ECE/CS 757: Advanced Computer Architecture II

Instructor:Mikko H Lipasti

Spring 2017

University of Wisconsin-Madison

Lecture notes based on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Natalie Enright Jerger, Michel Dubois, Murali Annavaram, Per Stenström and probably others

Transactional Memory

- Transactional programming model
- Hardware Implementation
- Virtual TM (brief)
- Hardware-assisted Software Transactional Memory (brief)
- Thread-level speculation (TLS)

© 2005 Mikko Lipasti

Readings

[16] T. Harris, J. Larus, and R. Rajwar, "Transactional Memory, 2nd edition," Chapter 1 & Chapter 5, Synthesis Lectures on Computer Architecture, http://www.morganclaypool.com/doi/abs/10.2200/S00272ED1V01Y201006CAC011

[17] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13), June 2013.

© 2005 Mikko Lipasti

Transactions

- Divide parallel programs into *transactions*:
  - Basic unit of parallel work, communication, coherence, consistency
  - May contain multiple loads and stores
    - all commit at end of transaction, or none commit

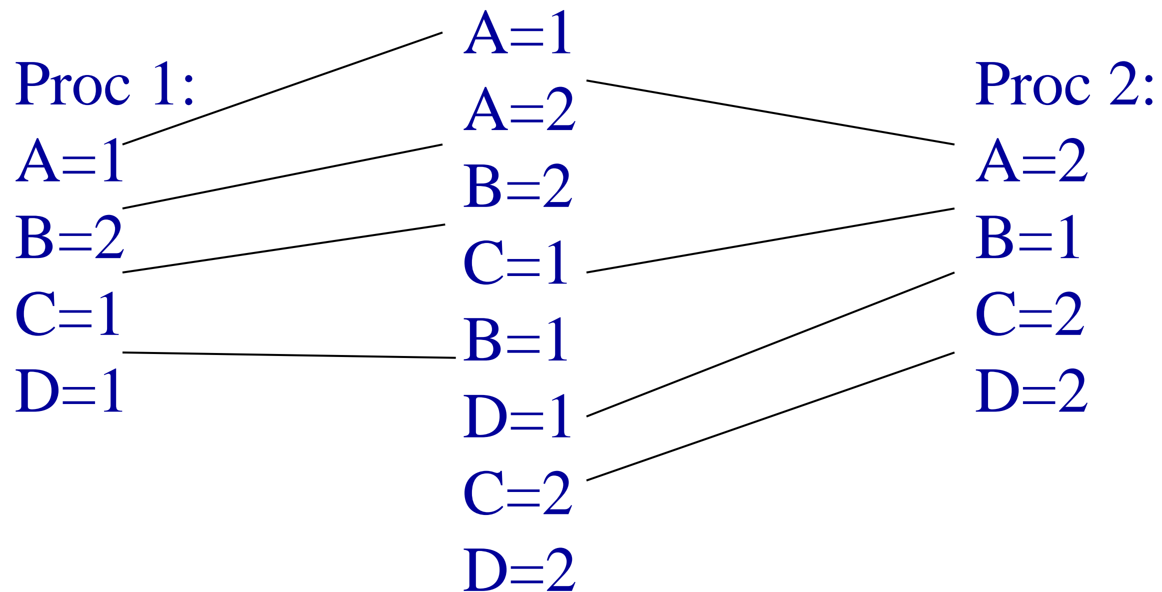
Simple Example

- Consider sequences of memory updates

Proc 1:	Proc 2:
A=1	A=2
B=2	B=1
C=1	C=2
D=1	D=2

Simple Example

- Sequential Consistency
  - Any interleaving is a legal sequence



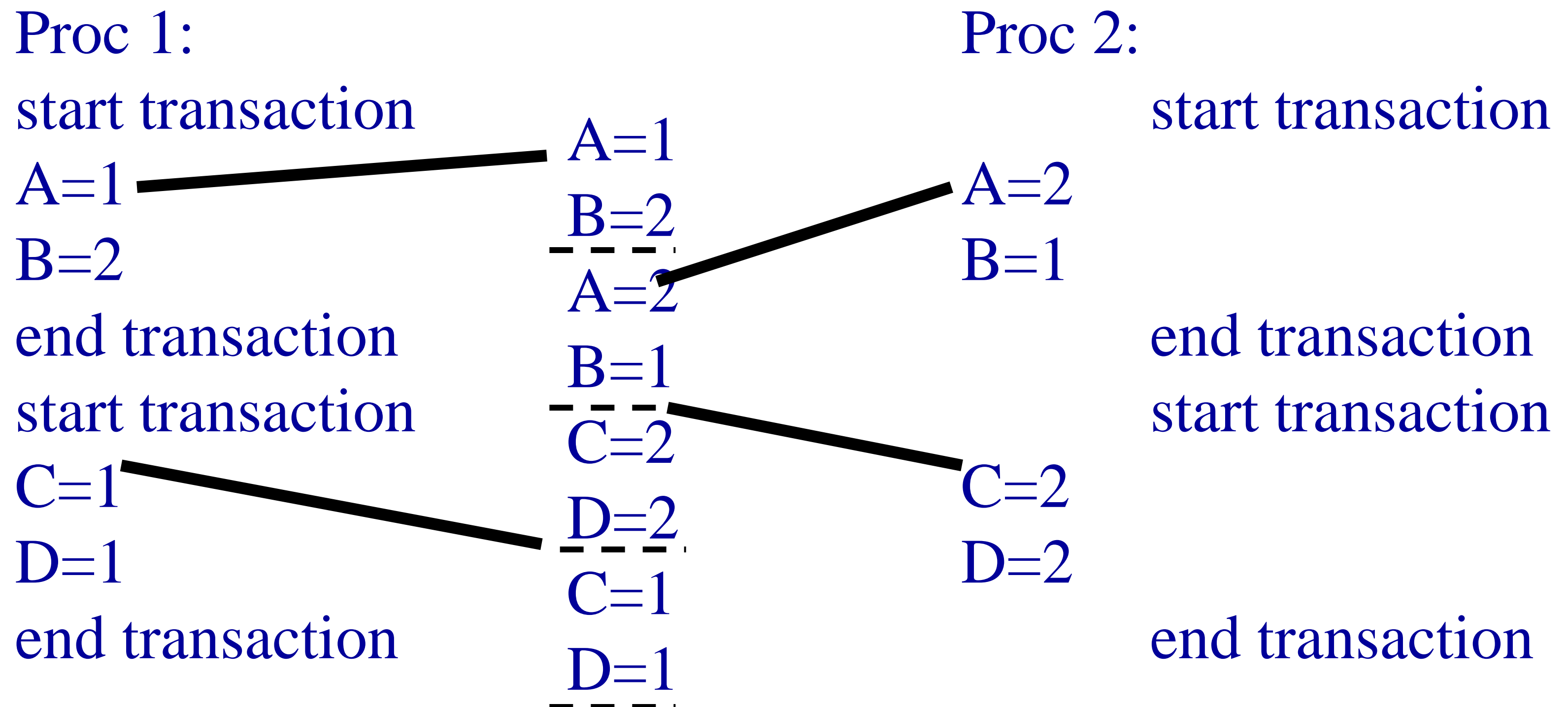
Example with Transactions

- Delineate transactions with start/end

Proc 1:	Proc 2:
start transaction	start transaction
A=1	A=2
B=2	B=1
end transaction	end transaction
start transaction	start transaction
C=1	C=2
D=1	D=2
end transaction	end transaction

Example

- Transactions provide consistency
  - A *subset* of the legal interleavings of sequential consistency



ECE/CS 757; copyright J. E. Smith, 2007

Example

- Transactions are atomic
  - Load/Stores w/in a transaction can be re-ordered in software or hardware

Proc 1:		Proc 1:
start transaction		start transaction
A=1		B=2
B=2	***Is equivalent to:***	A=1
end transaction		end transaction
start transaction		start transaction
C=1		D=1
D=1		C=1
end transaction		end transaction

ECE/CS 757; copyright J. E. Smith, 2007

Example: Histogram

- Read 10000 numbers between 1 and 100, and count the number of times each appears

```
#define n   10000
int buckets[100], data[n];
main()
{
input(data);
call histogram (0, n);
}
```

```
histogram (m, n);
int m, n;
{
for (i=0; i<n; i++) {
        buckets[data[i+m]]++;
return;
}
```


ECE/CS 757; copyright J. E. Smith, 2007

Example

```
#define n   10000
int buckets[100], data[n];
lock_type bucket_lock;
main()
int n, inc, is, incr;
{
if (procid == 0)
    {
    input(data);
    lock_init bucket_lock;
    }
inc = (n+nprocs-1)/nprocs;
is = inc*procid;
incr = min( inc, n-is);
call histogram (is, incr);
}
```

- Multi-threaded version
  - lock buckets before update;
  - lock becomes bottleneck;
- Could use fetch&add instead

```
histogram (m, n);
int m, n;
{
for (i=0; i<n; i++) {
        lock bucket_lock;
        buckets[a[i+m]]++;
        unlock bucket_lock;
return;
}
```

Example, contd.

```
#define n   10000
int buckets[100], data[n];
lock_type bucket_lock[100];
main()
int n, inc, is, incr;
{
if (procid == 0)
    {
    input(data);
    lock_init bucket_lock;
    }
inc = (n+nprocs-1)/nprocs;
is = inc*procid;
incr = min( inc, n-is);
call histogram (is, incr);
}
```

- Use array of locks
  - Still pay lock overhead when seldom needed
  - False sharing in lock array?

```
histogram (m, n);
int m, n;
{
for (i=0; i<n; i++) {
        lock bucket_lock[a[i+m]];
        buckets[a[i+m]]++;
        unlock bucket_lock[a[i+m]
return;
}
```

Example, with Transactions

- Make histogram update a transaction
  - Update becomes atomic

```
#define n   10000
int buckets[100], data[n];
main()
int n, inc, is, incr;
{
if (procid == 0)
    {
    input(data);
    }
inc = (n+nprocs-1)/nprocs;
is = inc*procid;
incr = min( inc, n-is);
call histogram (is, incr);
}
```

```
histogram (m, n);
int m, n;
{
for (i=0; i<n; i++) {
        start_transaction;
        buckets[a[i+m]]++;
        end_transaction;
return;
}
```

Advantages

- Shared memory programming model
  - Intuitive semantics
    - (unlike not-quite-sequential consistency)
  - Allows composition
- Simplified hardware implementation of SC
  - Implements atomic coarse-grain memory updates rather than individual load/stores
  - Amortizes overhead
  - Ordering of individual updates is relaxed
    - within a transaction

Programming Model

- How does it fit into languages?
  - Atomic blocks

```
// Insert a node into a doubly-linked list atomically
atomic {
newNode->prev = node;
newNode->next = node->next;
node->next->prev = newNode;
node->next = newNode;
 }
```

  - A simple form, but this is probably not enough
- Composability
  - Composing procedures containing transactions leads to nested transaction
  - Simple solution: Flatten transactions
    - Leads to large transaction (and buffer overflow problem)
    - Can lead to many replays (converts fine grain transaction into a coarse grain one)
- Enclosing system calls and I/O in a transaction
  - Complexities are discussed in Synthesis lecture reading

ECE/CS 757; copyright J. E. Smith, 2007

TM MECHANISMS

MECHANISMS TO ENFORCE ATOMICITY AND ISOLATION

- **ATOMICITY:**
  - ABILITY TO ROLL BACK AND RESTART A TRANSACTION
  - SPECULATIVE MEMORY DATA MANAGEMENT
    - STORE VALUES THAT ARE PART OF AN UNCOMMITTED TRANSACTION MUST REMAIN SEPARATE FROM VALUES OF COMMITTED TRANSACTIONS
    - VERSION MANAGEMENT
- **ISOLATION**
  - TRANSACTIONAL CONFLICT DETECTION (RACE)
  - CONCURRENCY CONTROL MECHANISM:
    - ON A CONFLICT DECIDE WHICH TRANSACTION ABORTS AND WHICH COMMITS

CAN BE IMPLEMENTED IN HARDWARE (HTM), IN SOFTWARE (STM) OR BOTH (HYBRID)

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

TM MECHANISMS

- **CONFLICT DETECTION:**
  - EACH TRANSACTION KEEPS TRACK OF ITS READ SET AND WRITE SET
    - NO OTHER TRANSACTION SHOULD READ OR WRITE TO AN ADDRESS THAT IS PART OF WRITE SET
    - NO THER TRANSACTION SHOULD WRITE TO AN ADDRESS THAT IS PART OF ITS READ SET
  - EAGER CONFLICT DETECTION: DETECTS AT THE TIME WHEN IT HAPPENS
  - LAZY CONFLICT DETECTION: DETECTS LATER, SUCH AS RIGHT BEFORE COMMITTING
    - CAUSES A DELAY IN ROLLBACK

- **VERSION MANAGEMENT:**
  - EAGER VERSION MANAGEMENT:
    - ORIGINAL (COMMITTED) VALUES ARE STORED IN A BUFFER
    - UNCOMMITTED VALUES ARE STORED DIRECTLY IN MEMORY
    - COMMIT IS FAST
  - LAZY VERSION MANAGEMENT
    - UNCOMMITTED VALUES ARE STORED IN A BUFFER
    - THEY PROPAGATE TO MEMORY WHEN THE TRANSACTION COMMITS
    - ROLLBACK IS FAST

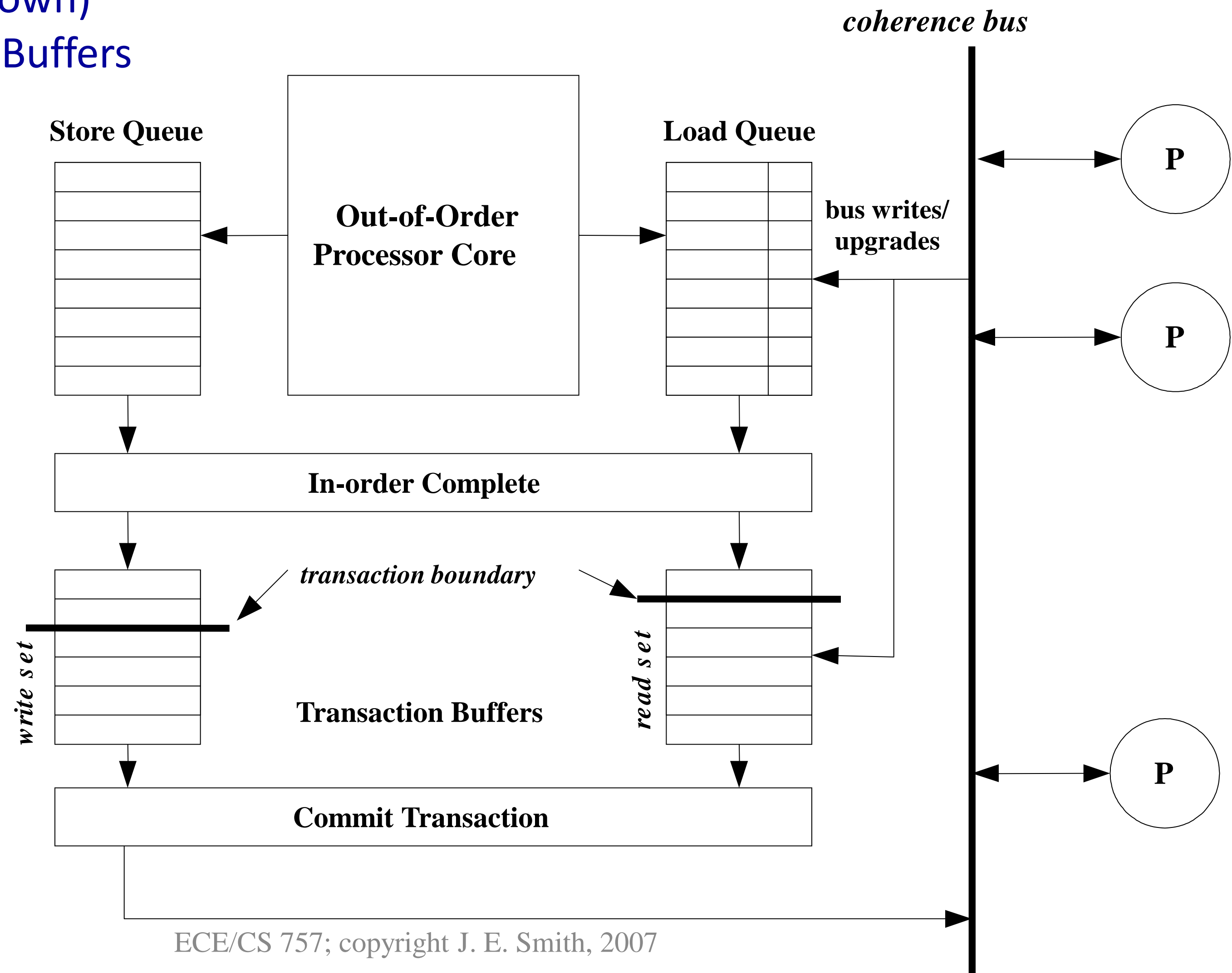
© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

Hardware Implementation Features

- Keep read set and write set
  - Read and write addresses of this transaction
- Buffer stores (the write set)
- Conflict detection with other transactions
  - Compare (pending) read set with writes by other transactions
  - Comparing at commit boundaries is one option
- Rollback if a conflict is detected
- Atomic commit of stores if no conflict

Implementation

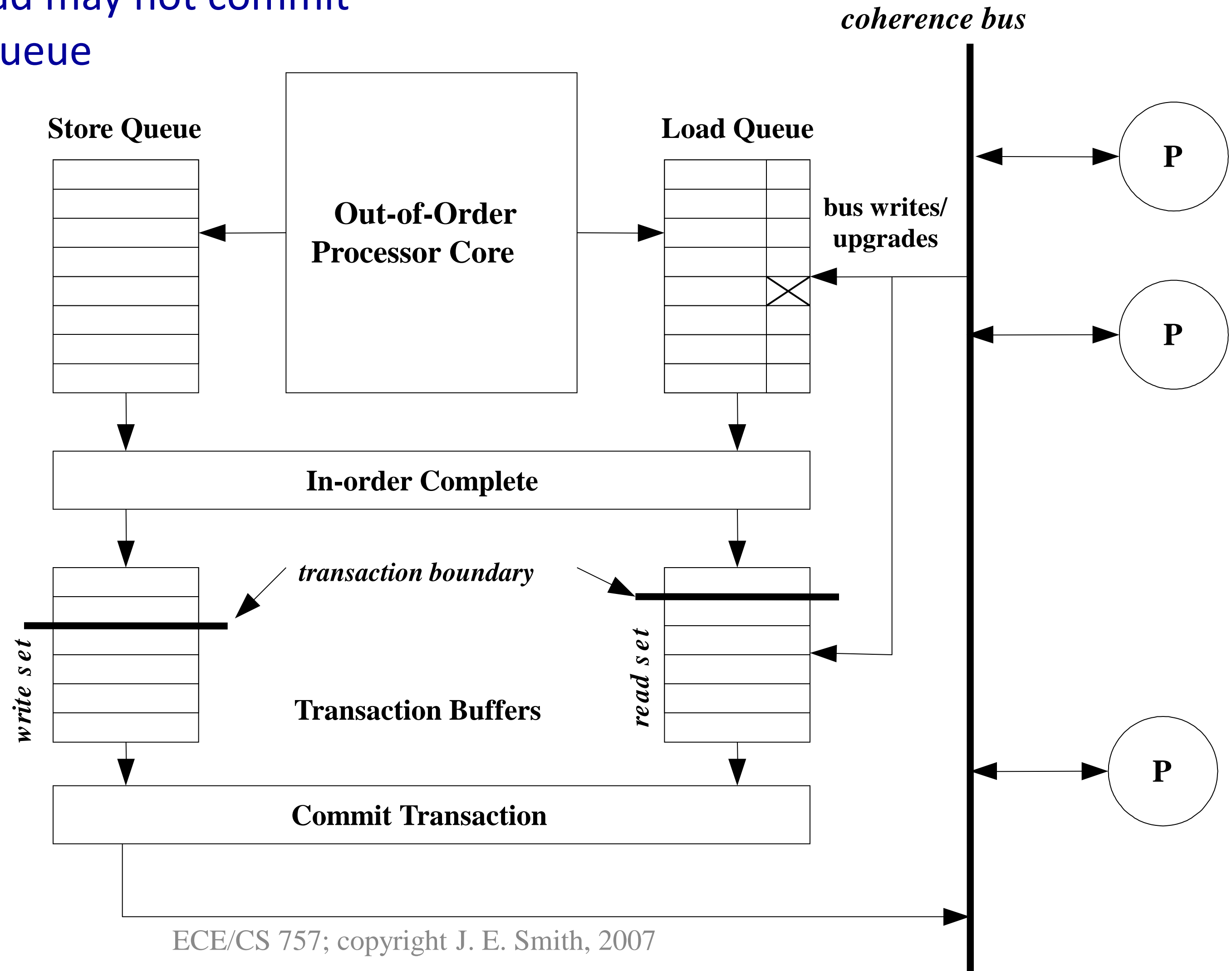
- Keep track of transaction boundaries
  - –in ROB (not shown)
  - –in Transaction Buffers



ECE/CS 757; copyright J. E. Smith, 2007

Implementation

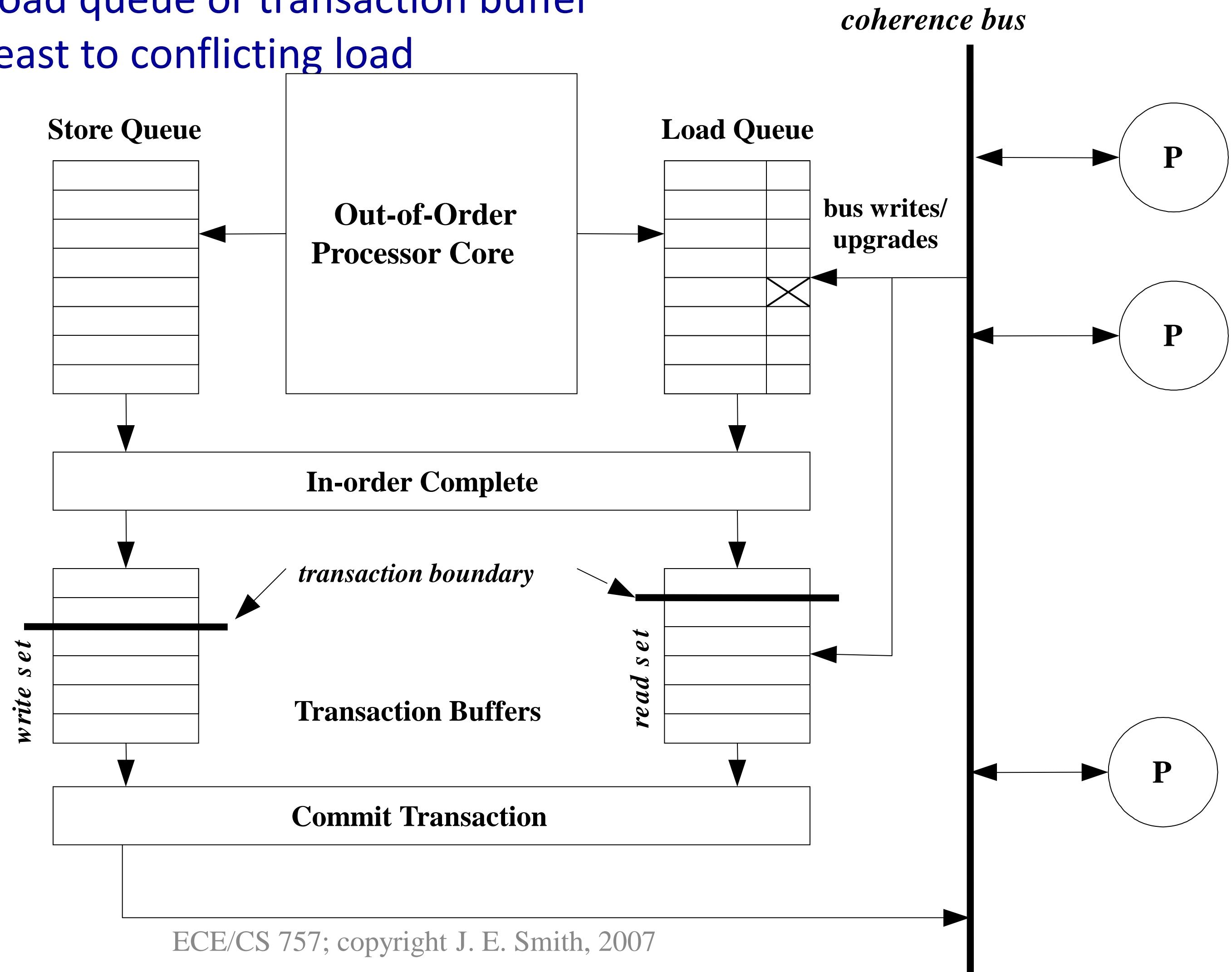
- Bus write/upgrade to speculative load
  - –Speculative load may not commit
  - –Mark in load queue



ECE/CS 757; copyright J. E. Smith, 2007

Implementation

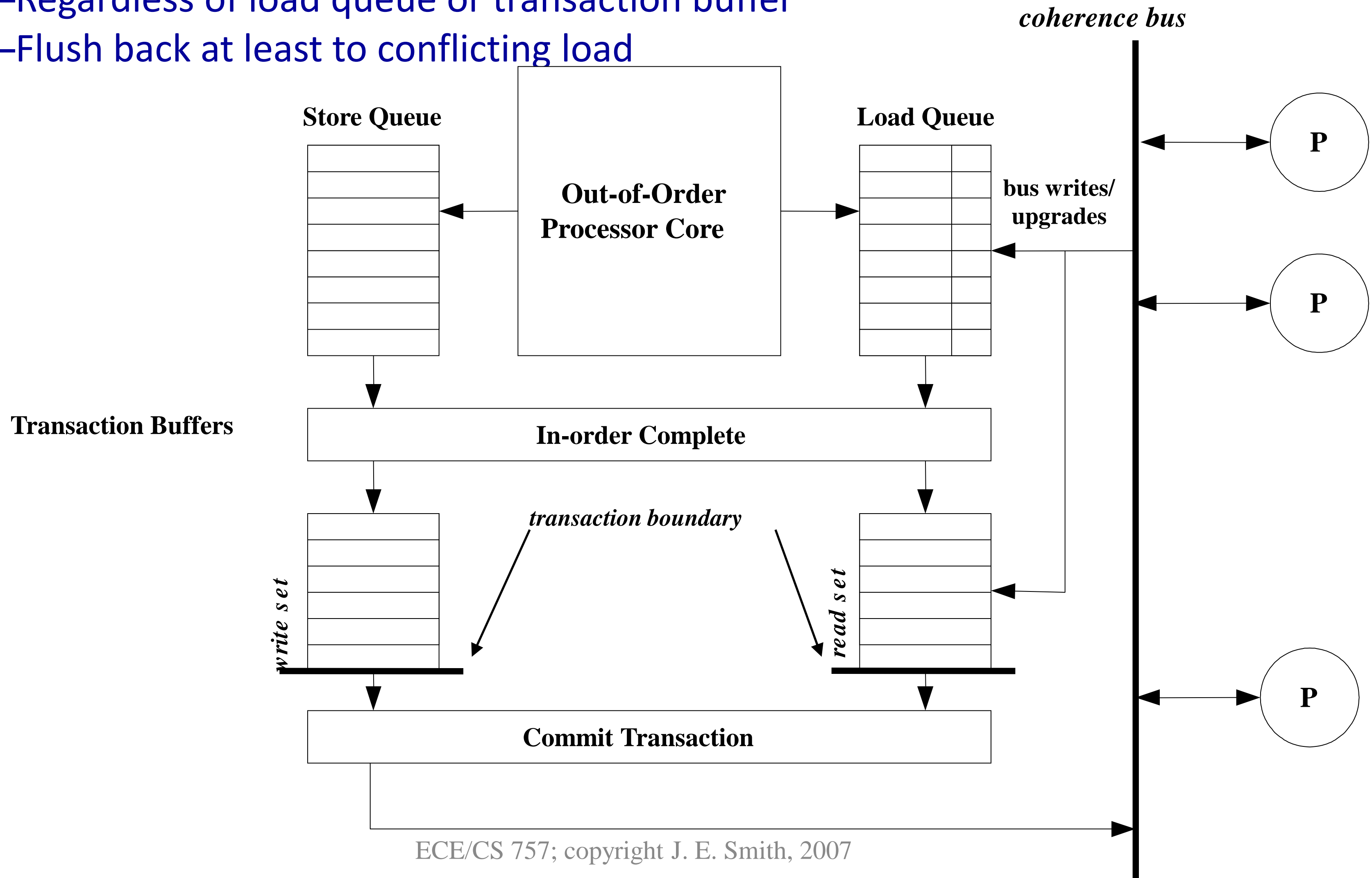
- Bus write/upgrade to non-speculative load
  - –Regardless of load queue or transaction buffer
  - –Flush back at least to conflicting load



ECE/CS 757; copyright J. E. Smith, 2007

Implementation

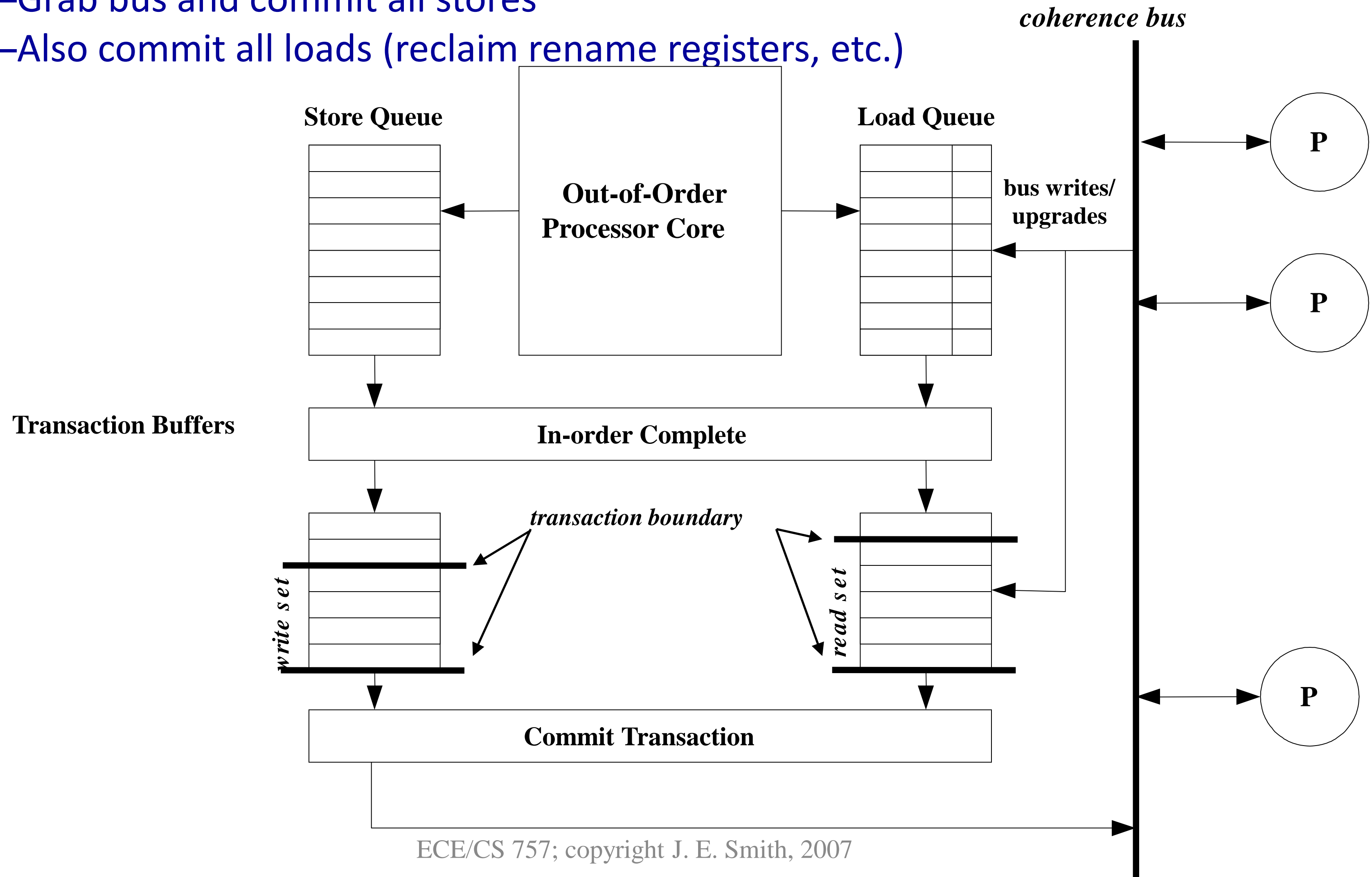
- Bus write/upgrade to non-speculative load
  - Regardless of load queue or transaction buffer
  - Flush back at least to conflicting load



ECE/CS 757; copyright J. E. Smith, 2007

Implementation

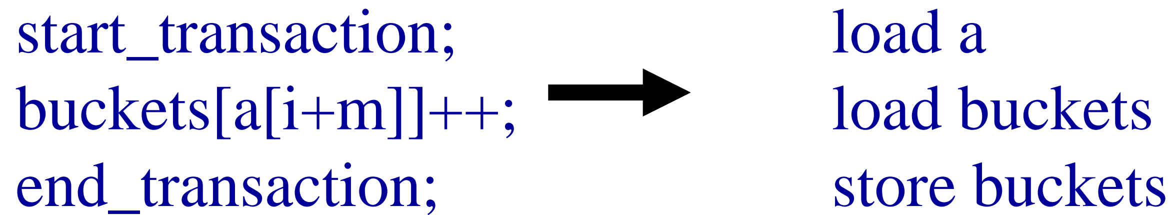
- When all instructions in current transaction are complete
  - –Grab bus and commit all stores
  - –Also commit all loads (reclaim rename registers, etc.)



ECE/CS 757; copyright J. E. Smith, 2007

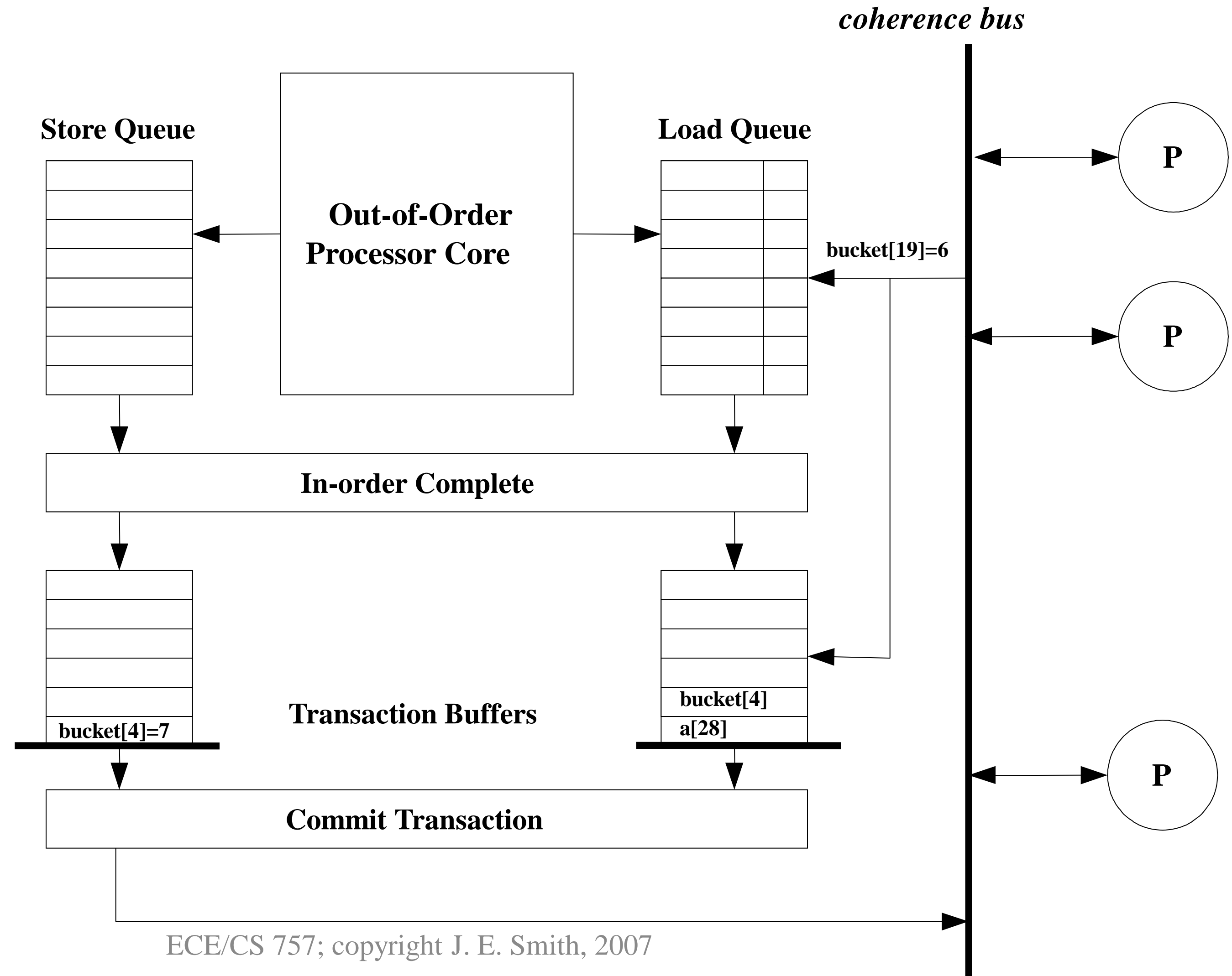
Transaction Example

- Inside transaction



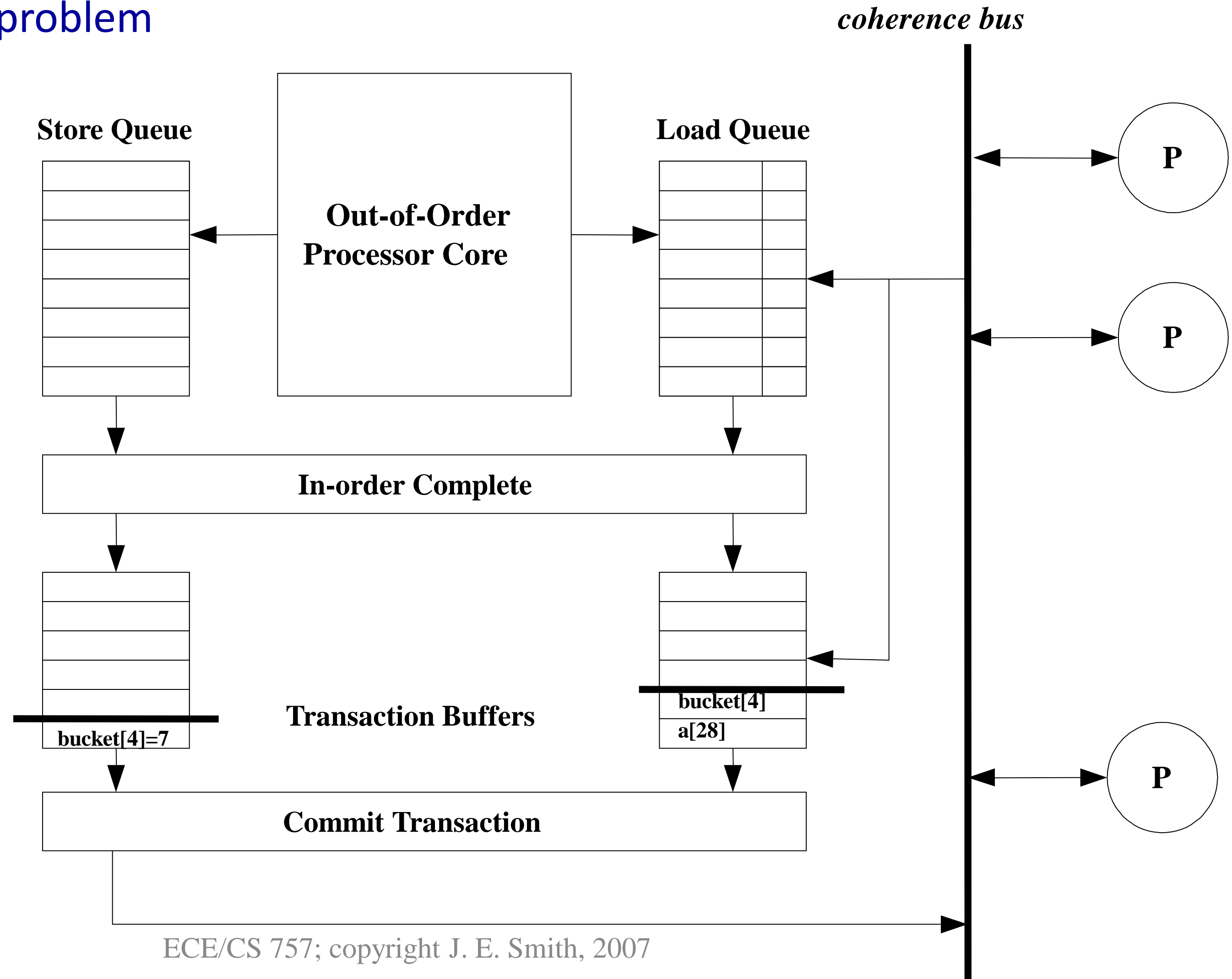
Histogram Example 1

- Each read-modify-write of a bucket is a single transaction



Histogram Example 1

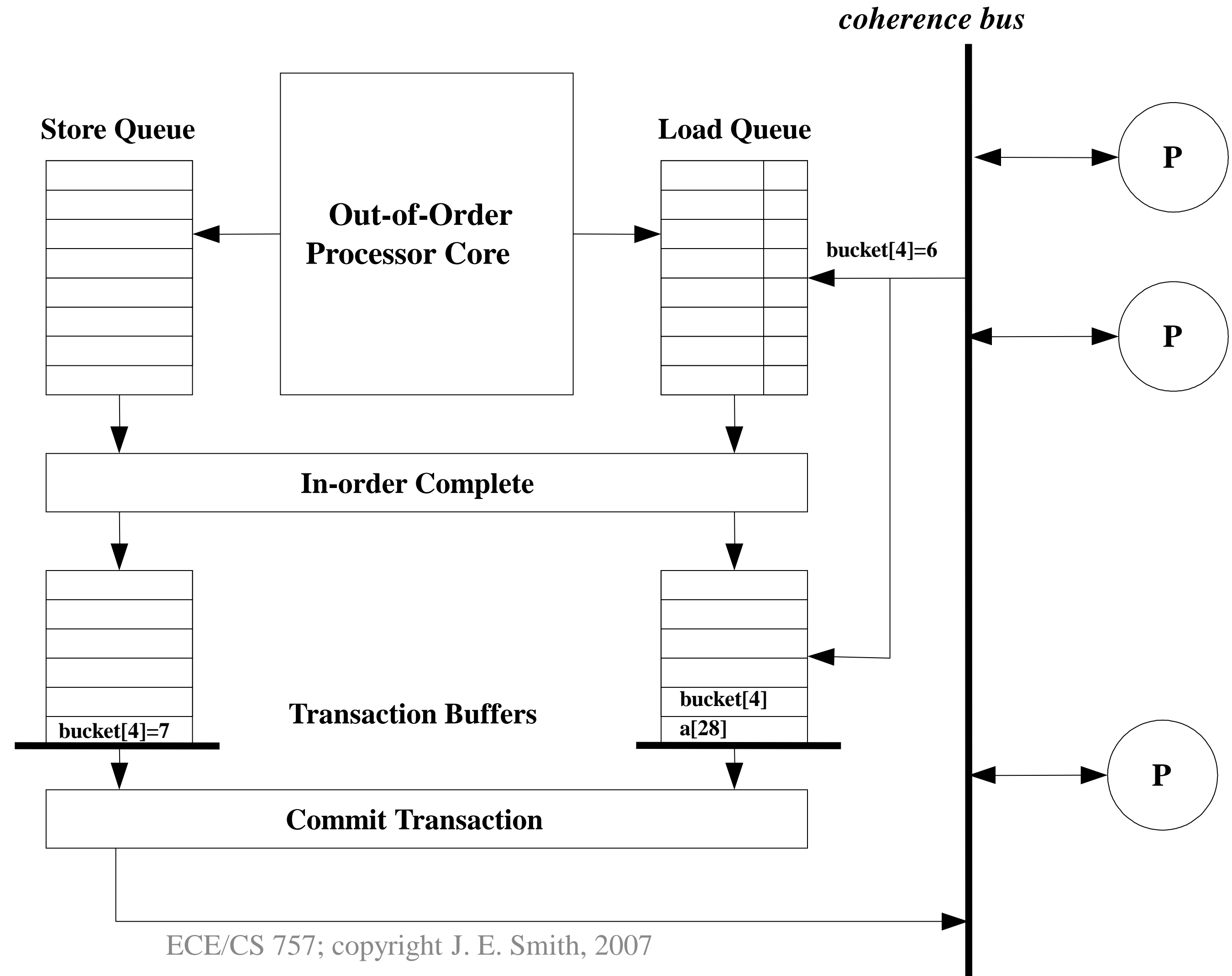
- Each read-modify-write of a bucket is a single transaction
  - –no conflict, no problem



ECE/CS 757; copyright J. E. Smith, 2007

Histogram Example 2

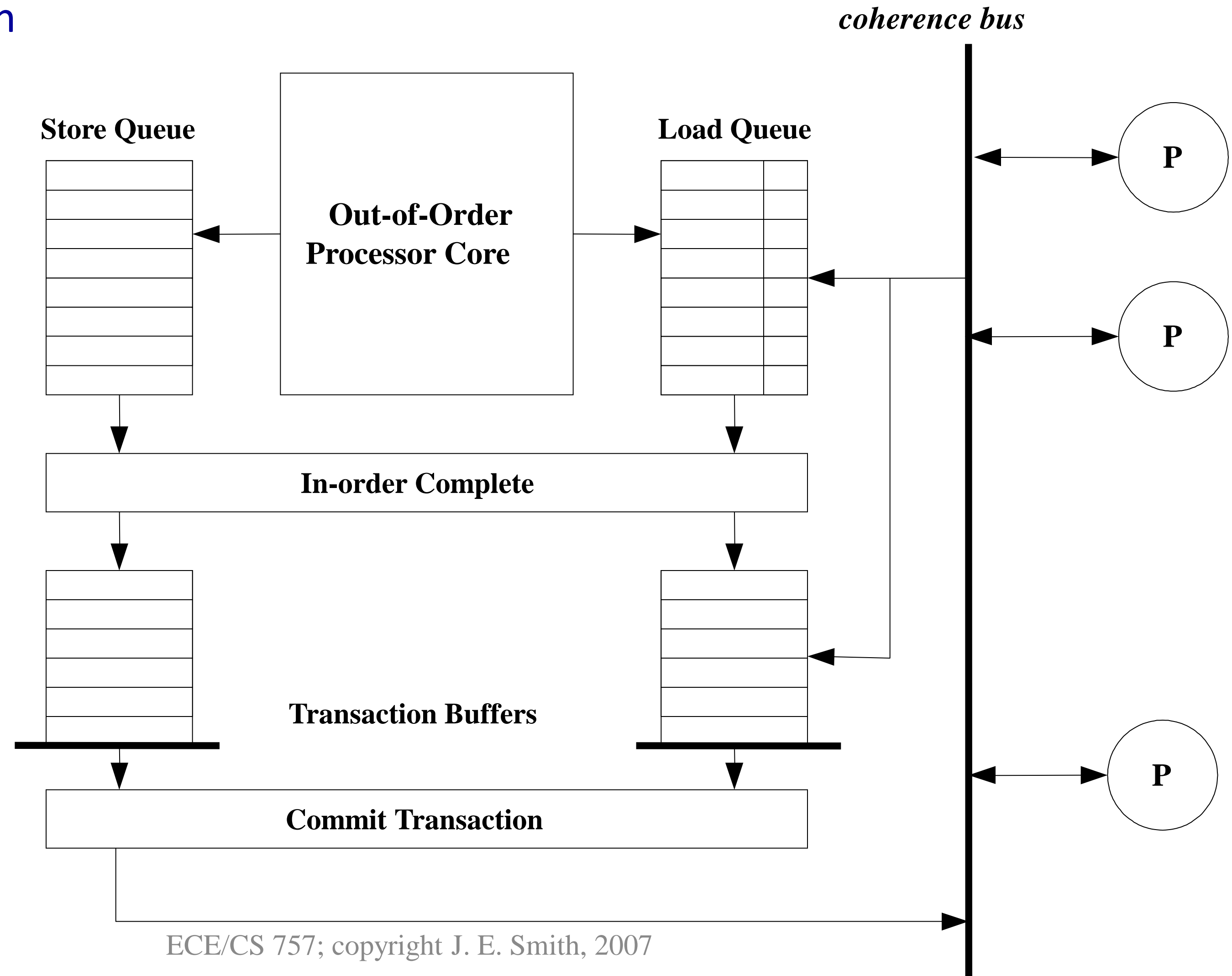
- Each read-modify-write of a bucket is a single transaction



ECE/CS 757; copyright J. E. Smith, 2007

Histogram Example 2

- Each read-modify-write of a bucket is a single transaction
  - –conflict => flush



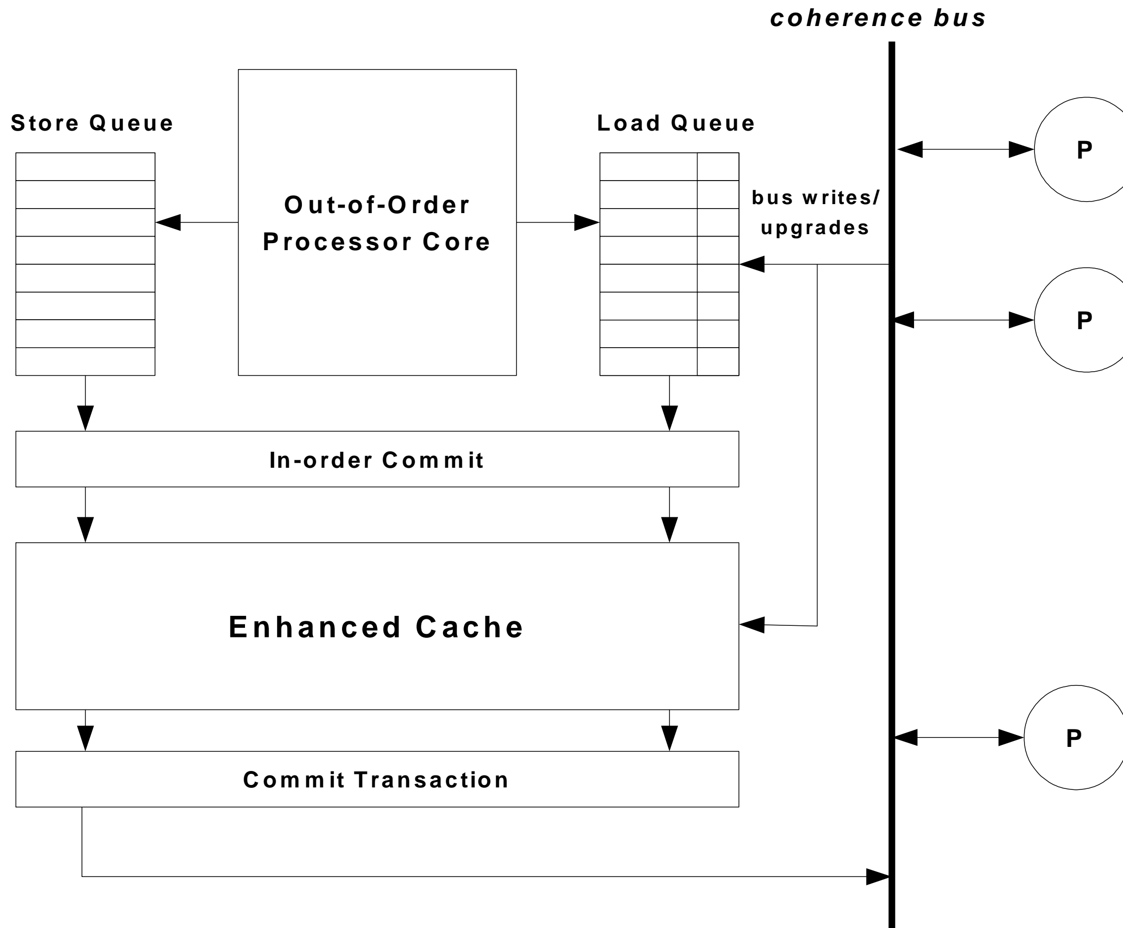
ECE/CS 757; copyright J. E. Smith, 2007

Hardware Implementation Issues

- When the Transaction Buffer Isn't Big Enough
  - Grab bus when buffer fills, and don't let go until transaction is done
- Deadlock
  - Conflict detection at commit time assures forward progress
- Can use explict buffers for read/write sets
  - OR, can use enhanced cache

Cache Implementation

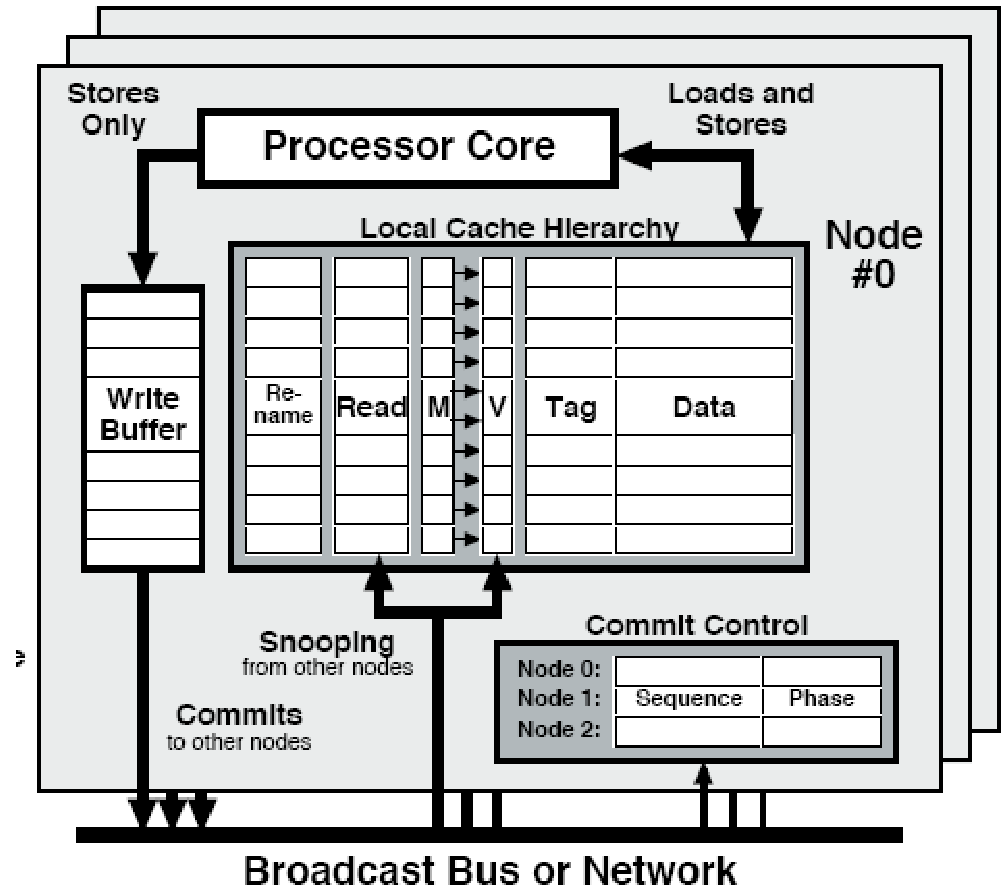
- As done in Hammond et al. ISCA paper



ECE/CS 757; copyright J. E. Smith, 2007

Cache Implementation

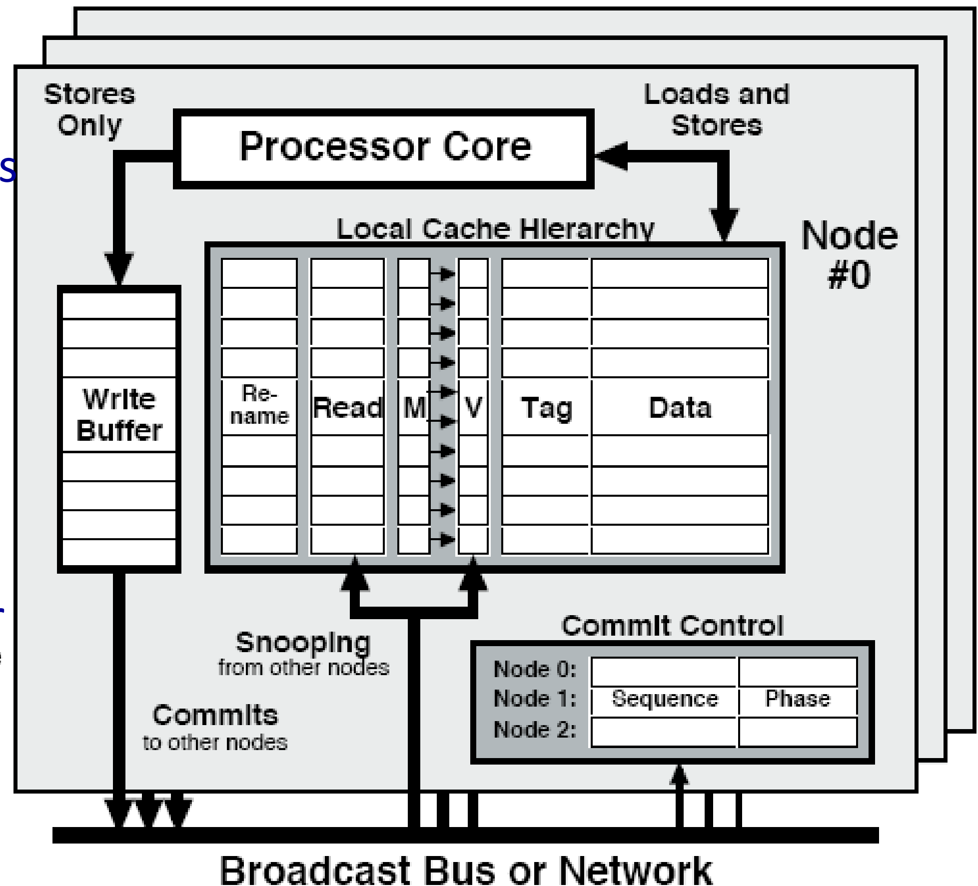
- New stuff shown on left side of cache directory
- Read bit(s)
  - Set to indicate "speculative" load
  - Snooped by bus writes & upgrades
  - Flush back to checkpoint on conflict
  - Bit per line (or finer to prevent false sharing)
- Write (M) bit
  - Set to indicate "speculative" store to line
- *Speculative $\Rightarrow$ in terms of transaction, not branches*



ECE/CS 757; copyright J. E. Smith, 2007

Cache Implementation, contd.

- Rename bits
  - Optional
  - Indicate locally modified sub-lines
  - If set, reads from these locations do not have to set read bit(s)
- Eviction
  - cannot be performed on speculative read/write data in mid-transaction (use victim buffer or stall)
- Write buffer
  - Holds addresses and/or data for commits
- Double buffering
  - Allows commit to go in parallel with next transaction



ECE/CS 757; copyright J. E. Smith, 2007

Cache Implementation, contd.

- Mark "speculative" loads and stores in cache
- On transaction commit:
  - Perform stores indicated in write buffer
  - Flash clear read/write bits in cache
- (Other caches) check committing write/upgrades
  - with marked reads in cache (and any pre-commit loads)
  - do not check marked writes
- On transaction conflict
  - Invalidate marked read/write lines in cache
  - rollback processor to nearest checkpoint preceding conflicting load
- Multiple uncommitted transactions w/in a processor
  - Paper doesn't seem to discuss it

Buffer Requirements

- Critical issue in Transactional Memory implementations
- Require a few KB to capture 90% of transactions
  - –Motivation for buffering in cache rather than externally

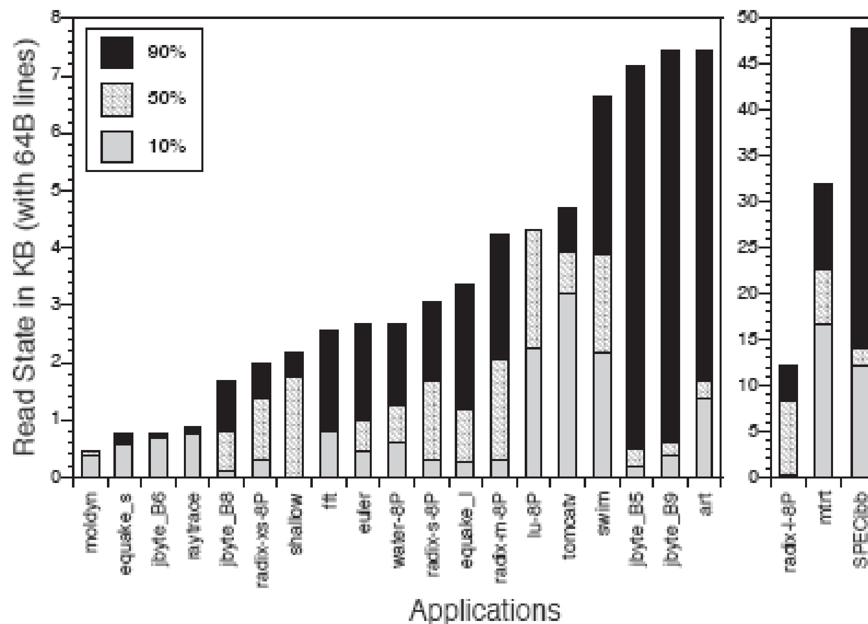


Figure 6: State read by individual transactions with store buffer granularity of 64-byte cache lines. We show state required by the smallest 10%, 50%, and 90% of iterations.

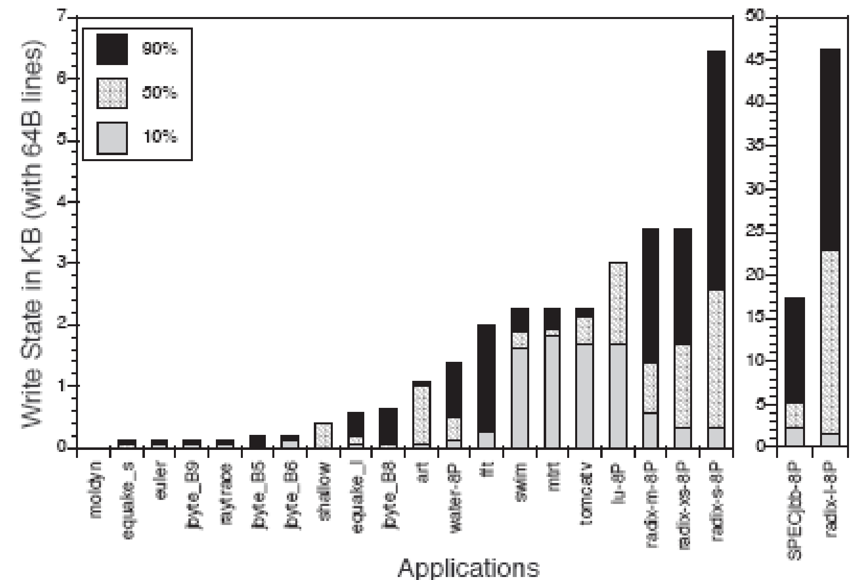


Figure 7: Same as Fig. 6, but for write state.

When the Transaction Buffer Isn't Big Enough

- Grab bus until commit done (as in Hammond et al.)
  - This could really hurt performance
  - Denial of service
- OR LogTM
  - UW work – Multifacet group
  - Store new values into memory
  - Save old values to the side in a VM-based log
  - Commits are very fast
  - On conflict, restore from log
    - Assumes conflicts are rare
- OR Spill buffered data into memory

Virtual Transactional Memory

- Problems with proposed hardware TM proposals occur because they are tied to physical resources
  - Finite buffers
  - Persistence over context switches
- Concept: use virtual resources
  - Spill buffer overflows into virtual memory
  - Save all buffer state in virtual memory on context switch
- R. Rajwar, et al., "Virtualizing Transactional Memory," Proc. International Symposium on Computer Architecture, June 2005.
  - Ravi is architect of Intel Haswell TM support

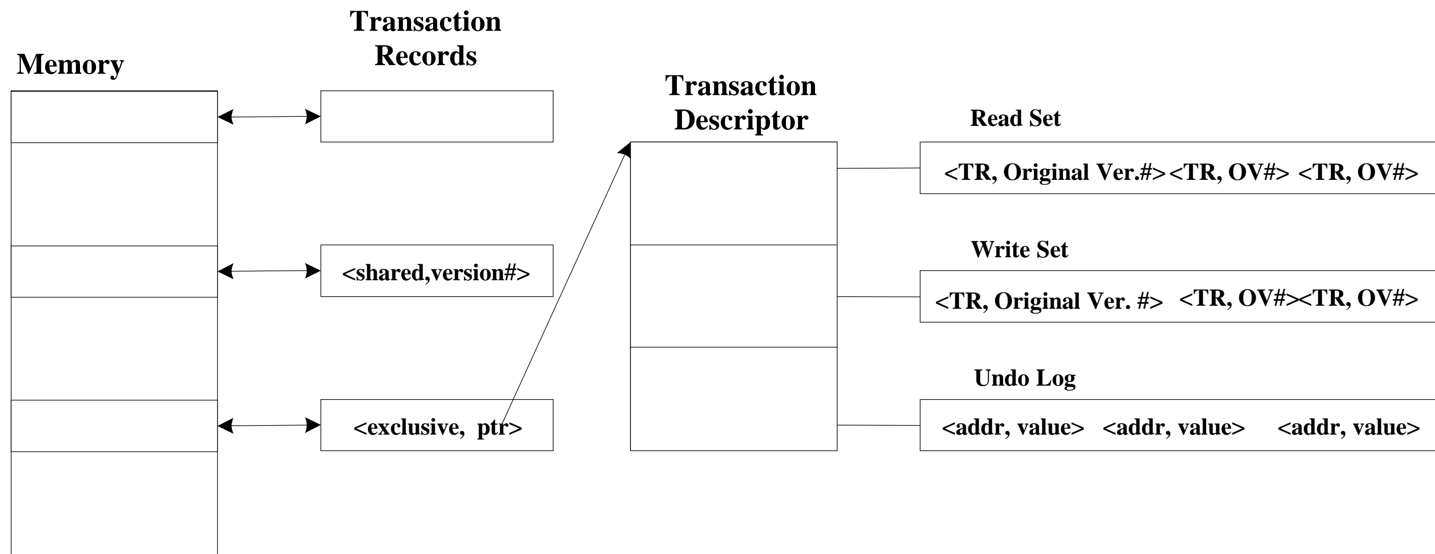
Hardware Supported STM (Saha et al.)

- Primitives, not Solutions
- Implement software transactional memory
  - Can be slow
- Add ISA primitives to accelerate
- B. Saha, A.-R. Adl-Tabatabai, Q. Jacobson, “Architectural Support for Software Transactional Memory,” 39th Int. Symp. on Microarchitecture, Dec. 2006, pp. 185-196.

ECE/CS 757; copyright J. E. Smith, 2007

Software Transactional Memory (Abbreviated)

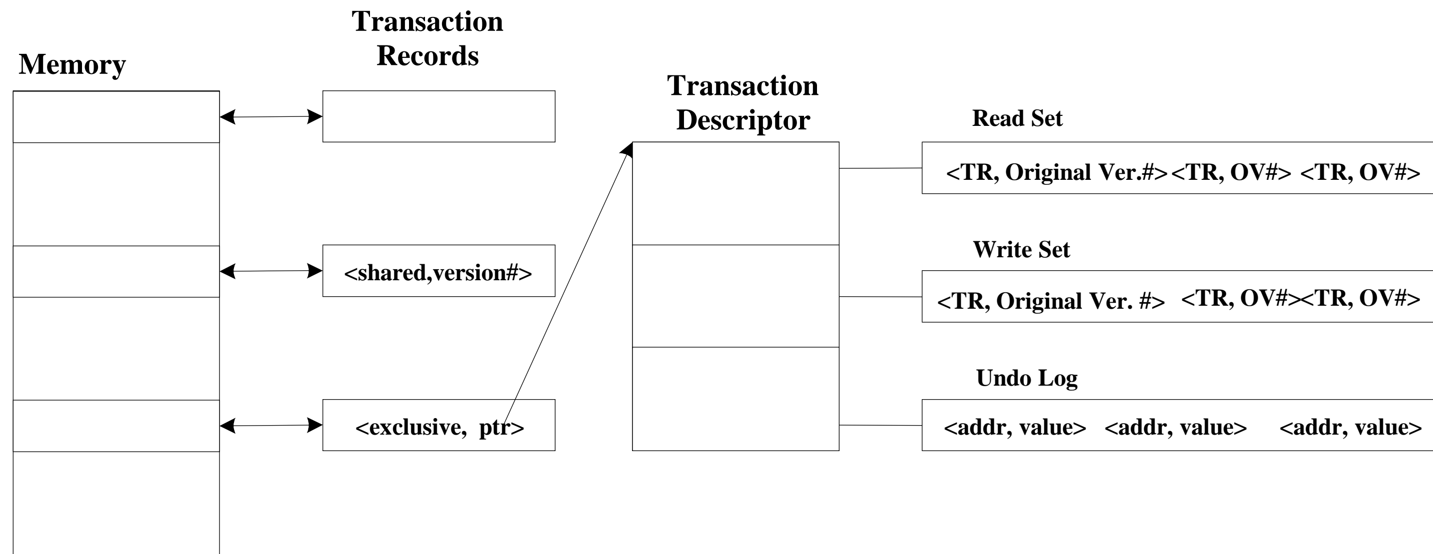
- Runtime-implemented
  - Runtime keeps a number of data structures
- Transaction Records
  - Track variables and/or objects
- When shared, keep version#
  - Version# updated whenever variable/object is written
- When exclusive, point to owner transaction
  - A transaction reserves transaction descriptor on first write



ECE/CS 757; copyright J. E. Smith, 2007

STM (Abbreviated)

- Transaction Descriptor
  - One per transaction
- Read Set
  - Contains read addresses and version# on first read
- Write Set
  - Contains write addresses and version# on first write
- Undo Log
  - Contains address/value pairs of overwritten values



ECE/CS 757; copyright J. E. Smith, 2007

STM Read Actions

- Before reading a variable, must first open for read

  Get transaction record;

  If transaction record owned by this transaction do nothing (return);

  If transaction record owned by another transaction,

  call contention manager (exponential backoff or abort);

  Else -- the record is shared --

  log (original) version# into read set;

- There may be redundant opens
  - If compiler cannot determine that read has been done before

ECE/CS 757; copyright J. E. Smith, 2007

STM Write Actions

- Before writing a variable, must first open for write

Get transaction record;

If transaction record unowned,

take ownership;

log (original) version# in write set

put address/original value into undo log

Else call contention manager;

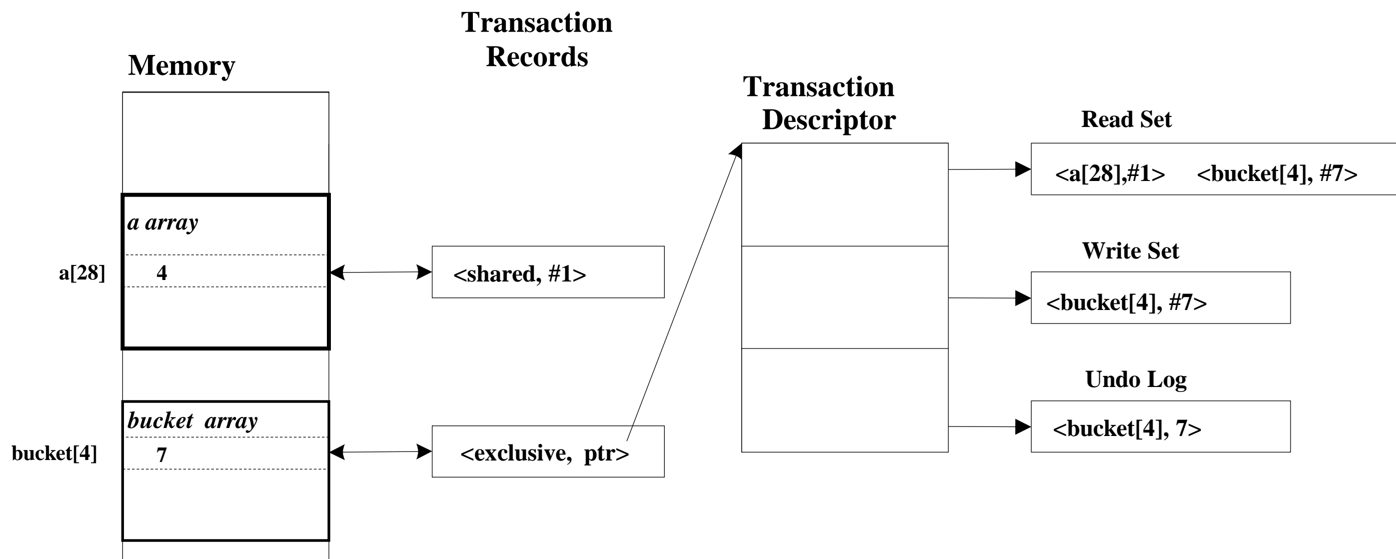
ECE/CS 757; copyright J. E. Smith, 2007

STM Commit

- At end of transaction
- Validate read set
  - Check read version numbers
  1) version# in read set == current version# in transaction record
     (no other transaction has done a write)
  2) version# in read set == version# of write in write set
     (write was done by this transaction)
- If read set validation succeeds
  - Update version numbers, release owned transaction records
- If validation fails
  - Abort transaction
  - Back up writes with Undo Log
  - Notify contention manager

Example: Histogram Problem

- State when transaction involving bucket#4 is nearly finished – ready to commit
- If some other transaction attempts to access bucket[4] anytime after this one acquires bucket[4] then it must wait
- Deadlock, Compiler optimizations, Garbage collection, etc. discussed in Adl-Tabatabai et al. 2006



ISA Supported Acceleration of STM

- Add and track *mark bits* in D-cache
  - Mark bits denote members of read set
  - Mark bit per 16 bytes (say)
  - The smaller the granularity, less false sharing (but more cost)
- Add Mark Counter
  - Tracks number of deleted marks
  - Incremented when marked line is removed from cache
  - Incremented when coherence write/upgrade to marked line
  - Set to max number on context switch
- New Instructions
  - loadSetMark(addr) – load from addr and set mark bit
  - loadResetMark(addr) – load from addr and clear mark bit
  - loadTestMark(addr) – load from addr and test mark bit (Cond. Branch)
  - resetMarkAll –reset all mark bits; increment Mark Counter
  - resetMarkCounter – reset Mark Counter
  - readMarkCounter – read Mark Counter

Cache States

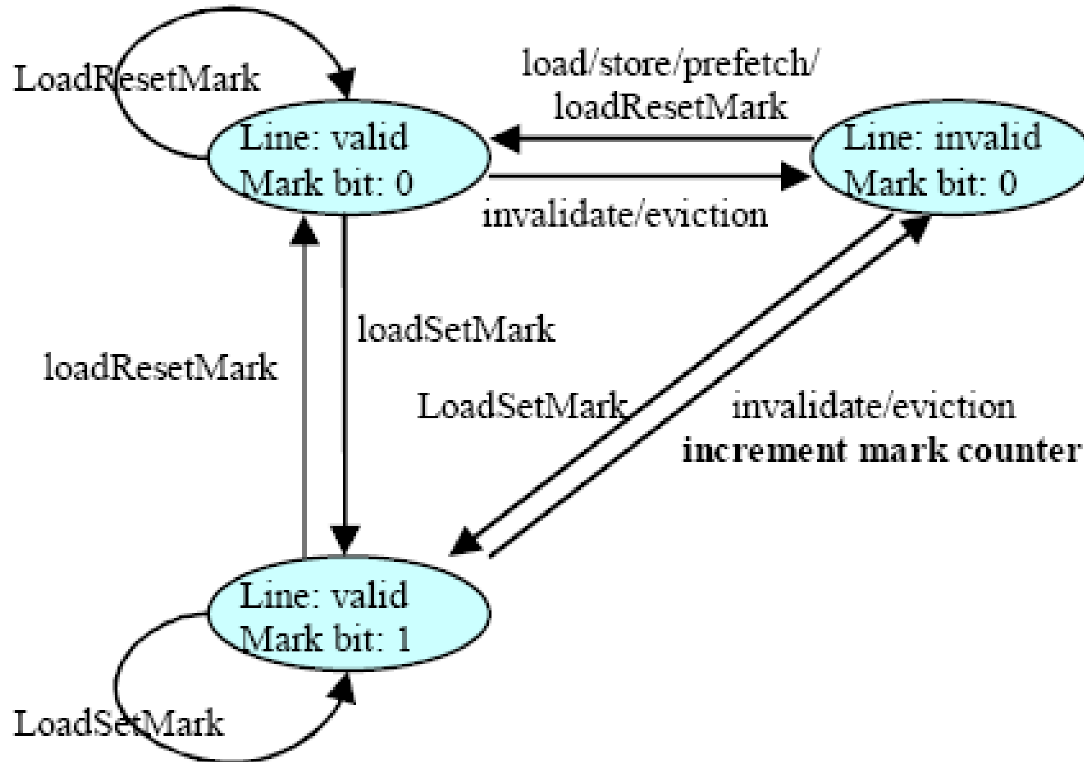


Figure 1: Cache line transitions

ECE/CS 757; copyright J. E. Smith, 2007

Hardware Acclerated STM (HASTM)

- Start with STM as described before
- Use mark bits for fast tracking/validation of read set

HASTM Read Actions

- Before reading a variable, must first open for read

  Perform loadTestMark;

  If mark bit set, do nothing (return);

  Perform LoadSetMark;

  Software:

  If transaction record owned by another transaction,

  call contention manager (exponential backoff or abort);

  Else -- the record is shared --

  log (original) version# into read set;

- Reduces fast path from 12 instructions to two instructions
  - Case where mark bit is already set

HASTM Write Actions

- Software same as before
- Before writing a variable, must first open for write

Get transaction record;
If transaction record unowned,
  take ownership;
  log (original) version# in write set
  put address/original value into undo log
Else call contention manager;

HASTM Commit

- At end of transaction
- Validate read set
  - Check mark count
    - If zero, then validation complete;
    - Else revert to software method;
- If read set validation succeeds
  - Update version numbers, release owned transaction records
- If validation fails
  - Abort transaction
  - Back up writes with Undo Log
  - Notify contention manager

Summary – Transactional Memory

- Transactional programming model
- Hardware Implementation
- Virtual TM (brief)
- Hardware-assisted Software Transactional Memory (brief)

- Real systems:
  - IBM Blue Gene, zSeries, Power
  - Intel Haswell

- Next: Thread-level speculation (TLS)

© 2005 Mikko Lipasti

THREAD-LEVEL SPECULATION (TLS)

- **PTHREAD, OPENMP AND TM ARE ALL EXPLICIT PARALLEL MODEL**
  - THE ADVANTAGE OF TM IS THAT LOCKS ARE REPLACED BY TRANSACTIONS
  - GOOD FOR NON-NUMERICAL CODES
  - STILL, THE PROGRAMMER HAS TO EXPRESS PARALLELISM EXPLICITLY IN ALL CASES

- **THREAD-LEVEL SPECULATION IS A TECHNIQUE TO PARALLELIZE SEQUENTIAL PROGRAMS AUTOMATICALLY**
  - VERY LITTLE HELP FROM THE PROGRAMMER
  - CORRECTNESS IS GUARANTEED
  - PROGRAMMER SIMPLY IDENTIFIES PROGRAM REGIONS TO PARALLELIZE
    - SUCH AS LOOPS, NESTED SUBROUTINES, RECURSIVE FUNCTION CALLS
    - USUALLY COMPILERS FOCUS ON LOOPS

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

LOOP PARALLELIZATION

- **LIMITED BY (TRUE) LOOP-CARRIED DEPENDENCIES**

```
main()
{
...
for(i:=m;i<M;i++)
{
        ...

A[i]:=A[i]+B[i]    ...
}
...
}
```

(a)

```
main()
{
...
for(i:=m;i<M;i++
{
        ...

A[i]:=A[i-4]+B[i]    ...
}
...
}
```

(b)

```
main()
{
...
for(i:=m;i<M;i++)
{
        ...
j:=C[i]
A[i]:=A[j]+B[i]
        ...
}
...
        }
```

(c)

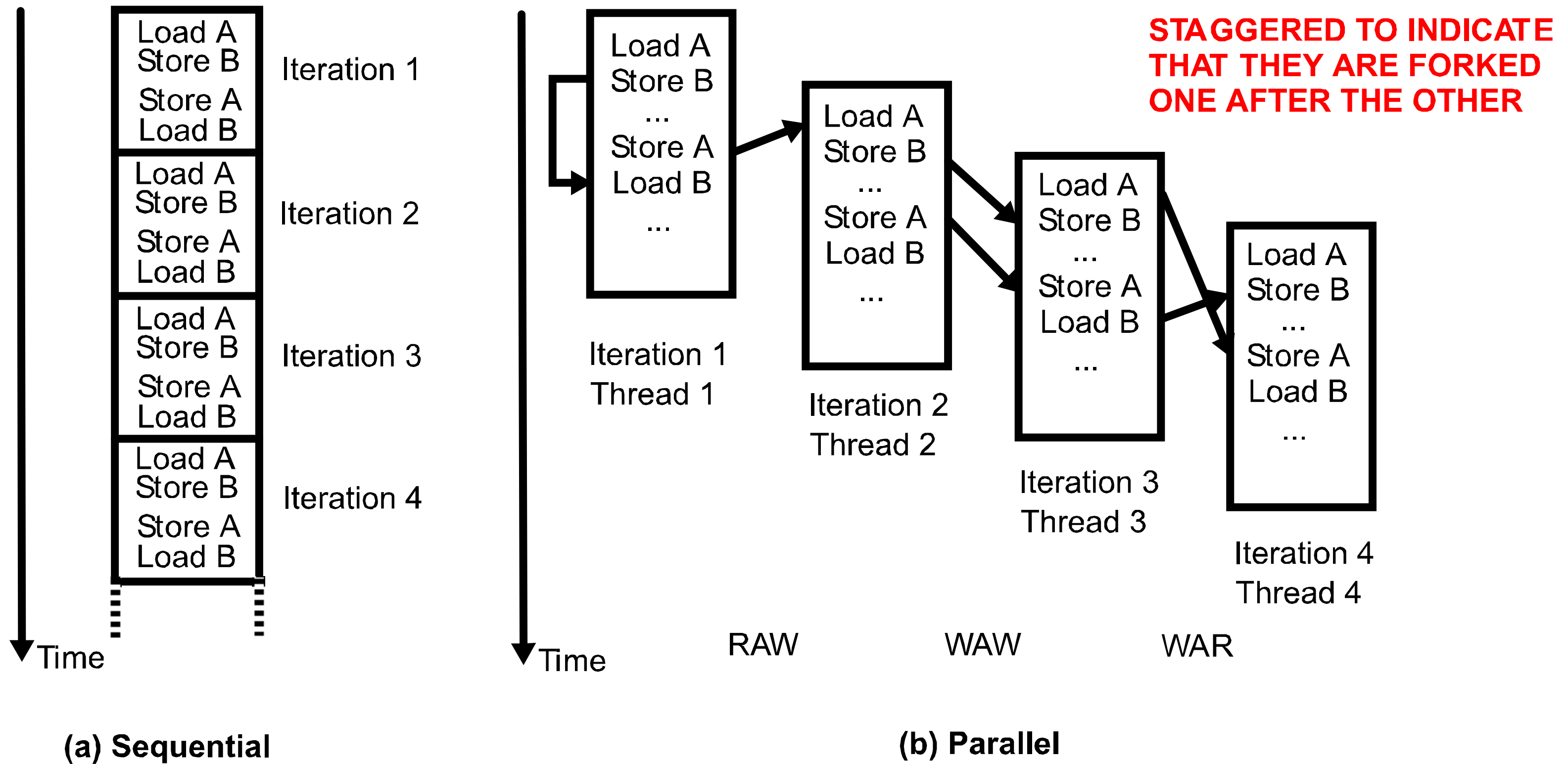
LOOP (a): PARALLELISM IS LIMITED BY THE NUMBER OF LOOP ITERATIONS
LOOP (b): PARALLELISM IS LIMITED BY THE RECURRENCE DISTANCE (4)
LOOP (c) NO PARALLELISM

- BECAUSE OF UNCERTAINTY CAUSED BY THE VALUE OF INDEX J (RAW LOOP-CARRIED DEPENDENCY)
- BUT J MAY BE GREATER THAN I OR I-J MAY BE SO LARGE THAT THE ITERATIONS COMPUTING A[I] AND A[J] NEVER OVERLAP IN TIME
- COMMON IN NON-NUMERICAL AND SOME NUMERICAL ALGORITHMS

- **A LARGE AMOUNT OF POTENTIAL PARALLELISM IS SQUANDERED IN (c)**

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

MEMORY HAZARDS IN LOOP PARALLELIZATION



- **LOAD B MUST RETURN THE VALUE OF STORE B IN THE SAME THREAD**
- **LOAD A MUST RETURN THE VALUE OF STORE A IN THE PREVIOUS THREAD**
  - THREAD 2 MUST SYNCHRONIZE OR VIOLATION MUST BE DETECTED AND THREAD MUST ROLLBACK
- **WAW HAZARDS ARE SHOWN BETWEEN THREADS 2 AND 3 (OK)**
- **WAR HAZARDS BETWEEN THREADS 3 AND 4 (A OK, B NOT OK)**

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

THREAD-LEVEL SPECULATION

- **PARALLELIZATION OF LOOPS WITH AMBIGUOUS LOOP-CARRIED DEPENDENCIES**
- **THE DEPENDENCY DISTANCE MUST BE QUITE LARGE (SINCE THIS LIMITS THE PARALLELISM)**
- **THREADS EXECUTES CONSECUTIVE ITERATION OF THE LOOP**
- **FOR EVERY ITERATION OF A LOOP**
  - A NEW THREAD IS SPECULATIVELY FORKED
  - SOME INPUTS TO THE THREAD MAY BE UNKNOWN
  - THREADS EXECUTE IN PARALLEL
  - THREADS ARE ORDERED AND NUMBERED
- **SO WE MAY HAVE RAW HAZARDS (DEPENDENCY VIOLATIONS)**
  - HAPPENS WHEN A WRITE FROM FROM AN EARLIER ITERATION UPDATES AN ADDRESS ALREADY READ BY A LATER ITERATION
  - MUST BE DETECTED
  - THE THREAD AND ALL SUBSEQUENT (MORE SPECULATIVE) THREADS MUST BE ROLLED BACK AND RE-RUN
- **WE ALSO NEED TO TAKE CARE OF WAW AND WAR DEPENDENCIES**
  - DONE BY MEMORY RENAMING IN L1 CACHES

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

SPECULATIVE PARALLELIZATION OF LOOPS

- **THREADS MUST APPEAR TO EXECUTE IN LOOP-INDEX ORDER**

- **THREADS HAVE A SEQUENCE NUMBER**
  - ORDER IN WHICH THEY WERE FORKED
  - ORDER OF LOOP INDEX

- **MORE SPECULATIVE THREADS HAVE HIGHER NUMBERS**
  - OLDEST (HEAD)THREAD WILL BECOME NON-SPECULATIVE
  - ALL OTHER THREADS REMAIN SPECULATIVE (PRECEDING WRITE POSSIBLE)
  - HEAD THREAD MUST COMMIT ITS RESULTS BEFORE THE NEXT THREAD MAY BECOME NON-SPECULATIVE
  - ONCE THE HEAD THREAD HAS COMMITTED, ITS CONTEXT CAN BE USED BY A NEW SPECULATIVE THREAD

- **BECAUSE THE HEAD THREAD WILL ALWAYS BECOME NON SPECULATIVE AND ALWAYS COMPLETE, THE COMPUTATION ALWAYS PROGRESSES**

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

TLS HARDWARE (OVERVIEW)

- **EACH CORE IS EQUIPPED WITH**
  - A THREAD SEQUENCE NUMBER REGISTER
  - A SPECULATION_ACTIVE BIT (SA) (SET DURING THE TIME THE THREAD IS SPECULATIVE)
  - A SPECULATION_FAIL BIT (SF) (SET WHEN A VIOLATION IS DETECTED)
- **EACH SPECULATIVE THREAD**
  - FIRST COMPLETES ITS EXECUTION
  - THEN IT WAITS TO BECOME THE HEAD THREAD
  - THEN IT CHECKS ITS SF BIT
  - IF SF IS SET
    - SPECULATION FAILED
    - IT SQUASHES ALL LATER THREADS (WITH HIGHER INDEX)
    - THESE SQUASHED THREADS SET THEIR SF BIT
    - THEN IT ROLLS BACK AND RESTARTS AS NON-SPECULATIVE
  - IF SF IS RESET
    - SPECULATION SUCCEEDED
    - IT BECOMES NON-SPECULATIVE
    - IT COMMITS ITS RESULTS TO MEMORY
    - IT RETIRES

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

TLS HARDWARE (OVERVIEW)

- **HARDWARE NEEDS TO DETECT VIOLATIONS AND SET SF BITS**
- **EXPLOIT THE CACHE HIERARCHY (PRIVATE L1s AND SHARED L2) AND PROTOCOL (MSI)**
  - L1s HOST THE SPECULATIVE VALUES
  - L2 HOSTS THE COMMITTED VALUES
  - SPECULATIVE VALUES ARE RENAMED IN L1 CACHES (WAW AND WAR HAZARDS)
- **BESIDES THE VALID AND DIRTY BIT PER LINE IN L1 WE ADD**
  - SL BIT (SPECULATIVELY LOADED) AND
  - SM BIT (SPECULATIVELY MODIFIED)
    - EACH IS SET ON A LOAD (SL) OR A WRITE (SM) WHILE IN SPECULATIVE MODE
  - WHEN EITHER BIT IS SET THE BLOCK IS IN THE *SPECULATIVE STATE* (Sp)
- **SPECULATIVE BLOCKS MUST REMAIN IN THE L1 CACHE**
  - STATE RECORDS THAT THE BLOCK WAS LOADED OR MODIFIED SPECULATIVELY
  - VALUE KEEPS TRACK OF THE SPECULATIVE MODIFICATIONS TO THE BLOCK
  - CANNOT PROPAGATE THROUGH THE MEMORY SYSTEM
  - CANNOT BE FLUSHED
  - CANNOT BE REPLACED

© Michel Dubois, Murali Annavaram, Per Stenström All rights reserved

TLS vs. TM

- TM first proposed in 1993 [Herlihy/Moss]
  - Dormant for a decade
- TLS: simplification of Multiscalar model
  - Speculative threads
  - Communicate only through memory/cache
  - CMU Stampede [Mowry, Steffan, Zhai, …]
  - Stanford Hydra [Olukotun, Hammond, …]
- Resurgence of TM
  - Same/similar mechanisms as TLS
  - Relax "transaction" order (implied sequential in TLS)

© 2005 Mikko Lipasti

Lecture Summary

- Transactional programming model
- Hardware Implementation
- Virtual TM (brief)
- Hardware-assisted Software Transactional Memory (brief)
- Thread-level speculation (TLS)