CSCI-GA.3033-017
Special Topics: Multicore Programming

Lecture 12
Transactional Memory

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
Database Background

• Databases have successfully exploited parallel hardware for decades
  • Multi-core computers
  • Multi-machine clusters

• Database performance optimizations
  • Execute queries concurrently
  • Run queries on multiple CPUs

• Query author need not know about parallelism
Database Background

• Database programming model: transactions
  • Computation executes as if it was the only computation accessing the database

• Query results are indistinguishable from serial execution
  • Serializability!

• Transactions allow concurrent operations to access a common database and still produce predictable, reproducible results
Transactional Memory

**Transactions**
- Inspired by databases
- Coordinate concurrent access
- Characteristics: ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability

**Memory**
- Main data storage during typical multicore programs
- Computations wrapped in transactions
- Much more primitive than database transactions
Transactions: ACID

A transaction is a sequence of actions that appears indivisible and instantaneous to an outside observer.

**Atomicity**
All constituent actions in a transaction complete successfully, or none appears to start executing.

**Consistency**
(Semantics depend on the application)

**Isolation**
Transactions do not interfere with each other while they are running, regardless of whether they are operating in parallel or not.

**Durability**
Once a transaction commits, its result is permanent.
### Transaction Example #1

<table>
<thead>
<tr>
<th>Thread 1 (Query 1)</th>
<th>Thread 2 (Query 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>transaction.begin(); a = a – 20; b = b + 20; a = a – b; c = c + 20; transaction.end();</td>
<td>transaction.begin(); c = c – 30; a = a + 30; transaction.end();</td>
</tr>
</tbody>
</table>

Thread 1’s (Query 1’s) accesses and updates to a, b, and c are atomic.

Thread 2 (Query 2) sees either all or none of Thread 1’s (Query 1’s) updates.
Transaction Example #2

```java
int x = 0, y = 0;

Thread 1 (Query 1)
transaction.begin();
x = 42;
y = 42;
transaction.end();

Thread 2 (Query 2)
transaction.begin();
tmp1 = x;
tmp2 = y;
transaction.end();
```

What values does Thread 2 (Query 2) see?
Transaction Example #3

Transaction A
- load 0xabba
  ...
- store 0xbeef

Transaction B
- load 0xabba
  ...
- load 0xcabb

Transaction C
- load 0xbeef
  ...
- load 0xcabb

Transaction C
- load 0xbeef
  ...
- load 0xcabb
Who Uses Transactional Memory?

• Programmer
• Compiler designer
  • To implement high-level language features
• Ancillary uses (general parallel programming)
  • Error recovery
  • Realtime programming
  • Multitasking
Transactional Memory in GCC/G++


```cpp
__transaction_relaxed {
  statements
}

__transaction_atomic {
  statements
}

__transaction_cancel;
```

• Works with both software and hardware transactional memory (more on that later)
Atomic Transactions

• Other code cannot see intermediate results from atomic transaction.
• Do not see intermediate results from other code.
• Limited subset of transaction-safe operations
• Can be cancelled
Relaxed Transactions

• Other code cannot see intermediate results from atomic transaction.
• Do not see intermediate results from other code.
• Any operation allowed
• Cannot be cancelled
• Act as if all relaxed transactions use a single mutex (serialization)
__attribute__((transaction_callable)) double foo(void);

double bar(void) {
    ...
    __transaction_atomic {
        ...
        foo();
    }
    ...
}
Concurrency Control: Conflict Detection/Resolution

• A conflict occurs when two transactions perform conflicting operations on the same piece of memory
  • 2 writes
  • 1 read, 1 write

• The conflict is detected by the underlying Transactional Memory (TM) system

• The conflict is resolved by the TM, e.g. by aborting, delaying, or repeating one or more transactions.
Concurrency Control: Conflict Detection/Resolution

- Motivating example: double-ended queue

```c
void pushLeft(Dqueue *q, int val) {
    QNode *qn = new Qnode;
    qn->val = val;
    QNode *leftSentinel = q->left;
    QNode *oldLeftNode =
        leftSentinel->right;
    qn->left = leftSentinel;
    qn->right = oldLeftNode;
    leftSentinel->right = qn;
    oldLeftNode->left = qn;
}
```

```c
void pushLeft(Dqueue *q, int val) {
    QNode *qn = new Qnode;
    qn->val = val;
    __transaction_atomic {
        QNode *leftSentinel = q->left;
        QNode *oldLeftNode =
            leftSentinel->right;
        qn->left = leftSentinel;
        qn->right = oldLeftNode;
        leftSentinel->right = qn;
        oldLeftNode->left = qn;
    }
}
```
Resolving Conflicts: Pessimistic Concurrency Control

- Some TMs use optimistic control, others use pessimistic control

Thread 1

Conflict occurs, is detected, and is resolved by delaying Thread 2’s transaction

Thread 2
Resolving Conflicts: Optimistic Concurrency Control

Some TMs use optimistic control, others use pessimistic control.

Conflict occurs, is detected, and is resolved by reexecuting one or both transactions.
## Classifying Available TMs

<table>
<thead>
<tr>
<th>Read</th>
<th>Write -&gt;</th>
<th>Optimistic</th>
<th>Pessimistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimistic</td>
<td></td>
<td>TCC</td>
<td>Intel C++ STM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TL2</td>
<td>Intel Java STM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SigTM</td>
<td>HASTM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Microsoft STM.NET</td>
</tr>
<tr>
<td>Pessimistic</td>
<td></td>
<td>LogTM</td>
<td>Intel C++ STM</td>
</tr>
</tbody>
</table>
Version Management

• How to handle writes before a transaction commit?

• Eager version management
  • Transaction directly modifies memory
  • Maintains undo log of overwritten data
  • Requires pessimistic concurrency control

• Lazy version management
  • Transaction records redo log of writes to apply
  • Updates delayed until transaction commits
  • Works with optimistic concurrency control
Conflict Detection

- Pessimistic approach: Easy
  - Locks

- Optimistic approach: Harder
  - Granularity of conflict (cache line, object, etc.)
  - Detection interval
    - When read/write set is declared: eager conflict detection
    - On validation: up to several instances throughout transaction
    - On commit: lazy conflict detection

- Conflicting accesses
  - Concurrent transactions
  - Concurrent and committed transactions
What Can Go Wrong?

Thread 1 (Query 1)
```c
__transaction_atomic {
    x = 1;
}
```

Thread 2 (Query 2)
```c
__transaction_atomic {
    int tmp1 = x;
    while (tmp1 == 0) {
    }
}
```
Providing TM

Software Transactional Memory (STM)

- User Program
- STL
- OS
- Hardware
  - STM Lib

Hardware Transactional Memory (HTM)

- User Program
- STL
- OS
- Hardware
  - HTM Hardware and Microcode
Software Transactional Memory (STM)

• Compiler instruments code with transaction prolog, epilog, and read/write functions

• Runtime tracks memory accesses, detects conflicts, and commits/aborts execution.

```c
__transaction_atomic {
  r = x
  y = r + 1;
}
```

```c
td = getTxnDesc();
txnBegin(td);
r = txnRead(td, &x);
txnWrite(td, &y, r + 1);
txnEnd(td);
```
Software Transactional Memory (STM)

- Components:
  - Transaction descriptor: per-transaction data structure
  - Undo log (eager version management)
  - Redo log (lazy version management)
  - Readset/writeset: tracks memory locations the transaction has read or written.
Hardware Transactional Memory (HTM)

- Three flavors
  1. Full hardware TM
  2. Software and hardware TM used together
  3. Hardware extensions to accelerate STM
Hardware Transactional Memory (HTM) Functions

- Identify memory locations for transactional accesses
- Manage readsets and writesets of transactions
- Detect and resolve data conflicts
- Manage architectural register state
- Commit or abort transactions
## Supporting Transactions

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Satisfied By</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffering</td>
<td>Transactional cache</td>
</tr>
<tr>
<td>Conflict Detection</td>
<td>Cache coherence protocol</td>
</tr>
<tr>
<td>Abort/Recovery</td>
<td>Invalidate transactional cache line</td>
</tr>
<tr>
<td>Commit</td>
<td>Validate transactional cache line</td>
</tr>
</tbody>
</table>
Hardware Transactional Memory (HTM)

- Extensions to the instruction set
- Tracking readsets and writesets using caches and buffers
- Coherence messages trigger conflict detection
- Nearly all existing HTMs perform eager conflict detection
IBM BlueGene/Q and HTM

- Multicore 64-bit PowerPC-based SoC
- 18 cores
  - 16 for computations
  - 1 for OS
  - 1 spare
- HTM in 32MB L2 cache
Intel Haswell and HTM

• Intel Haswell (2013): First consumer CPU to support HTM
  • Later disabled in Haswell and early Broadwell due to bugs

• Transactional Synchronization Extensions (TSX)
  • Hardware Lock Ellision (HLE): Backwards-compatible conversion from lock-based programs to transactional programs

• Restricted Transactional Memory (RTM): Complete, non-backwards-compatible transactional memory.
STM Advantages Over HTM

- More flexible than hardware
  - Permits implementation of a wider variety, more sophisticated algorithms
- Easier to modify and evolve
- Integrate more easily with existing systems and language features, such as garbage collection
- Fewer intrinsic limitations imposed by fixed-size hardware structures, such as caches.
HTM Advantages Over STM

- Execute applications with lower overhead
- Less reliant on compiler optimizations to provide performance
- Better power and energy profiles
- Can treat all memory accesses in a transaction as *implicitly* transactional
- Strong isolation without changing non-transactional memory accesses
- Well-suited to languages (like C/C++) without dynamic compilation, garbage collection, etc.
Conclusion

- Transactional Memory: Strong candidate for parallel programming
- Many different implementations, poor portability
  - Still a work-in-progress; details vary between languages and processors
- Trades bandwidth for simplicity and latency tolerance
- Transactions eliminate locks
  - Transactions are inherently atomic
  - Catches most common parallel programming errors
- Shared memory consistency is simplified
- Shared memory coherence is simplified