18-742 Fall 2012
Parallel Computer Architecture
Lecture 16: Speculation II

Prof. Onur Mutlu
Carnegie Mellon University
10/12/2012

Past Due: Review Assignments

- Was Due: Tuesday, October 9, 11:59pm.
- Sohi et al., "Multiscalar Processors," ISCA 1995.
- Was Due: Thursday, October 11, 11:59pm.
- Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
- Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," MICRO 1999.

New Review Assignments

- Due: Sunday, October 14, 11:59pm.
- Patel, "Processor-Memory Interconnections for Multiprocessors," ISCA 1979.

- Due: Tuesday, October 16, 11:59pm.
- Moscibroda and Mutlu, "A Case for Bufferless Routing in On-Chip Networks," ISCA 2009.

Project Milestone I Due (I)

- Deadline: October 20, 11:59pm (next Saturday)
- Format: Slides (no page limit) + presentation (10-15min)
- What you should turn in:
  - PPT/PDF slides describing the following:
    - The problem you are solving + your goal
    - Your solution ideas + strengths and weaknesses
    - **Your methodology to test your ideas**
    - **Concrete mechanisms you have implemented so far**
    - **Concrete results you have so far**
    - **What will you do next?**
    - What hypotheses you have for future?
    - How close were you to your target?

Project Milestone I Due (II)

- Next week (Oct 22-26)
  - Sign up for Milestone I presentation slots (10-15 min/group)
- Make a lot of progress and find breakthroughs
- Example milestones:
  - http://www.ece.cmu.edu/~ece742/2011spring/doku.php?id=project
  - http://www.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=milestone1_ausavarungnirun_meza_yoon.pptx
  - http://www.ece.cmu.edu/~ece742/2011spring/lib/exe/fetch.php?media=milestone1_tumanov_lin.pdf

Last Lecture

- Slipstream processors
- Dual-core execution
- Thread-level speculation
- Key concepts in speculative parallelization
- Multiscalar processors

Today

- More multiscalar
- Speculative lock elision
- More speculation

Readings: Speculation

- Required
  - Sohi et al., "Multiscalar Processors," ISCA 1995.
  - Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
- Recommended
  - Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.
  - Colohan et al., "A Scalable Approach to Thread-Level Speculation," ISCA 2000.
  - Akkary and Driscoll, "A dynamic multithreading processor," MICRO 1998.
- Reading list will be updated…

More Multiscalar

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

- Franklin and Sohi, "The expandable split window paradigm for exploiting fine-grain parallelism," ISCA 1992.
- Sohi et al., "Multiscalar processors," ISCA 1995.

Multiscalar vs. Large Instruction Windows

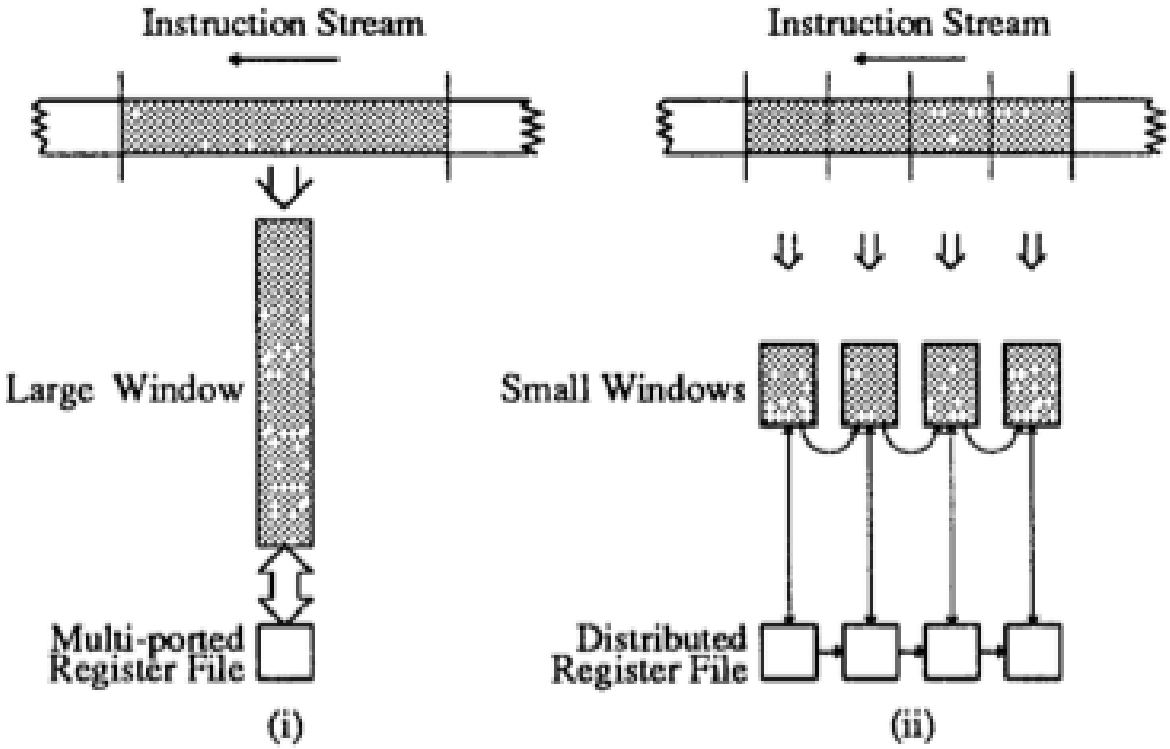
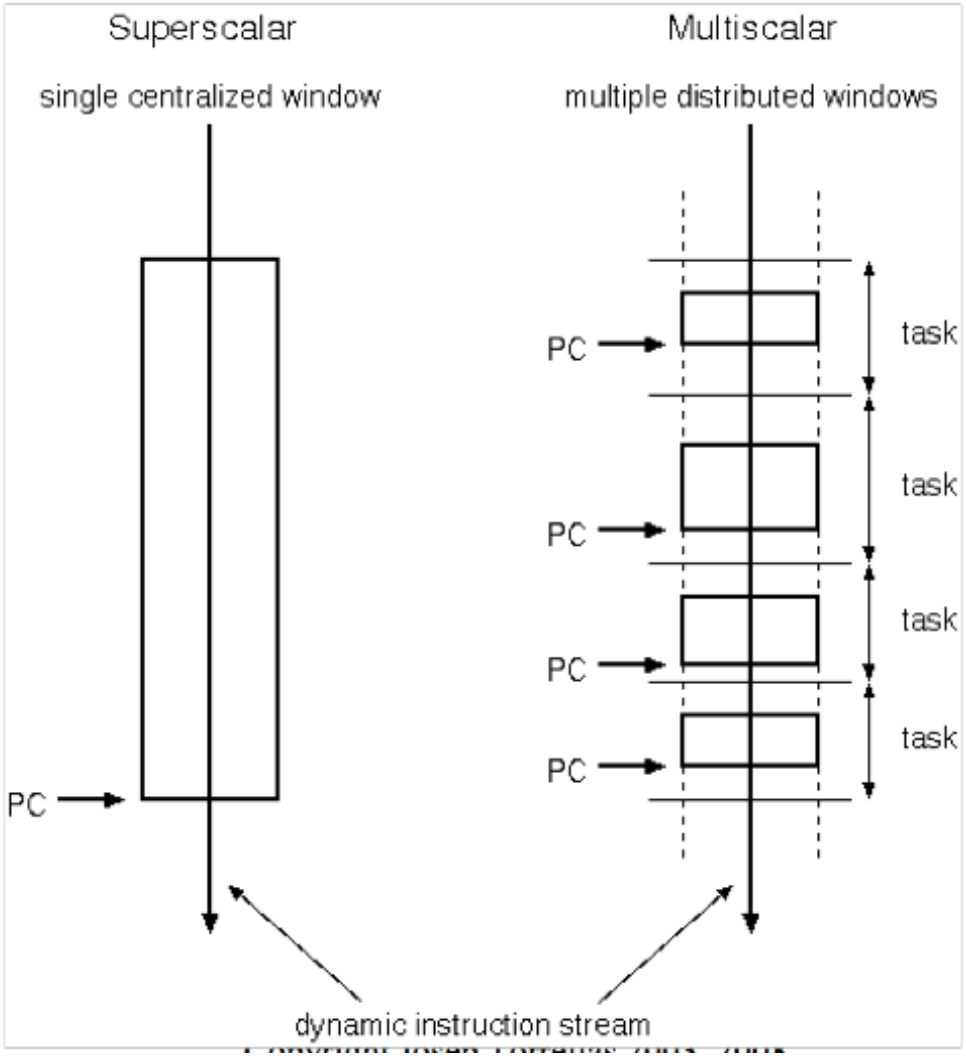


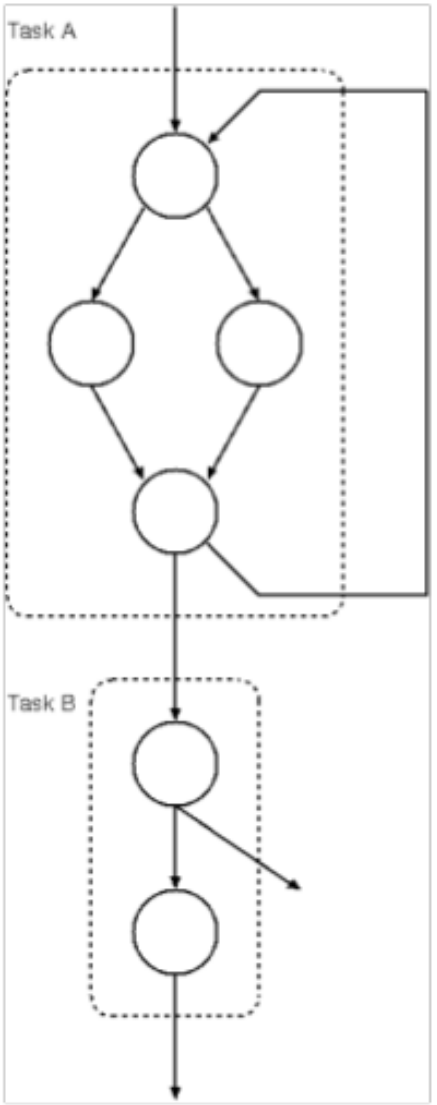
Figure 1: Splitting a large window of instructions into smaller windows
(i) A single large window (ii) A number of small windows

Multiscalar Model of Execution

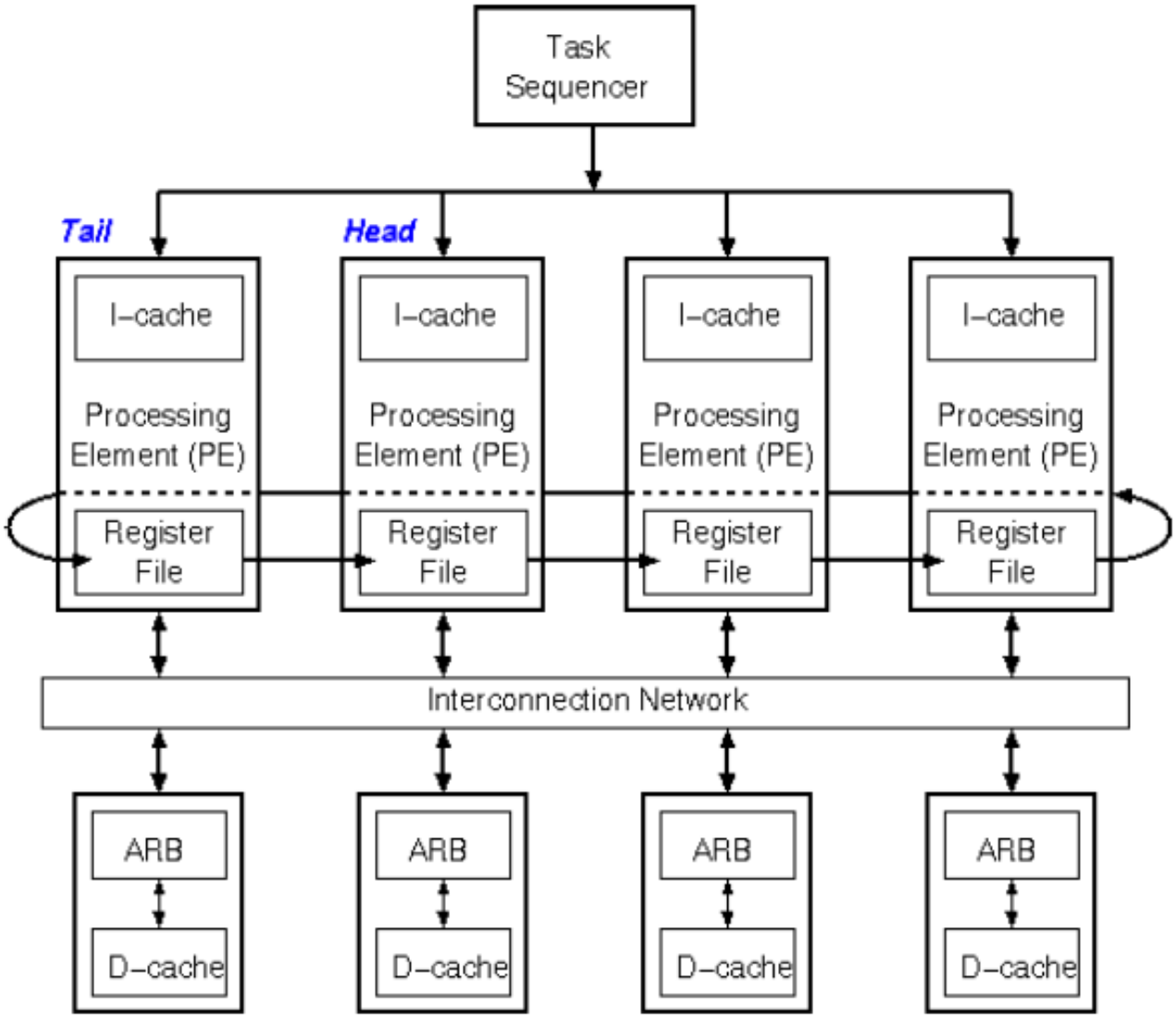


Multiscalar Tasks

- A task is a subgraph of the control flow graph (CFG)
  - e.g., a basic block, multiple basic blocks, loop body, function
- Tasks are selected by compiler and conveyed to hardware
- Tasks are predicted and scheduled by processor
- Tasks may have data and/or control dependences



Multiscalar Processor



Multiscalar Compiler

- Task selection: partition CFG into tasks
  - Load balance
  - Minimize inter-task data dependences
  - Minimize inter-task control dependences
    - By embedding hard-to-predict branches within tasks

- Convey task and communication information in the executable
  - Task headers
    - create_mask (1 bit per register)
      - Indicates all registers that are *possibly modified or created by the task (better: live-out of the task)*
      - *Don't forward instances received from prior tasks*
    - *PCs of successor tasks*
  - Release instructions: Release a register to be forwarded to a receiving task

Multiscalar Program Example

```
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search for the symbol in the list */
    for (list = listhd; list; list = LNEXT(list)) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
    }

    /* if symbol not found in the list, add to the tail */
    if (!list) {
        addlist(symbol);
    }
}
```

Figure 3: An Example Code Segment.

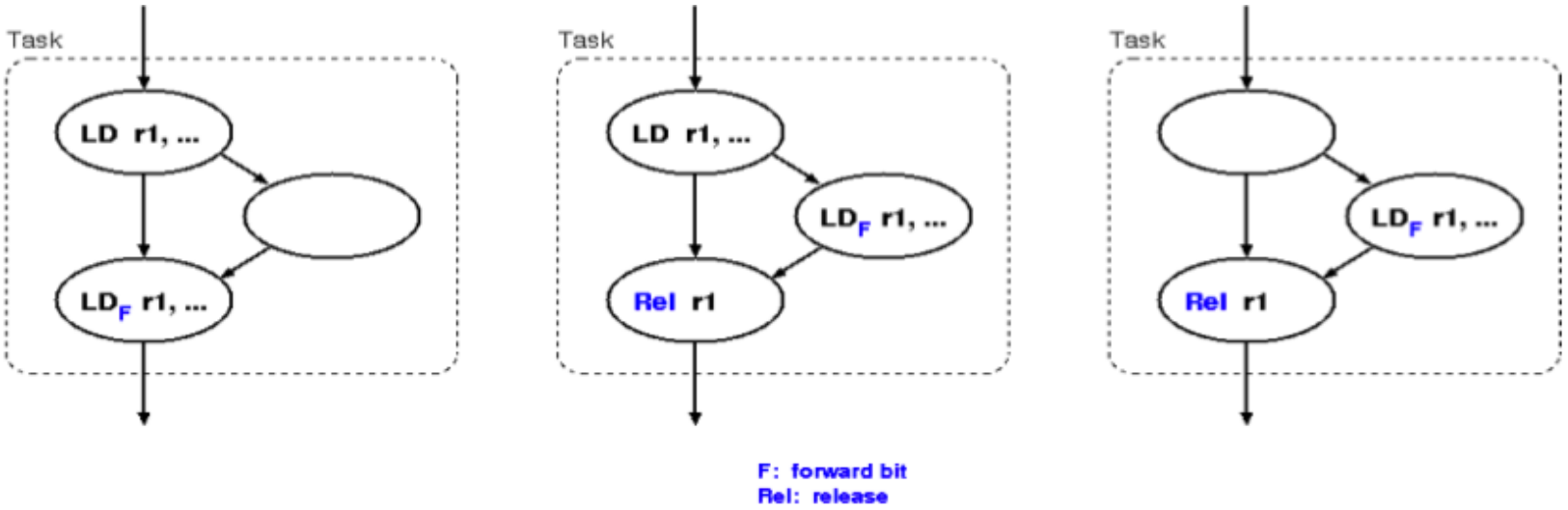
```
Targ Spec       Branch, Branch
Targ1           OUTER
Targ2           OUTERFALLOUT
Create mask     $4,$8,$17,$20,$23
```

```
                                        Forward Bits   Stop Bits
OUTER:
        addu    $20, $20, 16            F
        ld      $23, SYMVAL-16($20)     F
        move    $17, $21
        beq     $17, $0, SKIPINNER
INNER:
        ld      $8, LELE($17)
        bne     $8, $23, SKIPCALL
        move    $4, $17
        jal     process
        jump    INNERFALLOUT
SKIPCALL:
        ld      $17, NEXTLIST($17)
        bne     $17, $0, INNER
INNERFALLOUT:
        release $8, $17
        bne     $17, $0, SKIPINNER
        move    $4, $23                 F
        jal     addlist
SKIPINNER:
        release $4
        bne     $20, $16, OUTER                Stop Always
OUTERFALLOUT:
```

Figure 4: An Example of a Multiscalar Program.

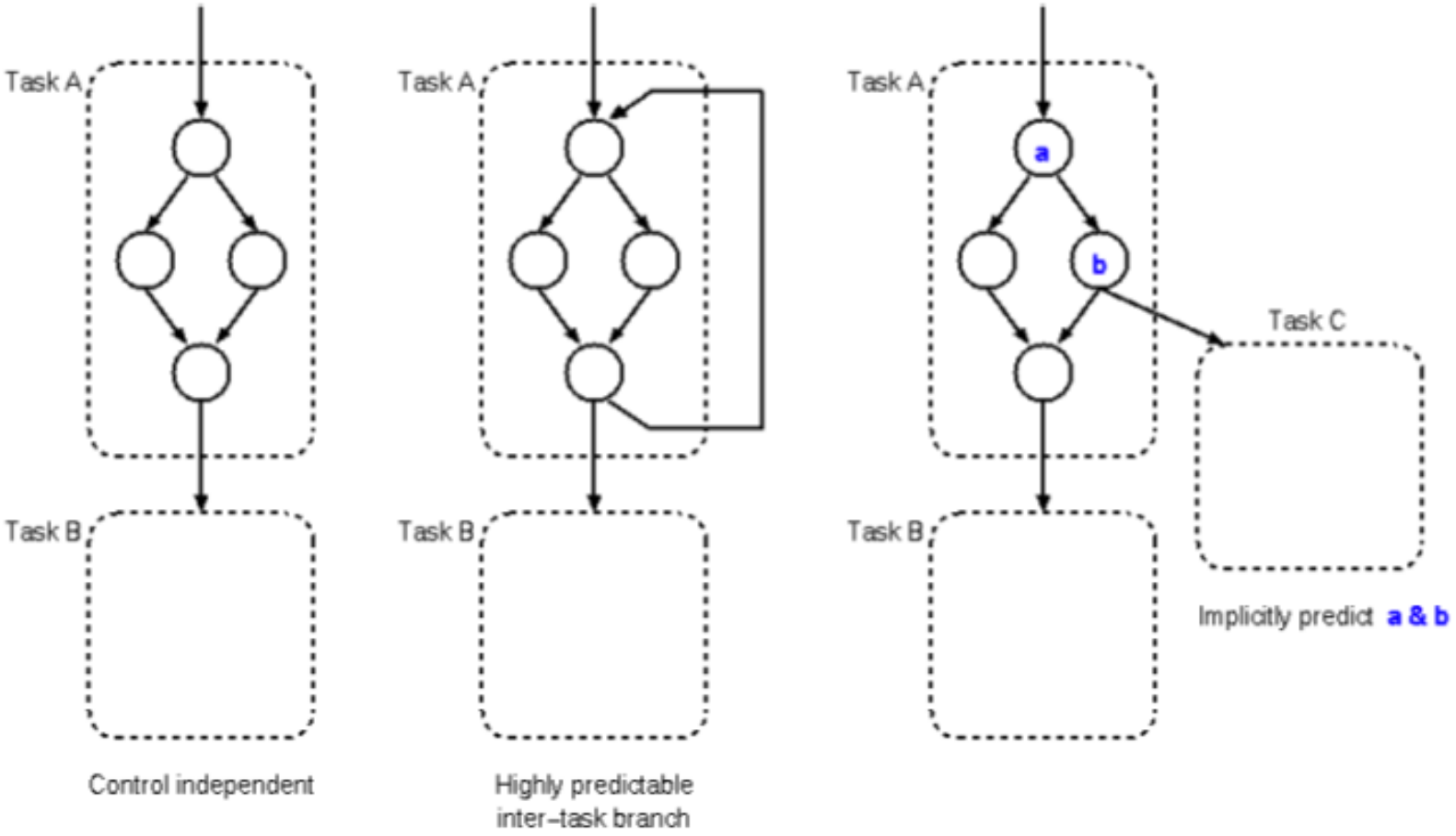
Forwarding Registers Between Tasks

- Compiler must identify the last instance of write to a register within a task
  - Opcodes that write a register have additional forward bit, indicating the instance should be forwarded
  - Stop bits - indicate end of task
  - Release instruction
    - tells PE to forward the register value



Task Sequencing

- Task prediction analogous to branch prediction
- Predict inter-task control flow



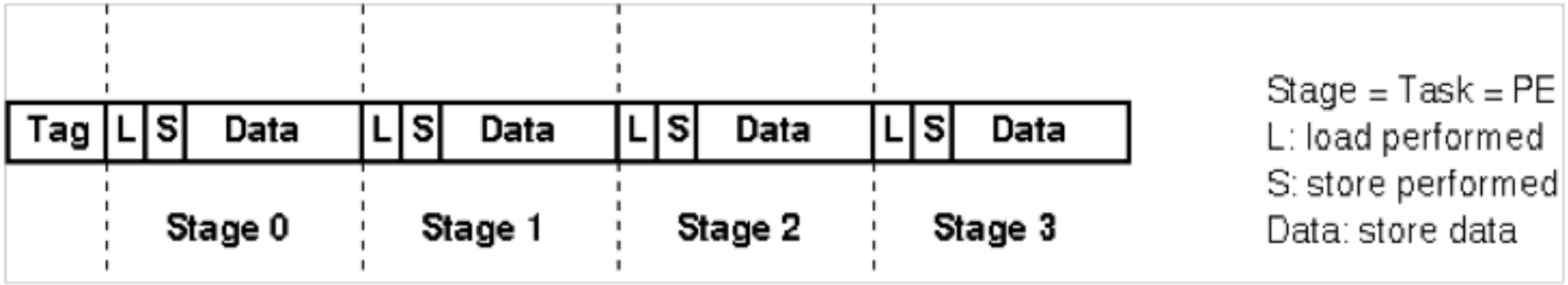
Handling Inter-Task Dependences

- Control dependences
  - Predict
  - Squash subsequent tasks on inter-task misprediction
    - Intra-task mispredictions do not need to cause flushing of later tasks
- Data dependences
  - Register file: mask bits and forwarding (stall until available)
  - Memory: address resolution buffer (speculative load, squash on violation)

Address Resolution Buffer

- Multiscalar issues loads to ARB/D-cache as soon as address is computed
- ARB is organized like a cache, maintaining state for all outstanding load/store addresses
- Franklin and Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," IEEE TC 1996.

- An ARB entry:



Address Resolution Buffer

- Loads
  - ARB miss: data comes from D-cache (no prior stores yet)
  - ARB hit: get most recent data to the load, which may be from D-cache, or nearest prior task with S=1

- Stores
  - ARB buffers speculative stores
  - If store from an older task finds a load from a younger task to the same address → misspeculation detected
  - When a task commits*, commit all of the task's stores into the D-cache*

Address Resolution Buffer

- Franklin and Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," IEEE TC 1996.

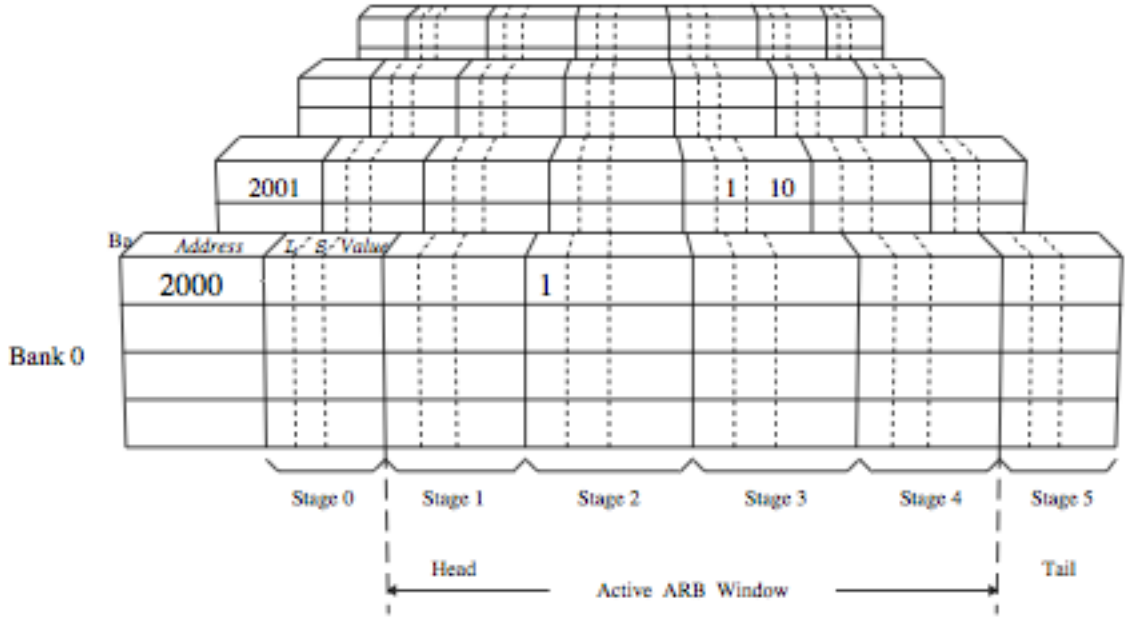


Figure 1: A 4-Way Interleaved, 6-stage ARB

Memory Dependence Prediction

- ARB performs memory renaming
- However, it does not perform dependence prediction
  - Can reduce intra-task dependency flushes by accurate memory dependence prediction

- Idea: Predict whether or not a load instruction will be dependent on a previous store (and predict which store). Delay the execution of the load if it is predicted to be dependent.

- Moshovos et al., "Dynamic Speculation and Synchronization of Data Dependences," ISCA 1997.
- Chrysos and Emer, "Memory Dependence Prediction using Store Sets," ISCA 1998.

740: Handling of Store-Load Dependencies

- A load's dependence status is not known until all previous store addresses are available.

- How does the OOO engine detect dependence of a load instruction on a previous store?
  - Option 1: Wait until all previous stores committed (no need to check)
  - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address

- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
  - Option 1: Assume load independent of all previous stores
  - Option 2: Assume load dependent on all previous stores
  - Option 3: Predict the dependence of a load on an outstanding store

740: Memory Disambiguation

- Option 1: Assume load independent of all previous stores

  + Simple and can be common case: no delay for independent loads

  -- Requires recovery and re-execution of load and dependents on misprediction

- Option 2: Assume load dependent on all previous stores

  + No need for recovery

  -- Too conservative: delays independent loads unnecessarily

- Option 3: Predict the dependence of a load on an outstanding store

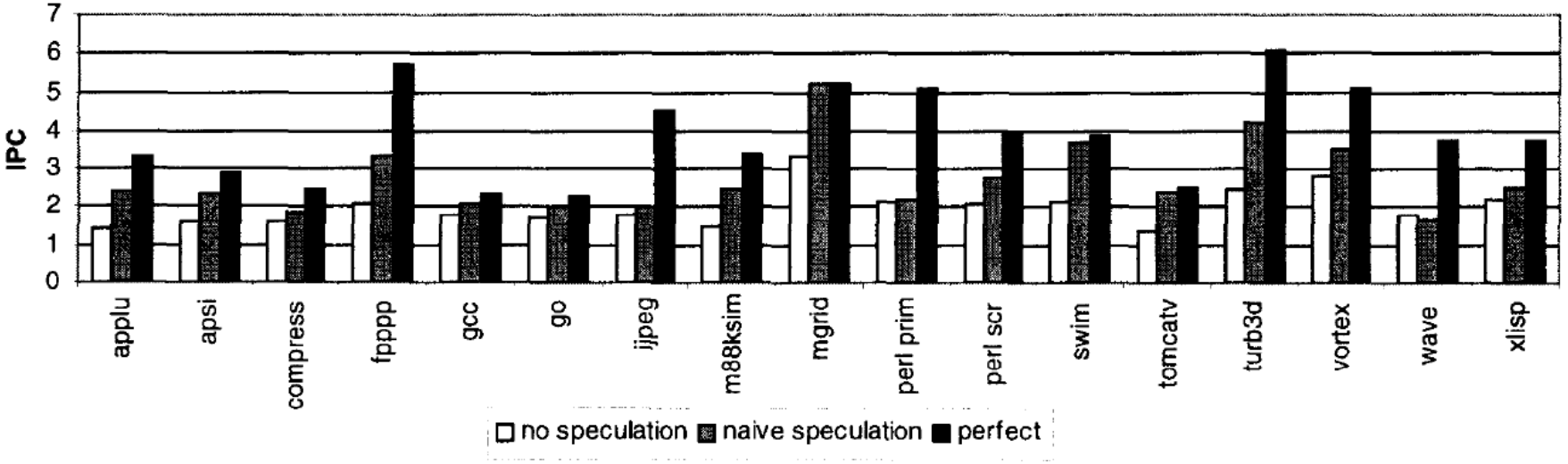
  + More accurate. Load store dependencies persist over time

  -- Still requires recovery/re-execution on misprediction

  - Alpha 21264 : Initially assume load independent, delay loads found to be dependent
  - Moshovos et al., "Dynamic speculation and synchronization of data dependences," ISCA 1997.
  - Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.

740: Memory Disambiguation

- Chrysos and Emer, "Memory Dependence Prediction Using Store Sets," ISCA 1998.



- Predicting store-load dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance

Multiscalar Comparisons and Questions

- vs. superscalar, out-of-order?
- vs. multi-core?
- vs. CMP and SMT-based thread-level speculation mechanisms
  - What is different in multiscalar hardware?

- Scalability of fine-grained register communication
- Scalability of memory renaming and dependence speculation

More Speculation

Speculation to Improve Parallel Programs

- Goal: reduce the impact of serializing bottlenecks
  - Improve performance
  - Improve programming ease

- Examples
  - Herlihy and Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," ISCA 1993.
  - Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.
  - Martinez and Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," ASPLOS 2002.
  - Rajwar and Goodman, "Transactional lock-free execution of lock-based programs," ASPLOS 2002.

Speculative Lock Elision

- Many programs use locks for synchronization
- Many locks are not necessary
  - Stores occur infrequently during execution
  - Updates can occur to disjoint parts of the data structure

- Idea:
  - Speculatively assume lock is not necessary and execute critical section without acquiring the lock
  - Check for conflicts within the critical section
  - Roll back if assumption is incorrect

- Rajwar and Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," MICRO 2001.

Dynamically Unnecessary Synchronization

a)

```
1.LOCK(locks->error_lock)
2.if (local_error > multi->err_multi)
3.      multi->err_multi = local_err;
4.UNLOCK(locks->error_lock)
```

b)

Thread 1

```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X)
if (!var)
    hash_tbl.add(X);
UNLOCK(hash_tbl.lock)
```

Thread 2

```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(Y)
if (!var)
    hash_tbl->add(Y);
UNLOCK(hash_tbl.lock)
```

Figure 1. ***Two examples of potential parallelism masked by dynamically unnecessary synchronization.***

Speculative Lock Elision: Issues

- Either the entire critical section is committed or none of it

- How to detect the lock
- How to keep track of dependencies and conflicts in a critical section
  - Read set and write set
- How to buffer speculative state
- How to check if "atomicity" is violated
  - Dependence violations with another thread
- How to support commit and rollback

Maintaining Atomicity

- If atomicity is maintained, all locks can be removed
- Conditions for atomicity:
  - Data read is not modified by another thread until critical section is complete
  - Data written is not accessed by another thread until critical section is complete

- If we know the beginning and end of a critical section, we can monitor the memory addresses read or written to by the critical section and check for conflicts
  - Using the underlying coherence mechanism

SLE Implementation

- Checkpoint register state before entering SLE mode
- In SLE mode:
  - Store: Buffer the update in the write buffer (do not make visible to other processors), request exclusive access
  - Store/Load: Set "access" bit for block in the cache
  - Trigger misspeculation on some coherence actions
    - If external invalidation to a block with "access" bit set
    - If exclusive access to request to a block with "access" bit set
  - If not enough buffering space, trigger misspeculation
- If end of critical section reached without misspeculation, commit all writes (needs to appear instantaneous)

Accelerated Critical Sections (ACS) vs. SLE

- ACS Advantages over SLE
  - + Speeds up each individual critical section
  - + Keeps shared data and locks in a single cache (improves shared data and lock locality)
  - + Does not incur re-execution overhead since it does not speculatively execute critical sections in parallel

- ACS Disadvantages over SLE
  - - Needs transfer of private data and control to a large core (reduces private data locality and incurs overhead)
  - - Executes non-conflicting critical sections serially
  - - Large core can reduce parallel throughput (assuming no SMT)

ACS vs. SLE

8.2 Hiding the Latency of Critical Sections

Several proposals try to hide the latency of a critical section by executing it speculatively with other instances of the same critical section *as long as they do not have data conflicts with each other*. Examples include transactional memory (TM) [14], speculative lock elision (SLE) [33], transactional lock removal (TLR) [34], and speculative synchronization (SS) [29]. SLE is a hardware technique that allows multiple threads to execute the critical sections speculatively without acquiring the lock. If a data conflict is detected, only one thread is allowed to complete the critical section while the remaining threads roll back to the beginning of the critical section and try again. TLR improves upon SLE by providing a timestamp-based conflict resolution scheme that enables lock-free execution. ACS is partly orthogonal to these approaches due to three major reasons:

Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009, IEEE Micro Top Picks 2010.

ACS vs. Transactional Lock Removal

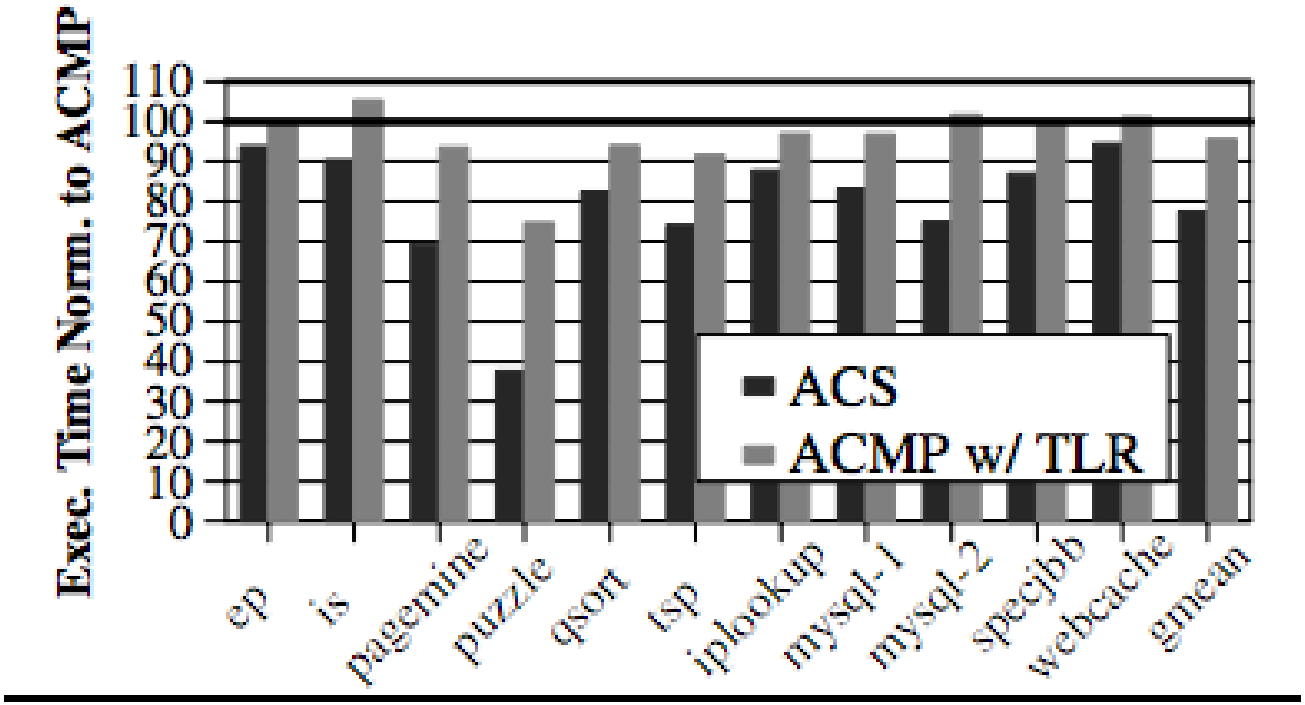


Figure 13. ACS vs. TLR performance.

ACS vs. SLE

- Can you combine both?
- How would you combine both?
- Can you do better than both?

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

We did not cover the following slides in lecture. These are for your preparation for the next lecture.

Transactional Memory

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
- Motivated by lack of concurrency (performance issues) in blocking synchronization (or "pessimistic concurrency")

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 → data races
- Deadlock due to lock ordering
- Blocking synchronization
- Conservative serialization

How transactions help:

+ Simpler interface/reasoning
+ No ordering
+ Nonblocking (Abort on conflict)
+ Serialization only on conflicts

- Locks → pessimistic concurrency
- Transactions → optimistic concurrency

Transactional Memory

- Transactional Memory (TM) allows arbitrary multiple memory locations to be updated atomically (all or none)

- Basic Mechanisms:
  - Isolation and conflict management: Track read/writes per transaction, detect when a conflict occurs between transactions
  - Version management: Record new/old values (where?)
  - Atomicity: Commit new values or abort back to old values → all or none semantics of a transaction

- 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: lazy vs. eager
- 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

Herlihy and Moss, ISCA 1993

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).

Current Implementations of TM/SLE

- Sun ROCK
- IBM Blue Gene
- IBM System Z: Two types of transactions
  - Best effort transactions: Programmer responsible for aborts
  - Guaranteed transactions are subject to many limitations
- Intel Haswell

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?