Reviews

Due Last Time (March 21, before class)

Due Saturday (March 26)
Speculative Parallelization Concepts

- **Idea:** Execute threads unsafely in parallel
  - Threads can be from a sequential or parallel application

- Hardware or software monitors for data dependence violations

- If data dependence ordering is violated
  - Offending thread is squashed and restarted

- If data dependences are not violated
  - Thread commits
  - If threads are from a sequential order, the sequential order needs to be preserved → threads commit one by one and in order
Multiscalar Processors (ISCA 1992, 1995)

- Exploit “implicit” thread-level parallelism within a serial program
- Compiler divides program into tasks
- Tasks scheduled on independent processing resources
- Hardware handles register dependences between tasks
  - Compiler specifies which registers should be communicated between tasks
- Memory speculation for memory dependences
  - Hardware detects and resolves misspeculation

Four Issues in Speculative Parallelization

- How to deal with unavailable values: predict vs. wait
- How to deal with speculative updates: Logging/buffering
- How to detect conflicts
- How and when to abort/rollback or commit
Transactional Memory

- **Idea:** Programmer specifies code to be executed atomically as transactions. Hardware/software guarantees atomicity for transactions.

- Motivated by difficulty of lock-based programming
Locking Issues

- Locks: objects only one thread can hold at a time
  - Organization: lock for each shared structure
  - Usage: (block) → acquire → access → release

- Correctness issues
  - Under-locking → data races
  - Acquires in different orders → deadlock

- Performance issues
  - Conservative serialization
  - Overhead of acquiring
  - Difficult to find right granularity
  - Blocking
Locks vs. Transactions

Lock issues:
- Under-locking
- Acquires in different orders
- Blocking
- Conservative serialization

How transactions help:
+ Simpler interface
+ No ordering
+ Can cancel transactions
+ Serialization only on conflicts
Transactional Memory

- Transactional Memory (TM) allows arbitrary multiple memory locations to be updated atomically

- Basic Mechanisms:
  - Isolation: Track read and writes, detect when a data conflict occurs between transactions
  - Version management: Record new/old values
  - Atomicity: Commit new values or abort back to old values

- Issues the same as other speculative parallelization schemes
  - Logging/buffering
  - Conflict detection
  - Abort/rollback
  - Commit
Four Issues in Transactional Memory

- How to deal with unavailable values: predict vs. wait
- How to deal with speculative updates: Logging/buffering
- How to detect conflicts
- How and when to abort/rollback or commit
Many Variations of TM

- **Software**
  - High performance overhead, but no virtualization issues

- **Hardware**
  - What if buffering is not enough?
  - Context switches, I/O within transactions?
  - Need support for virtualization

- **Hybrid HW/SW**
  - Switch to SW to handle large transactions and buffer overflows
Initial TM Ideas

- **Load Linked Store Conditional Operations**
  - Lock-free atomic update of a single cache line
  - Used to implement non-blocking synchronization
    - Alpha, MIPS, ARM, PowerPC
  - Load-linked returns current value of a location
  - A subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to the location

- **Herlihy and Moss, ISCA 1993**
  - Instructions explicitly identify transactional loads and stores
  - Used dedicated transaction cache
  - Size of transactions limited to transaction cache
Our transactions are intended to replace short critical sections. For example, a lock-free data structure would typically be implemented in the following stylized way (see Section 5 for specific examples). Instead of acquiring a lock, executing the critical section, and releasing the lock, a process would:

1. use LT or LTX to read from a set of locations,
2. use VALIDATE to check that the values read are consistent,
3. use ST to modify a set of locations, and
4. use COMMIT to make the changes permanent. If either the VALIDATE or the COMMIT fails, the process returns to Step (1).
TM Research Issues

- How to virtualize transactions (without much complexity)
  - Ensure long transactions execute correctly
  - In the presence of context switches, paging

- Handling I/O within transactions
  - No problem with locks

- Semantics of nested transactions (more of a language/programming research topic)

- Does TM increase programmer productivity?
  - Does the programmer need to optimize transactions?
Readings: Data Flow

- **Review papers**

- **Recommended papers**