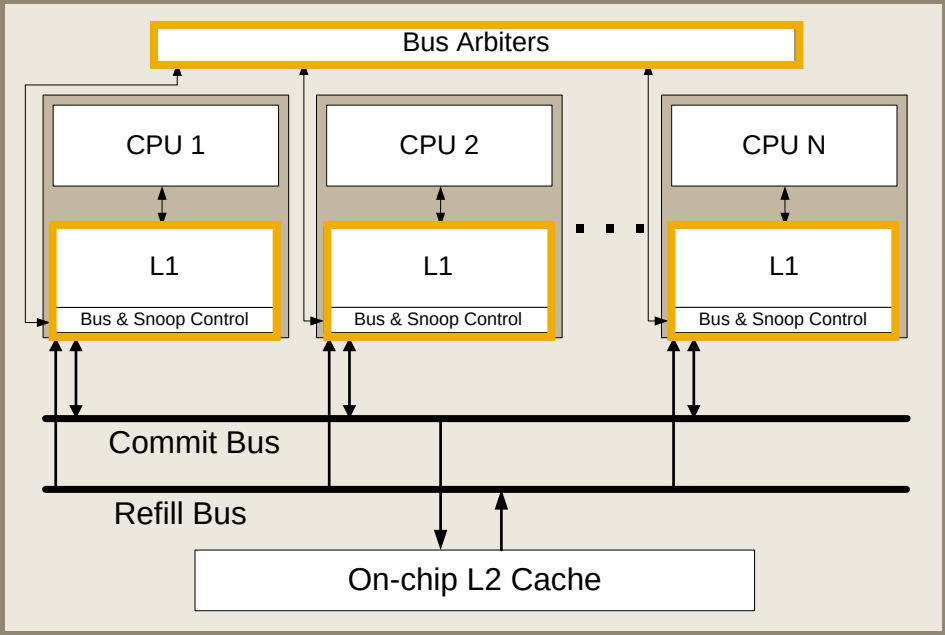
Optimistic Conflict Detection

- Need to detect conflicts at commit time
- Coherence protocol already detects sharing

Conflict Resolution

- The first committer wins
- Simple and guarantees forward progress
  Aggressive Conflict Resolution

LO HTM Specifics



Changes for TM

LO HTM Specifics

Read Bits:

```
ld 0xdeadbeef
```

Write Bits:

```
st 0xcafebabe
```

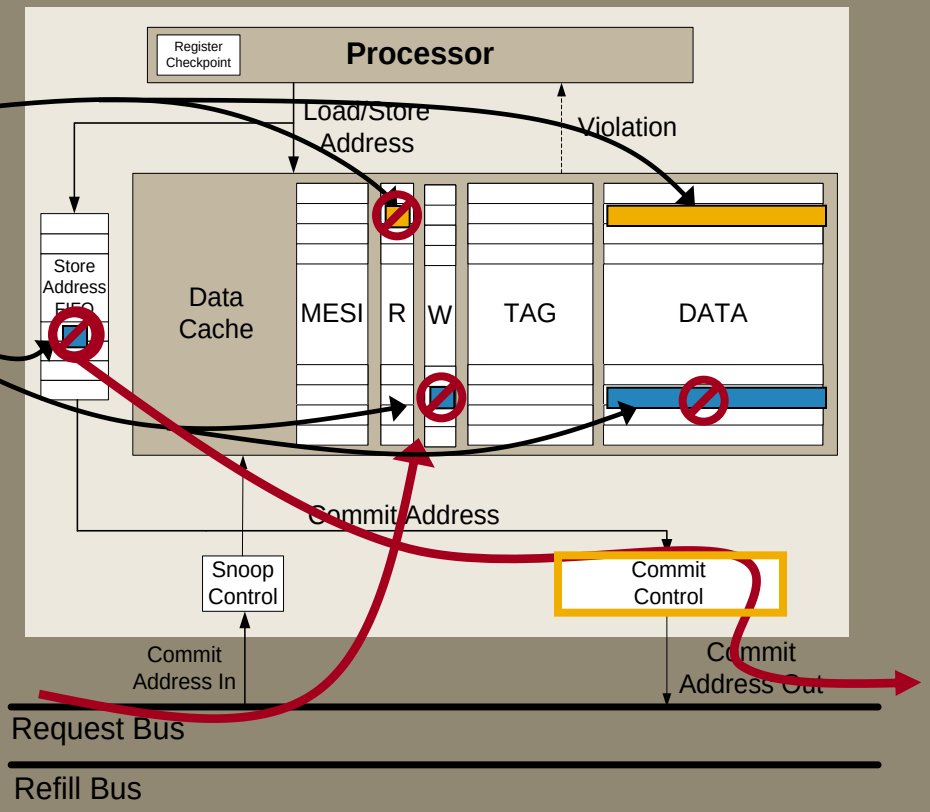
Commit:

Acquire permission to commit

Upgrade lines listed in Store Address FIFO

Conflict Detection:

Compare incoming address to R bits



Performance Questions

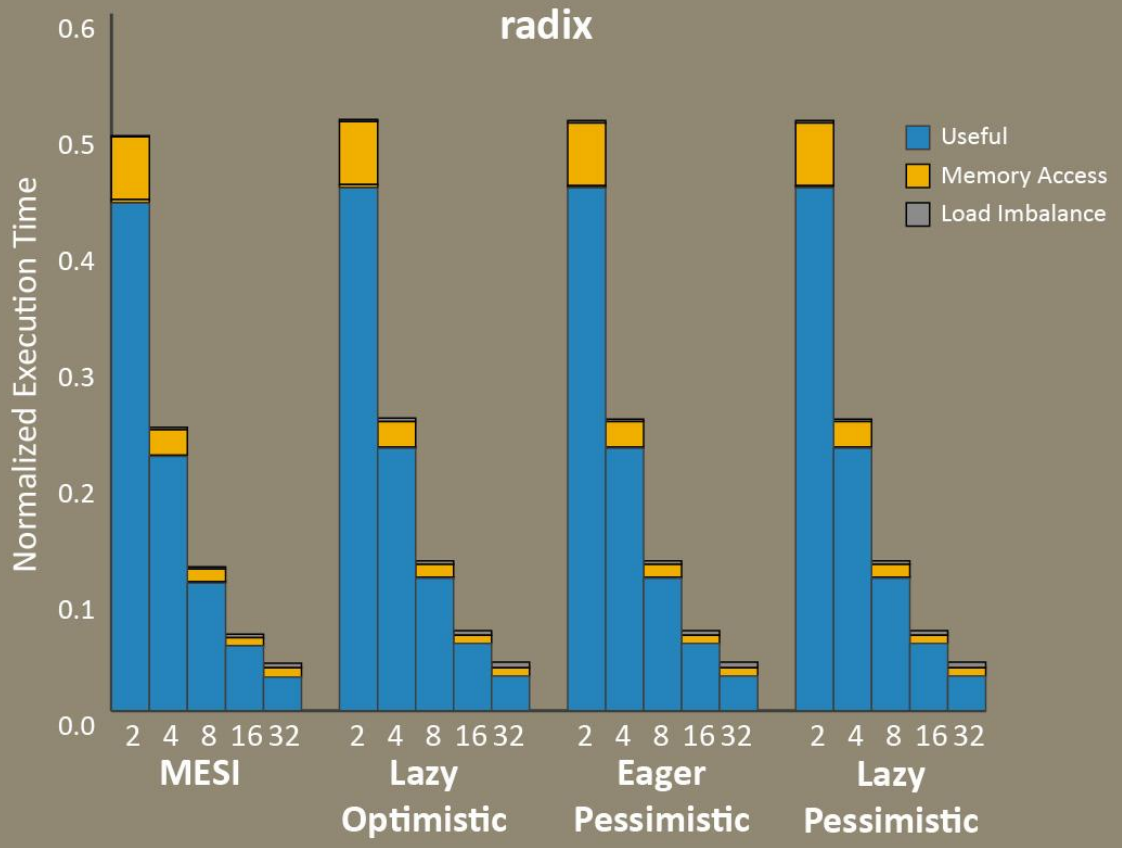
1. Will transactions perform as well as locks?

2. What is the best HTM system and why?

Methodology

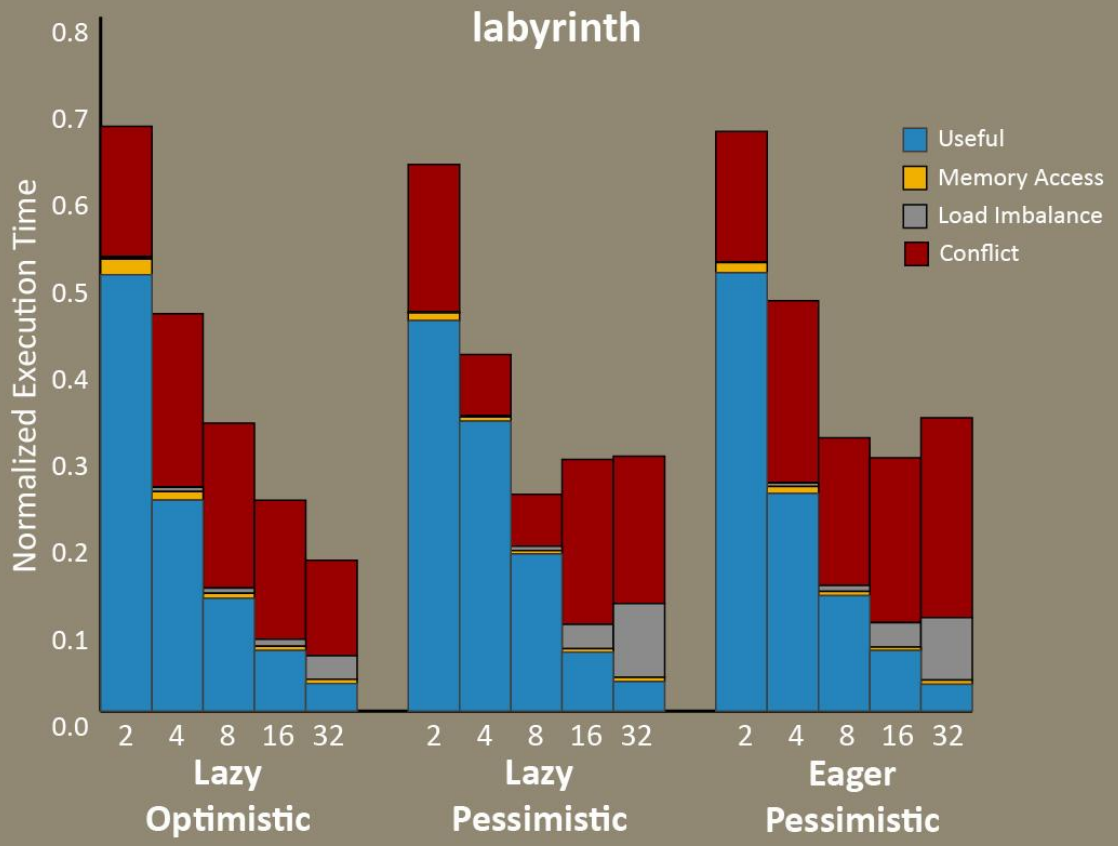
- Execution-driven x86 simulator
  - 1 IPC (except ld/st)
- SPLASH-2 Benchmarks
  - Heavily optimized for MESI
- STAMP
  - Representative applications for today's workloads
  - Wide range of transactional behaviors
  - Difficult to parallelize, TM only apps

1. TM vs Locks



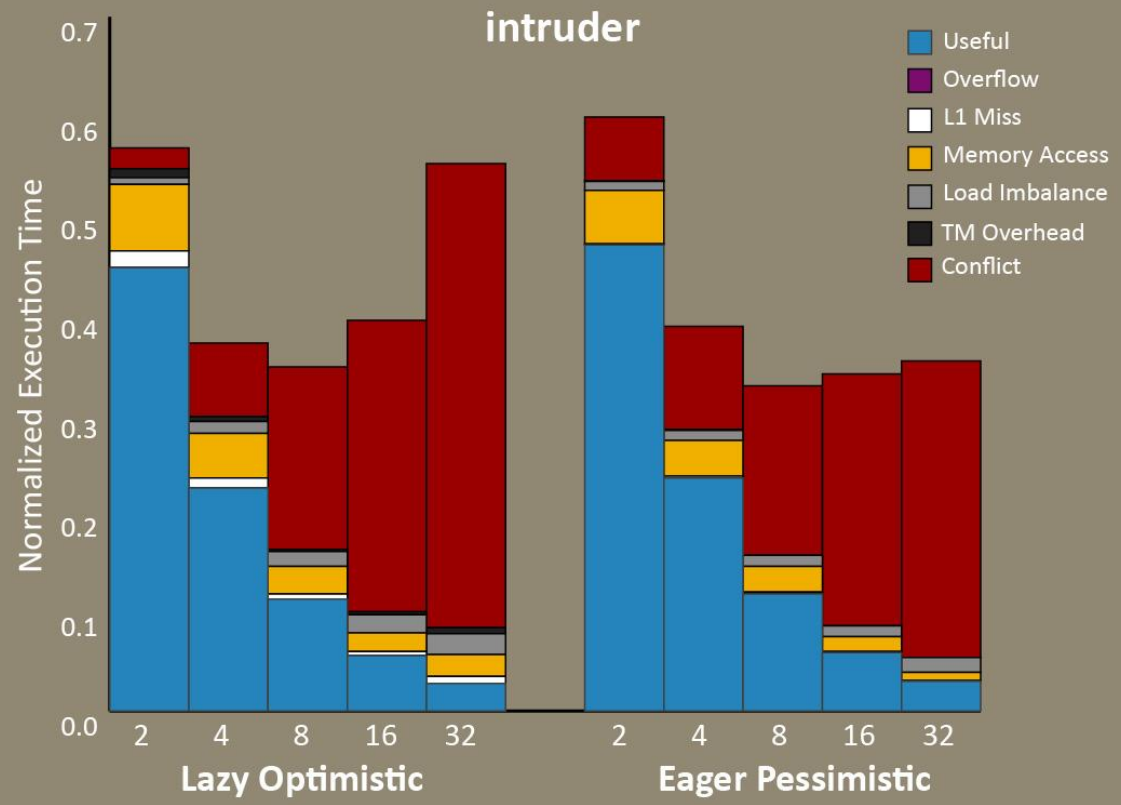
- Performs similar to locks
  - TM overhead is negligible [McDonald '05]
- Similar performance at low contention for all TM schemes

2. Which TM System is Best?



- Pessimistic conflict detection degrades performance
- Rolling back undo log in eager versioning is expensive

2. Which TM System is Best?



- Early conflict detection saves expensive memory accesses
  - High contention, many accesses / Tx

2. Which TM System is Best?

- Same for SPLASH applications
- Same: 2 of 8 STAMP
  - genome, kmeans
- LO Better: 4 of 8 STAMP
  - bayes, labyrinth, vacation, yada
- EP/LP Better: 2 of 8 STAMP
  - intruder, ssca2

- How can I decide on one system?

2. Which TM System is Best?

- Conflict Detection/Resolution principal offender
  - Need intelligent decisions on conflict
- Simple for Optimistic Conflict Detection
  - Priority/aging and random backoff all you need for progress and fairness [Scott '04]
- More complex for Pessimistic
  - More potential performance problems
  - Stall or Abort?
    - Need deadlock/livelock detection
  - Best solution requires hardware predictor [Bobba '08']

Summary of Results

- TM performs as well as locks
- Lazy-Optimistic is the best performing, simplest architecture for TM
- Resource overflow is not a problem

1. Motivation & Contributions
2. Building a TM system in hardware
3. An architecture with only transactions
4. What about the interface to software?
5. Conclusions

SIGNPOST

Only Transactions

Transactions manage communication

- Can we dispense with coherence/consistency protocols?
  - Should be no sharing outside of transactions
  - In transactions, only care about sharing at boundaries
- Easier to reason about parallel programs

TCC: Transactional Coherence and Consistency

[Hammond '04, McDonald '05]

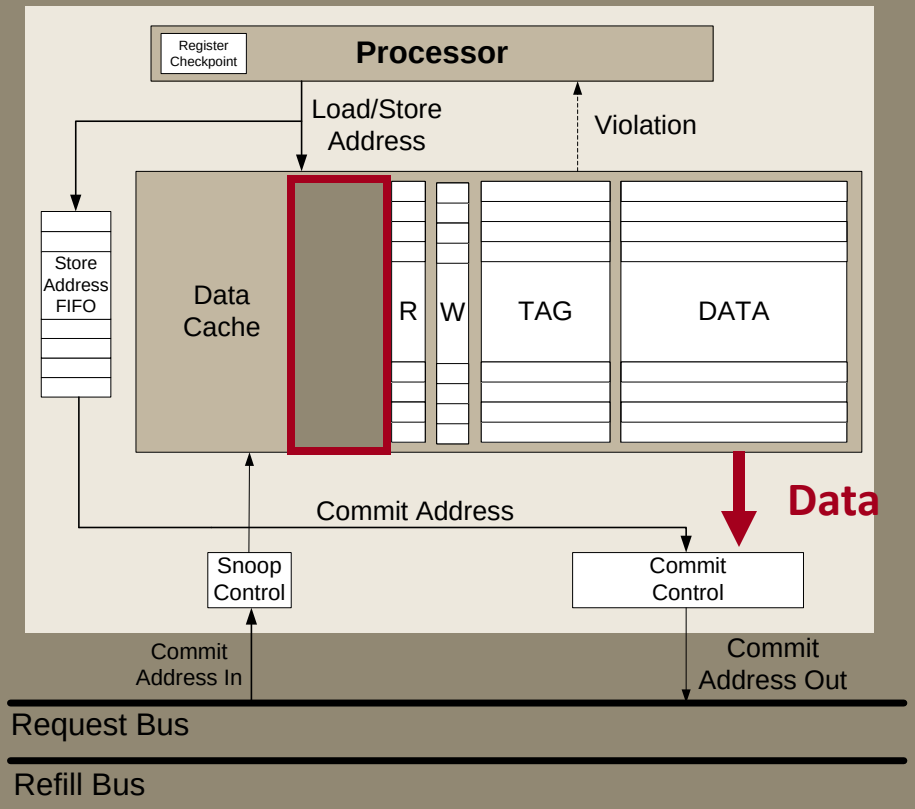
TCC

- Everything is run inside of a transaction [Hammond '04]
  - Even when you don't explicitly create one
- Still have explicit transactions
  - To ensure atomicity
  - Regions between explicit transactions can be split, by the system, into arbitrary transactions

- Simplified Reasoning
  - One mechanism to communicate between threads
    - Hardware is simpler
  - Debugging becomes easier [Chafi '05]
    - All accesses are tracked → detect missing explicit transactions
  - Deterministic replay [Wee '08]

TCC Modifies Lazy-Optimistic

- No need for MESI
- Commit
  - Send data
    - Only way to maintain coherence



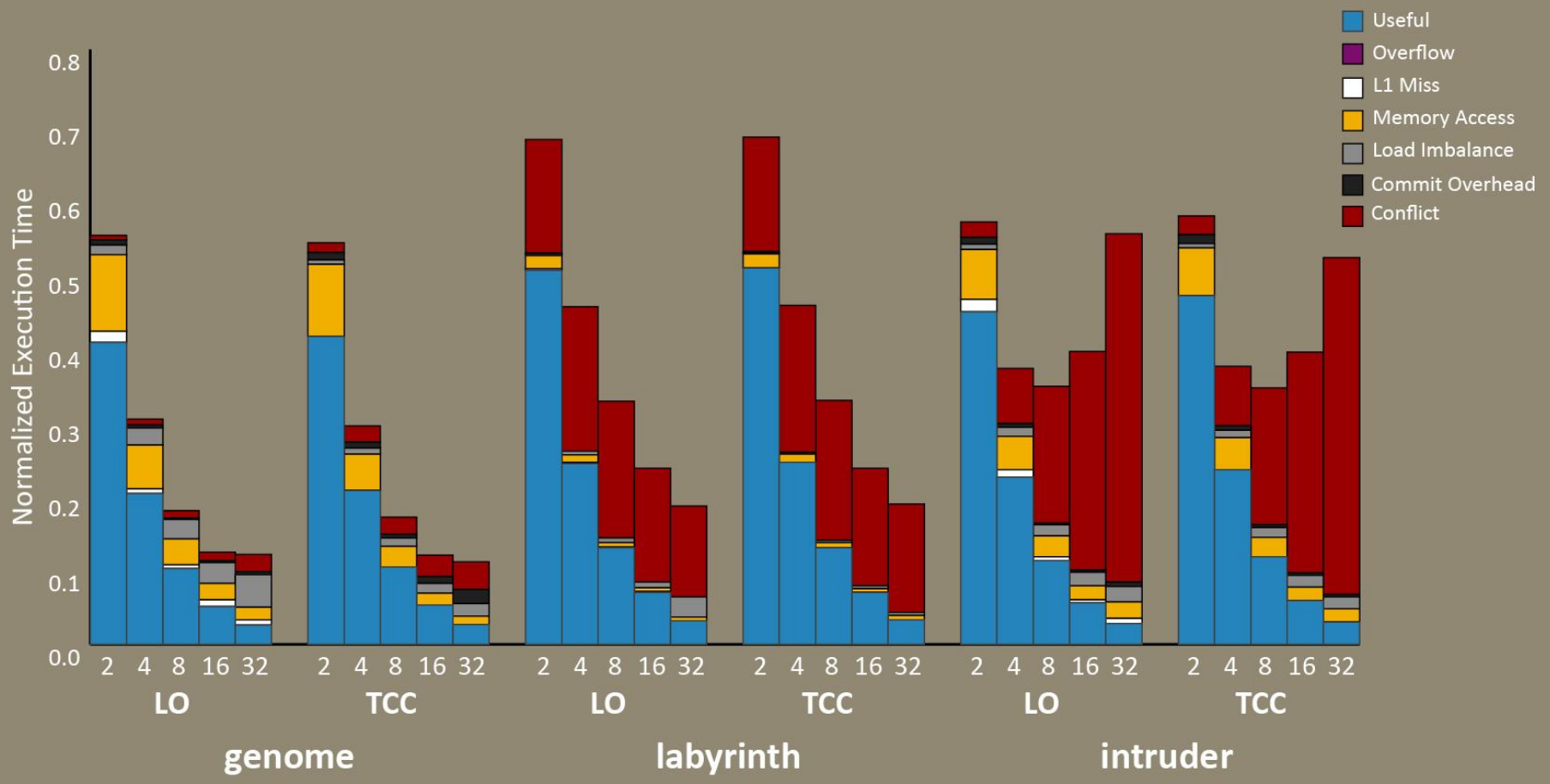
TCC Design Space

- Commit-through or Commit-back
  - Commit-through
  - Commit-back, snooping and M bit
- Line or word-level granularity
  - Communicating less often so word-level is possible
    - Avoids false sharing
    - Need word-level R, W, and V bits

TCC Performance

- Should be similar to LO
- More transactions means more transactional overhead
- Commits happen more often and **contain data**, not just addresses
  - Will bandwidth become a bottleneck?

TCC Performance



Summary of Results

- Neither overhead nor bandwidth are a problem
  - TCC performs similarly to LO and therefore to locks
- Word-level granularity helps alleviate false sharing
- Update does not significantly improve performance

  [McDonald '05]

1. Motivation & Contributions
2. Building a TM system in hardware
3. An architecture with only transactions
4. What about the interface to software?
5. Conclusions

SIGNPOST

Won't Someone Think of the Software

- How does TM interact with library-based software containing transactions?
- How do we handle I/O and system calls within transactions?
- How do we handle exceptions and contention within transactions?
- How do we implement TM programming languages?

Towards a TM ISA

- I defined a flexible, ISA-level semantics for TM
  - Any TM system

  [McDonald '06]
- Four primitives:
  - Two-phase Commit
  - Transactional Handlers
  - Nested Transactions
  - Non-Transactional Loads and Stores

Two-Phase Commit

- TM systems have monolithic commit
- Two-Phase Commit: **validate** and **commit**
  - Validate ensures no conflicts
  - Run code in between as part of the transaction

- Examples:
  - Finalize I/O operations started in the transaction

Transactional Handlers

- TM events processed by hardware
  - Prevents “smart” decisions on commit and violate
- Handlers: fast code on **commit**, **conflict**, and **abort**
  - Software can register multiple handlers per transaction
    - Stack of handlers maintained in software
  - Handlers have access to all transactional state
    - They decide what to commit or rollback, to re-execute or not, …
- Example:
  - Contention managers
  - I/O operations within transactions and conditional synchronization

Nested Transactions

- Early TM systems did not run transactions within transactions
  - Subsumption creates long dependency chains
- Nested Transactions: **closed** and **open**
  - Independent conflict tracking
  - Some cases, independent isolation/atomicity behavior

Closed Nesting

```
atomic {
  lots_of_work()
  count++
}
```

```
atomic {
  lots_of_work()
  atomic {
    count++
  }
}
```

- Performance improvement (reduce conflict penalty)
- Examples:
  - Composable libraries

Open Nesting

```
atomic {
  lots_of_work()
  malloc(…) {
    [modify free list]
  }
  lots_of_work()
}
```

```
atomic {
  lots_of_work()
  malloc(…) {
    openatomic {
      [modify free list]
    }
  }
  lots_of_work()
}
```

- Examples:
  - System calls, communication between transactions/OS/etc.
- Open nesting provides atomicity & isolation for enclosed code

Non-Transactional Loads and Stores

- Often, transactions contain dependencies that are irrelevant
- Non-Transactional Loads and Stores
  - Avoid creating unneeded dependencies
  - Prevent spurious conflicts

- Example:
  - Object-based TM (only dependence on header)

TM ISA Implementation

- Combinations of hardware and software
  - Nested Transactions like function calls
  - Handlers stored on a stack
    - Implemented like exceptions
- Need additional R/W bits or nesting level entry in cache lines

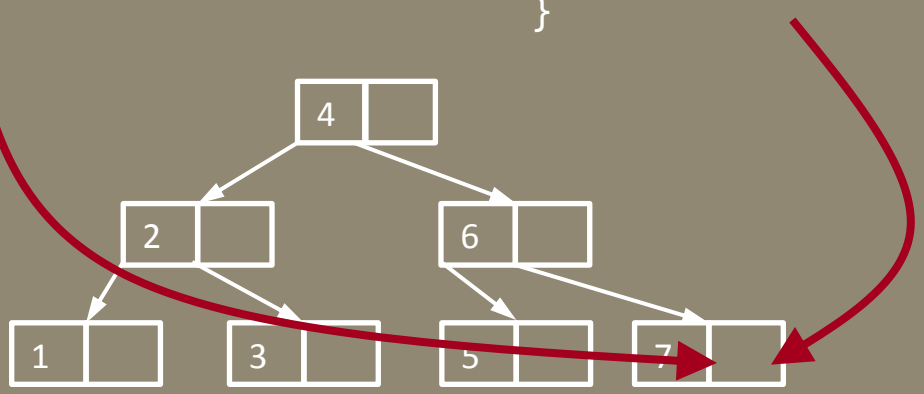
TM ISA Evaluation

- Will the overhead be prohibitive?
  - No, you've already seen it ☺
- Will the ISA be sufficient for all needs?
  - No formal proof
  - Examples [McDonald '06, Carlstrom '06, Carlstrom '07]

Semantic Concurrency Control

```
atomic {
  lots_of_work();
  insert(key=8, data1);
}
```

```
atomic {
  lots_of_work();
  insert(key=9, data2);
}
```



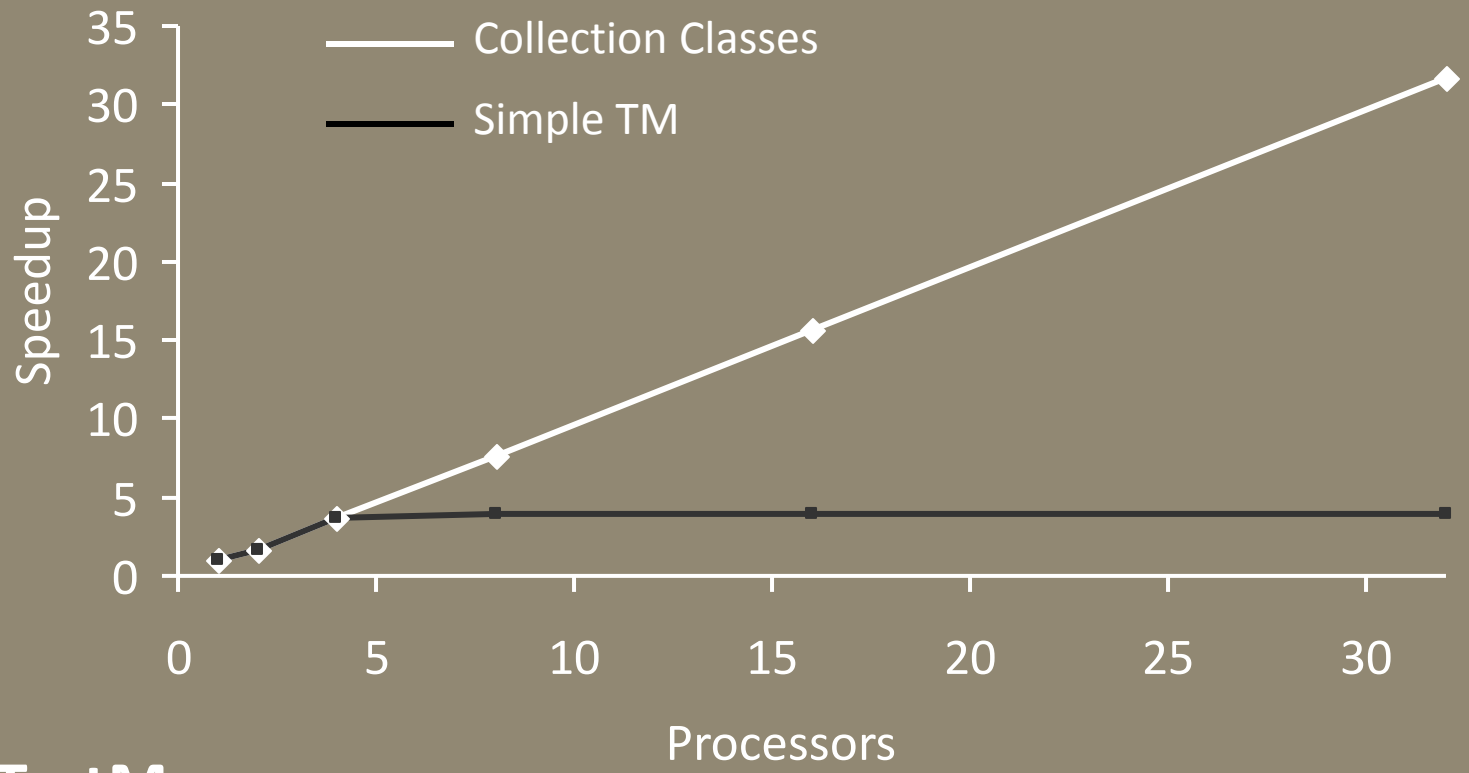
- Is there a conflict?
  - TM: yes, conflict on same memory location
  - Logically: no, operation on different keys
- Common performance loss in TM programs
  - Large, compound transactions

Transactional Collection Classes

- Read operations track semantic dependencies
  - Using **open nested transactions**
- Write operations deferred until commit
  - Using **open nested transactions**
- **Commit handler** checks for semantic conflicts
- **Commit handler** performs write operations
- **Commit/abort** handlers clear dependencies

[Carlstrom ’07]

Transactional Collection Classes



TestMap

- a long transaction containing a single map operation

Summary of Results

- TM needs rich semantics
  - Modern OS/PL
  - Changing underlying architectures
- Four primitives provide needed functionality
  - Two-Phase Commit
  - Transactional Handlers
  - Nested Transactions
  - Non-Transactional Loads and Stores
- These primitives are low overhead and sufficiently flexible

1. Motivation & Contributions
2. Building a TM system in hardware
3. An architecture with only transactions
4. What about the interface to software?
5. Conclusions

SIGNPOST

Contributions/Conclusions

- Evaluated hardware TM systems
  - The best system from efficiency/complexity standpoint is Lazy-Optimistic
- Replaced coherence and consistency with only transactions
  - Using only transactions for communication is advantageous and efficient
- Devised a hardware/software interface for TM
  - Simple primitives provide TM with flexible and needed semantics

Acknowledgements

- GOD
- Advisors: Christos (the Man) Kozyrakis and Kunle (Papa “K”) Olukotun
- Thesis/Defense Committee: Mendel, Phil, Eric
- Parents & Sister: Pete and Jane, Liz
  - (meet them, they’re here!)
- TCC Group
  - Brian Carlstrom, JaeWoong Chung, Chi Cao Minh, Hassan Chafi, Jared Casper, and Nathan Bronson
- Admins: Teresa and Darlene
- Aunt Elizabeth for the food
- GT Peeps
  - Advisor: Kenneth Mackenzie
  - Josh, Chad, Craig, Peter
- Friends

  Vijay, Kayvon, Jeff, Martin, Natasha, Doantam, Adam, Ted, Dan

  Zack, Nick, Brian & Rose, Asela, Ming, Danny, Doug, Zaz, Adam, Josh, Sam, Stone, Rich, Ray, Byron, Susan, Jynette, Kristi, Kokeb, Wendy, Adelaide, Ellen, Sean, Brogan & O’Haras, Rick, Shane, Lawrence, Eric, Burhan & Abby, Todd & Veronica, Anthony & Jasamine, Liz, Lucy, Rama, JT

The Difficulties with Parallel Programming

1. Finding independent tasks in the algorithm
2. Mapping tasks to execution units (e.g. threads)
3. Defining & implementing synchronization
   - Race conditions
   - Deadlock avoidance
   - Interactions with the memory model
4. Composing parallel tasks
5. Recovering from errors
6. Portable & predictable performance
7. Scalability
8. Locality management

And, of course, all the sequential issues…

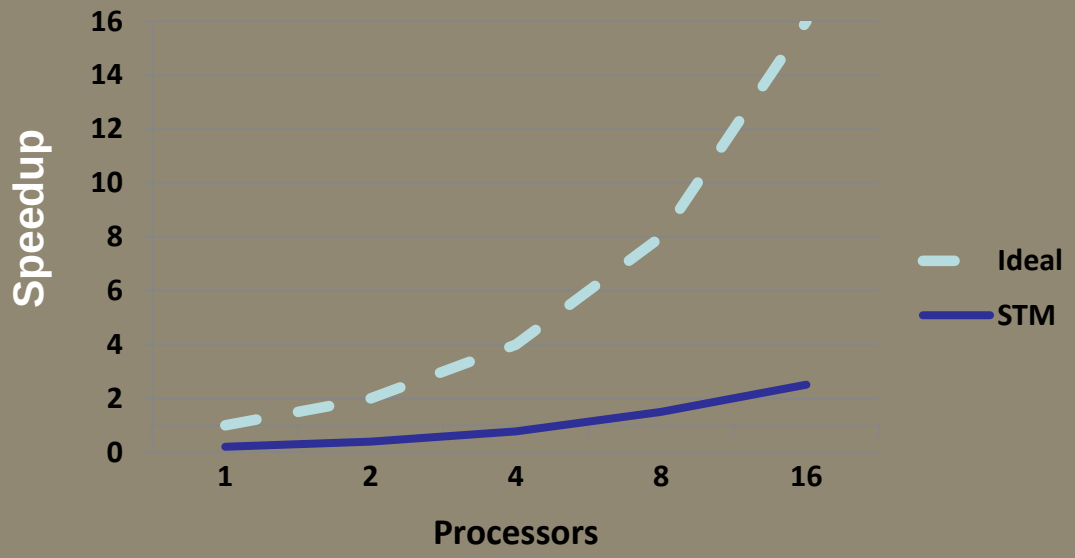
Simulation Parameters

- **CPU** 1–32 single-issue x86 cores
- **L1** 32-KB, 32-byte cache line, 4-way associative
- **Private L2** 512-KB, 32-byte cache line, 16-way associative, 3 cycle latency
- **L1/L2 Victim Cache** 16 entries fully associative
- **Bus Width** 32 bytes
- **Bus Arbitration** 3 pipelined cycles
- **Bus Transfer Latency** 3 pipelined cycles
- **Shared Cache** 8MB, 16-way, 20 cycles hit time
- **Main Memory** 100 cycles latency, up to 8 outstanding transfers

Table 4.2: The basic characterization of the STAMP applications. The number of instructions per transaction does not include instructions that occur as part of TM barriers. The Lazy HTM was used to measure the read and write sets and the amount of time spent in transactions. The amounts of read and write barriers were collected using the Lazy STM, and the Lazy HTM was used to find the fraction of time spent in transactions. For the number of retries, 16 threads were used on all systems. The transactional statistics for `genome` and `intruder` follow bimodal distributions, and those for `vacation` are trimodal. The rest of the applications have normal distributions.

Application	Per Transaction					Time in Transactions	Retries Per Transaction				Working Set	
							HTM		STM			
	Instructions (mean)	Read Set (90 pctile)	Write Set (90 pctile)	Read Barrier (90 pctile)	Write Barrier (90 pctile)		Lazy (mean)	Eager (mean)	Lazy (mean)	Eager (mean)	Small (KB)	Large (MB)
bayes	60584	452	304	24	9	83%	0.66	6.50	0.59	0.66	128	2
bayes+	57130	448	266	26	9	83%	0.69	5.78	0.61	0.69	128	2
genome	1717	98	15	32	2	97%	0.10	0.47	0.14	2.20	128	1
genome+	1709	108	15	30	2	97%	0.02	0.26	0.06	1.14	128	4
intruder	330	51	20	71	16	33%	1.79	6.27	3.54	3.31	32	1
intruder+	331	54	18	54	9	43%	0.67	2.05	1.95	2.96	128	2
kmeans-high	117	14	5	17	17	7%	0.07	0.13	2.73	3.10	16	1
kmeans-high+	153	16	6	25	25	6%	0.05	0.11	3.49	3.68	16	2
kmeans-low	117	14	5	17	17	3%	0.02	0.05	0.89	0.80	16	1
kmeans-low+	153	16	6	25	25	3%	0.01	0.02	0.81	0.70	16	2
labyrinth	219571	433	458	35	36	100%	0.72	2.64	0.94	1.11	64	1
labyrinth+	687809	783	779	46	47	100%	2.55	10.59	1.07	1.38	128	2
ssca2	50	10	4	1	2	17%	0.01	0.01	0.00	0.01	256	2
ssca2+	50	10	4	1	2	16%	0.00	0.00	0.00	0.00	512	4
vacation-high	3223	130	24	432	12	86%	0.37	1.01	0.00	0.01	256	2
vacation-high+	4193	173	23	608	12	92%	0.25	0.66	0.04	0.05	512	4
vacation-low	2420	99	22	287	8	86%	0.07	0.25	0.00	0.00	256	2
vacation-low+	3161	126	22	401	8	92%	0.05	0.18	0.02	0.03	512	4
yada	9795	250	142	256	108	100%	0.52	3.06	2.51	4.35	32	2
yada+	11596	274	145	282	108	100%	0.45	2.04	1.38	2.52	64	8

Hardware or Software TM?



- Software is slower: 2x to 8x overhead due to barriers
  - Short term: discourages parallel programming
  - Long term: wastes energy
- Software is harder: have to avoid programming pitfalls
  - Not the same semantics as locks
  - Strong vs Weak Isolation

Is STM Correct?

Thread 1

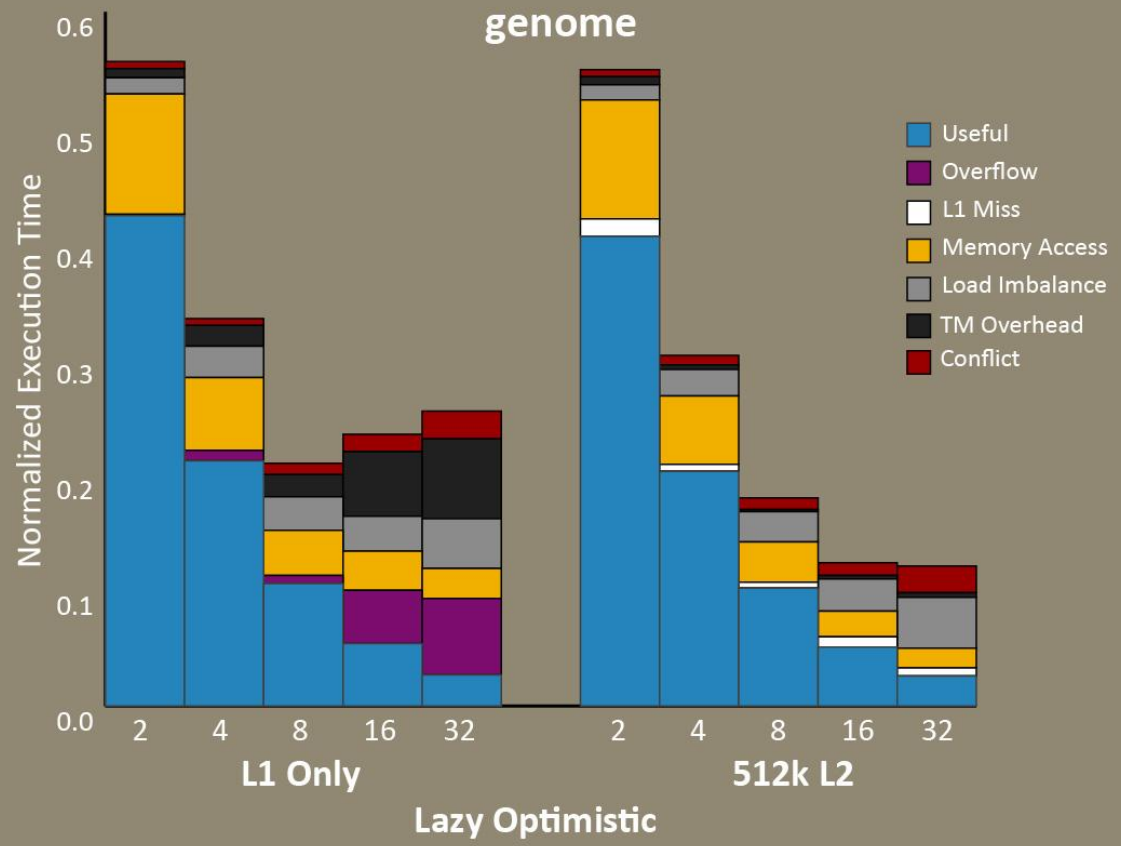
```
atomic{
 if (list != NULL) {
       e = list;
       list = e.next;
}}
r1 = e.x;
r2 = e.x;
assert(r1 == r2);
```

Thread 2

```
atomic{
  if (list != NULL) {
       p = list;
       p.x = 9;
}
```

- The privatization example
  - T1 removes a head; T2 increments head
  - Correctly synchronized code with locks
- Inconsistent results with all STMs
  - T1 assertion may fail from time to time

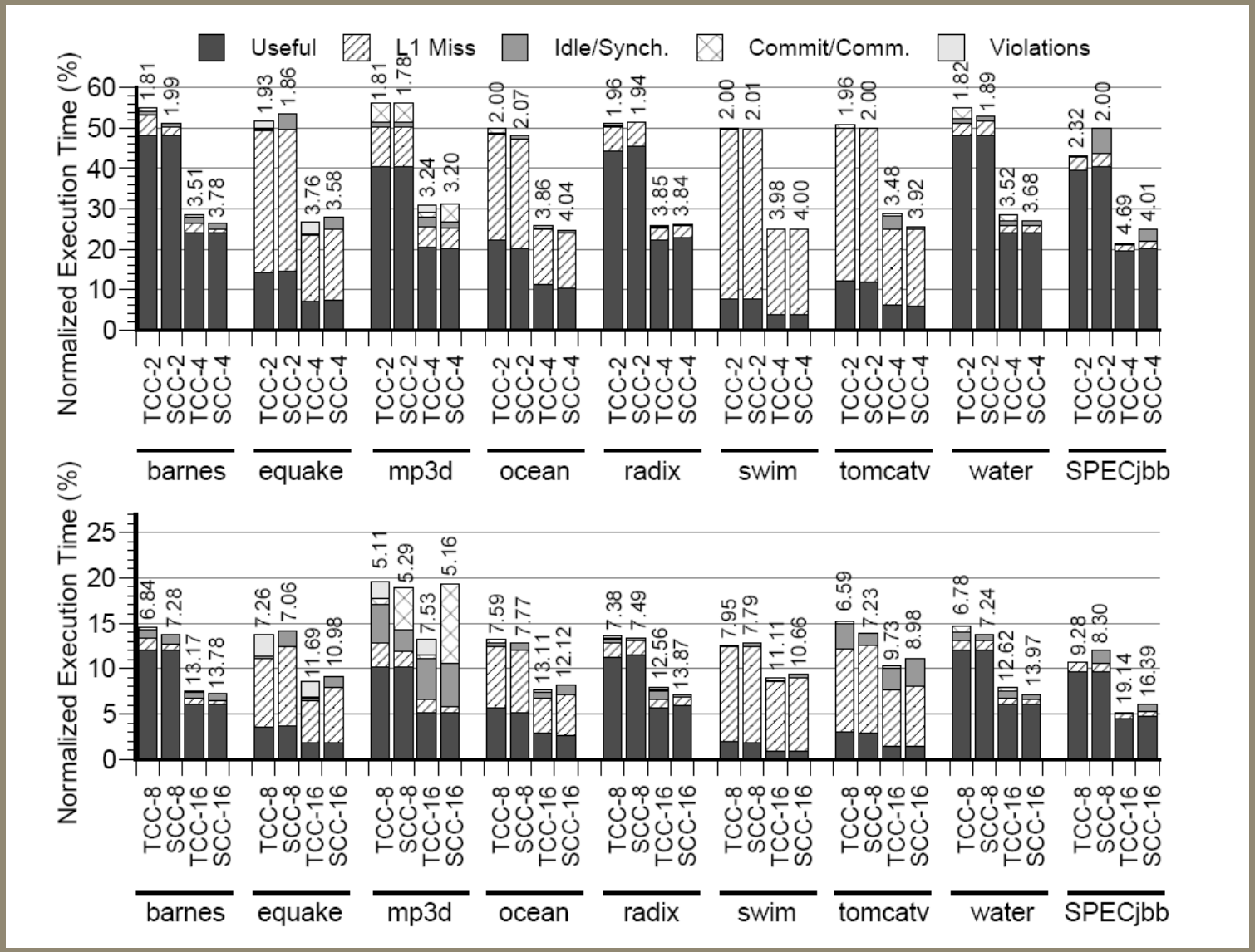
3. Resource Overflow



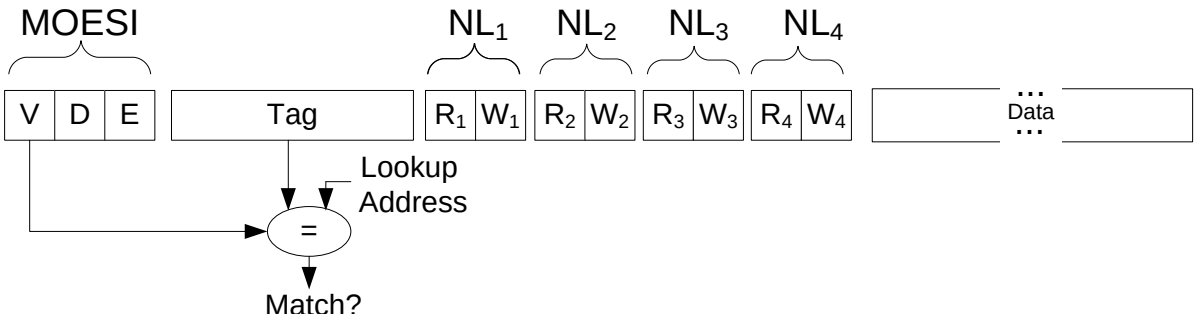
- Overflow mitigated by simple L2 and victim cache
- Virtualization [Chung '06]

Implementing HTM

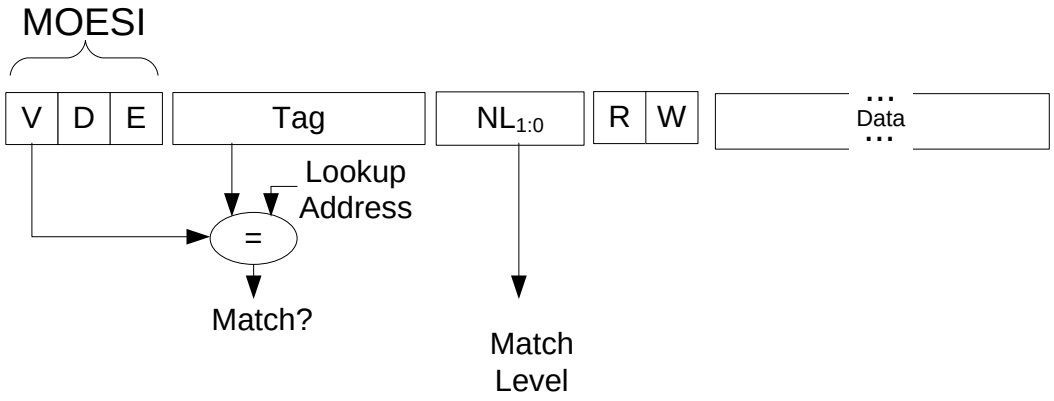
Conflict Detection	Versioning: Eager	Versioning: Lazy
Optimistic	Not logical in HW	Store new values on side Slow commits Fast aborts Conflicts at TX boundaries [Hammond '04, McDonald '05]
Pessimistic	Store new values in place Fast commits Undo log to store old values Slow aborts Conflicts at ld/st granularity [Moore '06]	Store new values on side Slow commits Fast aborts Conflicts at ld/st granularity [Ananian '05]


Useful
L1 Miss
Idle/Synch.
Commit/Comm.
Violations
Normalized Execution Time (%)
0
10
20
30
40
50
60
1.81 1.99 3.51 3.78
1.93 1.86 3.76 3.58
1.81 1.78 3.24 3.20
2.00 2.07 3.86 4.04
1.96 1.94 3.85 3.84
2.00 2.01 3.98 4.00
1.96 2.00 3.48 3.92
1.82 1.89 3.52 3.68
2.32 2.00 4.69 4.01
TCC-2
SCC-2
TCC-4
SCC-4
barnes
equake
mp3d
ocean
radix
swim
tomcatv
water
SPECjbb
Normalized Execution Time (%)
0
5
10
15
20
25
6.84 7.28 13.17 13.78
7.26 7.06 11.69 10.98
5.11 5.29 7.53 5.16
7.59 7.77 13.11 12.12
7.38 7.49 12.56 13.87
7.95 7.79 11.11 10.66
6.59 7.23 9.73 8.98
6.78 7.24 12.62 13.97
9.28 8.30 19.14 16.39
TCC-8
SCC-8
TCC-16
SCC-16
barnes
equake
mp3d
ocean
radix
swim
tomcatv
water
SPECjbb

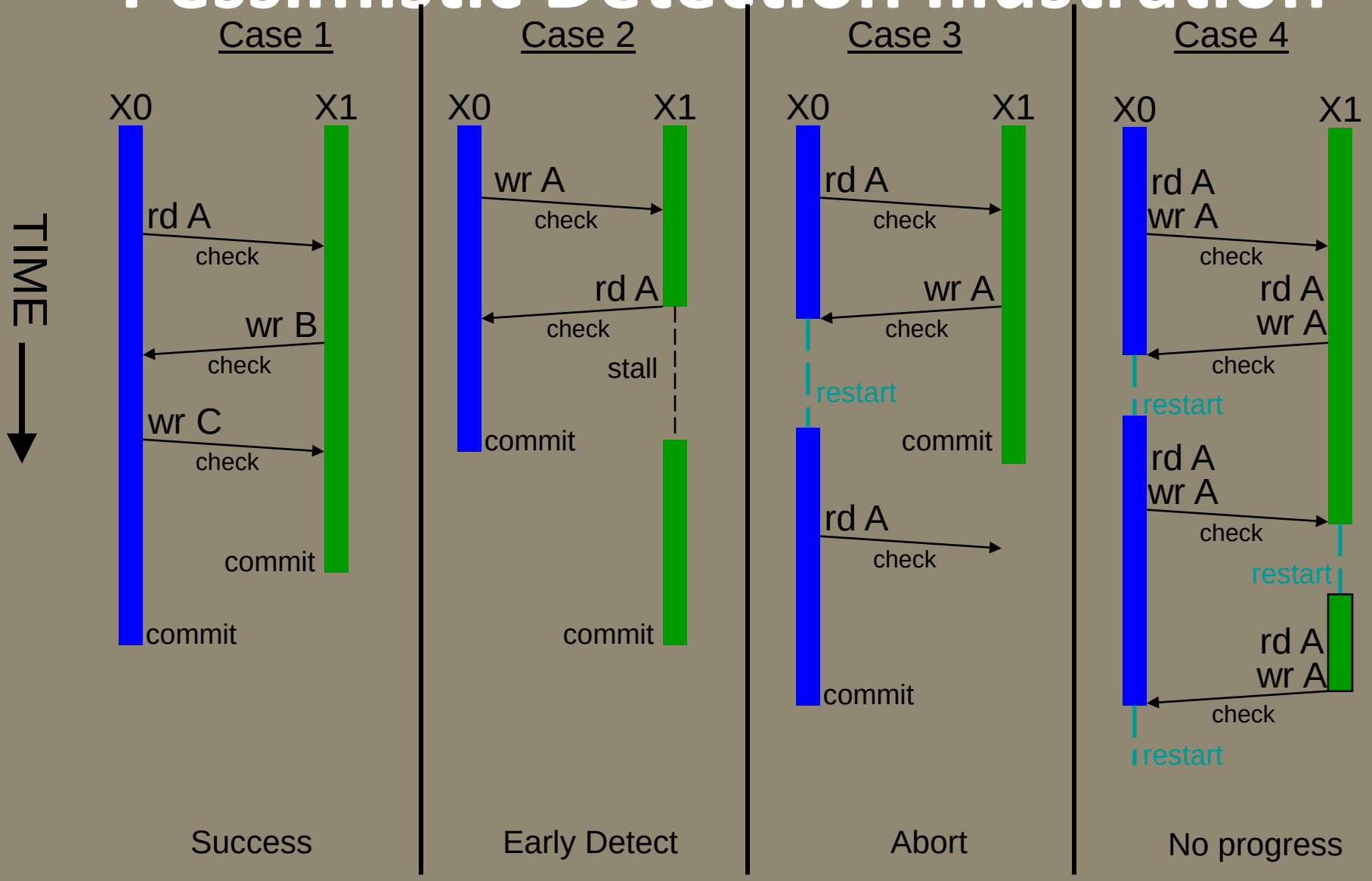
Multi-tracking



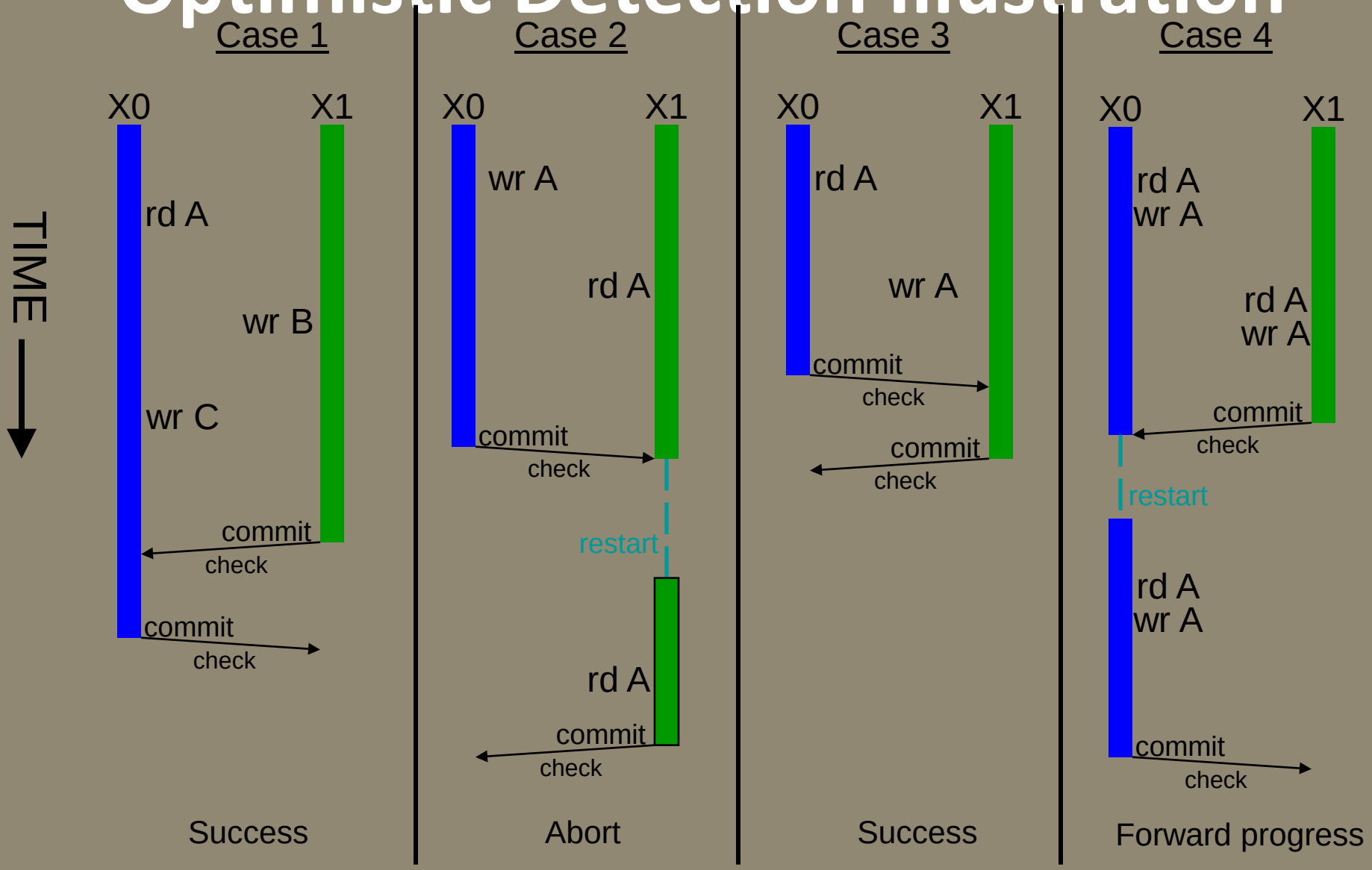
Associativity-based

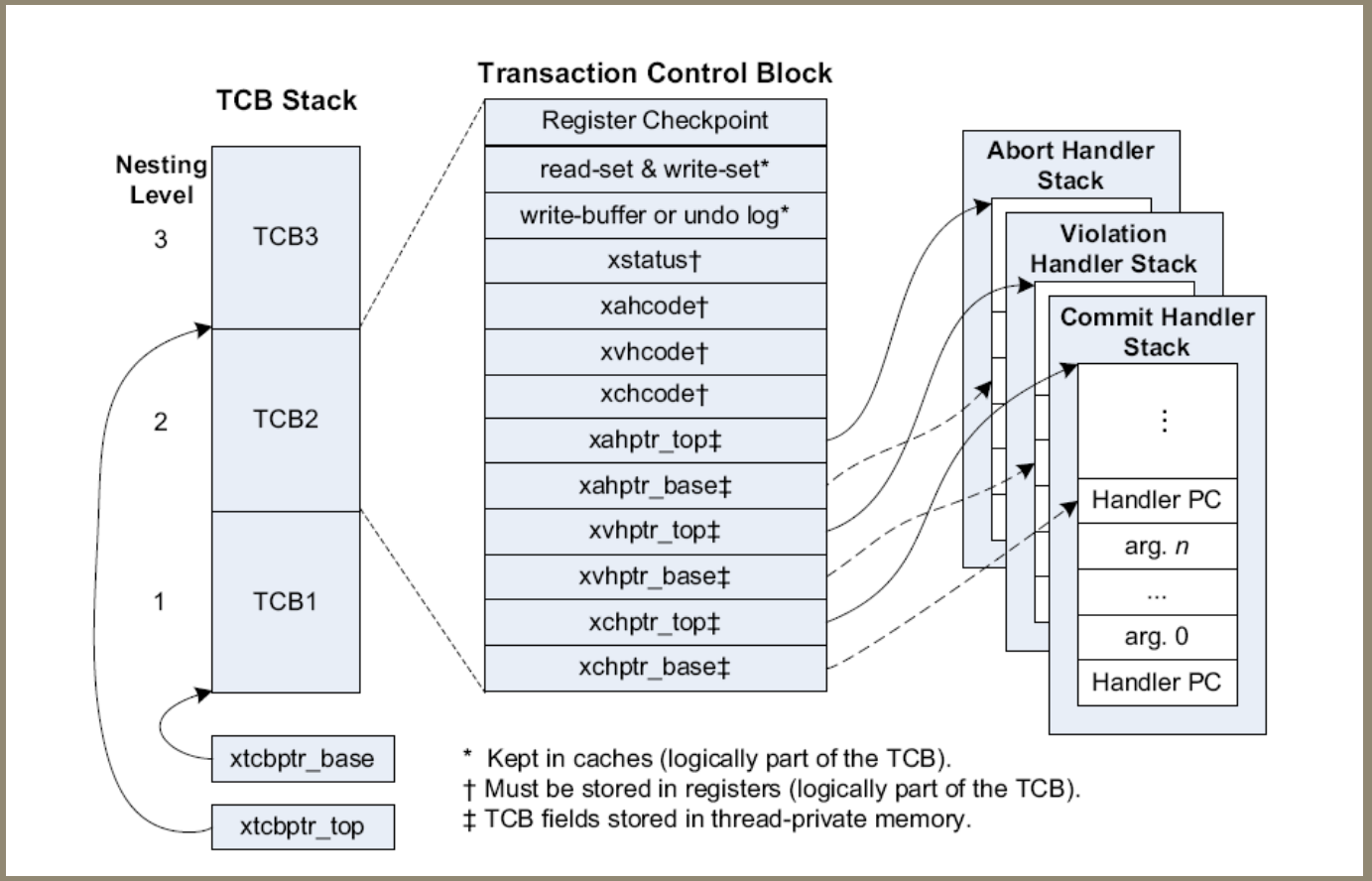


Pessimistic Detection Illustration



Optimistic Detection Illustration




TCB Stack
Nesting Level
3
2
1
TCB3
TCB2
TCB1
xtcbptr_base
xtcbptr_top
Transaction Control Block
Register Checkpoint
read-set & write-set*
write-buffer or undo log*
xstatus†
xahcode†
xvhcode†
xchcode†
xahptr_top‡
xahptr_base‡
xvhptr_top‡
xvhptr_base‡
xchptr_top‡
xchptr_base‡
Abort Handler Stack
Violation Handler Stack
Commit Handler Stack
Handler PC
arg. n
...
arg. 0
Handler PC
* Kept in caches (logically part of the TCB).
† Must be stored in registers (logically part of the TCB).
‡ TCB fields stored in thread-private memory.