Supporting Nested Transactional Memory in LogTM

Michelle J. Moravan, Jayaram Bobba,

Kevin E. Moore, Luke Yen, Mark D. Hill,

Ben Liblit, Michael M. Swift, David A. Wood

Twelfth International Conference on
Architectural Support for Programming
Languages and Operating Systems

Motivation

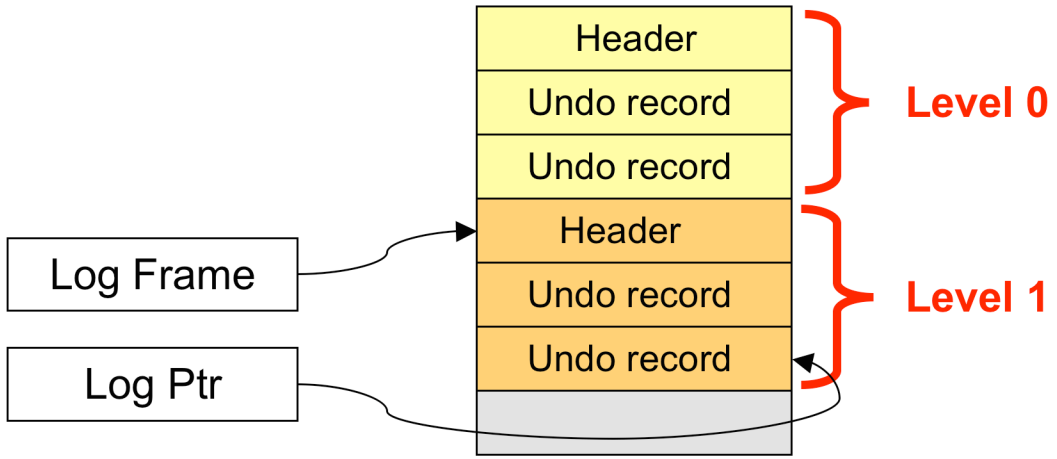
- **Composability**
  - Complex software built from existing modules
- **Concurrency**
  - "Intel has 10 projects in the works that contain four or more computing cores per chip"
    -- Paul Otellini, Intel CEO, Fall '05
- **Communication**
  - Operating system services
  - Run-time system interactions

Nested LogTM

- **Closed nesting**: transactions remain isolated until parent commits
- **Open nesting**: transactions commit independent of parent
- **Escape actions**: non-transactional execution for system calls and I/O

Nested LogTM

- Splits log into "frames"
- Replicates R/W bits
- Software abort processing naturally executes compesating actions



Composability

Composability: Closed Nested Transactions

Composability: Closed Nested Transactions

- Modules expose interfaces, **not** implementations

Composability: Closed Nested Transactions

- Modules expose interfaces, **not** implementations
- Must allow transactions **within** transactions

Composability: Closed Nested Transactions

- Modules expose interfaces, **not** implementations
- Must allow transactions **within** transactions
- Example
  - Insert() calls getID() from within a transaction
  - The getID() transaction is **nested** inside the insert() transaction

Composability: Closed Nested Transactions

- Modules expose interfaces, **not** implementations
- Must allow transactions **within** transactions
- Example
  - Insert() calls getID() from within a transaction
  - The getID() transaction is **nested** inside the insert() transaction

```
void insert(object o){
   begin_transaction();
      n = find_node();
      n.insert(getID(), o);
   commit_transaction();
}
```

```
int getID() {
   begin_transaction();
        id = global_id++;
   commit_transaction();
   return id;
}
```

Composability: Closed Nested Transactions

- Modules expose interfaces, **not** implementations
- Must allow transactions **within** transactions
- Example
  - Insert() calls getID() from within a transaction
  - The getID() transaction is **nested** inside the insert() transaction

Parent

```
void insert(object o){
   begin_transaction();
      n = find_node();
      n.insert(getID(), o);
   commit_transaction();
}
```

Child

```
int getID() {
   begin_transaction();
        id = global_id++;
   commit_transaction();
   return id;
}
```

Closed Nesting

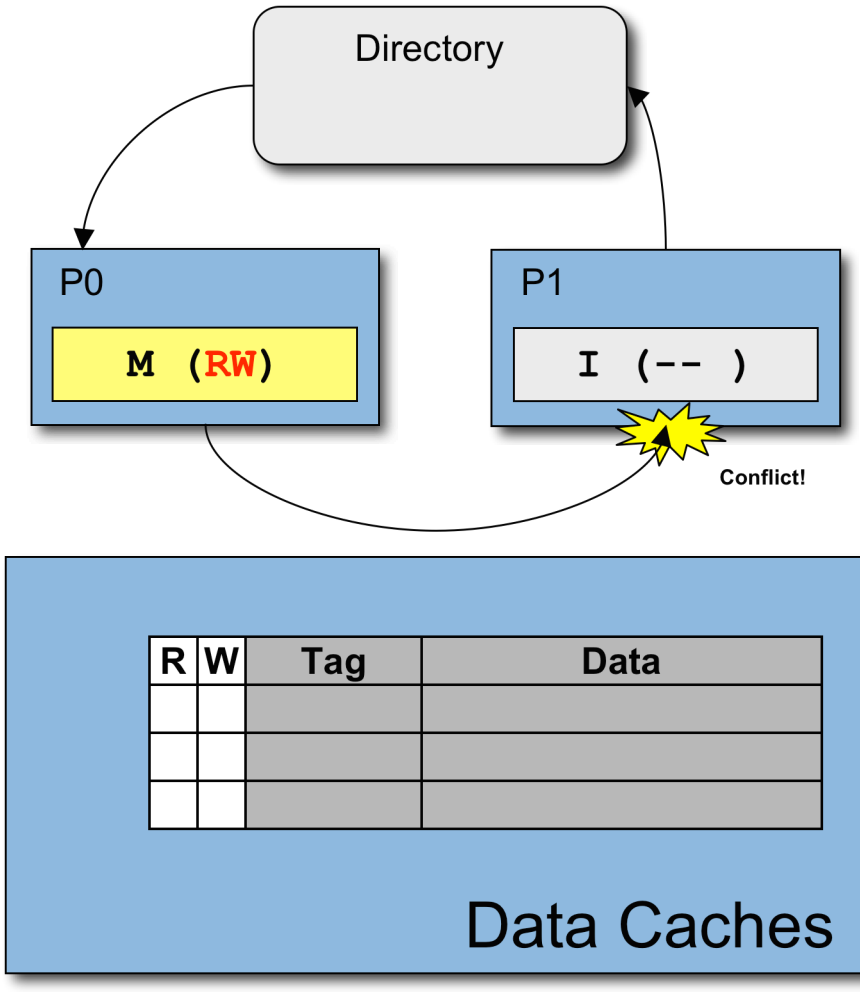
Child transactions remain isolated until parent commits

- On Commit child transaction is merged with its parent

- Flat
  - Nested transactions “flattened” into a single transaction
  - Only outermost begins/commits are meaningful
  - Any conflict aborts to outermost transaction
- Partial rollback
  - Child transaction can be aborted independently
  - Can avoid costly re-execution of parent transaction
  - But child merges transaction state with parent on commit
    - So most conflicts with child end up affecting the parent

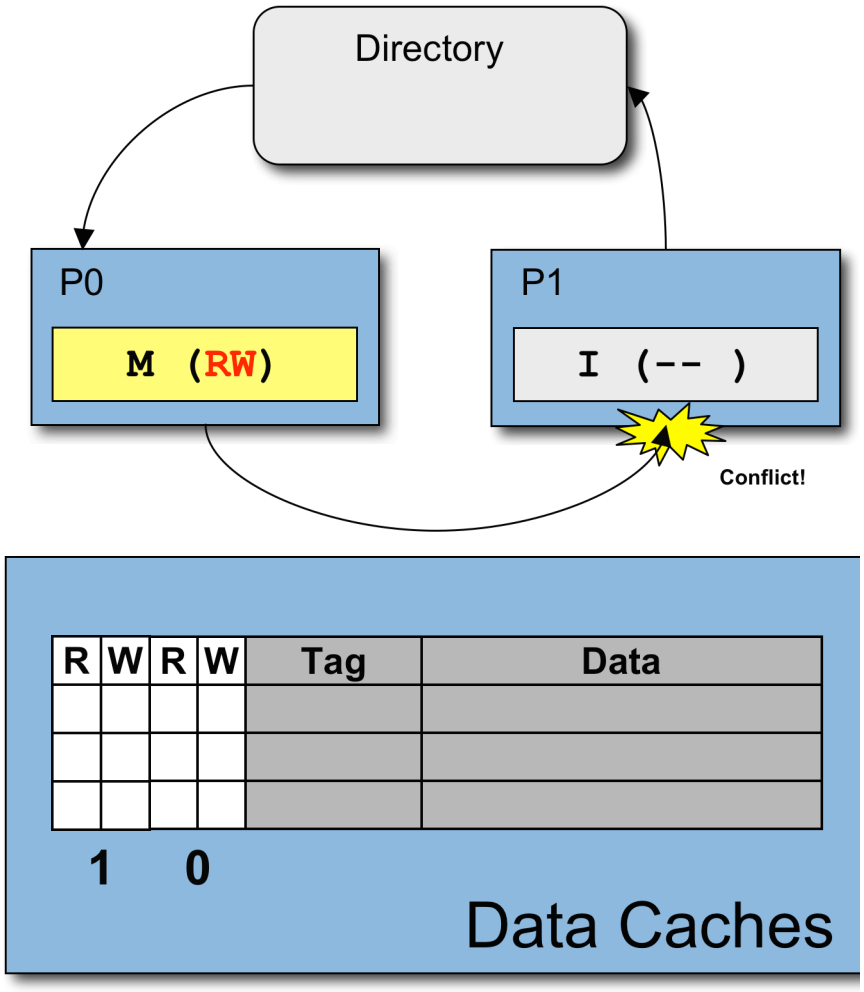
Conflict Detection in Nested LogTM

- Flat LogTM detects conflicts with directory coherence and Read (R) and Write (W) bits in caches
- Nested LogTM replicates R/W bits for each level
- Flash-Or circuit merges child and parent R/W bits



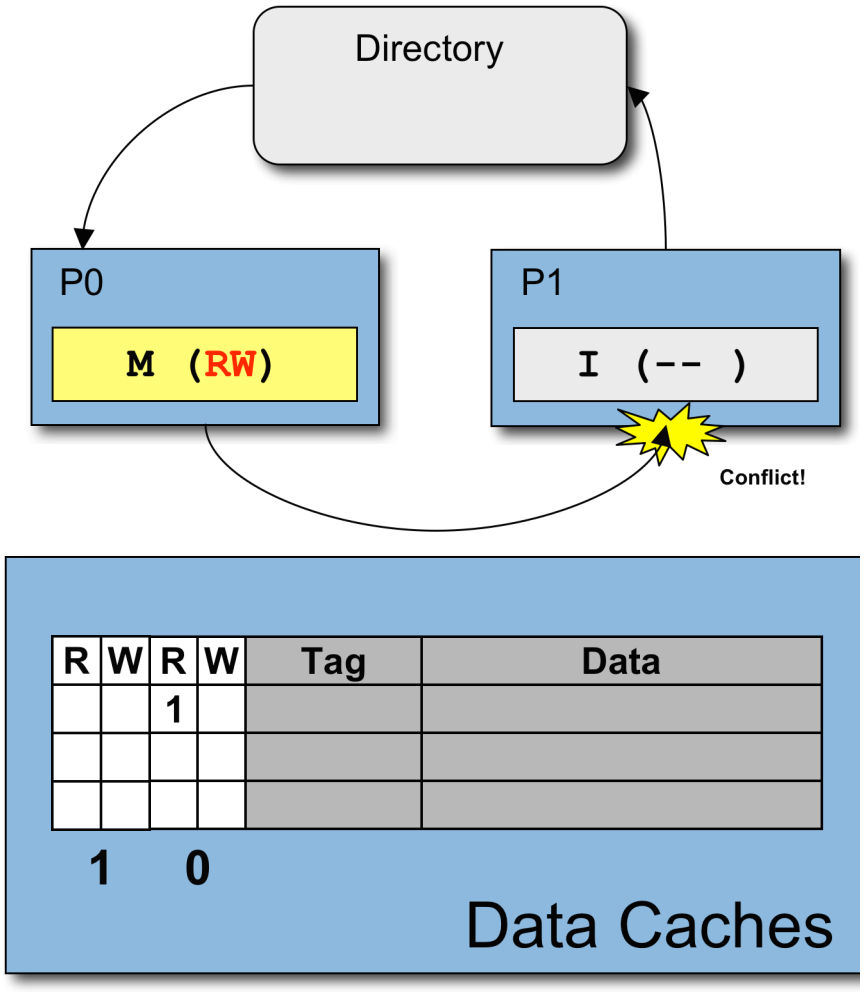
Conflict Detection in Nested LogTM

- Flat LogTM detects conflicts with directory coherence and Read (R) and Write (W) bits in caches
- Nested LogTM replicates R/W bits for each level
- Flash-Or circuit merges child and parent R/W bits



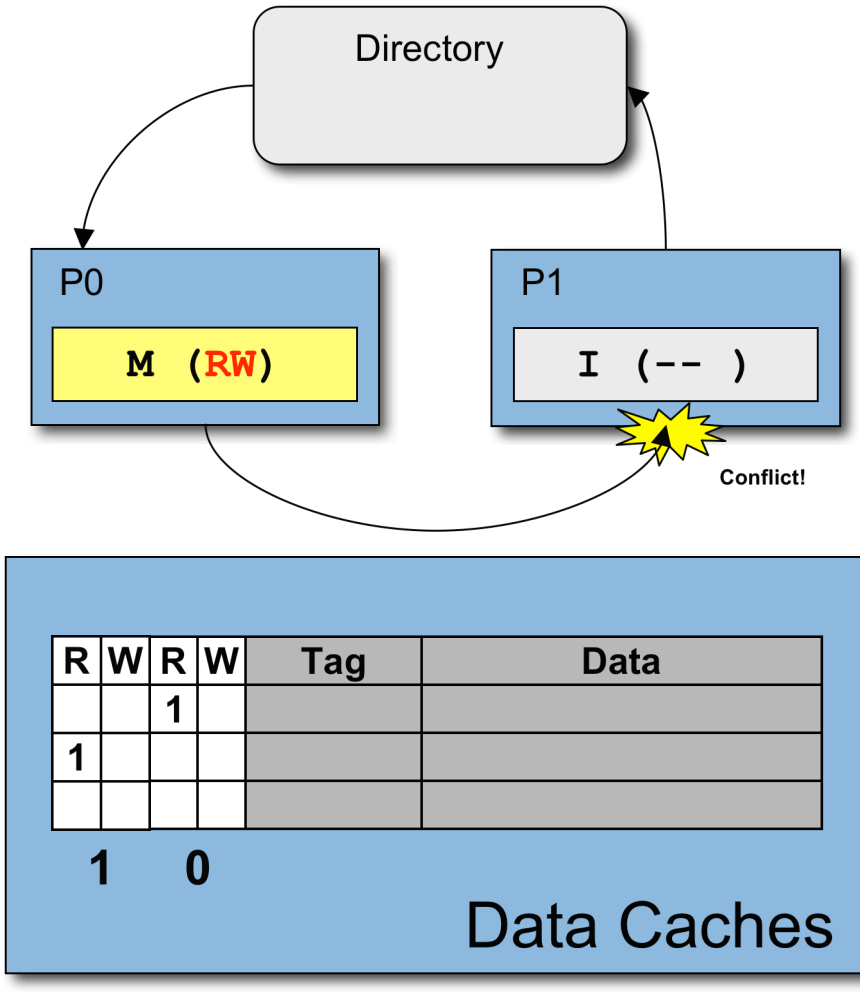
Conflict Detection in Nested LogTM

- Flat LogTM detects conflicts with directory coherence and Read (R) and Write (W) bits in caches
- Nested LogTM replicates R/W bits for each level
- Flash-Or circuit merges child and parent R/W bits



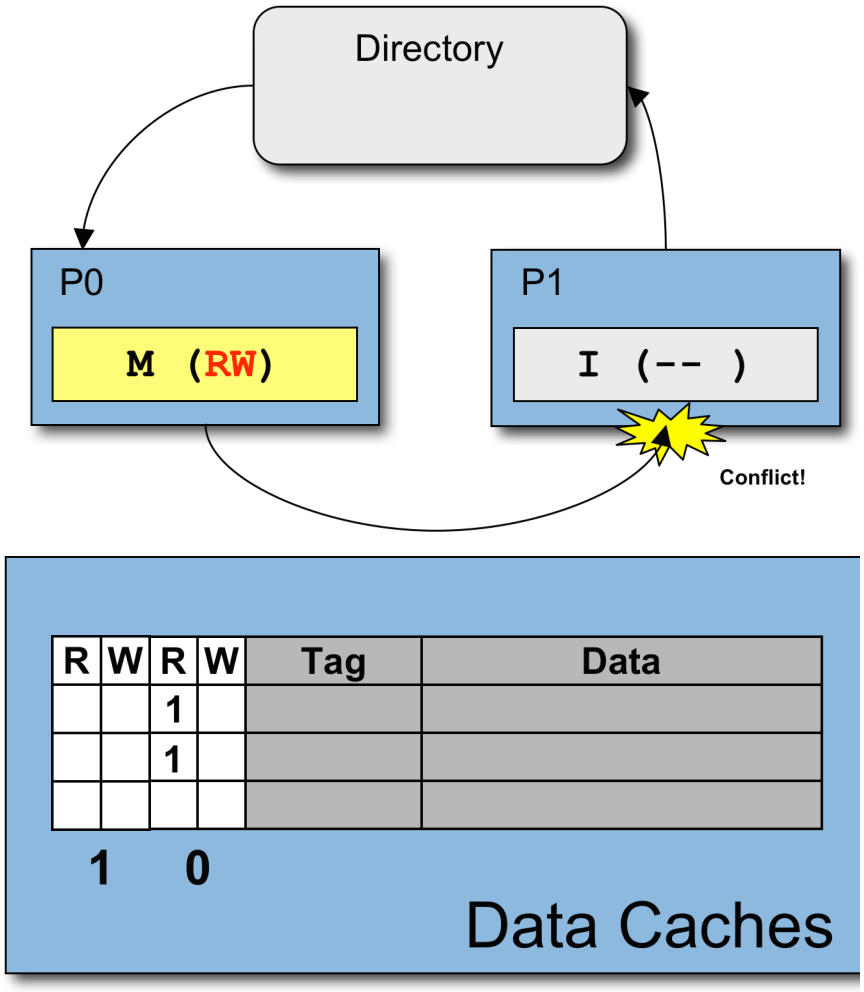
Conflict Detection in Nested LogTM

- Flat LogTM detects conflicts with directory coherence and Read (R) and Write (W) bits in caches
- Nested LogTM replicates R/W bits for each level
- Flash-Or circuit merges child and parent R/W bits



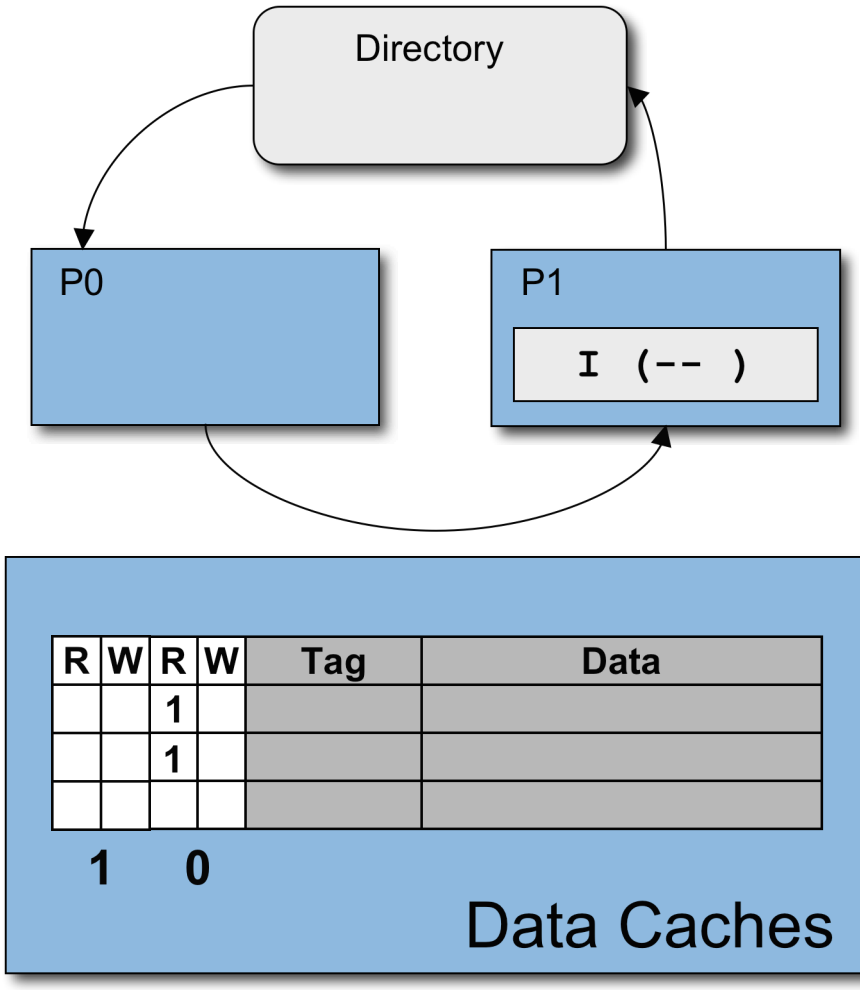
Conflict Detection in Nested LogTM

- Flat LogTM detects conflicts with directory coherence and Read (R) and Write (W) bits in caches
- Nested LogTM replicates R/W bits for each level
- Flash-Or circuit merges child and parent R/W bits

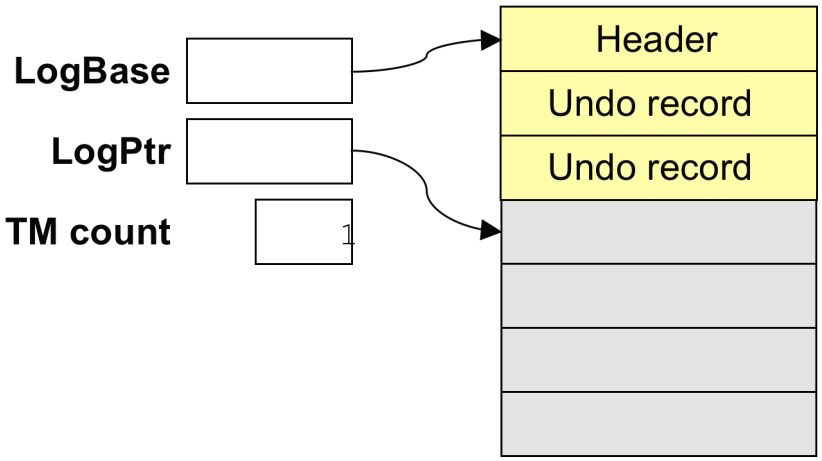


Conflict Detection in Nested LogTM

- Flat LogTM detects conflicts with directory coherence and Read (R) and Write (W) bits in caches
- Nested LogTM replicates R/W bits for each level
- Flash-Or circuit merges child and parent R/W bits

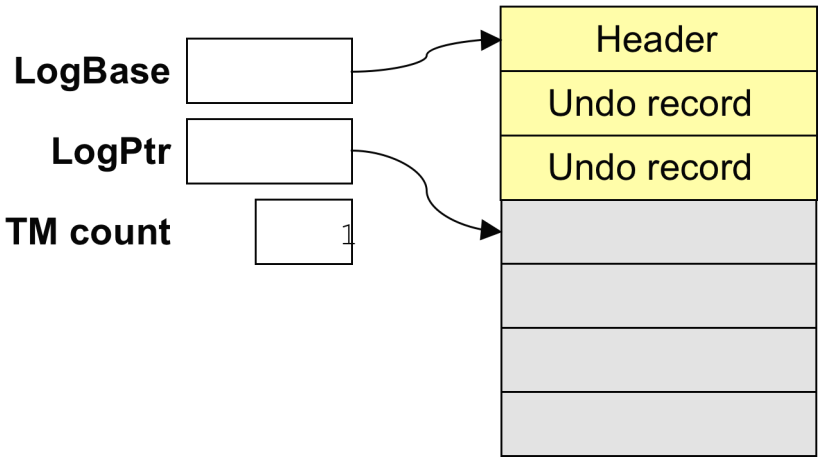


Version Management in Nested LogTM



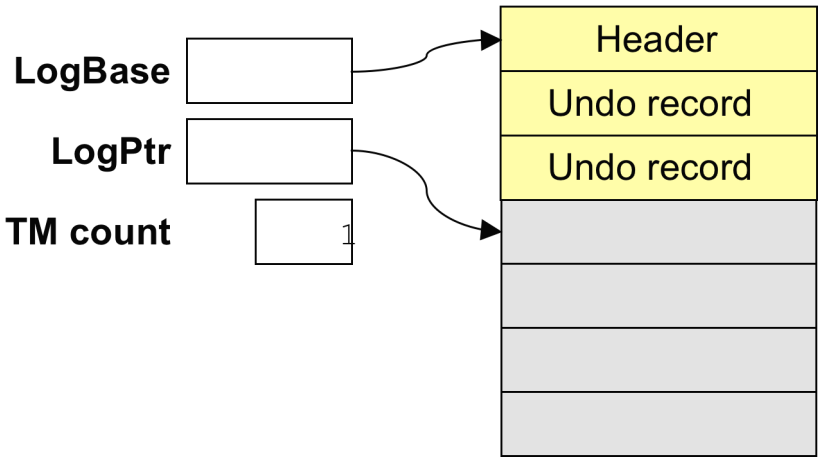
Version Management in Nested LogTM

- Flat LogTM's log is a single frame (header + undo records)



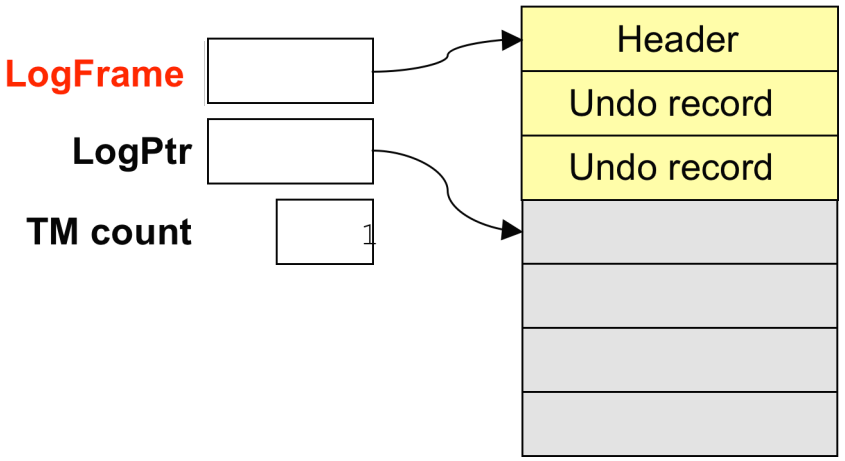
Version Management in Nested LogTM

- Flat LogTM's log is a single frame (header + undo records)
- Nested LogTM's log is a stack of frames



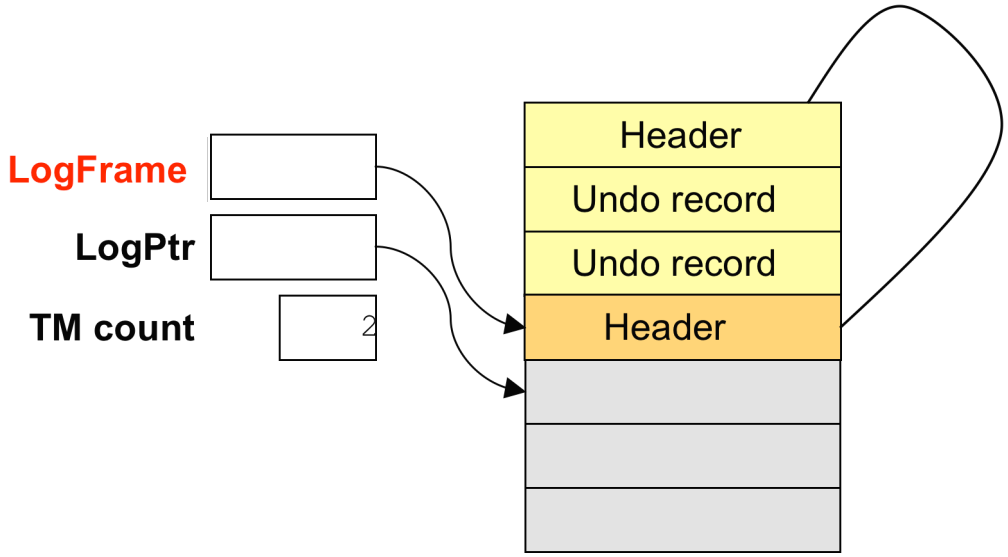
Version Management in Nested LogTM

- Flat LogTM's log is a single frame (header + undo records)
- Nested LogTM's log is a stack of frames



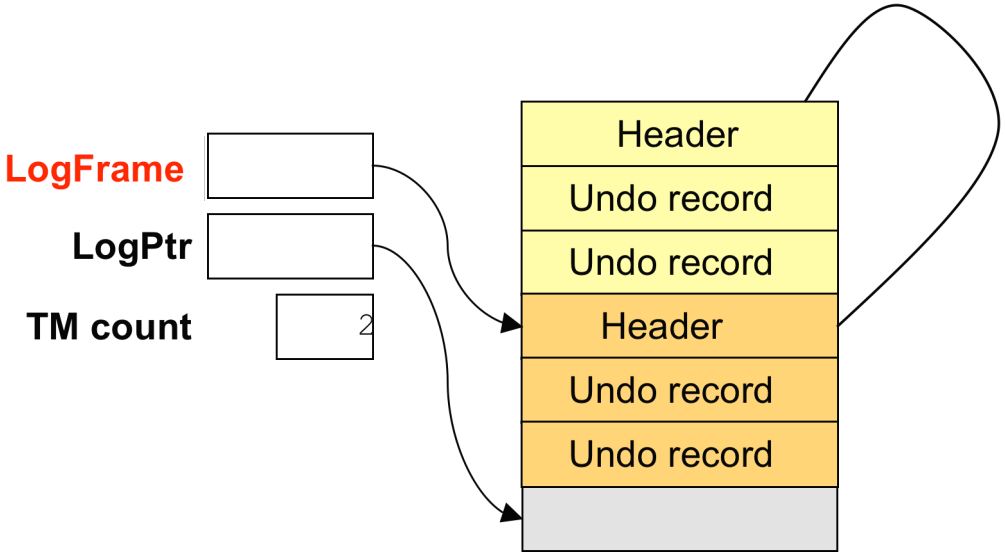
Version Management in Nested LogTM

- Flat LogTM's log is a single frame (header + undo records)
- Nested LogTM's log is a stack of frames
- A frame contains:
  - Header (including saved registers and pointer to parent's frame)



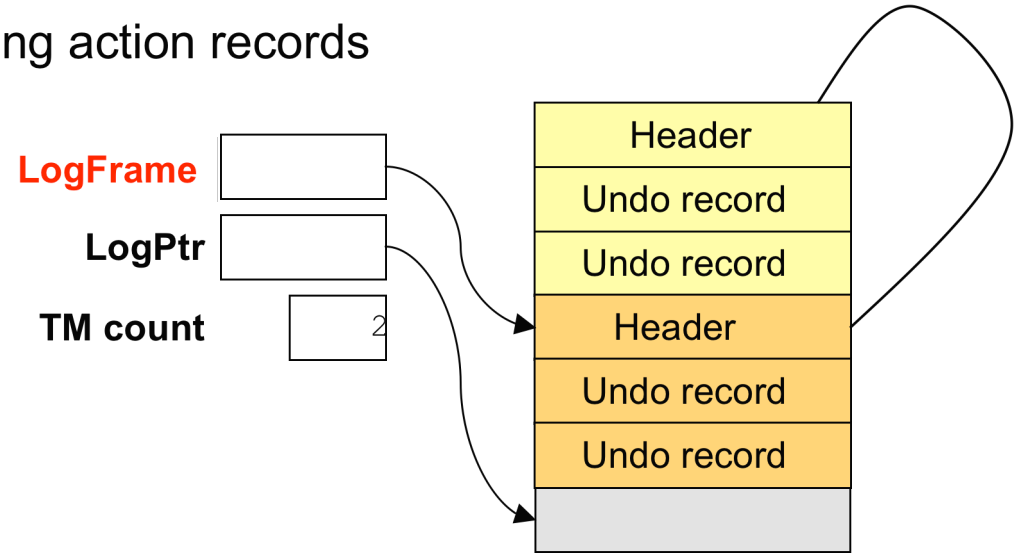
Version Management in Nested LogTM

- Flat LogTM's log is a single frame (header + undo records)
- Nested LogTM's log is a stack of frames
- A frame contains:
  - Header (including saved registers and pointer to parent's frame)
  - Undo records (block address, old value pairs)



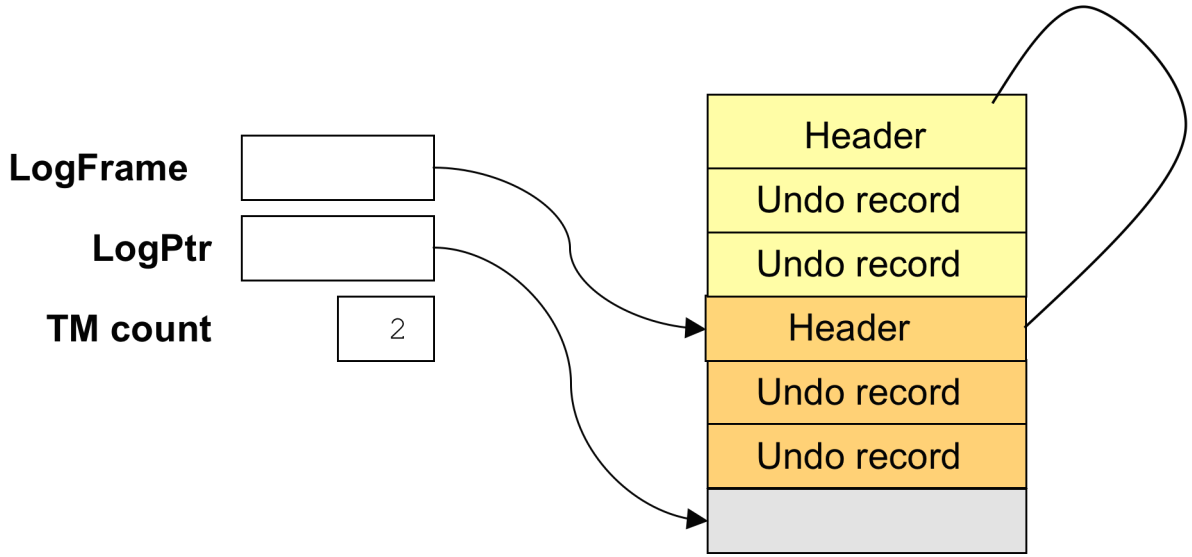
Version Management in Nested LogTM

- Flat LogTM's log is a single frame (header + undo records)
- Nested LogTM's log is a stack of frames
- A frame contains:
  - Header (including saved registers and pointer to parent's frame)
  - Undo records (block address, old value pairs)
  - Garbage headers (headers of committed closed transactions)
  - Commit action records
  - Compensating action records



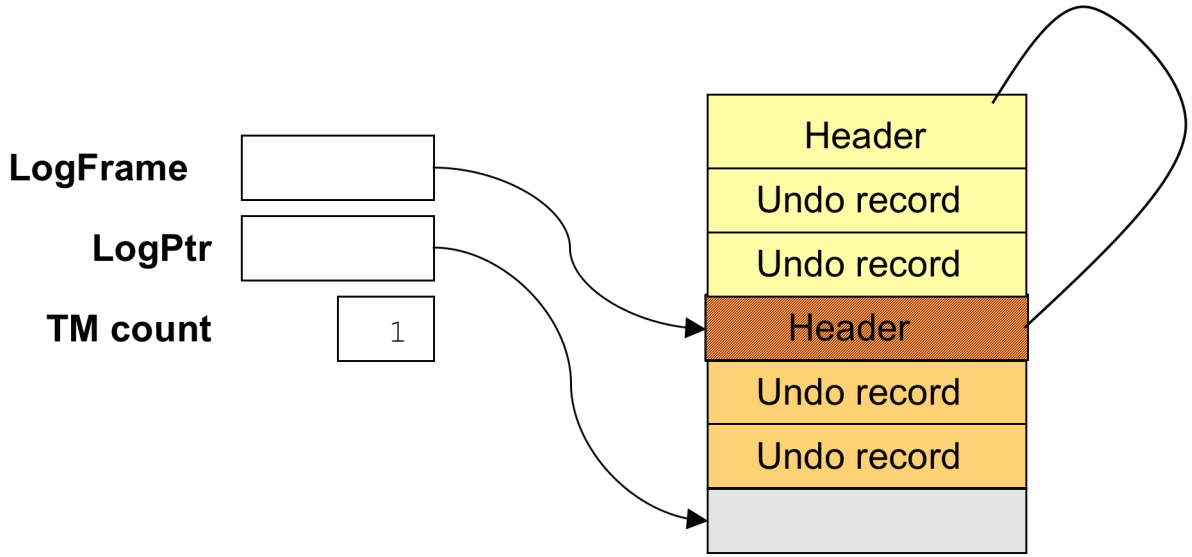
Closed Nested Commit

- Merge child's log frame with parent's



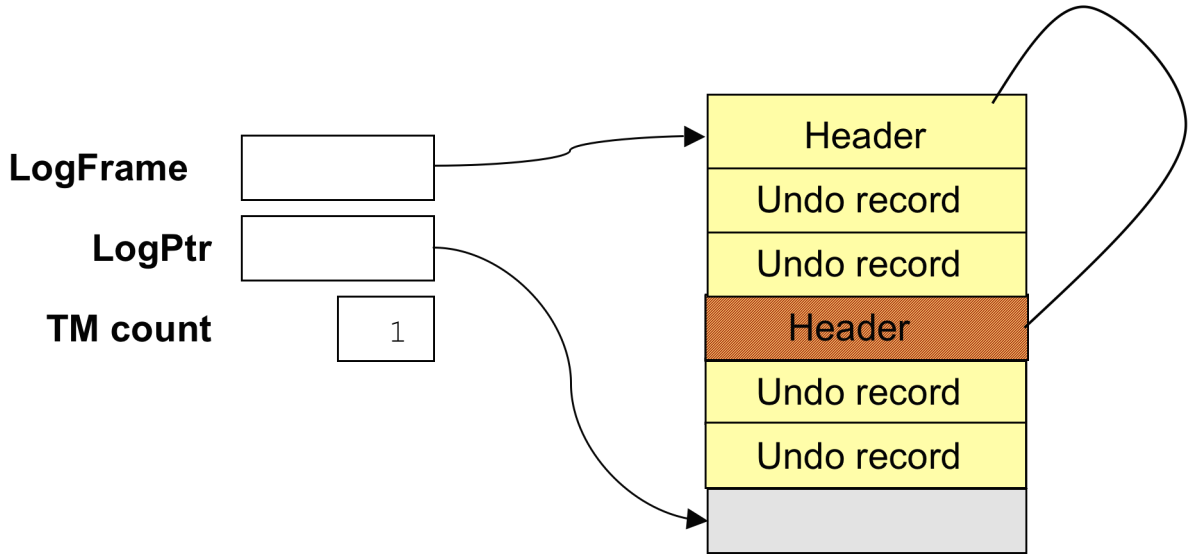
Closed Nested Commit

- Merge child's log frame with parent's
  - Mark child's header as a "garbage header"



Closed Nested Commit

- Merge child's log frame with parent's
  - Mark child's header as a "garbage header"
  - Copy pointer from child's header to LogFrame



Concurrency

Increased Concurrency: Open Nesting

Increased Concurrency: Open Nesting

```
insert(int key, int value) {
   begin_transaction();
        leaf = find_leaf(key);
        entry = insert_into_leaf(key, value);
        // lock entry to isolate node

   commit_transaction();

}

insert_set(set S) {
   begin_transaction();
        while ((key,value) = next(S)){
               insert(key, value);
        }
   commit_transaction();
}
```

Open Nesting

Child transaction exposes state on commit
(i.e., **before the parent commits**)

- Higher-level **atomicity**
  - Child's memory updates not undone if parent aborts
  - Use **compensating action** to undo the child's forward action at a higher-level of abstraction
    - E.g., malloc() compensated by free()
- Higher-level **isolation**
  - Release memory-level isolation
  - Programmer enforce isolation at higher level (e.g., locks)
  - Use **commit action** to release isolation at parent commit

Increased Concurrency: Open Nesting

```
insert(int key, int value) {
   open_begin;
        leaf = find_leaf(key);
        entry = insert_into_leaf(key, value);
        // lock entry to isolate node
        entry->lock = 1;
   open_commit(abort_action(delete(key)),

          commit_action(unlock(key)));
}

insert_set(set S) {
   open_begin;
        while ((key,value) = next(S))
              insert(key, value);
   open_commit(abort_action(delete_set(S)));
}
```

Increased Concurrency: Open Nesting

```
insert(int key, int value) {
   open_begin;
        leaf = find_leaf(key);
        entry = insert_into_leaf(key, value);
        // lock entry to isolate node
        entry->lock = 1;
   open_commit(abort_action(delete(key)),
          commit_action(unlock(key)));
}

insert_set(set S) {
   open_begin;
        while ((key,value) = next(S))
              insert(key, value);
   open_commit(abort_action(delete_set(S)));
}
```

← Delete entry if ancestor aborts

Increased Concurrency: Open Nesting

```
insert(int key, int value) {
   open_begin;
        leaf = find_leaf(key);
        entry = insert_into_leaf(key, value);
        // lock entry to isolate node
        entry->lock = 1; ← Isolate entry at higher-level of abstraction
   open_commit(abort_action(delete(key)),← Delete entry if
                                          ancestor aborts
          commit_action(unlock(key)));
}

insert_set(set S) {
   open_begin;
        while ((key,value) = next(S))
              insert(key, value);
   open_commit(abort_action(delete_set(S)));
}
```

Increased Concurrency: Open Nesting

```
insert(int key, int value) {
   open_begin;
        leaf = find_leaf(key);
        entry = insert_into_leaf(key, value);
        // lock entry to isolate node
        entry->lock = 1; ← Isolate entry at higher-level of abstraction
   open_commit(abort_action(delete(key)),← Delete entry if
                                          ancestor aborts
          commit_action(unlock(key)));← Release high-level
}                                       isolation on ancestor commit

insert_set(set S) {
   open_begin;
        while ((key,value) = next(S))
              insert(key, value);
   open_commit(abort_action(delete_set(S)));
}
```

Increased Concurrency: Open Nesting

```
insert(int key, int value) {
   open_begin;
        leaf = find_leaf(key);
        entry = insert_into_leaf(key, value);
        // lock entry to isolate node
        entry->lock = 1; ← Isolate entry at higher-level of abstraction
   open_commit(abort_action(delete(key)),← Delete entry if
                                          ancestor aborts
          commit_action(unlock(key)));← Release high-level
}                                       isolation on ancestor commit

insert_set(set S) {
   open_begin;
        while ((key,value) = next(S))
              insert(key, value);
   open_commit(abort_action(delete_set(S))); ← Replace
}                                    compensating action with higher-
                                     level action on commit
```

Commit and Compensating Actions

- **Commit Actions**
  - Execute in FIFO order when innermost open ancestor commits
    - Outermost transaction is considered open
- **Compensating Actions**
  - Discard when innermost open ancestor commits
  - Execute in LIFO order when ancestor aborts
  - Execute “in the state that held when its forward action commited” [Moss, TRANSACT ‘06]

Timing of Compensating Actions

```
// initialize to 0
counter = 0;
transaction_begin(); // top-level 1
    counter++; // counter gets 1
    open_begin(); // level 2
       counter++; // counter gets 2
    open_commit(abort_action(counter--));
    ...
// Abort and run compensating action
// Expect counter to be restored to 0
...
transaction_commit(); // not executed
```

Condition O1

Condition O1: An open nested child transaction never modifies a memory location that has been modified by any ancestor.

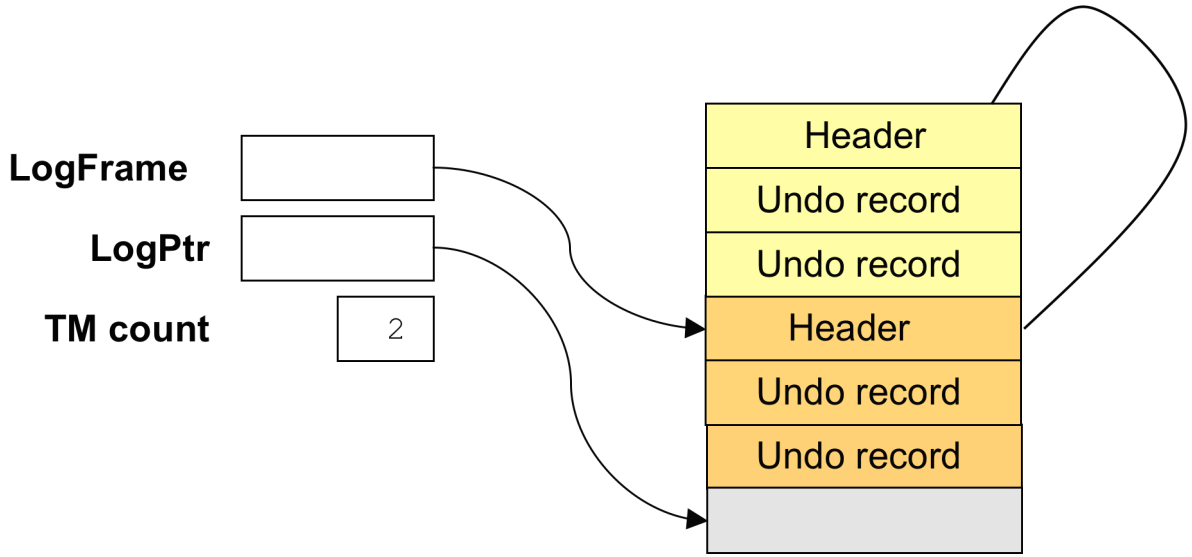
- If condition O1 holds programmers need not reason about the interaction between **compensation** and **undo**
- All implementations of nesting (so far) agree on semantics when O1 holds

Open Nesting in LogTM

- Conflict Detection
  - R/W bits **cleared** on open commit
  - (no flash or)
- Version Management
  - Open commit pops the most recent frame off the log
  - (Optionally) add commit and compensating action records
  - Compensating actions are run by the software abort handler
  - Software handler **interleaves** restoration of memory state and compensating action execution

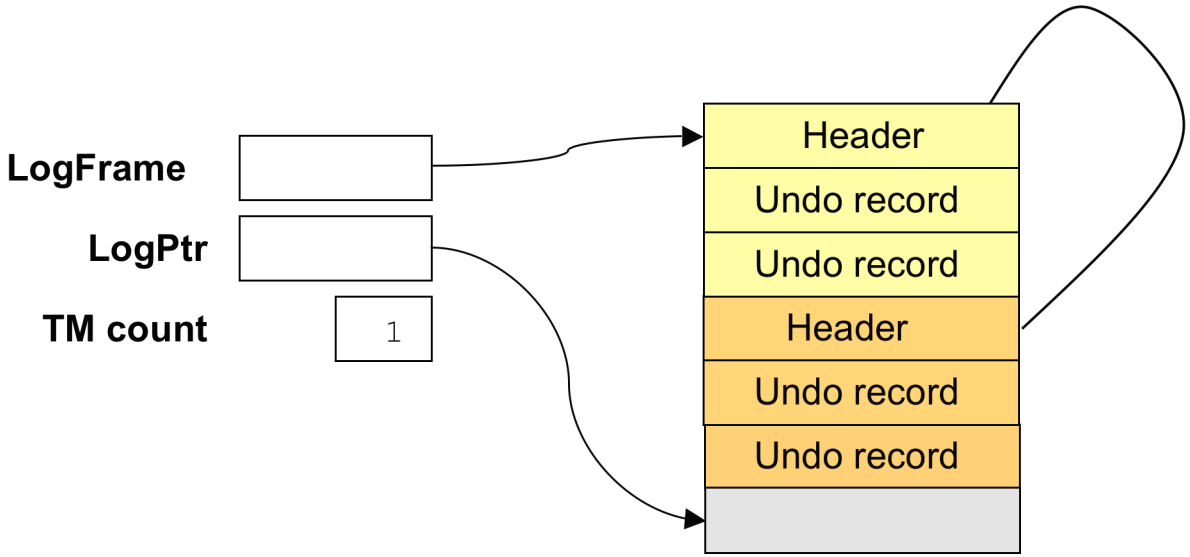
Open Nested Commit

- Discard child's log frame



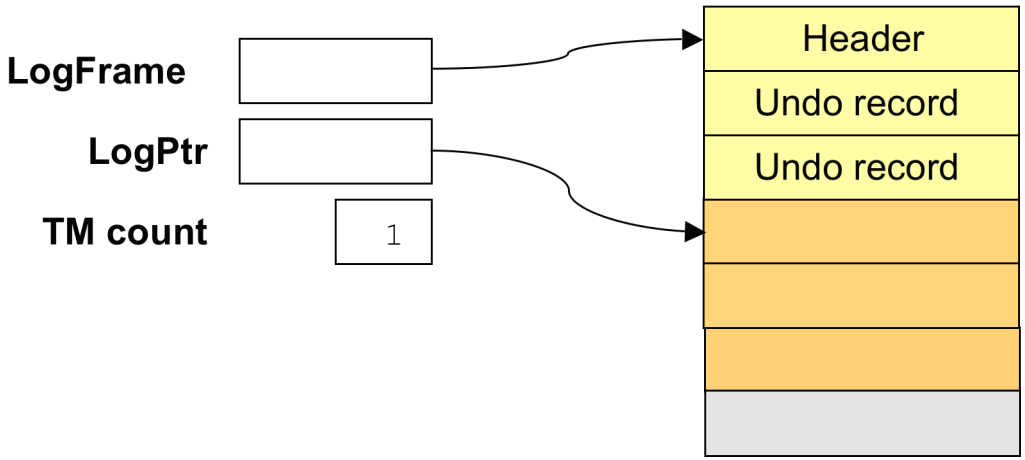
Open Nested Commit

- Discard child's log frame



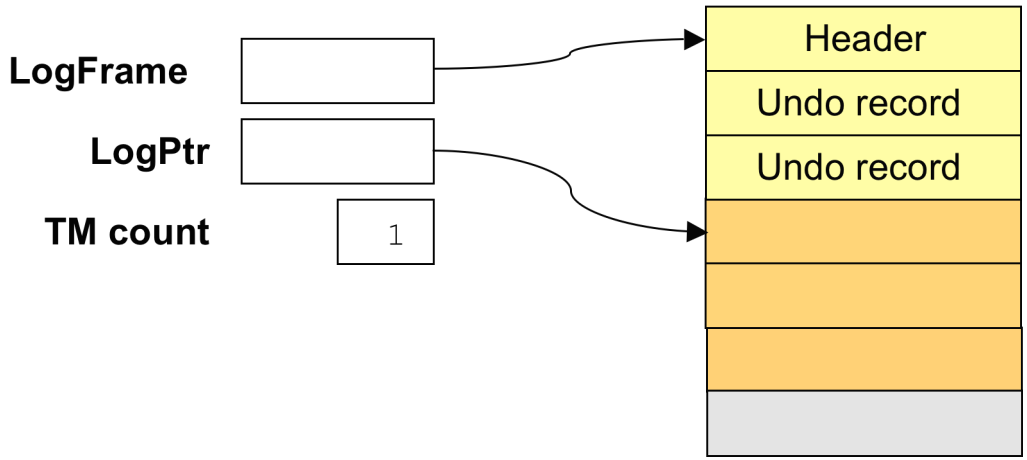
Open Nested Commit

- Discard child's log frame



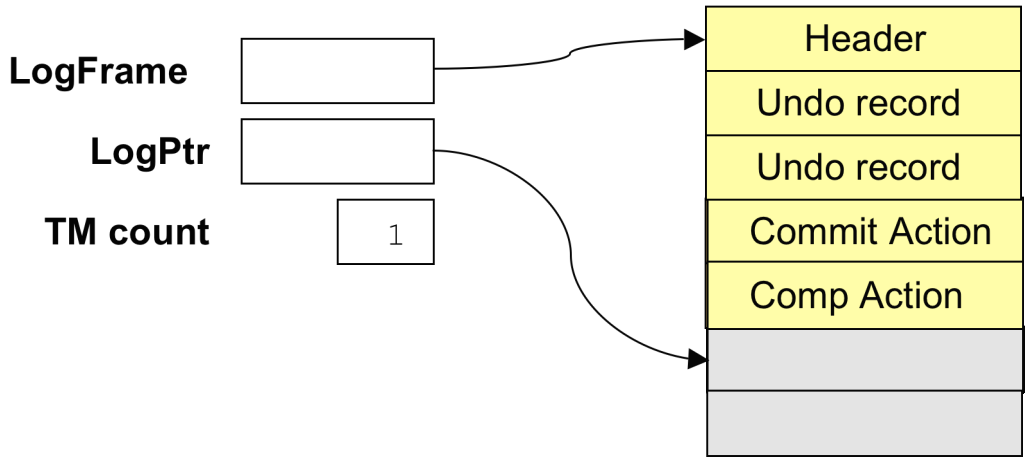
Open Nested Commit

- Discard child's log frame
- (Optionally) append commit and compensating actions to log



Open Nested Commit

- Discard child's log frame
- (Optionally) append commit and compensating actions to log



Timing of Compensating Actions

```
// initialize to 0
counter = 0;
transaction_begin(); // top-level 1
    counter++; // counter gets 1
    open_begin(); // level 2
       counter++; // counter gets 2
    open_commit(abort_action(counter--));
    ...
// Abort and run compensating action
// Expect counter to be restored to 0
...
transaction_commit(); // not executed
```

LogTM behaves correctly:

Compensating action sees the state of the counter when the open transaction committed (2)

Decrement restores the value to what it was before the open nest executed (1)

Undo of the parent restores the value back to (0)

Timing of Compensating Actions

```
// initialize to 0
counter = 0;
transaction_begin(); // top-level 1
    counter++; // counter gets 1
    open_begin(); // level 2
       counter++; // counter gets 2
    open_commit(abort_action(counter--));
    ...
// Abort and run compensating action
// Expect counter to be restored to 0
...
transaction_commit(); // not executed
```

LogTM behaves correctly:

Compensating action sees the state of the counter when the open transaction committed (2)

Decrement restores the value to what it was before the open nest executed (1)

Undo of the parent restores the value back to (0)

Condition O1: No writes to blocks written by an ancestor transaction.

Communication

Communication: Escape Actions

- “Real world” is not transactional
- Current OS’s are not transactional

- Systems should allow non-transactional escapes from a transaction
- Interact with OS, VM, devices, etc.

Escape Actions

Escape actions bypass transaction isolation and version management.

- Escape actions never:
  - Abort
  - Stall
  - Cause other transactions to abort
  - Cause other transactions to stall
- Commit and compensating actions
  - similar to open nested transactions

Not recommended for the average programmer!

Case Study: System Calls in Solaris

Category	#	Examples
Read-only	57	getpid, times, stat, access, mincore, sync, pread, gettimeofday
Undoable (without global side effects)	40	chdir, dup, umask, seteuid, nice, seek, mprotect
Undoable (with global side effects)	27	chmod, mkdir, link, mknod, stime
Calls **not** handled by escape actions	89	kill, fork, exec, umount

Escape Actions in LogTM

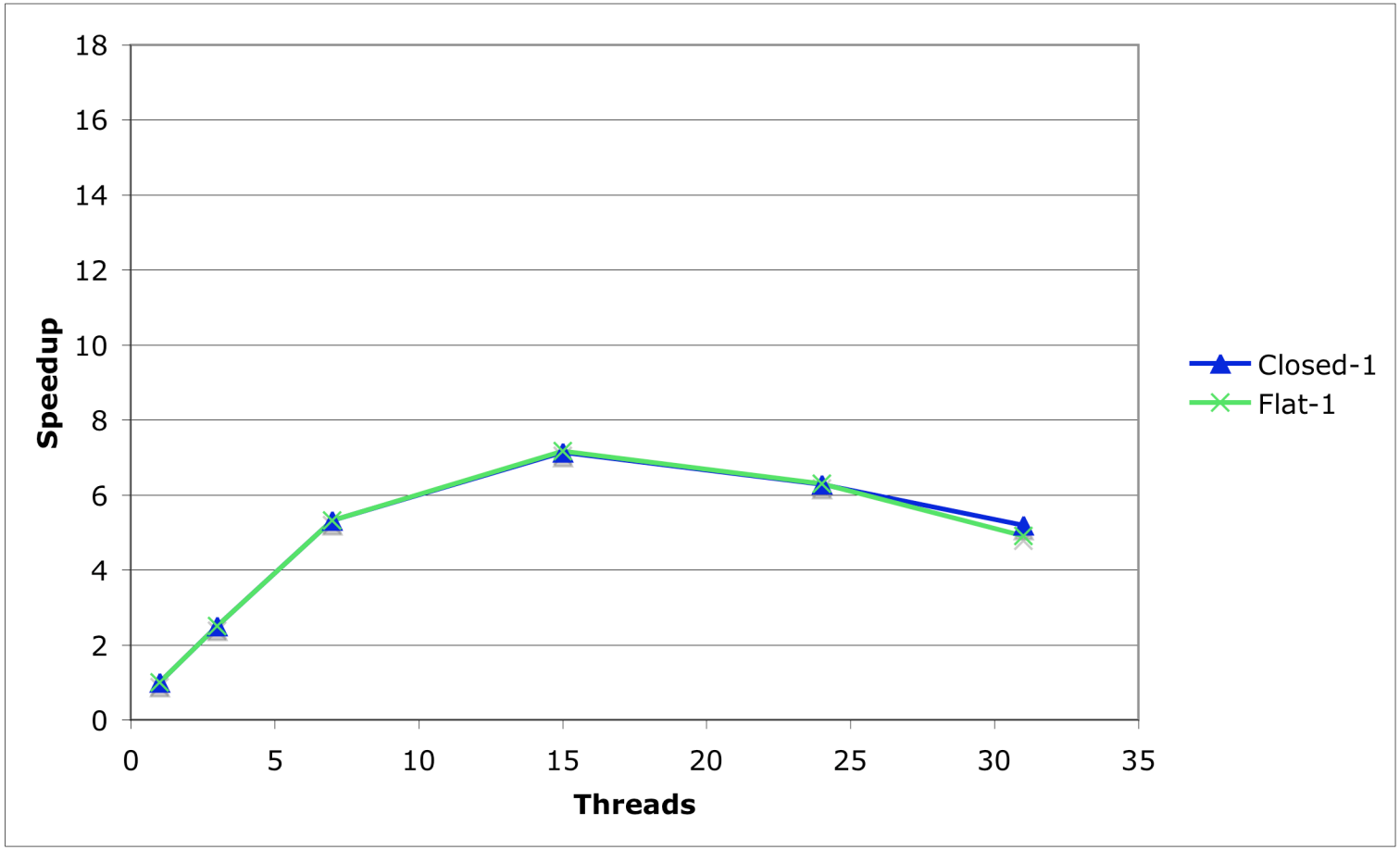
- Loads and stores to non-transactional blocks behave as normal coherent accesses
- Loads return the latest value in coherent memory
  - Loads to a transactionally modified cache block triggers a writeback (sticky-M state)
  - Memory responds with an uncacheable copy of the block
- Stores modify coherent memory
  - Stores to transactionally modified blocks trigger writebacks (sticky-M)
  - Updates the value in memory (non-cacheable write through)

Methods

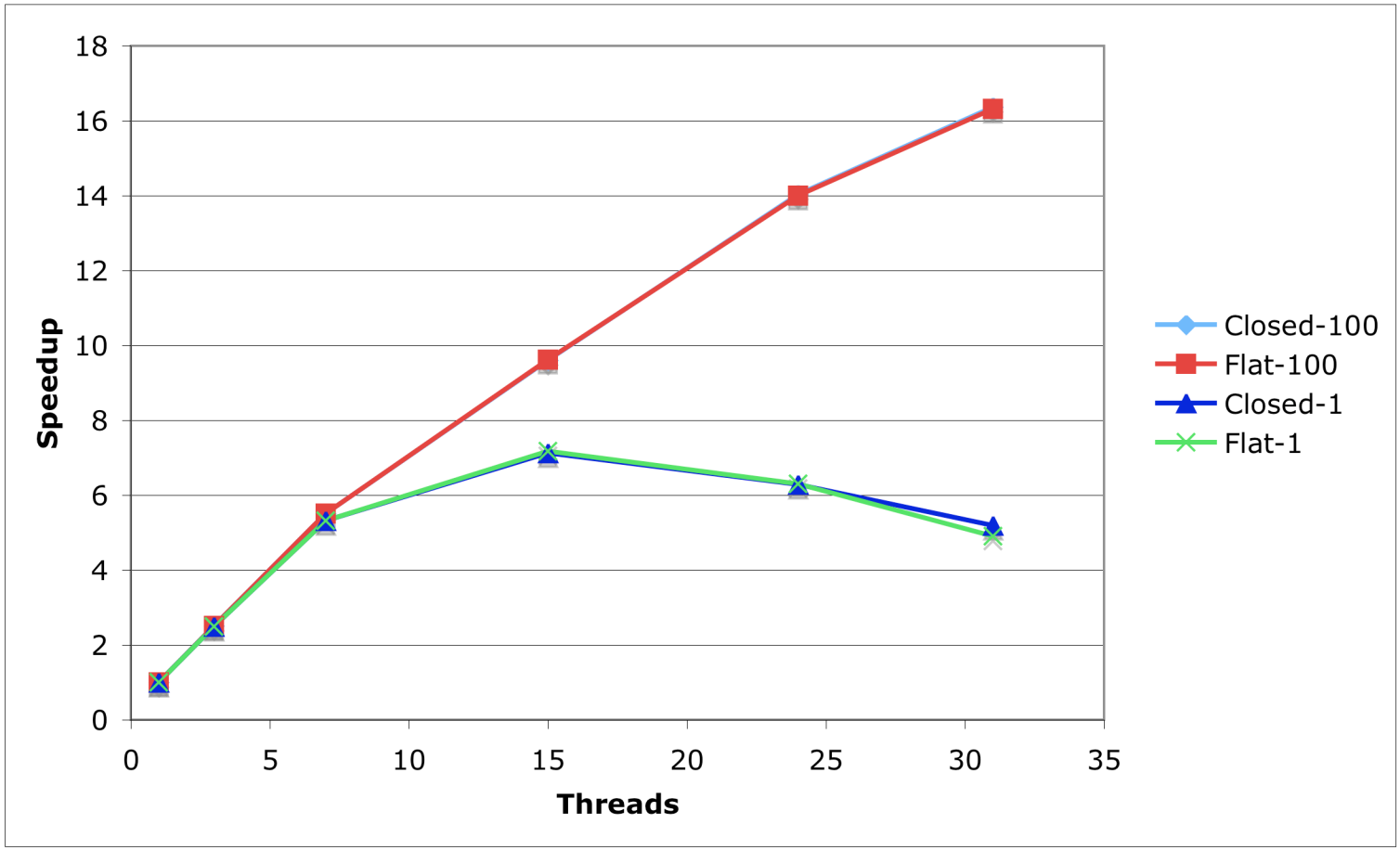
- Simulated Machine: 32-way non-CMP
  - 32 SPARC V9 processors running Solaris 9 OS
  - 1 GHz in-order processors w/ ideal IPC=1 & private caches
  - 16 kB 4-way split L1 cache, 1 cycle latency
  - 4 MB 4-way unified L2 cache, 12 cycle latency
  - 4 GB main memory, 80-cycle access latency
  - Full-bit vector directory w/ directory cache
  - Hierarchical switch interconnect, 14-cycle latency

- Simulation Infrastructure
  - Virtutech Simics for full-system function
  - Multifacet GEMS for memory system timing (Ruby only)
    GPL Release: http://www.cs.wisc.edu/gems/
  - Magic no-ops instructions for `begin_transaction()` etc.

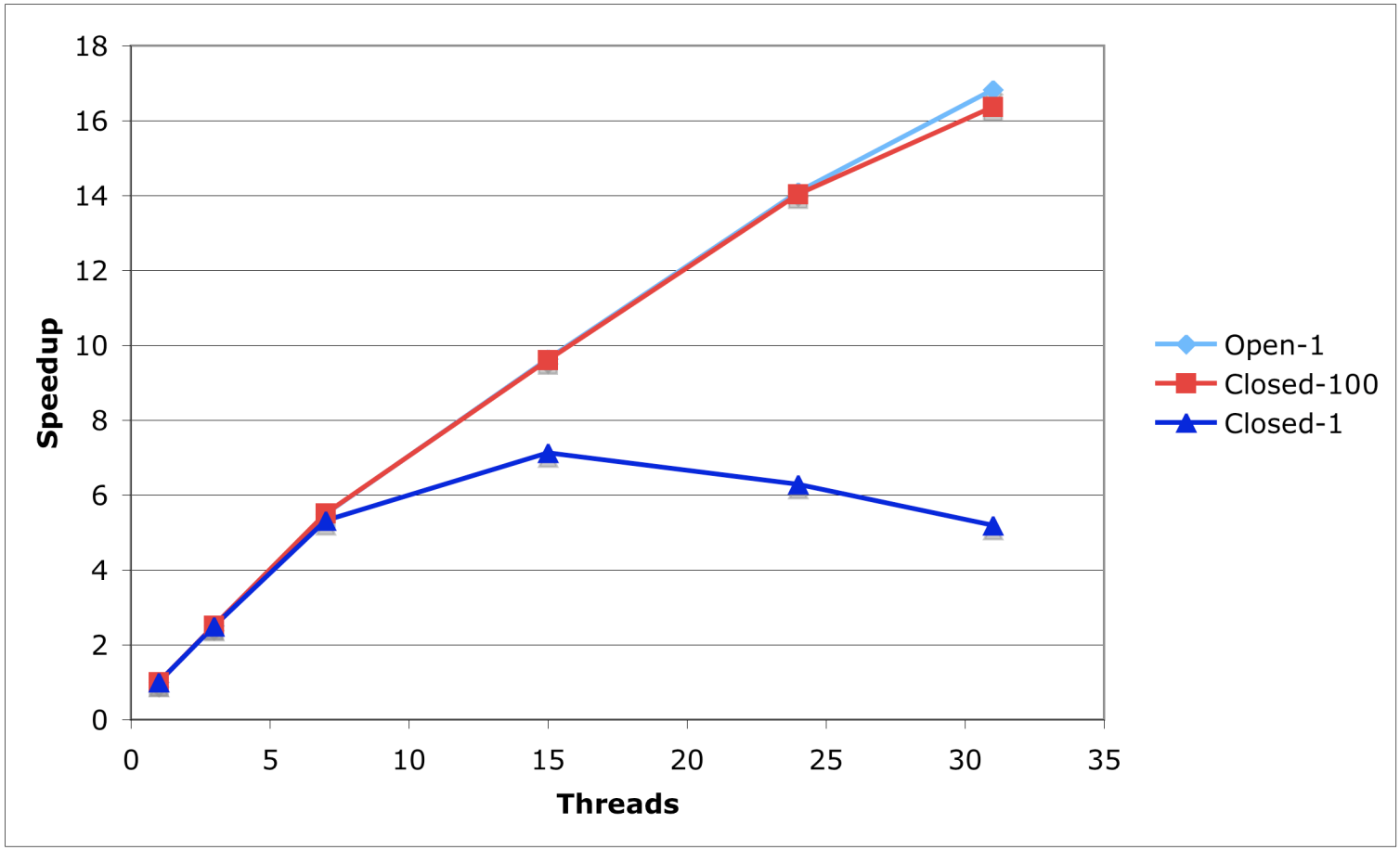
B-Tree: Closed Nesting



B-Tree: Closed Nesting



B-Tree: Open Nesting



Conclusions

- **Closed Nesting** (partial rollback)
  - Easy to implement--segment the transaction log (stack of log frames)/Replicate R & W bits
  - Small performance gains for LogTM
- **Open Nesting**
  - Easy to implement--software abort handling allows easy execution of **commit actions** and **compensating actions**
  - Big performance gains
  - Added software complexity
- **Escape Actions**
  - Provide non-transactional operations inside transactions
  - Sufficient for most Solaris system calls

BACKUP SLIDES

How Do Transactional Memory Systems Differ?

- (Data) Version Management
  - **Eager**: record old values “elsewhere”; update “in place”
  - **Lazy**: update “elsewhere”; keep old values “in place”

How Do Transactional Memory Systems Differ?

- (Data) Version Management
  - **Eager**: record old values “elsewhere”; update “in place” ← Fas commi
  - **Lazy**: update “elsewhere”; keep old values “in place”

How Do Transactional Memory Systems Differ?

- (Data) Version Management
  - **Eager**: record old values "elsewhere"; update "in place" ← Fas commi
  - **Lazy**: update "elsewhere"; keep old values "in place"
- (Data) Conflict Detection
  - **Eager**: detect conflict on every read/write
  - **Lazy**: detect conflict at end (commit/abort)

How Do Transactional Memory Systems Differ?

- (Data) Version Management
  - **Eager**: record old values "elsewhere"; update "in place" ← Fas commi
  - **Lazy**: update "elsewhere"; keep old values "in place"
- (Data) Conflict Detection
  - **Eager**: detect conflict on every read/write ← Less wasted work
  - **Lazy**: detect conflict at end (commit/abort)

Microbenchmark Analysis

- Shared Counter
  - All threads update the same counter
  - High contention
  - Small Transactions
- LogTM v. Locks
  - EXP - Test-And-Test-And-Set Locks with Exponential Backoff
  - MCS - Software Queue-Based Locks

```
BEGIN_TRANSACTION();

 new_total = total.count + 1;
 private_data[ id] .count++;
 total.count = new_total;

COMMIT_TRANSACTION();
```

Locks are Hard

```
// WITH LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Locks are Hard

```
// WITH LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Thread 0

```
move(a, b, key1);
```

Thread 1

```
move(b, a, key2);
```

Locks are Hard

```
// WITH LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Thread 0

```
move(a, b, key1);
```

Thread 1

```
move(b, a, key2);
```

DEADLOCK!

Locks are Hard

```
// WITH LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Thread 0

```
move(a, b, key1);
```

Thread 1

```
move(b, a, key2);
```

DEADLOCK!

Moreover

Coarse-grain locking limits concurrency

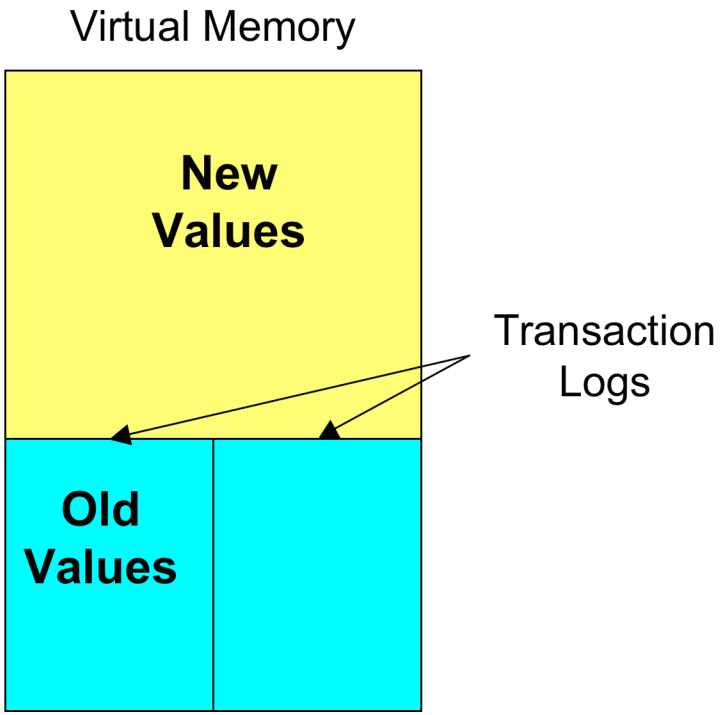
Fine-grain locking difficult

Eager Version Management Discussion

- Advantages:
  - No extra indirection (unlike STM)
  - Fast Commits
    - No copying
    - Common case
- Disadvantages
  - Slow/Complex Aborts
    - Undo aborting transaction
  - Relies on Eager Conflict Detection/Prevention

Log-Based Transactional Memory (LogTM)

- New values stored in place (even in main memory)
- Old values stored in a thread-private **transaction log**
- Aborts processed in software



HPCA 2006 - **LogTM: Log-Based Transactional Memory**, Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill and David A. Wood

Strided Array

Sticky States

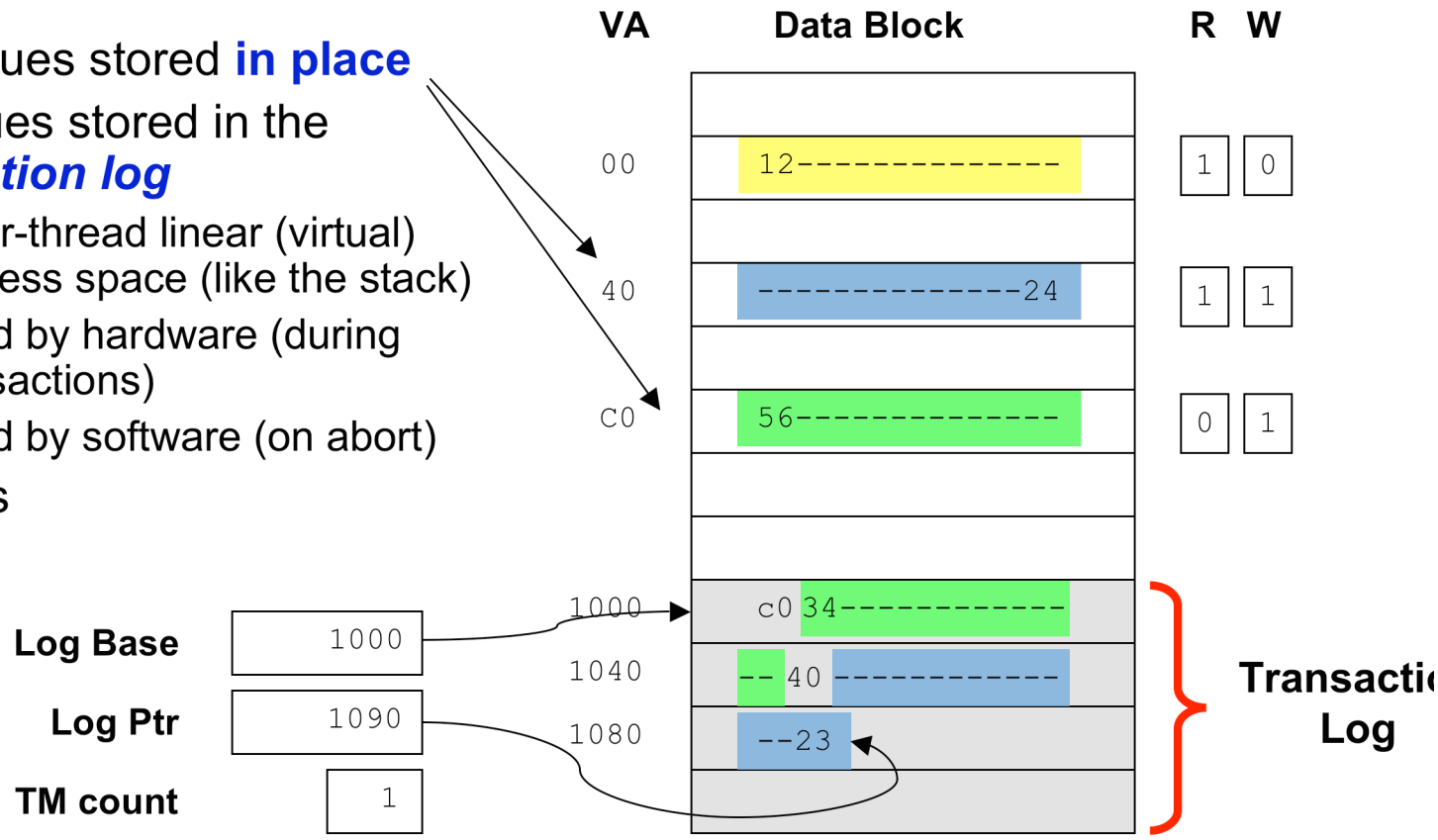
Directory State

Cache State

	M	S	I
M	**M**		
E	**E**		
S		**S**	
I	**Sticky-M**	**Sticky-S**	**I**

Flat LogTM (HPCA'06)

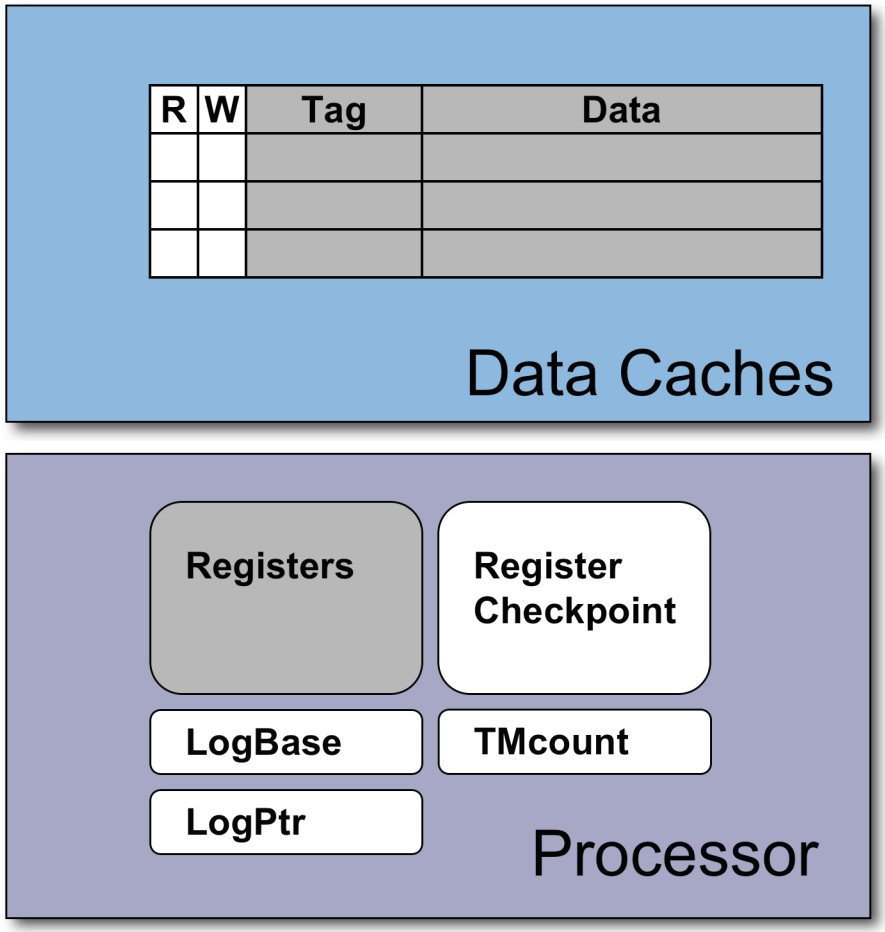
- New values stored **in place**
- Old values stored in the ***transaction log***
  - A per-thread linear (virtual) address space (like the stack)
  - Filled by hardware (during transactions)
  - Read by software (on abort)
- R/W bits



<example>

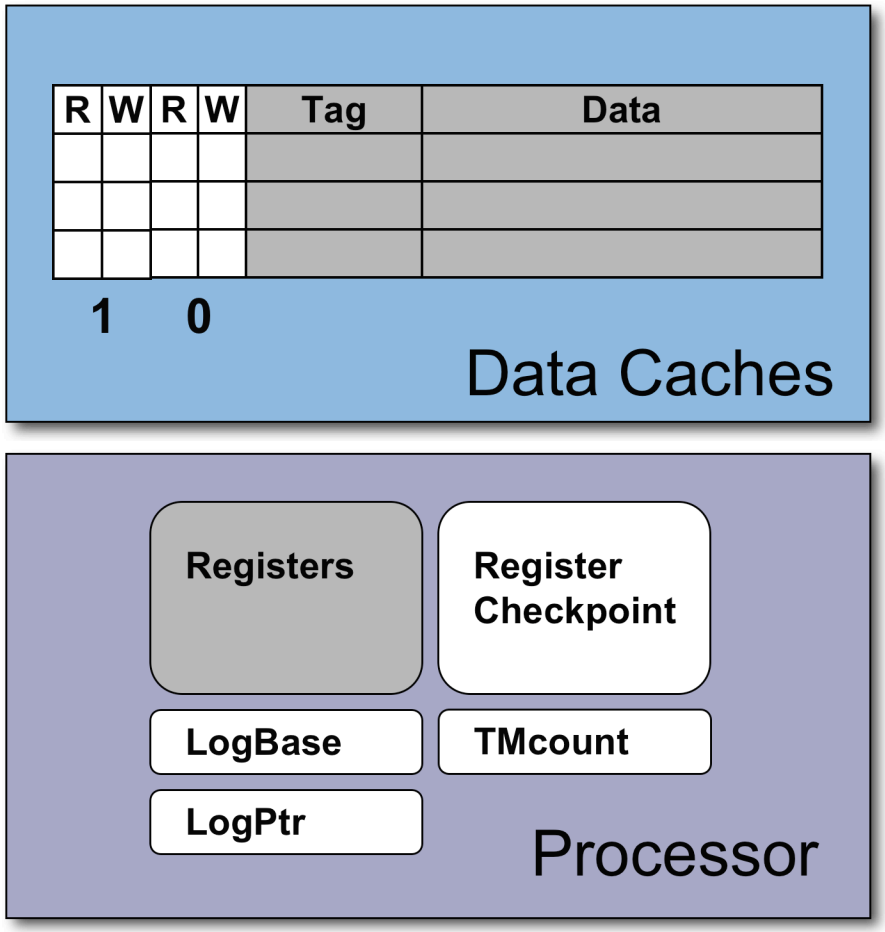
Conflict Detection in LogTM

- Conflict Detection
  - Nested LogTM replicates R/W bits for each level
  - Flash-Or circuit merges child and parent R/W bits
- Version Management
  - Nested LogTM segments the log into frames
  - (similar to a stack of activation records)



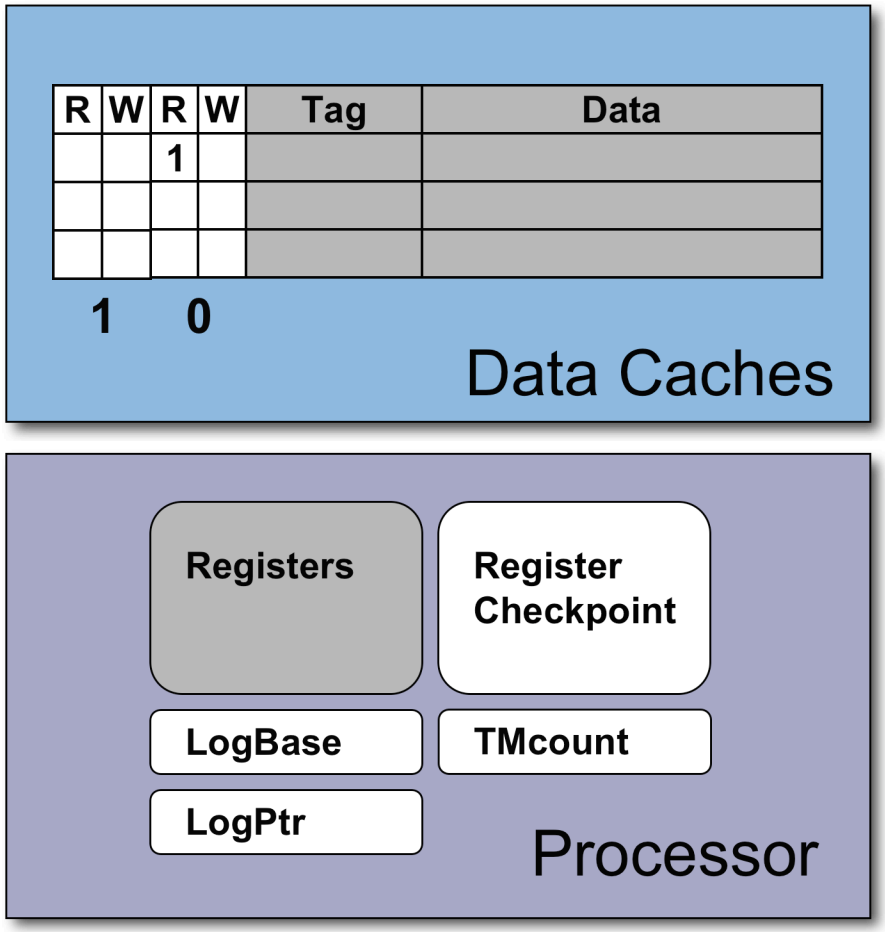
Conflict Detection in LogTM

- Conflict Detection
  - Nested LogTM replicates R/W bits for each level
  - Flash-Or circuit merges child and parent R/W bits
- Version Management
  - Nested LogTM segments the log into frames
  - (similar to a stack of activation records)



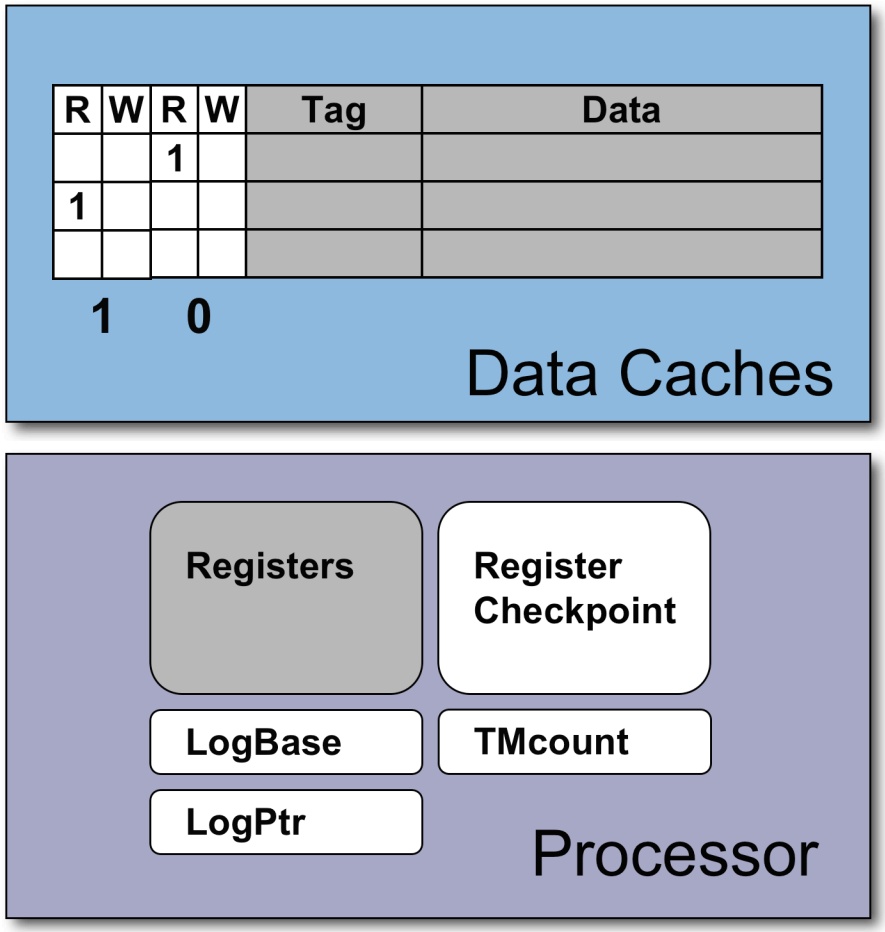
Conflict Detection in LogTM

- Conflict Detection
  - Nested LogTM replicates R/W bits for each level
  - Flash-Or circuit merges child and parent R/W bits
- Version Management
  - Nested LogTM segments the log into frames
  - (similar to a stack of activation records)



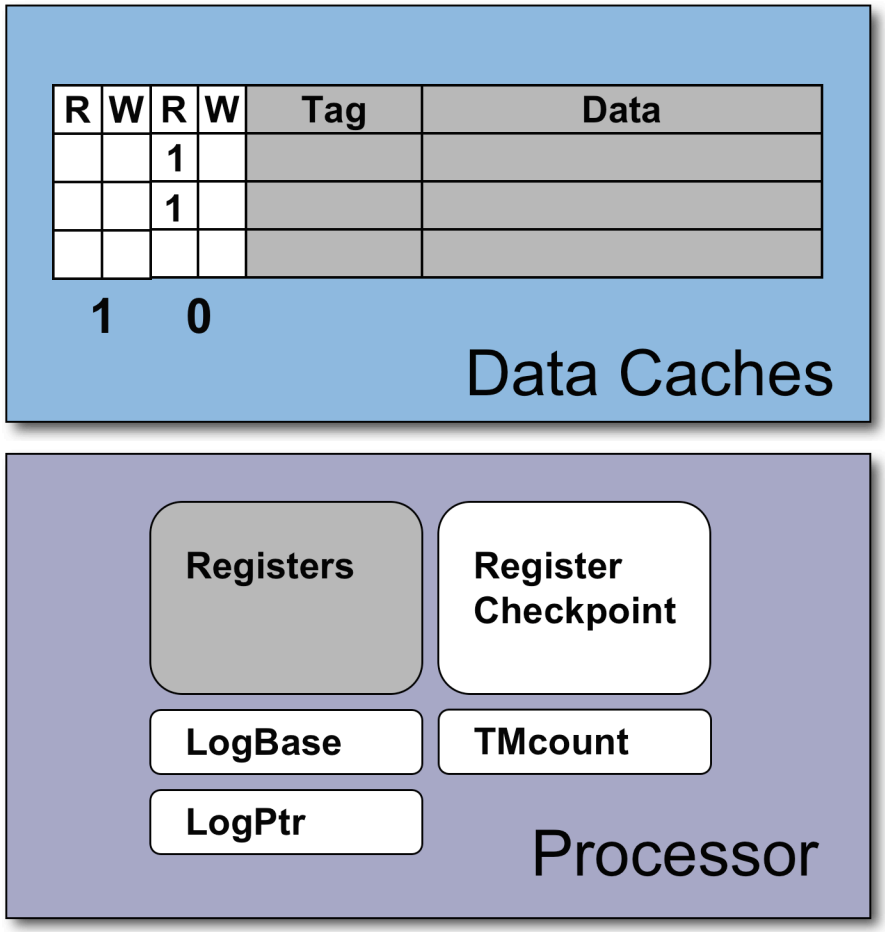
Conflict Detection in LogTM

- Conflict Detection
  - Nested LogTM replicates R/W bits for each level
  - Flash-Or circuit merges child and parent R/W bits
- Version Management
  - Nested LogTM segments the log into frames
  - (similar to a stack of activation records)



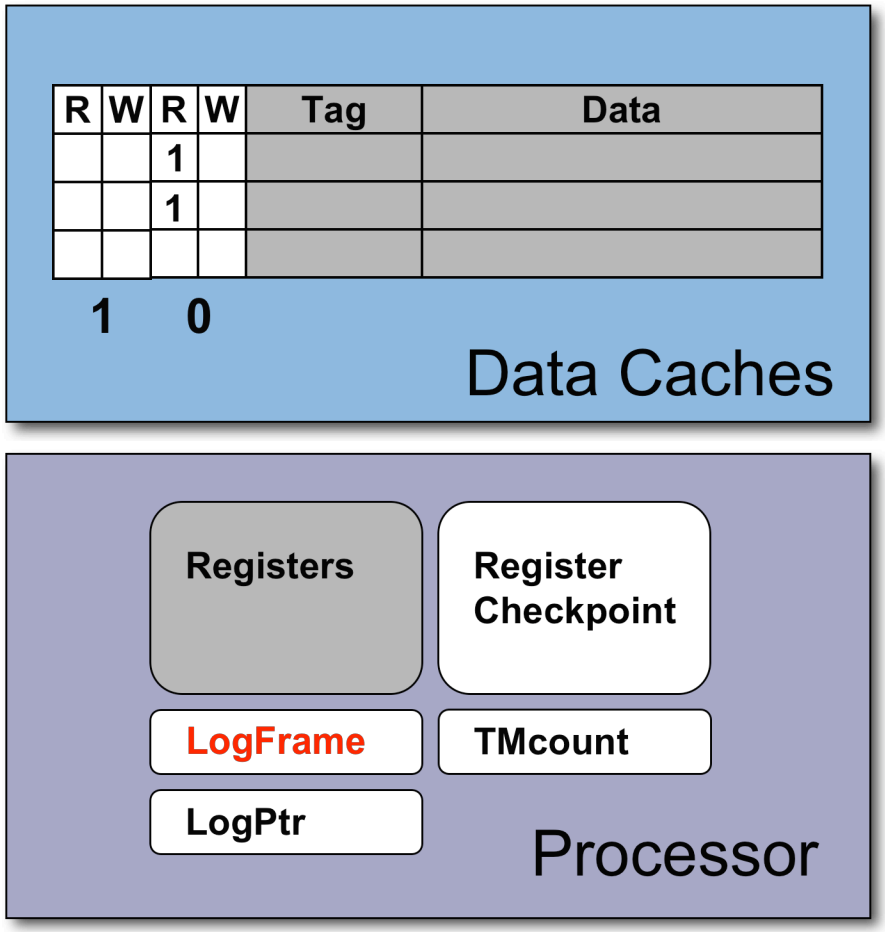
Conflict Detection in LogTM

- Conflict Detection
  - Nested LogTM replicates R/W bits for each level
  - Flash-Or circuit merges child and parent R/W bits
- Version Management
  - Nested LogTM segments the log into frames
  - (similar to a stack of activation records)



Conflict Detection in LogTM

- Conflict Detection
  - Nested LogTM replicates R/W bits for each level
  - Flash-Or circuit merges child and parent R/W bits
- Version Management
  - Nested LogTM segments the log into frames
  - (similar to a stack of activation records)



Motivation: Transactional Memory

- Chip-multiprocessors/Multi-core/Many-core are here

Motivation: Transactional Memory

- Chip-multiprocessors/Multi-core/Many-core are here
  - "Intel has 10 projects in the works that contain four or more computing cores per chip" -- Paul Otellini, Intel CEO, Fall '05

Motivation: Transactional Memory

- Chip-multiprocessors/Multi-core/Many-core are here
  - "Intel has 10 projects in the works that contain four or more computing cores per chip" -- Paul Otellini, Intel CEO, Fall '05

- We must effectively program these systems
  - But programming with locks is challenging
  - "Blocking on a mutex is a surprisingly delicate dance"

    -- OpenSolaris, mutex.c

Conclusions

- **Nested LogTM** supports:
  - Closed nesting facilitates software composition
  - Open nesting increases concurrency (but adds complexity)
  - Escape actions support non-transactional actions
- Nested LogTM Version Management
  - Segments the transaction log (stack of log frames)
  - Software abort handling allows easy execution of **commit actions** and **compensating actions**
- Nested LogTM Conflict Detection
  - Replicates R/W in caches
  - Flash-Or merges closed child with parent