ECE 454
Computer Systems Programming

Avoiding Locks

Ashvin Goel
ECE Dept, University of Toronto
http://www.eecg.toronto.edu/~ashvin

With thanks to Angela Demke Brown, Tom Hart, Paul McKenney
Overview

• Challenges with Locking
• Non-Blocking Synchronization
• Read-Copy Update
Challenges with Locking
Locking: A Necessary Evil?

- Locks - easy solution to critical section problem
  - Protect shared data from corruption due to simultaneous updates
  - Protect against inconsistent views of intermediate states

- But locks have lots of problems
  - 1. Deadlock
  - 2. Priority inversion
  - 3. Not fault tolerant
  - 4. Convoying
  - 5. Expensive, even when uncontended

- Not easy to use correctly!
1. Deadlock

- Textbook definition: Set of threads blocked waiting for event that can only be caused by another thread in the same set

  /* a threaded program with a potential for deadlock */

  Thread1()
  { 
      lock(a);
      lock(b);
      do_work();
      unlock(b);
      unlock(a);
  }

  Thread2()
  { 
      lock(b);
      lock(a);
      do_work();
      unlock(a);
      unlock(b);
  }

- Solutions exists but add complexity
  - E.g., specify lock order
2. Priority Inversion

- Lower priority thread gets spinlock
- Higher priority thread becomes runnable and preempts it
  - Needs lock, starts spinning
  - Lock holder can’t run and release lock

- Solutions exist but add complexity
  - E.g. disable preemption while holding spinlock, implement priority inheritance, etc.
3. Not Fault Tolerant

- If lock holder crashes, or gets delayed, no one makes progress

- Delays can happen due to preemption, page faults
  - Disable such delays, e.g., pin pages in memory
  - Avoid critical sections when delays will happen

- Crashes require abort / restart
4. Convoying

- Threads started at different times occasionally access shared data
- Expect shared data accesses to be spread out over time
  - Lock contention should be low
- Delay of lock holder allows other threads to catch up
  - Lock becomes contended and tends to stay that way
  - => Convoying
## 5. Expensive, Even When Uncontended!

<table>
<thead>
<tr>
<th>Operation</th>
<th>Nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>0.24</td>
</tr>
<tr>
<td>Clock Cycle</td>
<td>0.69</td>
</tr>
<tr>
<td>Atomic Increment</td>
<td>42.09</td>
</tr>
<tr>
<td>Cmpxchg Blind Cache Transfer</td>
<td>56.80</td>
</tr>
<tr>
<td>Cmpxchg Cache Transfer and Invalidate</td>
<td>59.10</td>
</tr>
<tr>
<td>SMP Memory Barrier (eieio)</td>
<td>75.53</td>
</tr>
<tr>
<td>Full Memory Barrier (sync)</td>
<td>92.16</td>
</tr>
<tr>
<td>CPU-Local Lock</td>
<td>243.10</td>
</tr>
</tbody>
</table>

McKenney, 2005 – 8-CPU 1.45 GHz PPC
Critical Section Efficiency

- Assuming little to no contention, and no caching effects in CS

\[ \text{Efficiency} = \frac{T_c}{T_c + T_a + T_r} \]

- Ta and Tr can take 100+ cycles, even with no contention
- Critical section efficiency must be addressed!
Causes: Deeper Memory Hierarchy

• Memory speeds have not kept up with CPU speeds
  • 1984: no caches needed, since instructions slower than memory accesses
  • after ~2005: 3-4 level cache hierarchies, since instruction speeds are orders of magnitude faster than memory accesses

• Synchronization ops typically execute at memory speed
Causes: Deeper Pipelines

<table>
<thead>
<tr>
<th>Then:</th>
<th>Now:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Fetch</td>
</tr>
<tr>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>Retire</td>
<td>Retire</td>
</tr>
</tbody>
</table>

- **1984**: Many cycles per instruction
- **2005**: Many instructions per cycle
  - 20 stage pipelines
  - CPU logic executes instructions out-of-order to keep pipeline full
  - Synchronization instructions cannot be reordered
  - => Synchronization stalls the pipeline
Performance

• **Main issue with lock performance used to be contention**
  • Techniques were developed to reduce overheads in contended case
    • E.g., MCS locks

• **Today, issue is degraded performance even when locks are always available**
  • Together with other concerns about locks
Locks: A Necessary Evil?

Idea: Don’t lock if we don’t need to!

- Use “lockless” synchronization
  - Design data structures so that locks are not required
Non-Blocking Synchronization
Non-Blocking Synchronization (NBS) Basics

- Think of NBS as a “lockless” synchronization scheme
  - With locking, threads access shared object under mutual exclusion
  - With NBS, threads can access shared object concurrently

- Idea: make change optimistically, if conflict detected, roll back

```c
// atomically increment *counter using CAS
atomic_inc(int *counter) {
    int value;
    do {
        value = *counter;  // save value of counter
    } while (!CAS(counter, value, value+1));
}
```

- Complex updates (e.g. modifying multiple values in a structure) are hidden behind a single commit point using atomic instructions
Example: Lock-Based Stack

class Node {
    Node *next;
    int data;
};

Node *head; Lock *l;

Node *pop() {
    int current = NULL;
    lock(l);
    if (head) {
        current = head;
        head = head->next;
    }
    unlock(l);
    return current;
}

void push(Node *node) {
    lock(l);
    node->next = head;
    head = node;
    unlock(l);
}

Example: Lock-Free Stack

```c
void push(Node *node) {
    do {
        node->next = head;
    } while (!CAS(&head, node->next, node));
}

Node *pop() {
    Node *current = head;
    while (current) {
        if (CAS(&head, current, current->next)) {
            return current;
        }
        current = head; // head may have changed
    }
    return NULL;
}
```

class Node {
    Node *next;
    int data;
};

Node *head;

Anything wrong?
ABA Problem

- Notice that `pop` reads `head` twice.
- If the value of `head` hasn’t changed, then `head` is updated.
- What if another thread updates `head` in between, does other work, and then changes `head` back to the old value?

```c
Node *pop() {
    Node *current = head;
    while (current) {
        if (CAS(&head, current, current->next)) {
            return current;
        }
    }
    ...
}
```
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:
  - Ti: starts pop()
    - head is A
    - current is A
    - current->next is B (loaded in reg)
    - Ti **interrupted before** it performs: CAS(&head, current, current->next), i.e., before head is assigned to B

```
Node *current = head;
...
if (CAS(&head, current, current->next))
```

![Diagram](head → A → B → C)
ABA Problem

• Say Ti, Tj are both doing pops and pushes on this stack:
ABA Problem

• Say Ti, Tj are both doing pops and pushes on this stack:

• Tj:
  • a=pop()
ABA Problem

• Say Ti, Tj are both doing pops and pushes on this stack:

• Tj:
  • a=pop()
  • b=pop()
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:

- Tj:
  - a=pop()
  - b=pop()
  - push(N)
ABA Problem

• Say Ti, Tj are both doing pops and pushes on this stack:

  • Tj:
    • a = pop()
    • b = pop()
    • push(N)
    • push(a)
    • ‘a’ is the same node that was returned by first pop()
ABA Problem

- Say Ti, Tj are both doing pops and pushes on this stack:
  - Tj:
    - a=pop()
    - b=pop()
    - push(N)
    - push(a)
  - Ti resumes: head is A
    - current is A, current->next is B
    - CAS succeeds, sets head to B!
    - Returns A, A->next set to NULL
    - Stack should have been N, C
One Solution

- Include a version number with every pointer
  - `pointer_t = <pointer, version>`
  - Increment version number every time pointer is modified
    - Need atomic update to pointer and increment
    - Requires double-word CAS operation
      - Not every architecture provides this operation

- Version number ensures CAS will fail if pointer has changed

- Old versions of pointers need to be freed
  - Use garbage collection to reclaim memory later
  - May restrict reuse of memory
Using NBS

- Generally used for simple, update-heavy data structures
  - E.g., linked list
  - See [https://en.wikipedia.org/wiki/Non-blocking_linked_list](https://en.wikipedia.org/wiki/Non-blocking_linked_list)
  - Hard to design data structures that use NBS
When do we need NBS Guarantees?

- When we need linearizability
  - Everyone agrees on all intermediate states
    - All updates appear instantaneous, occur in total order
    - Reads return value of last completed write
  - Imposes dependency between operations
    - Limits parallelism

- Do we always need linearizability?
  - Consider “top” program that lists all existing processes
Read-Copy Update (RCU)
Read-Copy Update (RCU)

• What is RCU?
  • Paul McKenney’s PhD thesis
  • A key part of the Linux scalability effort

• Reader-writer synchronization mechanism
  • Supports concurrency between multiple readers + single updater
  • Readers use no locks
    • Hence best for read-mostly data structures
  • Writers create new versions atomically
    • Either using atomic instructions or by locking out other writers
  • Readers may continue to access old versions
    • Old versions must be deleted at some point
Why RCU?

• Consider concurrent hash table example
  • Hash function selects bucket (entry in an array)
  • Collisions handled by chaining (linked list per bucket)
  • Use per-bucket locks to increase concurrency

• But recall costs of synchronization operations…
What about NBS?

• Non-blocking synchronization is possible for hash table operations
  • But still expensive, even for read-only operations

• Consider concurrent lookup and remove operations:

  T1: read N
  T1 obtains pointer to Node N. Need to ensure N continues to exist until T1 is done using it.

  T2: remove N
  T2 must detect that Node N is in use and defer deletion.
Reference Counting Solution

- T1 can increment reference count on N
  - Requires atomic update for each node along path to N on a read!
- T2 must defer deletion of a node with elevated reference count

T1: read N

T1: atomic_inc(N->refcount)

T2: remove N

T2: while(N->refcount > 1) {};

Reader/Writer locks?

- Concurrent reads, exclusive writes

  - CPU 0
    - Reader
    - Reader
    - Blocked
    - Reader

  - CPU 1
    - Reader
    - Reader
    - Blocked
    - Reader

  - CPU 2
    - Reader
    - Reader
    - Blocked
    - Reader

  - CPU 3
    - Reader
    - Reader
    - Spin
    - Writer
    - Reader

- Lots of “dead time” as all readers wait for single writer to finish
RCU Design Principle

- Avoid mutual exclusion!

- No more “dead time”

- But how can this be implemented?
RCU Basics

- Three key ideas
  - Use publish/subscribe ordering mechanism
    - Orders operations so readers see consistent, atomic updates
  - Maintain multiple versions of recently updated objects
    - Ensures readers that are concurrent with writers will read consistent (perhaps stale) data versions
  - Wait for previous readers to complete
    - For deleting old versions

- All three together ensure that reads can be performed correctly without using locks

- See LWN article: http://lwn.net/Articles/262464
Is This Code Correct?

• No locks are being used by reader
• When is it safe to dereference the gp pointer?

/* definitions */
struct foo {
    int a;
};

/* gp == global ptr */
struct foo *gp = NULL;

T1 (Writer):
    p = malloc(sizeof(*p));
    p->a = 1;
    gp = p;  // gp can be read by others

T2 (Reader):
    retry:
    p = gp;  // get ptr to shared data
    if (p == NULL)
        goto retry;
    use(p->a);
Memory Order “Writer Mischief”

Compiler, CPU can reorder memory assignments and reads

Problem 1

T1 (Writer):
\[
p = \text{malloc}(\text{sizeof}(\ast p)); \\
p \rightarrow a = 1; \\
gp = p;
\]

T1 (Writer):
\[
p = \text{kmalloc}(\text{sizeof}(\ast p)); \\
gp = p; \\
p \rightarrow a = 1;
\]

T2 (Reader):
\[
\text{retry:} \\
p = gp; \quad // \text{get ptr to shared data} \\
\text{if} \ (p == \text{NULL}) \\
\quad \text{goto retry;} \\
\text{use}(p \rightarrow a); \quad // \text{may read uninitialized value!}
\]
Memory Order “Reader Mischief”

Compiler, CPU can reorder memory assignments and reads

T1 (Writer):

\[
p = \text{malloc}(\text{sizeof}(*p));
\]
\[
p->a = 1;
\]
\[
gp = p; \quad // \text{gp can be read by others}
\]

T2 (Reader):

retry:
\[
p = \text{gp};
\]
\[
\text{if} \ (p == \text{NULL})
\]
\[
goto \text{retry};
\]
\[
\text{use}(p->a);
\]

Problem 2

T2 (Reader):

retry:
\[
p = \text{guess}(gp);
\]
\[
\text{use}(p->a); \quad // \text{old value}
\]
\[
\text{if} \ (p \neq gp) \quad // \text{fails!}
\]
\[
goto \text{retry};
\]
RCU Publish/Subscribe Ordering Mechanism

- Enforce ordering with `rcu_assign_pointer/rcu_dereference`

```c
/* definitions */
struct foo {
    int a;
};

/* gp == global ptr */
struct foo *gp = NULL;

T1 (Writer):
    p = malloc(sizeof(*p));
    p->a = 1;
    gp = p;  rcu_assign_pointer(gp,p);

T2 (Reader):
    retry
    p = gp;  p = rcu_dereference(gp);
    if (p == NULL)
        goto retry;
    use(p->a);
```
Maintaining Multiple Versions

• Two examples using linked list
  • Update
  • Deletion
RCU List Element Update

- T1 traversing linked list, T2 updates an element:

  T1: read N

  T2: update N
RCU List Element Update

• T1 traversing linked list, T2 updates an element:

T1: read N

RC: T2 reads and makes a copy of N

T2: update N

N

N'
RCU List Element Update

- T1 traversing linked list, T2 updates an element:

  T1: read N
  T2: update N
  RC: T2 Reads and makes a Copy of N
  U: T2 Updates prev to N’ atomically

Why make a copy of N?
When is it ok to delete N (and reuse the memory for something else)?
RCU List Element Deletion

- T1 traversing linked list, T2 removes an element:

![Diagram showing the process of T1 traversing a linked list and T2 removing an element N.]

T1: read N

T2: remove N
RCU List Element Deletion

- After removal – T1 continues to use N and later nodes in the list

T1: read N

T2: remove N

T2 updates prev to next atomically

When is it ok to delete N (and reuse the memory for something else)?
Waiting for Previous Readers

- RCU needs to wait for previous readers to reclaim old versions
- RCU uses quiescent-state based reclamation (QSBR) to handle these read-reclaim races
- Definition: A quiescent state for a thread T is a state in which T holds no references to any shared data
- Definition: A grace period is an interval in which every thread has passed through at least one quiescent state
- QSBR idea: elements removed from a data structure can be reclaimed after a grace period, since no thread can still be holding a reference to the old element at that point
Element removed at this point

Element can be reclaimed after this point

Grace Period

Thread 1
Thread 2
Thread 3

Time
How to define Quiescent States?

• Application dependent!

• For OS kernels, some natural ones exist
  • Assume that references to RCU data structures are only held within critical sections
  • Assume that read-side critical sections do not block
    • i.e., No context switch occurs within a read-side critical section

• Then, a context switch is a quiescent state
  • No references are held across a context switch
Reader-Side Quiescence Primitives: Read Lock/Read Unlock

```c
/* definitions */
struct foo {
    int a;
};

/* gp == global ptr */
struct foo *gp = NULL;
```

T1 (Writer):
```
p = malloc(sizeof(*p));
p->a = 1;
rcu_assign_pointer(gp, p);
```

T2 (Reader):
```
rcu_read_lock(); // notice, no lock var
p = rcu_dereference(gp);
if (p != NULL)
    use(p->a);
rcu_read_unlock();
```

- `rcu_read_lock/unlock` do not spin or block!
- They help detect when a reader is in a critical section by disabling context switch within read-side critical section.
Writer-Side Quiescence Primitive: Synchronize RCU

- **synchronize_rcu()**
  - Wait until all pre-existing RCU read-side critical sections complete

- Implementation:

```c
synchronize_rcu() {
    for_each_online_cpu(cpu)
        run_on(cpu); // runs the current thread on cpu
}
```

- **synchronize_rcu()** runs the current thread on all CPUs
  - Forces context switches on each of the CPUs
  - Ensures that it waits for the grace period
RCU Synchronization

- `rcu_dereference()`
- `rcu_assign_pointer()`
- `rcu_read_lock()`, `rcu_read_unlock()`
- `synchronize_rcu()`
// Reader traverses a linked list
rcu_read_lock();
for_each_entry_rcu(p, q, head, list) {
    // p is a linked list node
    do_something(p->value);
}
rcu_read_unlock();

// Writer searches and updates a list element
p = search(head, key);
if (p == NULL) {
    /* unlock and return. */
}
q = kmalloc(sizeof(*p), GFP_KERNEL);
*q = *p; // read and copy
q->value = ...;
// atomically replace p with q
list_replace_rcu(&p->list, &q->list);
// wait for grace period
synchronize_rcu();
// free previous version
kfree(p);
PPC Hash Table with RCU

![Graph showing searches per unit time normalized to ideal versus number of CPUs for different strategies: "ideal", "RCU", "HPBR", "spinbkt", "brlock", "globalrw".](image)
Growth of RCU Use in Linux

...but Still Small in Comparison

Graph from http://www.rdrop.com/users/paulmck/RCU/linuxusage.html
(Nov. 26, 2022, generated daily)
When to Use Which Tool?

- Read-mostly situations
  - If algorithm can handle concurrent reads + single updater: RCU

- Update-heavy situations
  - Simple data structures and algorithms: NBS
  - Complex data structures and algorithms: Locking

- When you only have a hammer, everything looks like a nail
- It’s good to have lots of tools in your toolbox!
Some Resources

• LWN article on lockless algorithms
  https://lwn.net/Kernel/Index/#Lockless_algorithms

• Load dependent ordering behavior in Alpha:
  http://www.cs.umd.edu/~pugh/java/memoryModel/AlphaReordering.html

• An excellent book on multi-processor synchronization and lockless algorithm is: The art of multiprocessor programming by Maurice Herlihy & Nir Shavit
Transactional Memory

Active research! Here be dragons…
Challenges of Synchronization

- Two major issues:
- Performance scalability
  - We have looked at some techniques for improving performance
    - Better spinlocks
    - Lockless strategies (NBS, RCU)
- Programmability
  - Locks are hard to use correctly
  - Lockless data structures are hard to design
What’s Missing?

• Lack of support for abstraction and composition

• E.g., Suppose we have thread-safe stack with (abstract) push and pop operations
  • In sequential programs, can use these operations without regard to their implementation
  • In parallel programs, internal details may be needed
    • Consider the task of moving an item from one stack to another
      • pop followed by push
    • Need to expose stack locking mechanism to compose the operations
“Magic” Wish List

• Let programmers express desired outcome that a block of code should run atomically
  • E.g., move (pop followed by push) should be atomic

• Allow abstractions to hide implementation and be composable
  • E.g., two different moves together should be atomic

• Let run-time system or hardware support make it happen

• A new programming model is needed!
Database Systems

- Database systems allow multiple queries to run in parallel
- Database programmers write queries without worrying about concurrency!
  - Complex queries can be composed out of simpler ones
- Can we use the DB programming model as a general parallel programming model?
Database Transactions

- Main idea in programming model: everything is a transaction
  - A transaction executes as if it were the only computation accessing the database
- Strong ACID guarantees
  - Atomic – all updates become visible at once, or none
  - Consistent – transactions leave database in consistent state
  - Isolated – no interference with or from other transactions, ensures serializability (transactions appear to execute in some serial order)
  - Durable – once committed, updates are permanent

- Database implementation
  - Controls all accesses, hides complex implementation details
  - Programmer only sees a simple interface
Transactional Memory: Some History

• 1977 – D.B. Lomet (IBM Research, now at Microsoft Research) suggests database transaction model for concurrent programming
  • No practical implementation provided

• 1983 – Kung & Robinson propose optimistic concurrency control for databases

• 1988 – Chang & Mergen describe IBM 801 storage manager
  • HW provided lock bits for each 128 byte range of a page; page tables & TLB extended

• 1993 – Herlihy & Moss describe a hardware proposal for transactional memory
Transaction Memory (TM) Programming Model

• Atomic block
  • Delimits code that should execute in a transaction
  • Ensures no two atomic sections interfere with each other

atomic {
  if (x!=null)
    x.foo();
  y = true;
}

• Dynamically-scoped
  • Code in foo() executes in transaction as well

• Atomic block does not name shared resources
  • Unlike lock-based programming, e.g., lock(x), lock(y)

• 3 possible outcomes
  • Commits, aborts, non-termination
TM System

Source Code:

```java
... atomic {
  ...
  access_shared_data();
  ...
}
...```

Programmer: Specifies atomic regions in source code

TM System: Executes transactions optimistically in parallel

1) Checkpoints execution
2) Detects conflicts
3) Commits or aborts and re-executes
Differences from DB Transactions

• Memory vs. disk
  • Disk access takes 100-10000 X longer than memory access, so
database systems can use relatively heavy-weight software solution
  • In-memory transaction systems need to be much more efficient

• No need for durability
  • Memory is transient anyway => simplifies TM implementations

• Existing languages, libraries and systems
  • Databases are closed systems in which all code executes as a
transaction
  • Programs using TM must coexist with libraries, OSs that do not
use transactions => complicates TM implementations
TM Implementations

- **Hardware TM (HTM)**
  - Changes to computer system and ISA, register checkpoint
  - Extended coherence protocol to track conflicts, special transaction instructions
  - Support for buffering a limited number of memory locations

- **Software TM (STM)**
  - Language runtime (or library) + extensions to specify transaction
  - Exploit current commodity hardware (multicores)
  - Java: DSTM (Marathe et al.), ASTM (Herlihy et al.)
  - Intel’s C++ STM compiler, gcc compiler

- **Hybrid TM (HyTM)**
Caution!

- Programmers can still use `atomic` incorrectly

```c
bool flagA = false;
bool flagB = false;

Thread 1:                        Thread 2:
atomic {
    while (!flagA);
    flagB = true;
}
atomic {
    flagA = true;
    while (!flagB);
}
```

- What’s wrong?
  - Atomic sections can’t be serialized
  - Deadlock occurs
Semantics

- Not yet formally specified!

- Useful ways to reason about TM:
  - Database correctness criteria: serializability
    - Useful for understanding transaction behavior
    - Says nothing about interaction of transactions with code outside of transactions
  - Operational semantics – single-lock atomicity (SLA)
    - Program executes as if all atomic blocks were protected by single global lock
    - Attractive, but does not fully support failure atomicity, certain forms of nesting, etc.
Implementation Basics

- For all (non-stack) write instructions:
  - Track write addresses and values (write set)

- For all (non-stack) read instructions:
  - Track read addresses and values (read set)

- When a transaction completes:
  - Atomically
    - Validate read set (conflict detection)
      - Check that values in read set haven’t been overwritten
    - Commit write set
Implementation Options

• Transaction Granularity
  • Unit of storage over which TM system detects conflicts
    • Similar to notion of cache coherence
    • Word or cache block size for HTM, object for OO STMs

• Direct or Deferred Update
  • Direct – transaction directly modifies the object itself
    • Must log previous value for undo in case of abort
  • Deferred – modify private copy, propagate at commit
  • Both get complicated in the presence of data races

• Optimistic or Pessimistic Concurrency Control
  • TM typically optimistic; need to detect and resolve conflict
Location-Based Conflict Detection

Transaction 1:
Strip versions:

Main Memory:
Strip versions:

Transaction 2:
Strip versions:

Legend:
- Read
- Written

Strips
Location-Based Conflict Detection

**Transaction 1:**
Strip versions:

**Main Memory:**
Strip versions:

**Transaction 2:**
Strip versions:

Legend:
- **Red** Read
- **Green** Written

Strips
Location-Based Conflict Detection

Transaction 1:
Strip versions:

Main Memory:
Strip versions:

Transaction 2:
Strip versions:

Legend:
- Read
- Written

Strips
# Location-Based Conflict Detection

## Transaction 1:
- **Strip versions:**
- **Main Memory:**
  - Strip versions:

## Transaction 2:
- **Strip versions:**

### T2 commit step:
1) Validate Read Set ✓

### Legend:
- **Read**
- **Written**
Location-Based Conflict Detection

Transaction 1:
- Strip versions: [2, 3, 0]

Main Memory:
- Strip versions: [6, 9, 3, 5, 0, 0]

Transaction 2:
- Strip versions: [6, 9, 0, 1]

T2 commit step:
1) Validate Read Set ✓
2) Publish writes, inc versions

Legend:
- Read
- Written
Location-Based Conflict Detection

Transaction 1:
Strip versions:

Main Memory:
Strip versions:

Transaction 2:
Strip versions:

Legend:
- Read
- Written

Strips
Location-Based Conflict Detection

Transaction 1:
- Strip versions: 2 3

Main Memory:
- Strip versions: 0 1 0

Transaction 2:
- Strip versions: 0 1

T1 commit step:
1) Validate Read Set ×

note: all txns must maintain strip versions

Legend:
- Read
- Written
Value-Based Conflict Detection

Transaction 1:

Main Memory: 6 2 3 5

Transaction 2:

Legend:
- Red: Read
- Green: Written
Value-Based Conflict Detection

Transaction 1:

Main Memory:

Transaction 2:

Legend:

- Read
- Written
Value-Based Conflict Detection

Transaction 1:

Main Memory:

Transaction 2:

Legend:
- Red: Read
- Green: Written
Value-Based Conflict Detection

Transaction 1: 

Main Memory: 

Transaction 2: 

T2 commit step:
1) Validate Read Set ✓

Legend:
- Brown: Read
- Green: Written
Value-Based Conflict Detection

Transaction 1:  
Main Memory:  
Transaction 2:  

T2 commit step:
1) Validate Read Set ✓
2) Publish writes

Legend:
- Brown: Read
- Green: Written
Value-Based Conflict Detection

Transaction 1:  
Main Memory:  
Transaction 2:  

Legend:  
- Read  
- Written
Value-Based Conflict Detection

Transaction 1:

Main Memory:

Transaction 2:

T1 commit step:
1) Validate Read Set  ✗

Legend:

Read

Written

Note: no version information needed
TM Weaknesses

- Some operations are hard to abort/retry
  - Essentially anything not idempotent, e.g. I/O
- In practice, TM does not interact well with locking
- Some variables are prone to high conflict rates
  - Frequent true sharing & dependences
- Conflict resolution needs to avoid starving long-running, large transactions
- Poor interaction with standard software tools like debuggers
  - Getting better though ...
• Hardware TM is a reality
  • Sun’s Rock processor, 2009 (canceled by Oracle)
  • IBM Blue Gene/Q, Sequoia supercomputer, 2011
  • IBM POWER8 and newer
  • Intel Transactional Synchronization Extensions (TSX), 2013
    • Available in select Haswell-based processors and newer

• Software TM has performance problems
  • But some applications are a nice fit, e.g. parallel game server
  • With GCC 4.7 and newer, transactional memory support utilizing a hybrid implementation is available