Lecture 11: Transactional Memory
active research: here there be dragons

CSC 469H1F / CSC 2208H1F
Fall 2007
Angela Demke Brown

Challenges of Synchronization

- Two major issues:
  - Performance
  - Scalability
  - Base cost
  - We've looked at some techniques that address this
    - 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 task of moving an item from one stack to another
    - Need to expose stack locking mechanism

“Magic” Wish List

- Let programmers express desired outcome (i.e. “This block of code should appear atomic“)
- Let run-time system or hardware support make it happen
- Allow abstractions to hide implementation and be composable

A new programming model is needed
Database Transactions

- Database systems allow multiple queries to run in parallel
- Query authors don't worry about concurrency
- Complex queries can be composed out of simpler ones
- **Key Programming Model:** everything is a transaction
  - A transaction executes as if it were the only computation accessing the database
  - Atomic - all updates become visible, or none
  - Consistent - transactions leave database in consistent state
  - Isolated - no interference with or from other transactions
  - Durable - once committed, updates are permanent

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**

Transactional Memory (TM)

**Source Code:**
```
atomic {
  ... access_shared_data(); ...
}
```

**Transactions:**
```
? ?
  ⬇️ ⬇️ ⬇️
  ⬆️ ⬆️ ⬆️
```

**TM System:** Execute transactions optimistically in parallel
1. Checkpoints execution
2. Detects conflicts
3. Commits or aborts and re-executes

Programmer: Specifies threads/transactions in source code

Differences from DB Transactions

- **Memory vs. disk**
  - Disk access takes 100X longer than memory access → database systems can use relatively heavy-weight software solutions
- **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
**TM Implementations**

- **Hardware TM (HTM)**
  - Changes to computer system and ISA
  - Extra cache to buffer writes, extended coherence protocol to track conflicts, special transaction instructions
  - Support for limited number of memory locations
- **Software TM (STM)**
  - Language runtime (or library) + extensions to specify trans.
  - Exploit current commodity hardware (multicores)
  - Get experience with transactional programming model
  - Java: DSTM (Marathe et al.), ASTM (Herlihy et al.)
  - C/C++: McRT-STM (Saha et al.), TL2 (Dice et al.), RSTM
- **Hybrid TM (HyTM)**

**Programming Constructs**

- **Atomic block**

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

- Delimits code that should execute in a transaction
- Dynamically-scoped – code in foo executes in transaction as well
- Does not name shared resources (unlike monitors or lock-based programming)
- 3 possible outcomes – commits, aborts, non-termination

**Caution!**

- Programmers can still use this construct incorrectly

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

- Deadlock results

**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
  - Program executes as if all atomic blocks were protected by single global lock
  - Does not capture failure atomicity
  - Can describe effect of non-transactional accesses
  - Conflict and data race concepts from lock-based programming
Additional Considerations

- **Weak vs. Strong Atomicity (or isolation)**
  - **Weak** – conflicting memory reference outside transaction may not follow protocols of TM system
  - **Strong** – all operations outside atomic blocks are converted into individual transactions, guaranteeing all accesses obey TM protocols
  - Equivalent w.r.t semantics of single-lock atomicity in programs without data races
- **Nested transactions - required for composability**
  - **Flattened** - inner transaction essentially removed
  - **Closed** - effect of inner transaction only visible to surrounding one; abort affects only inner
  - **Open** - effect of inner becomes visible to all after commit; abort affects only inner

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)
    - Commit write set

Implementation Options

- **Transaction Granularity**
  - Unit of storage over which TM system detects conflicts
  - Akin to notion of cache coherence
  - Word or block typical for HTM, object common for STMs that extend OO language
- **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:
- Strips
- Read
- Written
**Location-Based Conflict Detection**

Transaction 1:  
Strip versions:  
Main Memory:  
Strip versions:  
Transaction 2:  
Strip versions:  

Legend:  
![Read](image)  
![Written](image)  

Note: all transactions must maintain strip version #s

**Value-Based Conflict Detection**

Transaction 1:  
Main Memory:  
Transaction 2:  

Legend:  
![Read](image)  
![Written](image)  

Commit step 1) Validate Read Set  
Commit step 2) Publish Writes (and inc version #s)  
Abort!
### Value-Based Conflict Detection

#### Transaction 1:
```
2 3 5
```

#### Main Memory:
```
6 2 3 5
```

#### Transaction 2:
```
6 9
```

---

#### Commit step 1) Validate Read Set
- Verify that all reads in the read set are consistent with the main memory values.

#### Commit step 2) Publish Writes
- If validation passes, commit the transaction.
- If validation fails, abort the transaction.

---

#### Note: no version information to maintain