

# *Efficient Transactions in Hardware and Software*

**C. Scott Ananian**

**Computer Science and Artificial Intelligence Laboratory  
Massachusetts Institute of Technology**

**April 11, 2007**

**thesis committee**

**Martin Rinard, Charles Leiserson, Bradley Kuszmaul**

# *Outline*

- **Why Transactions?**
- **An efficient software transactional memory.**
- **How close are we to the performance we want?**
- **Transactions in hardware: LTM & UTM.**
- **Combining software and hardware.**
- **Future directions.**

# *The Age of Parallel Computers*

- **Parallel computers are here.**
  - multicore, etc
- **We want to write software for them**
- **Standard approaches:**
  - multiple threads
  - shared address space
  - locks for coordination
- **Standard way to use locks**
  - associate locks with data
  - acquire lock before touching the data
  - release lock when we're done
- **Desired result: no undesirable interleavings**

# Locks are not our friends

```
void pushFlow(Vertex v1, Vertex v2, int flow) {
```

```
    lock_t lock1, lock2;
```

```
    if (v1.id < v2.id) { /* avoid deadlock */
```

```
        lock1 = v1.lock; lock2 = v2.lock;
```

```
    } else {
```

```
        lock1 = v2.lock; lock2 = v1.lock;
```

```
    }
```

```
    lock(lock1);
```

```
    lock(lock2);
```

```
    if (v2.excess > f) {
```

```
        /* move excess flow */
```

```
        v1.excess += f;
```

```
        v2.excess -= f;
```

```
    }
```

```
    unlock(lock2);
```

```
    unlock(lock1);
```

```
}
```



- Deadlocks/ordering
- Multi-object operations
- Priority inversion
- Faults in critical regions
- Inefficient

# Locks are not our friends

```
void pushFlow(Vertex v1, Vertex v2, double flow) {
```

```
    atomic {  
        if (v2.excess > f) {  
            /* move excess flow */  
            v1.excess += f;  
            v2.excess -= f;  
        }  
    }
```

- Use an atomic region
  - implement using a non-blocking transaction

# What am I really trying to accomplish?

- I want to perform **atomic operations**
  - as if there was no interleaving at all
- We propose to let people write it just like that!

```
void pushFlow(Vertex v1, Vertex v2, int flow) {  
    atomic {  
        if (v2.excess > f) {  
            /* move excess flow */  
            v1.excess += f;  
            v2.excess -= f;  
        }  
    }  
}
```

# *Transactional Memory* *(definition)*



- A transaction is a sequence of **memory loads and stores** that either **commits** or **aborts**
- If a transaction commits, all the loads and stores appear to have executed **atomically**
- If a transaction aborts, none of its stores take effect
- Transaction operations aren't visible until they commit or abort
- Simplified version of traditional ACID database transactions (no durability, for example)
- For this talk, we assume no I/O within transactions

# *Non-blocking synchronization*

- Although transactions can be implemented with mutual exclusion (locks), we are interested only in **non-blocking** implementations.
- In a non-blocking implementation, the failure of one process cannot prevent other processes from making progress. This leads to:
  - **Scalable parallelism**
  - **Fault-tolerance**
  - **Safety**: freedom from some problems which require careful bookkeeping with locks, including priority inversion and deadlocks
- Little known requirement: limits on trans. suicide

# *Transactions: Philosophy*

- Transactions will be large & small, short & long
  - Mechanisms should be *unbounded*
- They will be **frequent** and **visible** in user code
  - Easy to use
  - Not hidden in libraries
- Implemented with **general-purpose** mechanisms
  - In addition to synchronization, useful for fault tolerance, exception handling, backtracking, priority scheduling...
- **Object-based** transactions
  - Expose a richer abstraction
  - Move beyond emulating an unavailable HTM

# *Making things practical: Things to keep in mind*

- There is both transactional and non-transaction code in real systems
  - A robust mechanism won't allow violations of transactional atomicity (*strong atomicity*)
- Non-transactional code should be fast!
- Transaction duration may reach 100M memory operations
- Transactional reads out-number transactional writes 3 to 1

# *APEX: Efficient Transactions in Software*

- **Design space for this implementation:**
  - **Pure software system**
    - but requires **load-linked** and **store-conditional** operations on the processor.
  - **Strongly atomic**
    - but at **low cost for non-transactional code**
  - **Object-based**

# *Why object-based transactions?*

- **Synchronization abstraction matches programming abstraction**
  - **No false sharing** due to variables incidentally colocated in same word/cache line/page.  
Possible deadlock!
- **Matching the programming abstraction allows better compiler analysis and optimization of transactional code**
  - For example, **escape analysis**
- **Potential performance benefits for long-running transactions**
  - Pay cloning costs up-front, then **run at full-speed** in own copy of the object graph

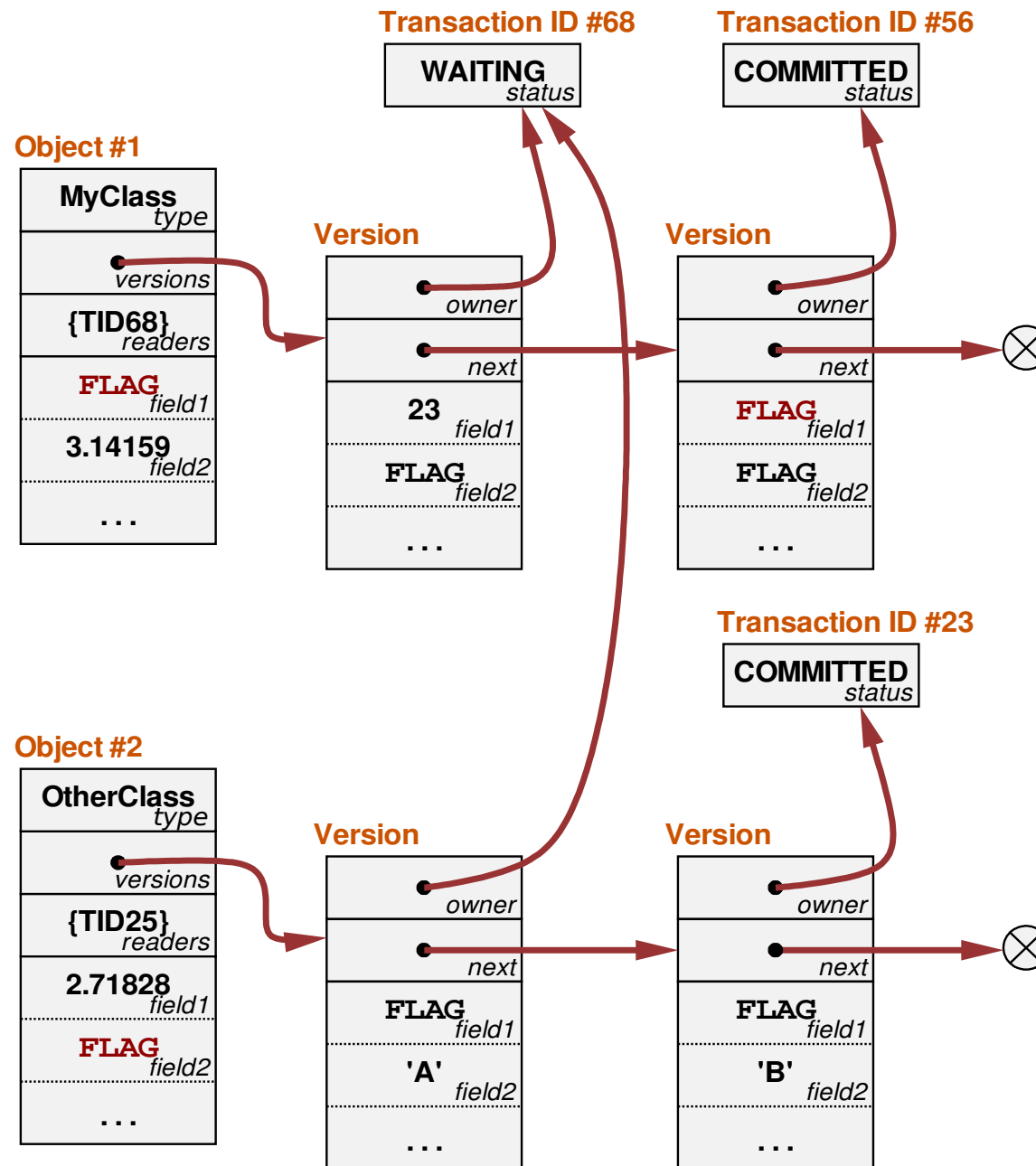
# *APEX Software Transactions*

- **Goals:**
  - Non-transactional operations should be fast
  - Reads should be faster than writes
  - Minimal amount of object bloat
- **Solution:**
  - Use special `FLAG` value to indicate “location involved in a transaction”
  - Object points to a linked list of **versions**, containing values written by (in-progress, committed, or aborted) transactions
  - Semantic value of `FLAGged` field is: “value of the first version owned by a committed transaction on the version list”
  - Values which are “really” `FLAG` are handled with an escape mechanism (**we call these “false flags”**)

# *How do we maintain atomicity?*

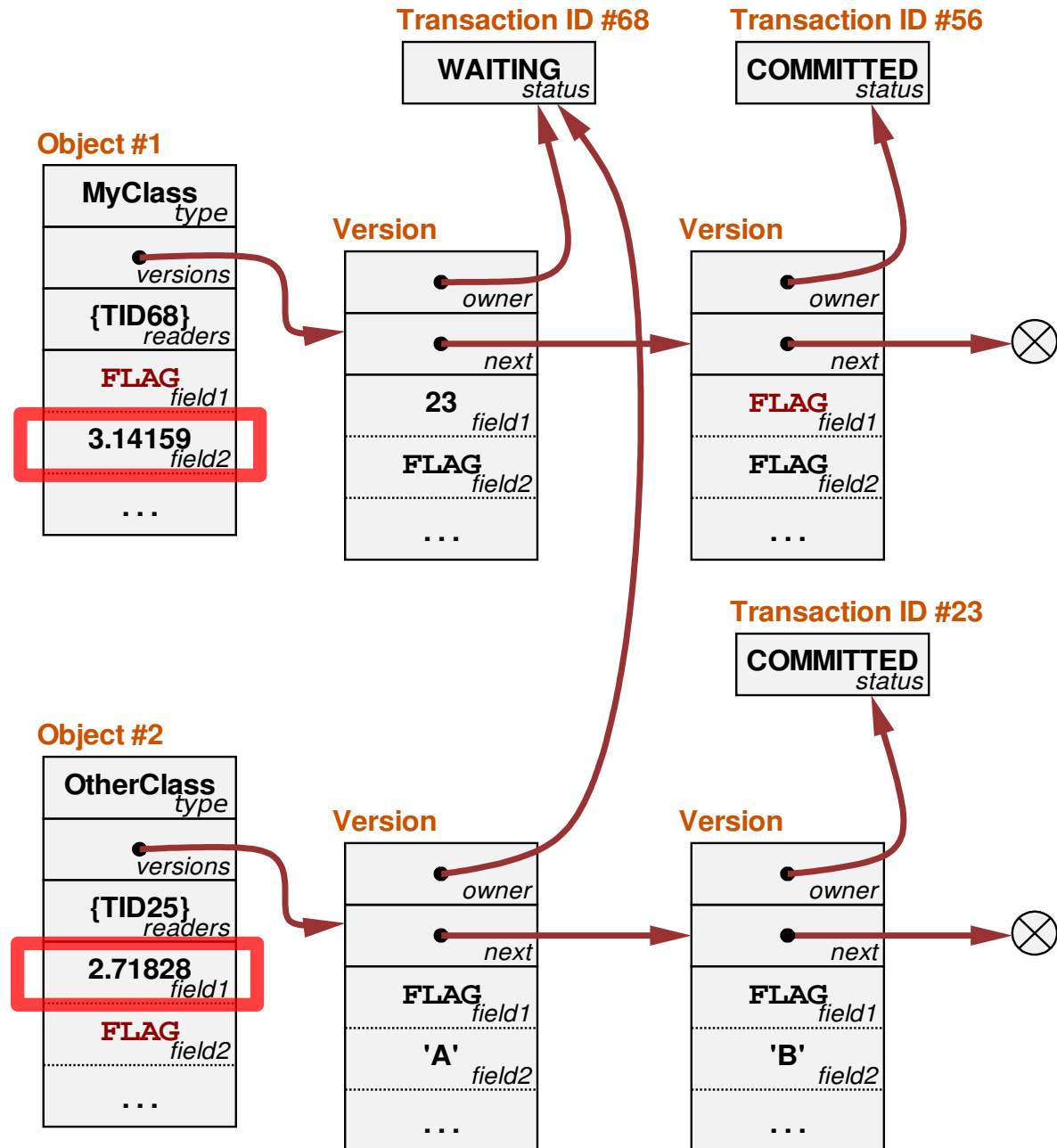
- **Allow multiple readers, but a single writer**
- **If you write a field, you must ensure that all prior readers and writers are committed or aborted.**
- **If you read a field, you must ensure that all prior writers are committed or aborted.**

# Transactions using version lists



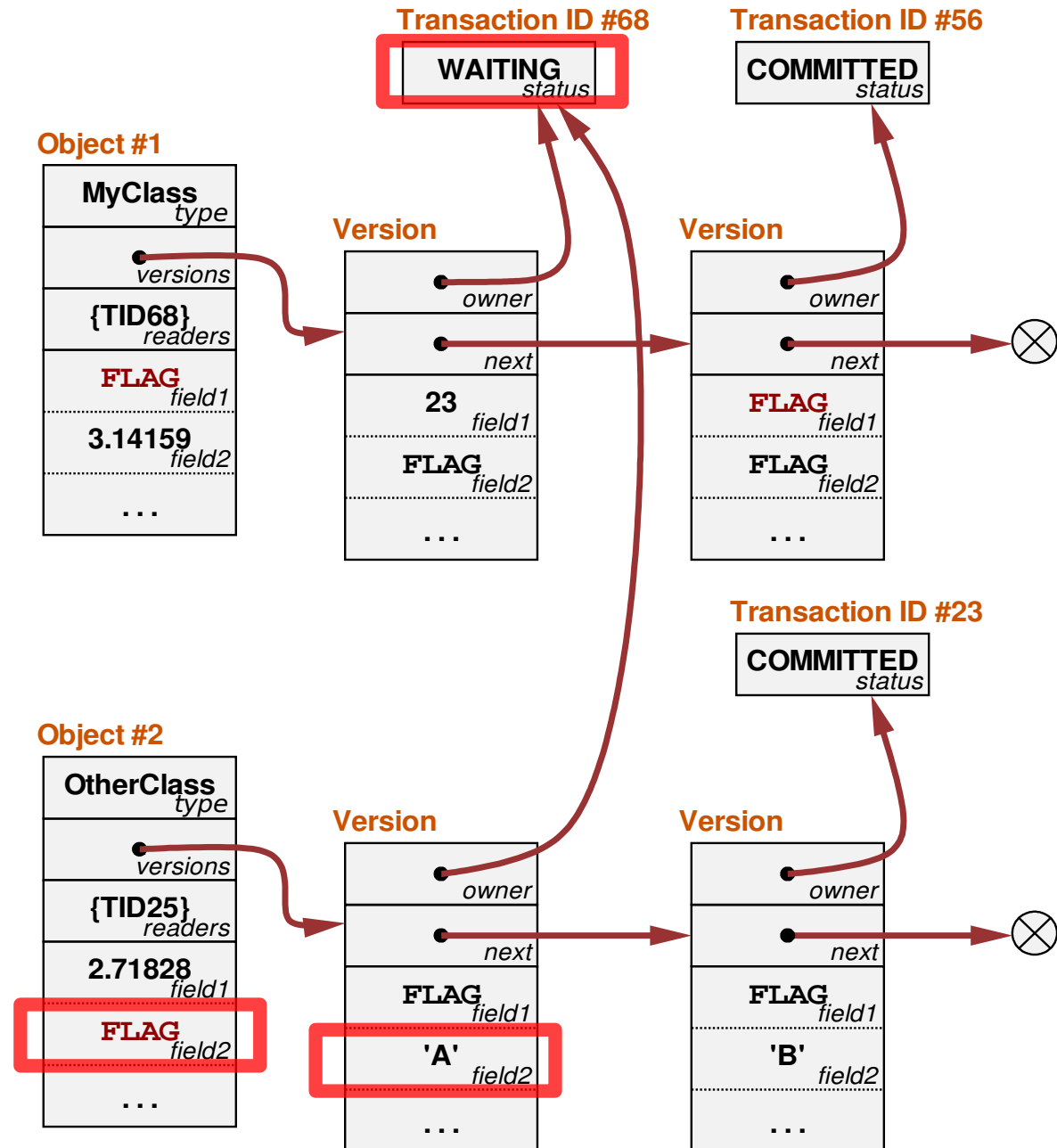
# Non-transactional Read (ReadNT)

- Begins with a normal read of the field.
- If value is not FLAG, we're done!



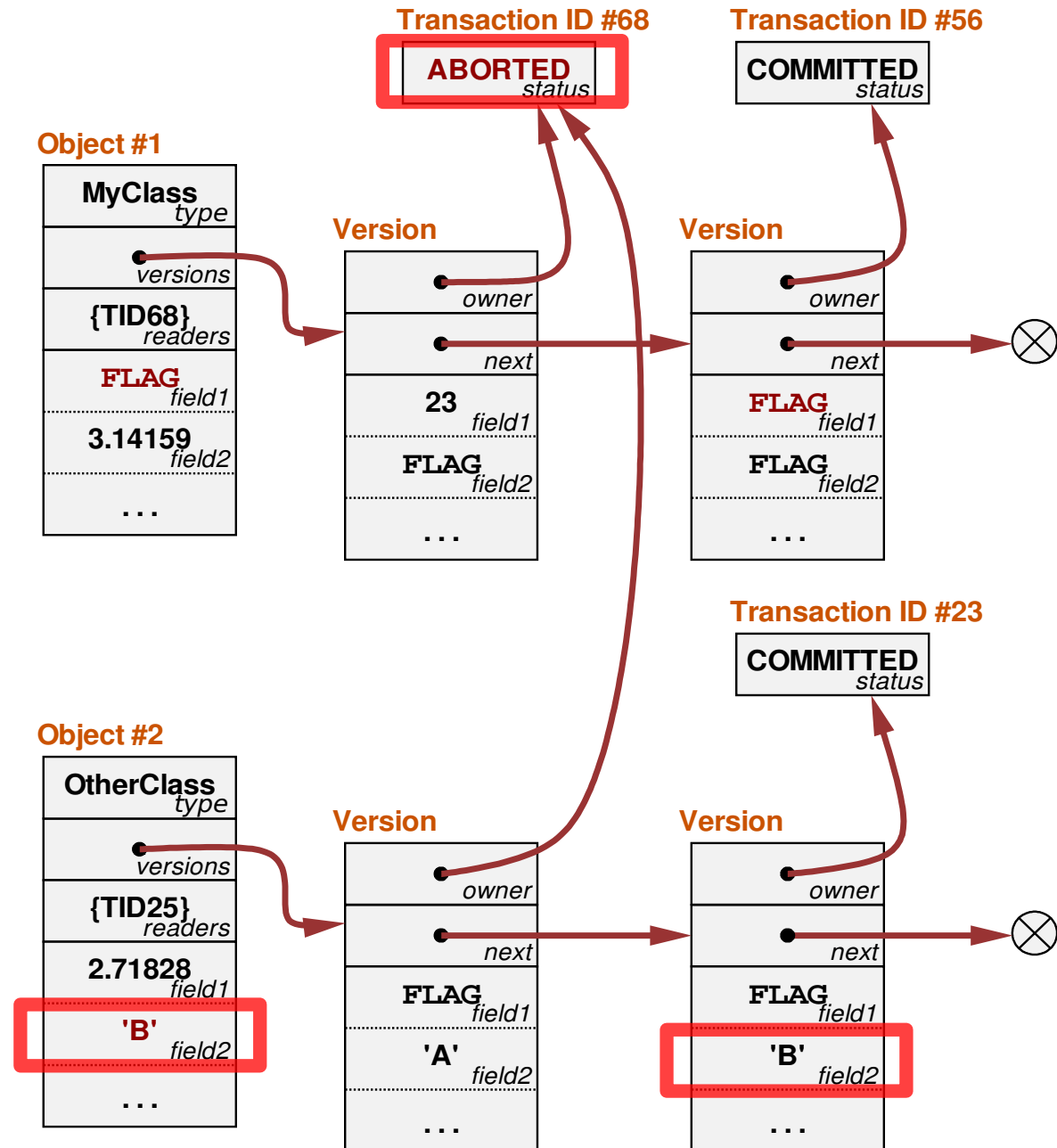
# Non-transactional Read (ReadNT)

- Begins with a normal read of the field...
- Otherwise:
  - kill writers



# Non-transactional Read (ReadNT)

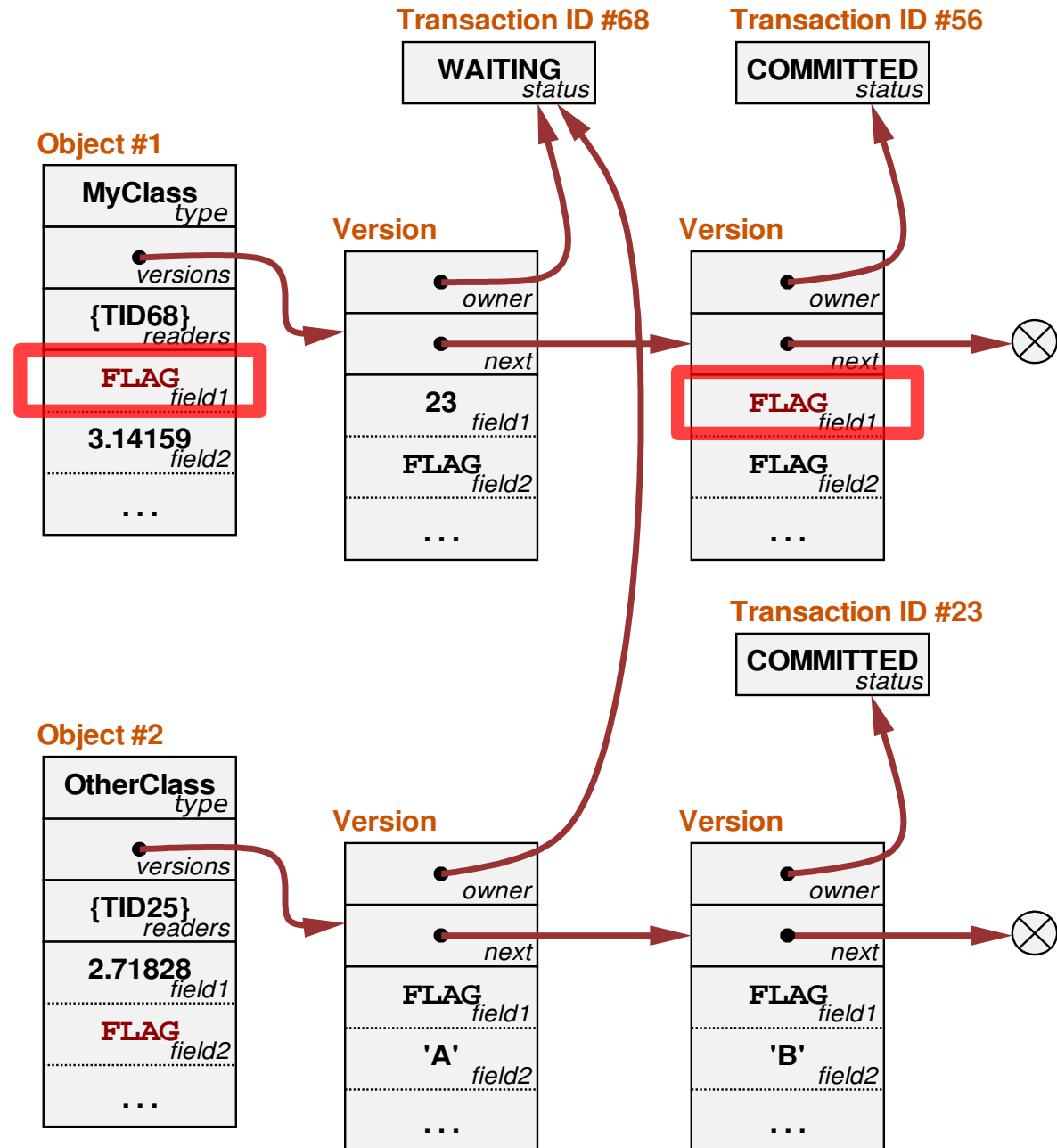
- Begins with a normal read of the field...
- Otherwise:
  - kill writers
  - copy back field





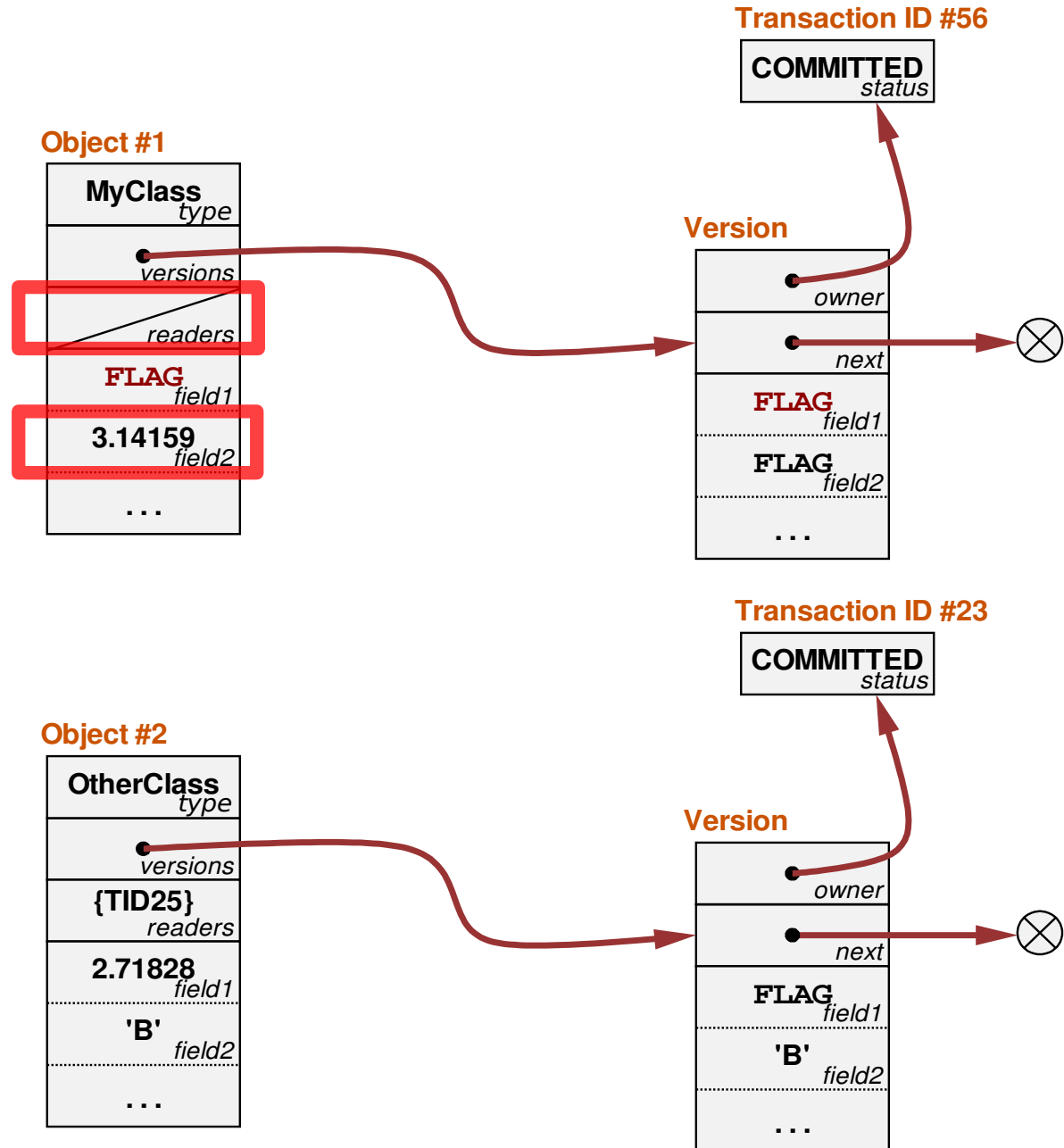
# Non-transactional Read (ReadNT)

- Begins with a normal read of the field...
- “False flags” are discovered during copy-back; the read value is FLAG in this case.



# Non-transactional Write (WriteNT)

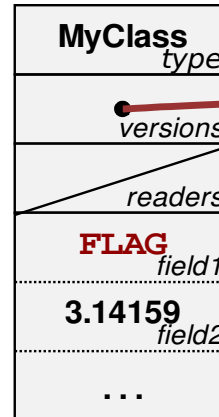
- If value-to-write is not FLAG:
  - LL(readers)
  - check that it's empty
  - SC(field)



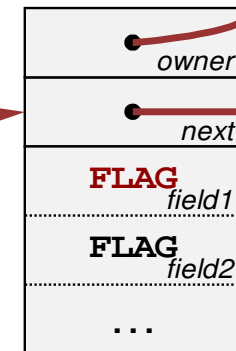
# Non-transactional Write (WriteNT)

- If value-to-write is not FLAG:
  - LL(readers)
  - check that it's empty
  - SC(field)

Object #1



Version



Transaction ID #56

<b>COMMITTED</b> <i>status</i>
-----------------------------------

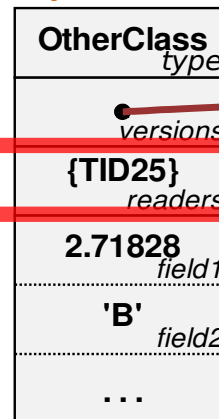
- LL(readers)

- check that it's empty

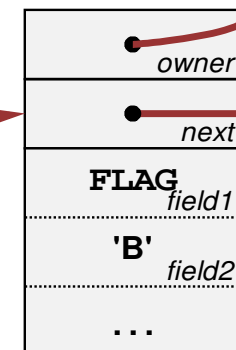
- SC(field)

- If unsuccessful
  - kill readers and writers
  - repeat

Object #2



Version

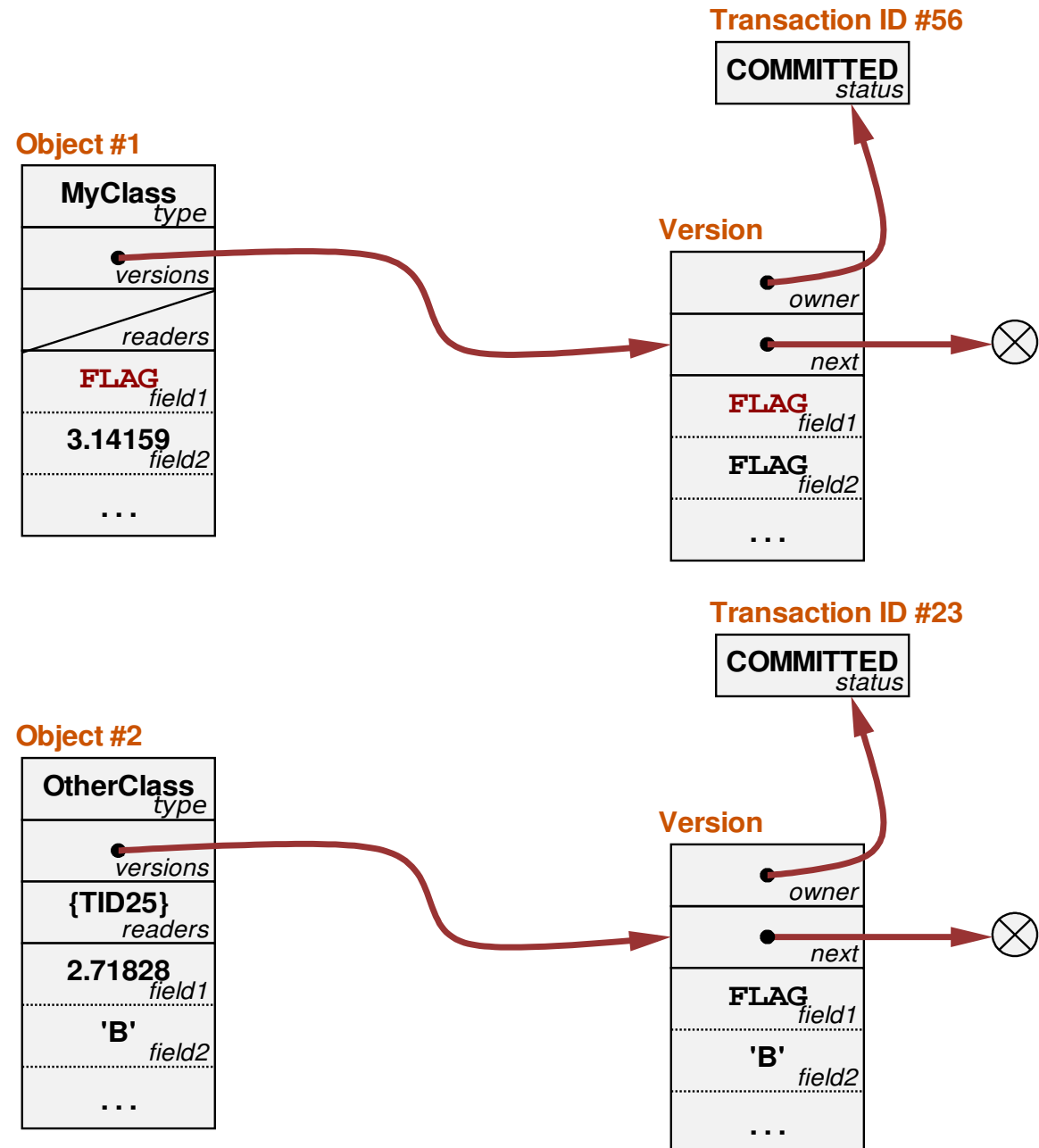


Transaction ID #23

<b>COMMITTED</b> <i>status</i>
-----------------------------------

# Non-transactional Write (WriteNT)

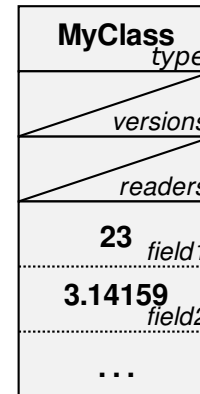
- If value-to-write **is** FLAG...
  - make this a short transactional write (WriteT)



# Transactional Write (*WriteT*)

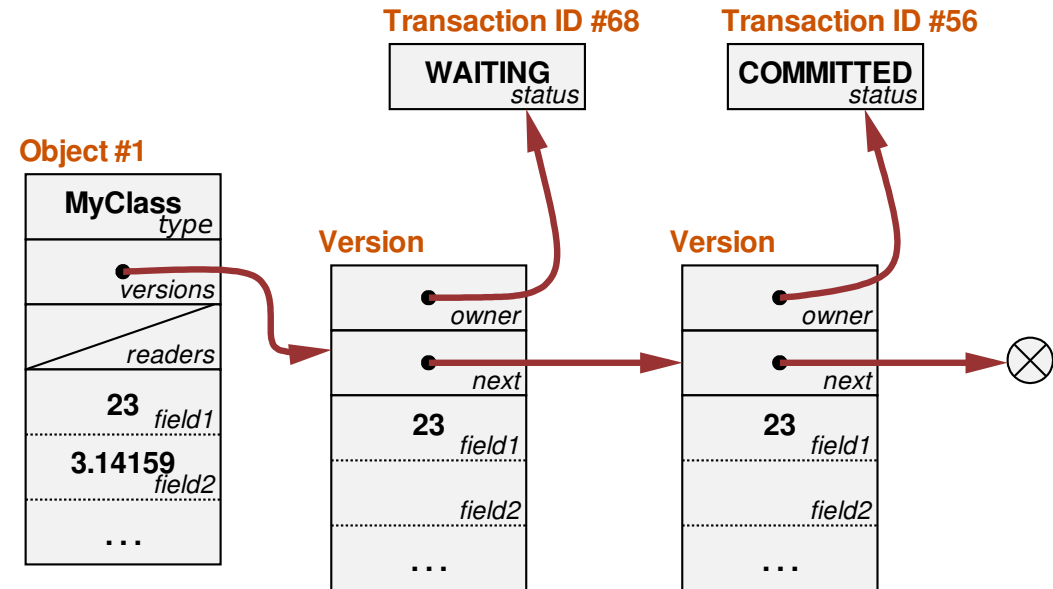
- **Once** per object written in this transaction:
  - find writable version...

Object #1



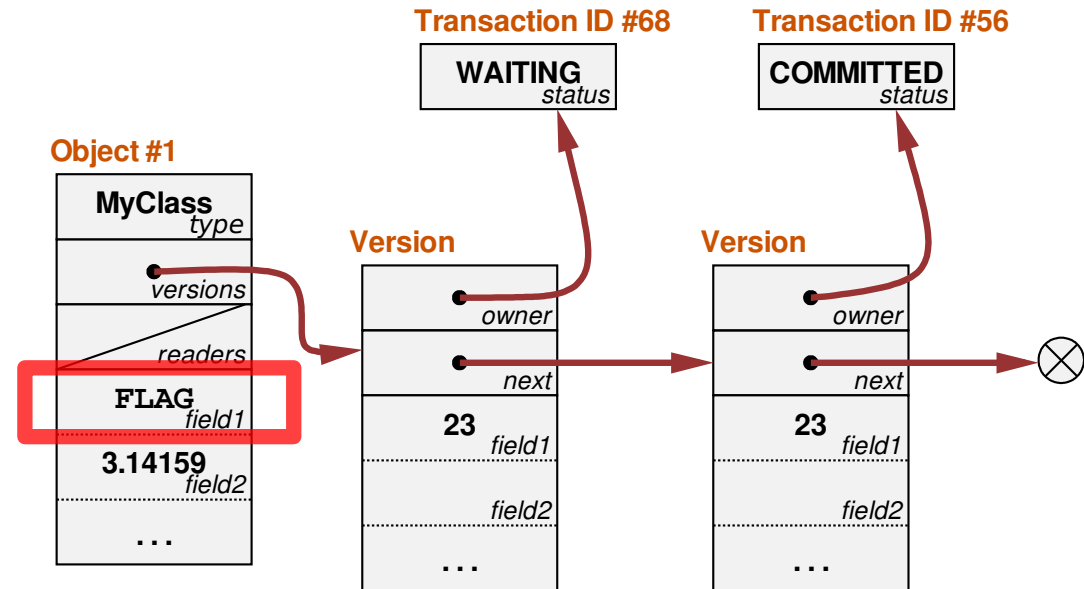
# Transactional Write (WriteT)

- **Once** per object written in this transaction:
  - find writable version
  - create (by cloning) if necessary



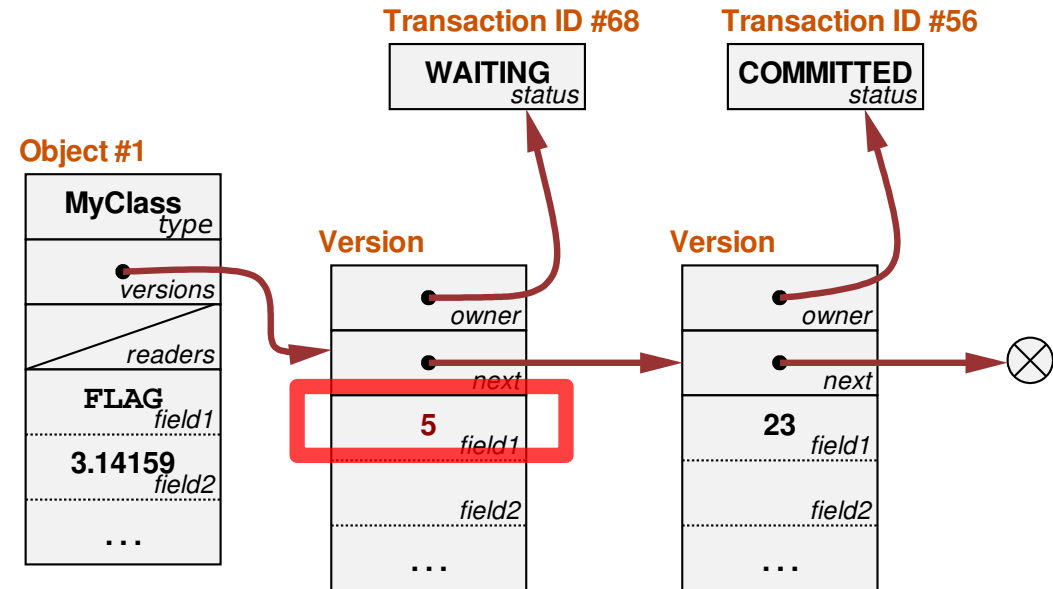
# Transactional Write (WriteT)

- **Once** per object written in this transaction:
  - find writable version
  - create (by cloning) if necessary
- **Once** per field written:
  - ensure field is FLAG



# Transactional Write (WriteT)

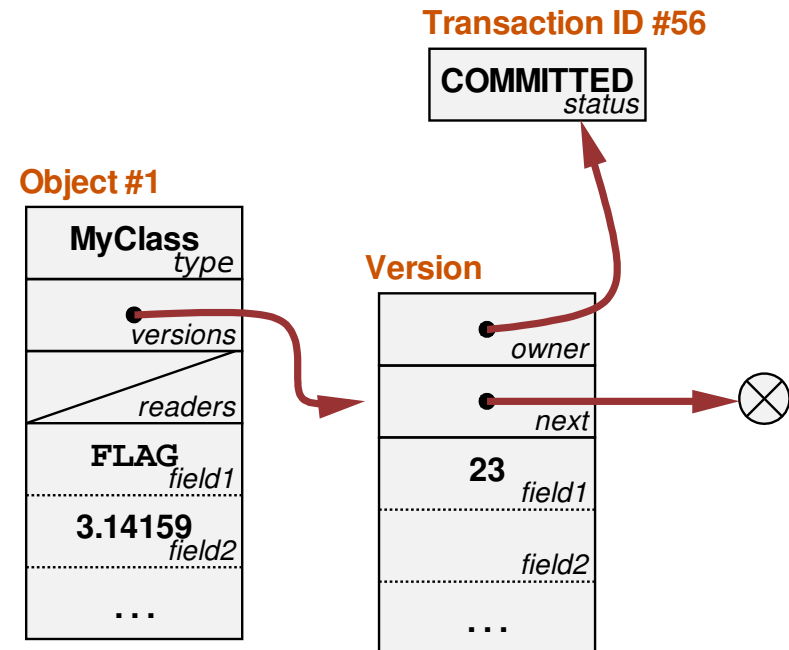
- **Once** per object written in this transaction:
  - find writable version
  - create (by cloning) if necessary
- **Once** per field written:
  - ensure field is FLAG
- Then, **just write to the version.**



Opportunity for  
program analysis and  
transformation!

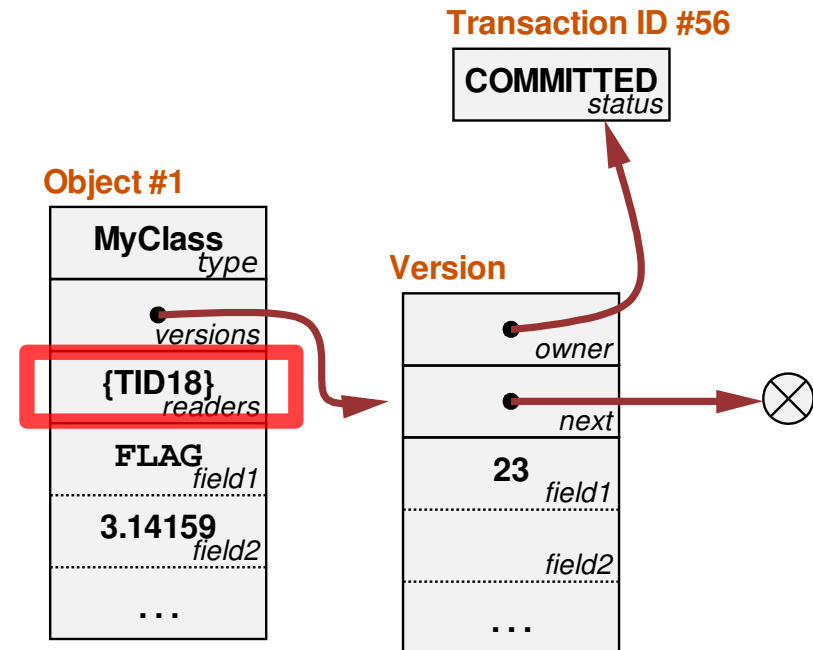
# Transactional Read (ReadT)

- **Once** per object read in this transaction:
  - ensure we're on list of readers
  - kill any writers



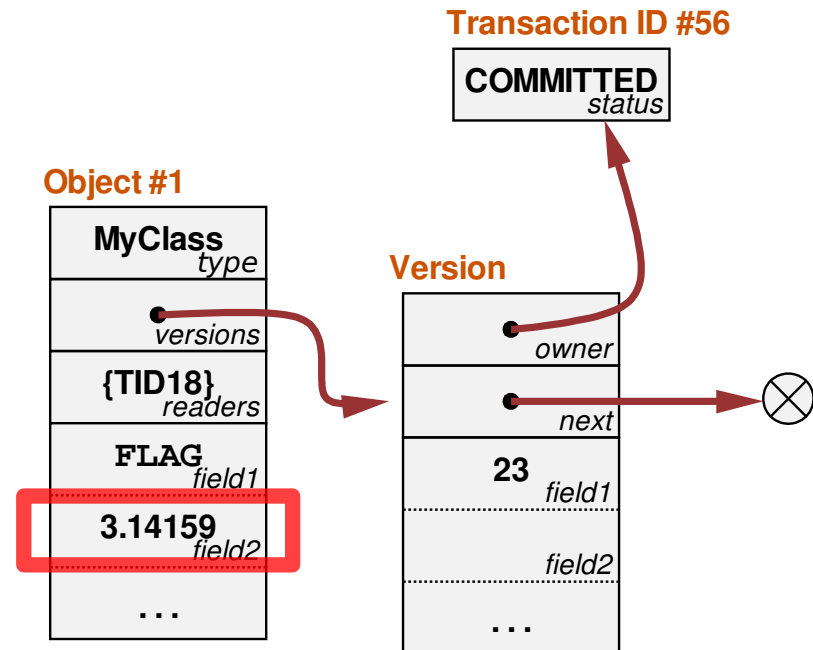
# Transactional Read (ReadT)

- **Once** per object read in this transaction:
  - ensure we're on list of readers
  - kill any writers



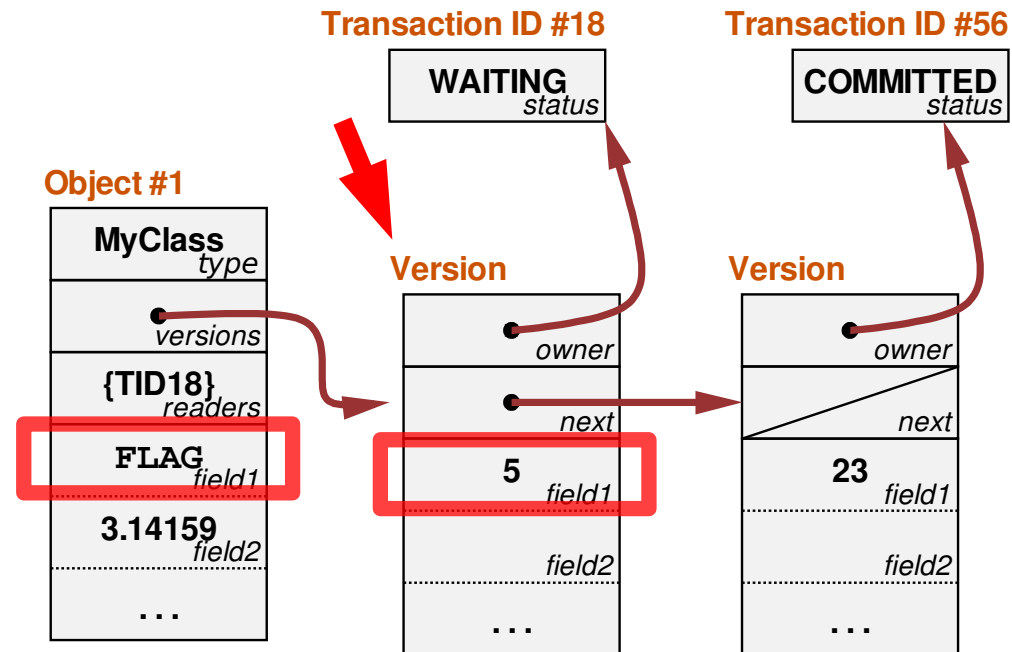
# Transactional Read (ReadT)

- **Once** per object read in this transaction:
  - ensure we're on list of readers
  - kill any writers
- Read field of object
- If this is not FLAG, you're done!



# Transactional Read (ReadT)

- **Once** per object read in this transaction:
  - ensure we're on list of readers
  - kill any writers
- Read field of object
- If this is `FLAG`, then read field from version
  - remember version for next time!

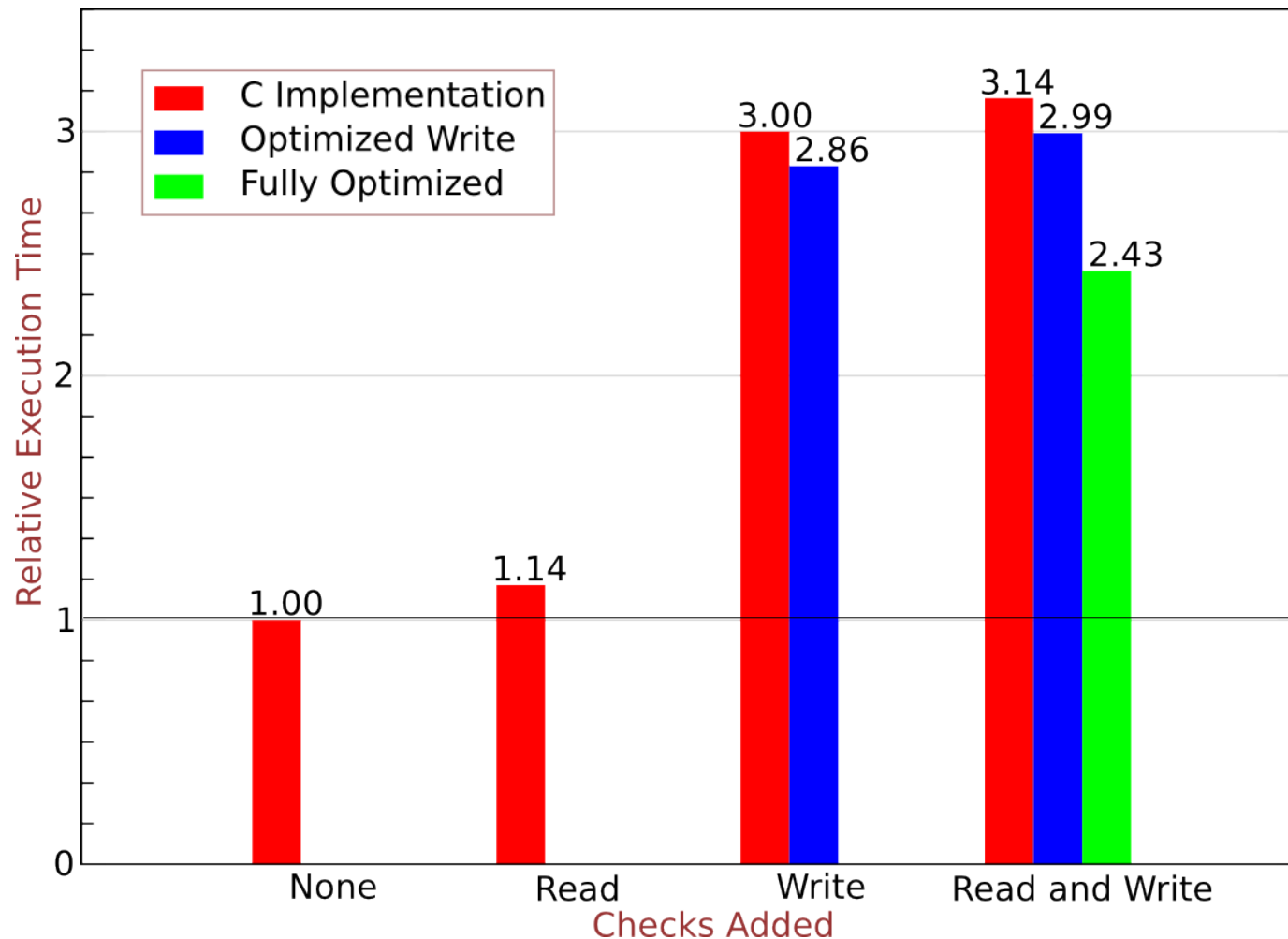


# Performance

- **Non-transactional code only needs to check whether a memory operand is FLAG before continuing.**
  - On superscalar processors, there are plenty of extra functional units to do the check
  - The branch is extremely predictable
- **Once FLAGged, transactional code operates mostly on the object's “version”**
  - if we know it's been written once
  - and we keep forgetting
- **Creating versions can be an issue for large arrays; “functional arrays” are one approach**

# Read/Write Check Overheads

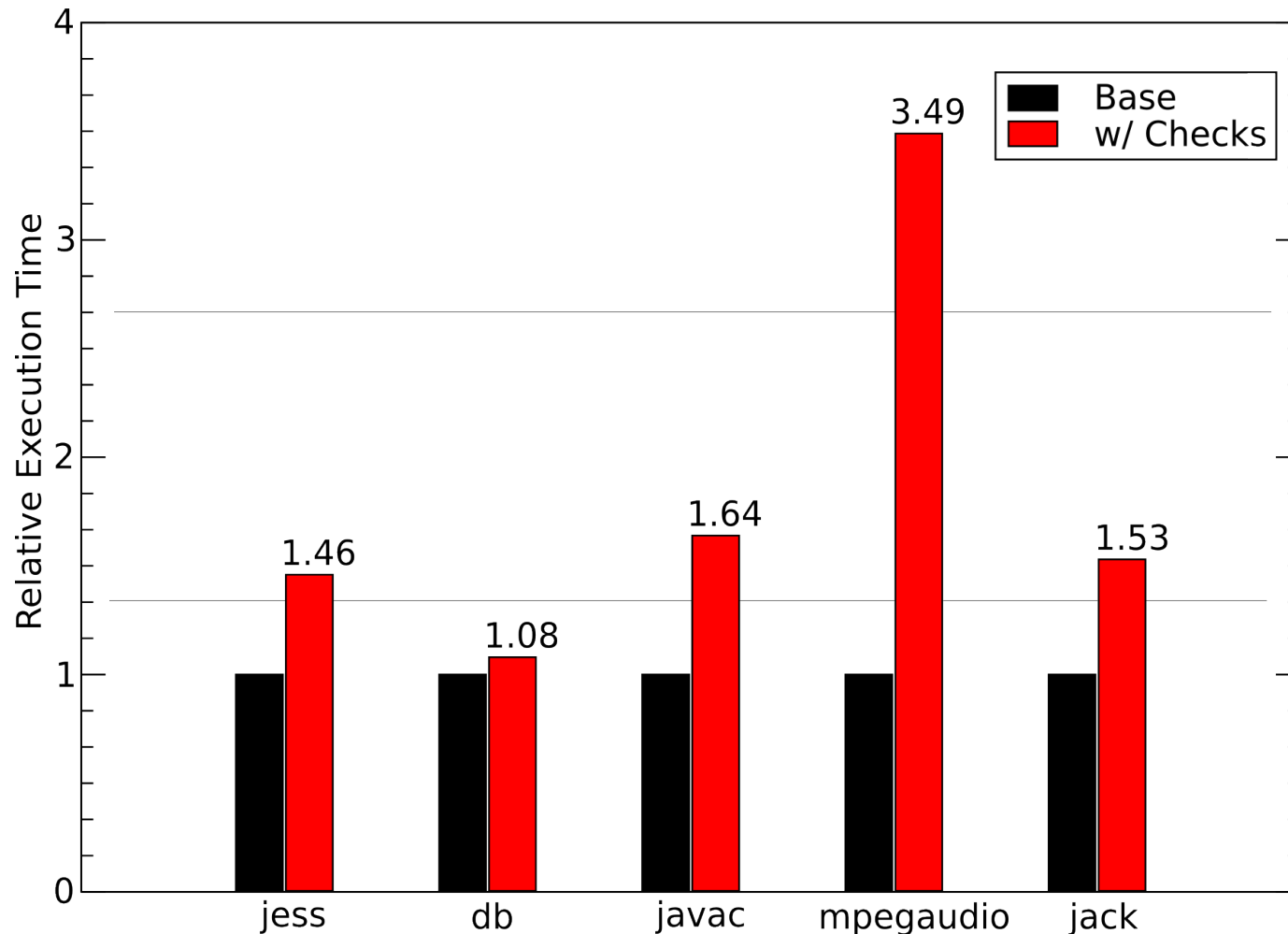
## Counter Microbenchmark



- Hand-tuned test code shows that the read check is fast, but writes can be slow

# Non-transactional Check Overhead

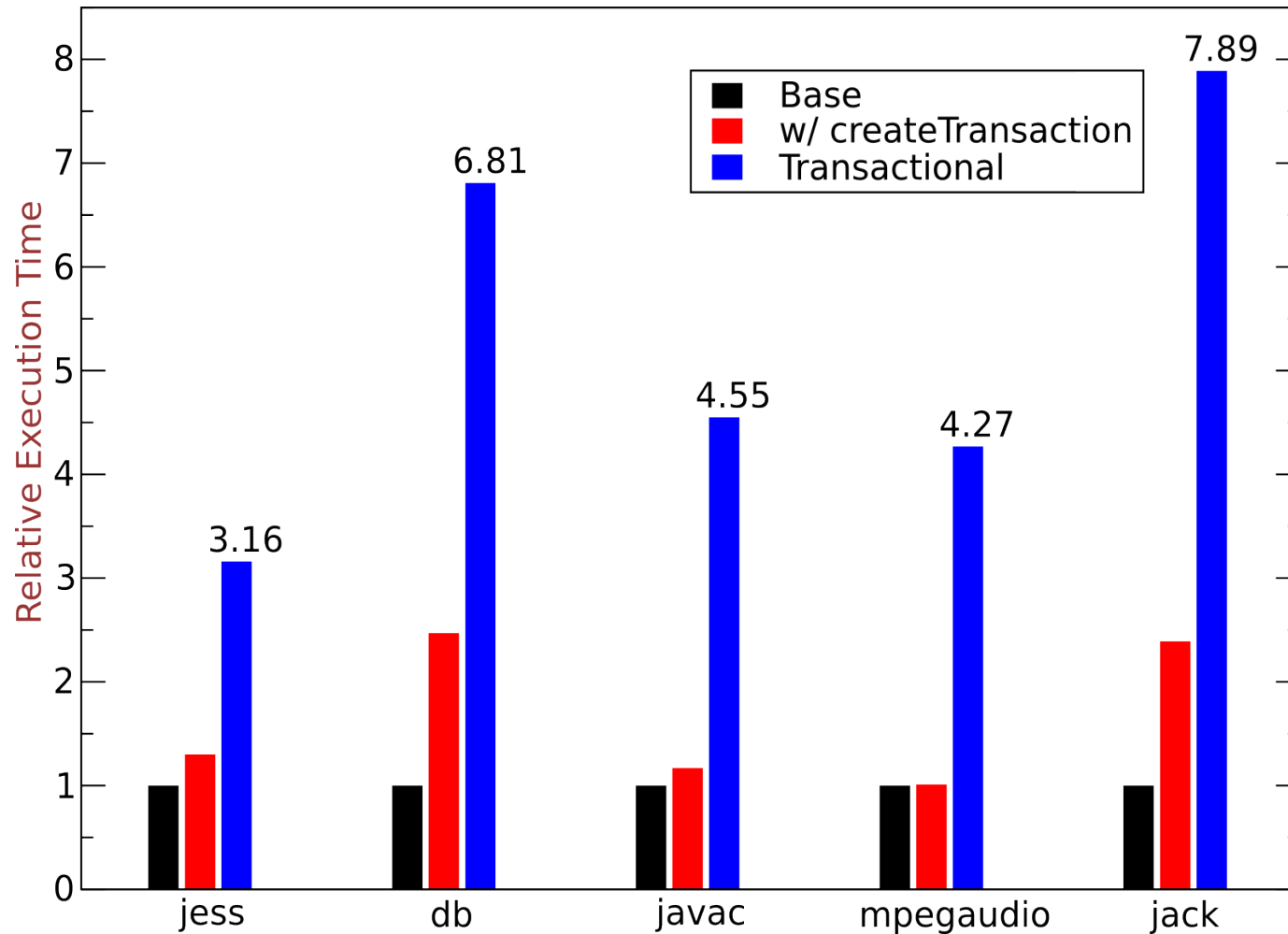
## SPECjvm98



- Strip all synchronization; just perform readNT/writeNT protocols
  - mpegaudio is an outlier

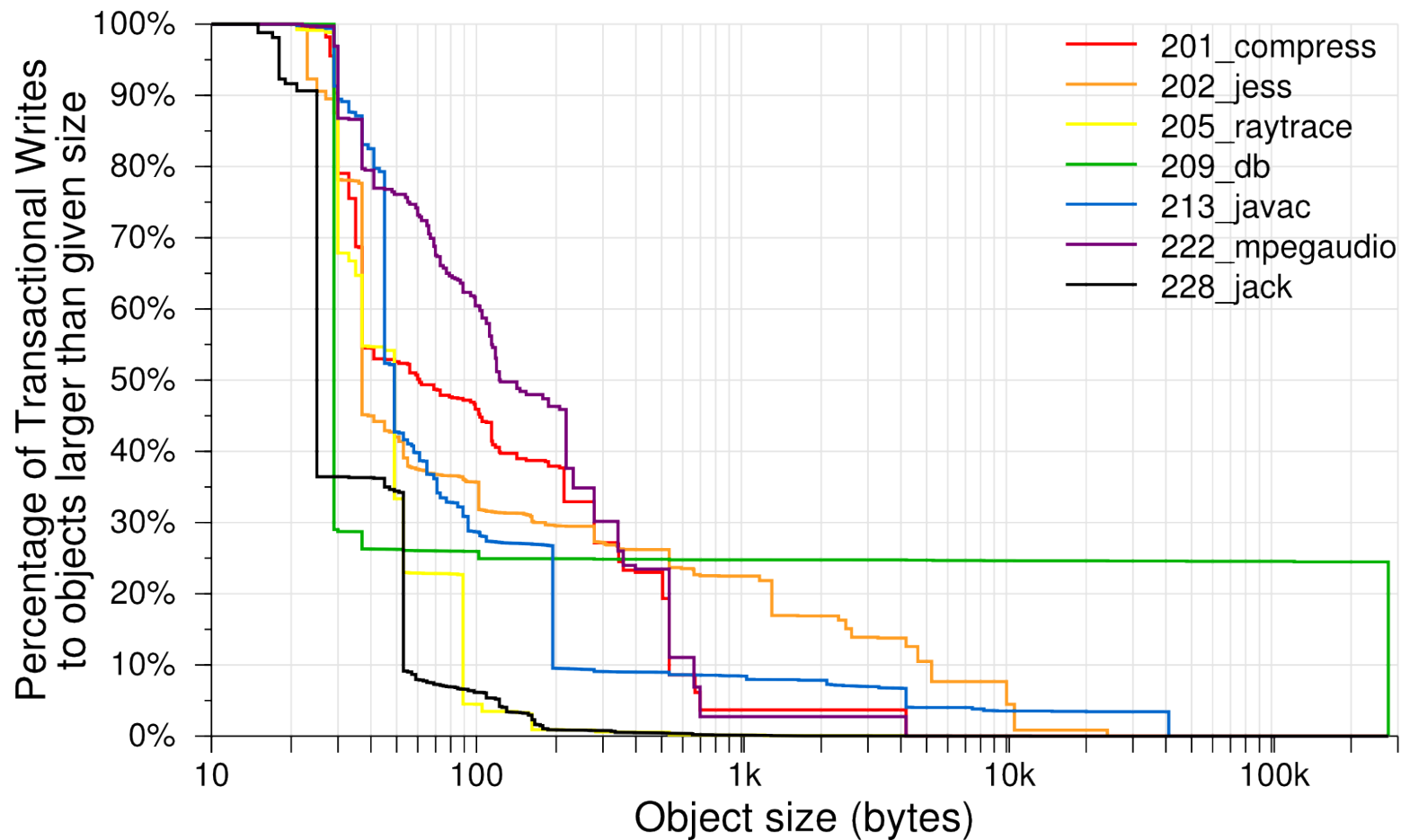
# Transaction Overhead

## Transactified SPECjvm98 benchmarks



# Transactional-write distribution

## SPECjvm98 benchmarks

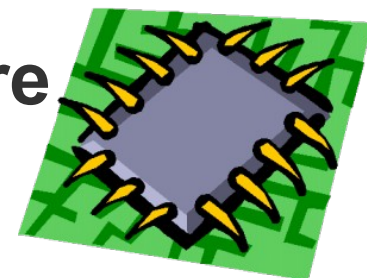


# *Can we do better?*

- **What if you want better performance?**
  - **recode parts of your application**
  - **fast allocation of transaction objects**
  - **chunk large objects**
  - **aggressively (and interprocedurally) hoist transaction checks**
- **Or we can use hardware support...**

# *LTM: Visible, Large, Frequent, Scalable*

- **“Large Transactional Memory”**
  - large bounded xactions, but simple and cheap
- **Minimal architectural changes, high performance**
  - Small mods to cache and processor core
  - No changes to main memory, cache coherence protocols or messages
  - **Can be pin-compatible with conventional proc**
- **Design presented here based on SGI Origin 3000 shared-memory multi-proc**
  - distributed memory
  - directory-based write-invalidate coherency protocol

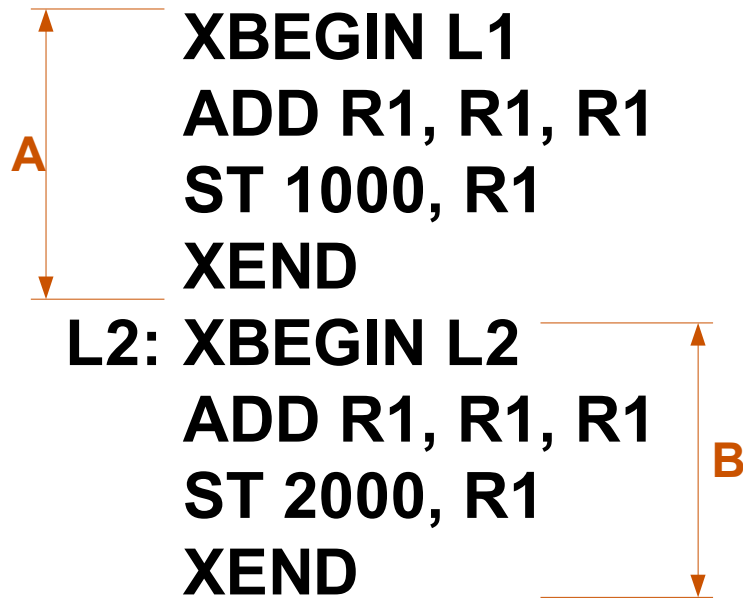


# Two new instructions



- **XBEGIN pc**
  - Begin a new transaction. Entry point to an *abort handler* specified by `pc`.
  - If transaction must fail, roll back processor and memory state to what it was when XBEGIN was executed, and jump to `pc`.
    - Think of this as a mispredicted branch.
- **XEND**
  - End the current transaction. If XEND completes, the xaction is committed and appeared atomic.

# Transaction Semantics



- **Two transactions**
  - “A” has an abort handler at L1
  - “B” has an abort handler at L2
    - Here, very simplistic retry. Other choices!
- **Always need “current” and “rollback” values for both registers and memory**

# Handling conflicts

## Processor 1

XBEGIN L1

ADD R1, R1, R1

→ ST 1000, R1

XEND

L2: XBEGIN L2

ADD R1, R1, R1

ST 2000, R1

XEND

## Processor 2

ST 1000, 65 ←

- We need to track locations read/written by transactional and non-transactional code
- When we find a conflict, transaction(s) must be aborted
  - We always “kill the other guy”
  - This leads to non-blocking systems

# Restoring register state



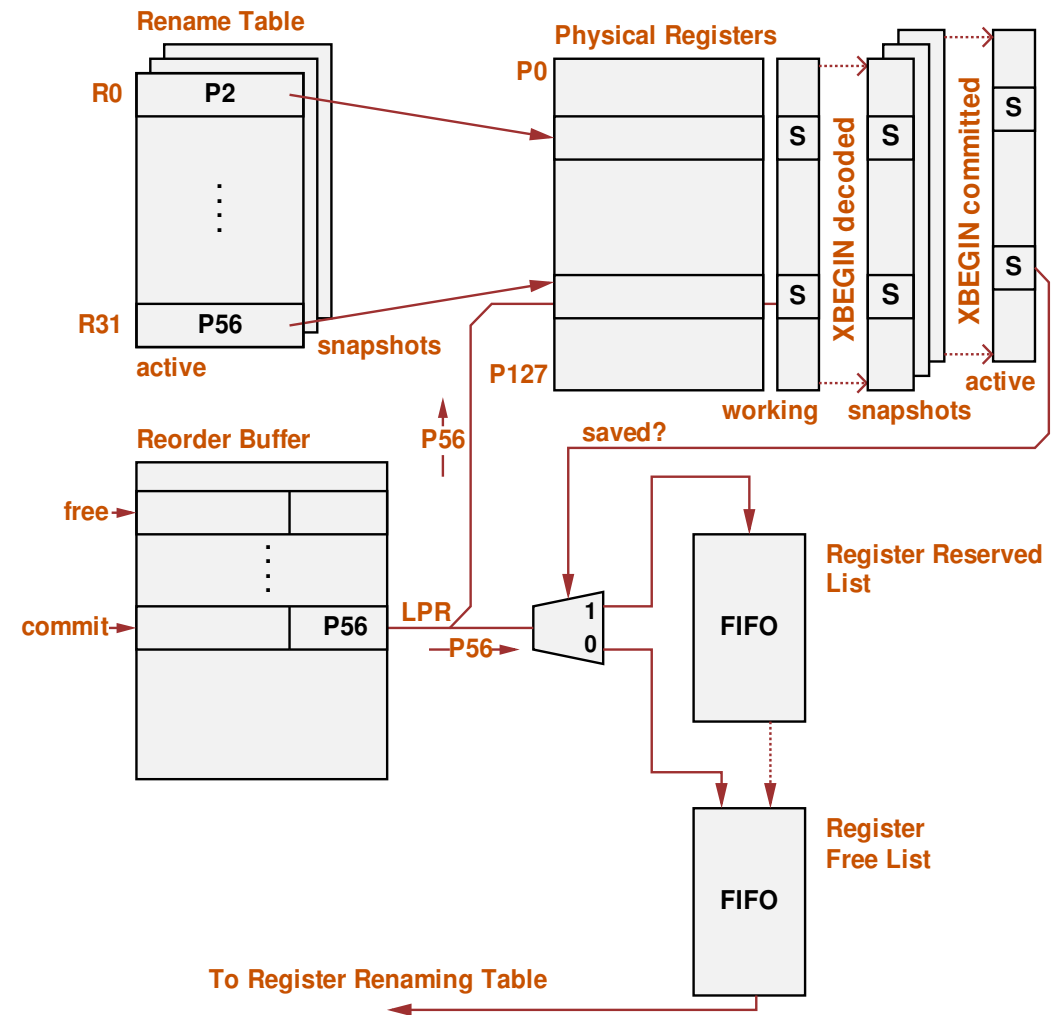
- Minimally invasive changes; build on existing rename mechanism
- Both “current” and “rollback” architectural register values **stored in physical registers**
- In conventional speculation, “rollback” values stored until the speculative instruction graduates (order **100 instrs**)
- Here, we keep these until the transaction commits or aborts (**unbounded # of instrs**)
- But we only need one copy!
  - only one transaction in the memory system per processor

## *LTM implementation, cont.*

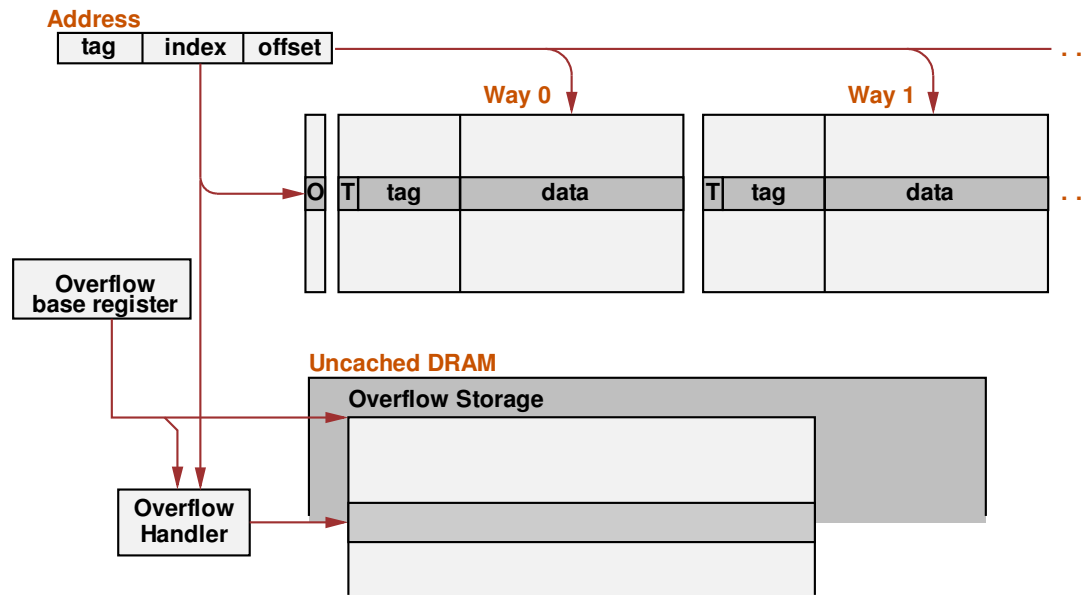
- Info about pending transactions stored in the cache
  - No special fully-associative cache needed
  - Main memory contains “committed” data
- Conflicts among pending transactions detected using existing cache-coherency mechanisms
  - Request from another proc for cache line with transactional data indicates conflict
- Overflow mechanism **allows large transactions to spill from the cache into main memory**

# LTM pipeline modifications

- Register snapshot stored with rename mechanism
- Limited # of regs reserved even if multiple xactions are in-flight
- Architectural changes are kept small



# LTM cache modifications



- **O bit per cache set**
  - indicates if set has overflowed
- **T bit per cache line**
  - set if accessed during current transaction
- **Overflow storage in uncached DRAM**
  - maintained by hardware
  - OS sets size/location via OBR, etc

# *Cycle-level LTM simulation*

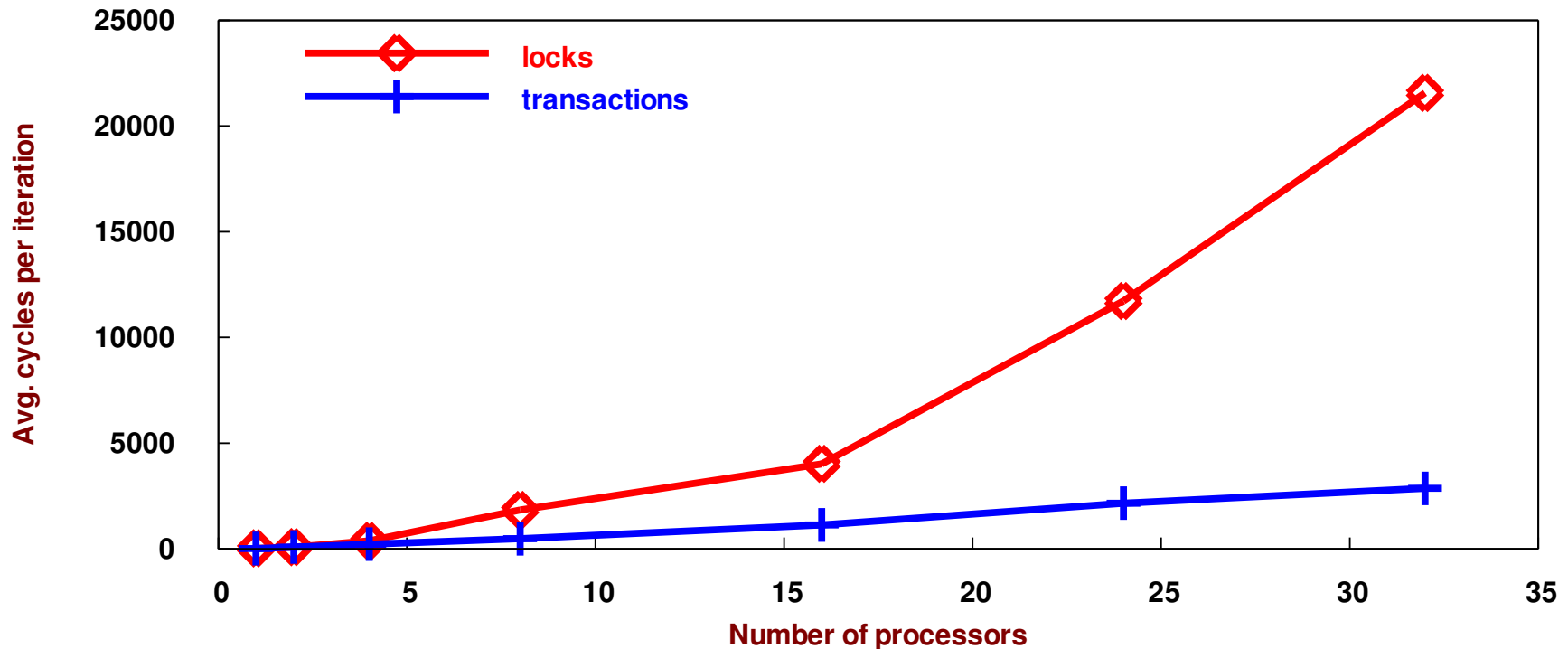
- **LTM implemented on top of UVSIM (itself built on RSIM)**
  - shared-memory multiprocessor model
  - directory-based write-invalidate coherence
- **Contention behavior:**
  - C microbenchmarks w/ inline assembly
  - Up to 32 processors
- **Overhead measurements:**
  - Modified MIT FLEX Java compiler
  - Compared no-sync, spin-lock, and LTM xaction
  - Single-threaded, single processor

*RSIM*

**Flex**  
Compiler Infrastructure



# Contention behavior

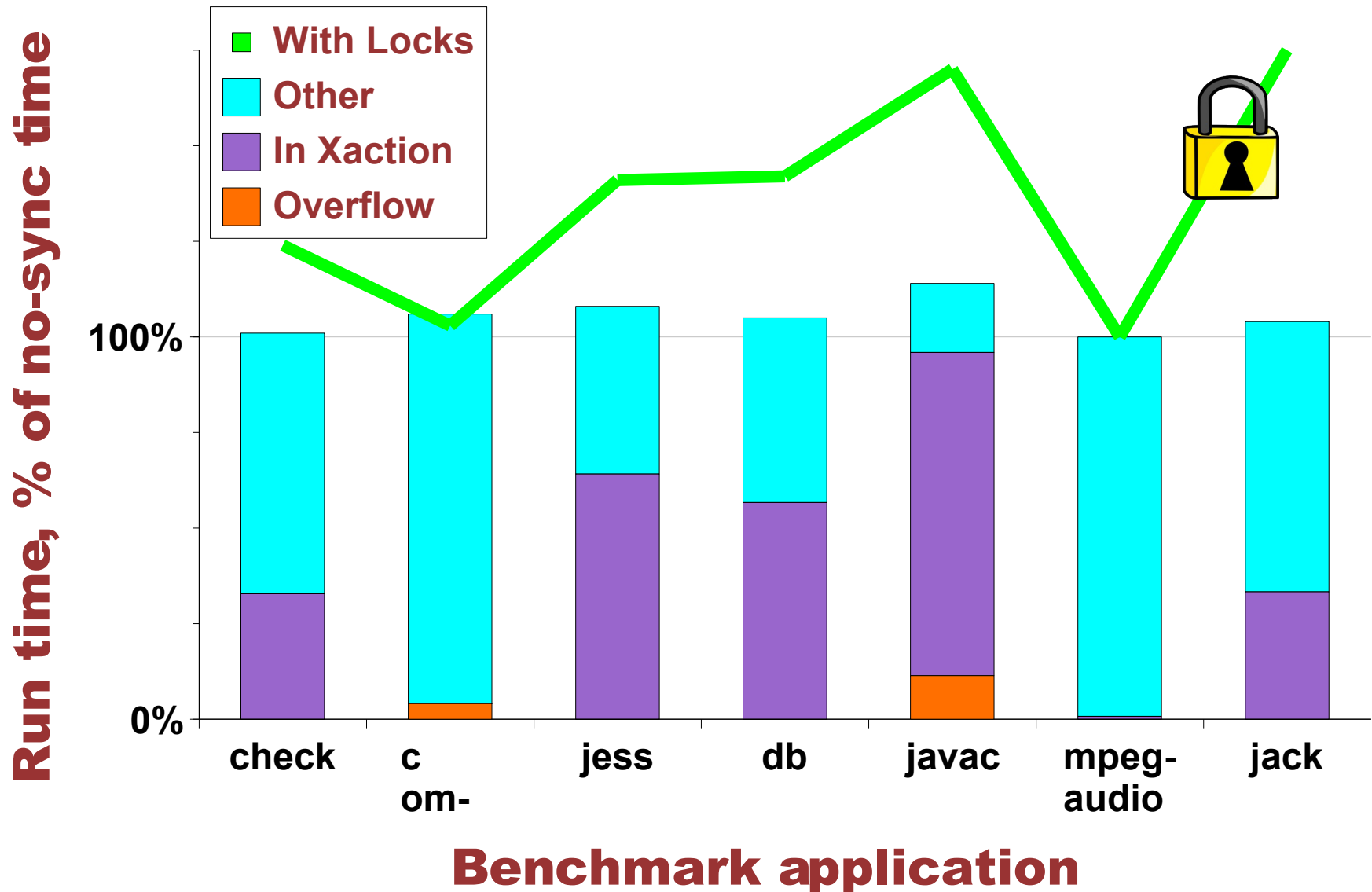


- **Contention microbenchmark: 'Counter'**
  - 1 shared variable; each processor repeatedly adds
  - locking version uses global LLSC spinlock
  - **Small xactions commit even with high contention**
  - Spin-lock causes lots of cache interventions even when it can't be taken (standard SGI library impl)

# *SPECjvm98 LTM benchmarks*

- **Compiled three versions of each benchmark using modified FLEX compiler**
  - **Base** with no synchronization
  - **Locks** with spinlocks
  - **Trans** with LTM xactions for synchronization
- **Run on one processor of UVSIM**
  - Looking at overhead, not contention

# LTM Overhead: SPECjvm98



# *Is this good enough?*

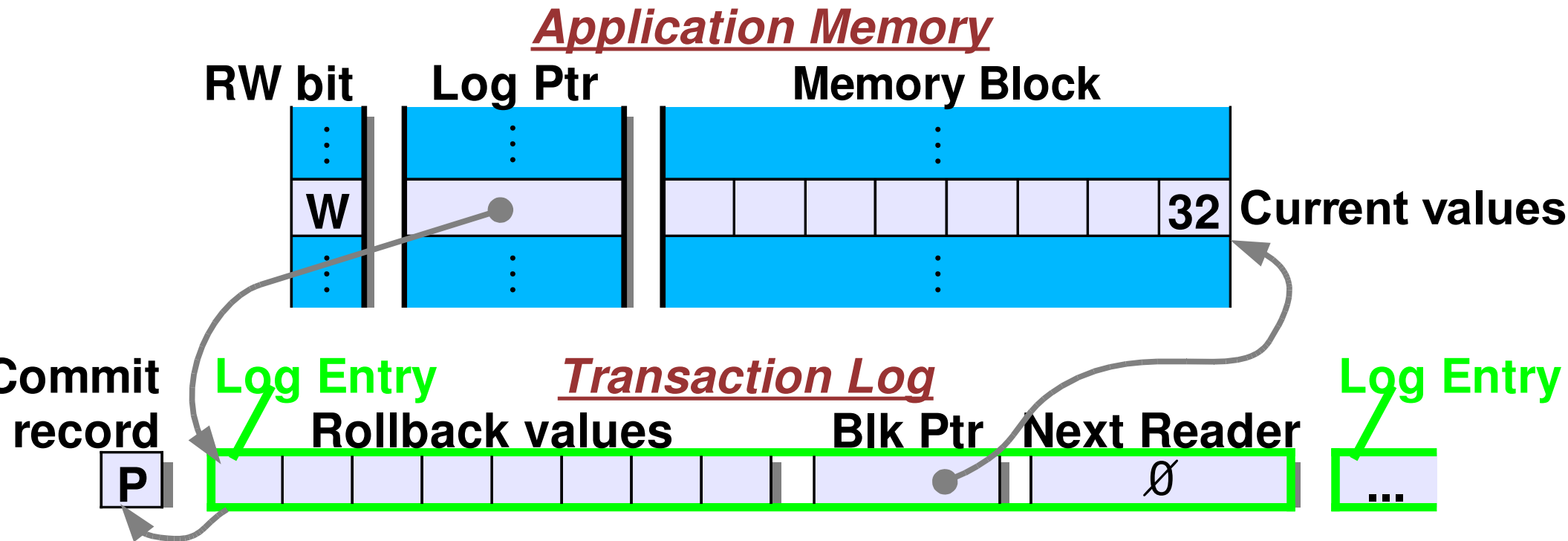
- **Problems solved:**
  - **Xactions as large as physical memory**
  - **Scalable** overflow and commit
  - **Easy to implement!**
  - **Low overhead**
  - **May speed up Linux!**
- **Open Problems...**
  - Is “physical memory” **large enough?**
  - What about **duration?**
    - **Time-slice interrupts!**

# Beyond LTM: UTM



- We can do better!
- The UTM architecture allows transactions as **large as virtual memory**, of **unlimited duration**, which can **migrate** without restart
- Same XBEGIN pc/XEND ISA; same register rollback mechanism
- Canonical transaction info is now stored in single **xstate** data struct in main memory

# xstate data structure



- Transaction log in DRAM for each active transaction
  - **commit record**: PENDING, COMMITTED, ABORTED
  - vector of **log entries** w/ “rollback” values
    - each corresponds to a block in main memory
- Log ptr & **RW bit** for each application memory block
  - **Log ptr/next reader** form linked list of all log entries for a given block

# Caching in UTM



- **Most log entries don't need to be created**
- **Transaction state stored in cache/overflow DRAM and monitored using cache-coherence, as in LTM**
- **Only create transaction log when thread is descheduled, or run out of physical mem.**
- **Can discard all log entries when xaction commits or aborts**
  - **Commit – block left in X state in cache**
  - **Abort – use old value in main memory**
- **In-cache representation need not match xstate representation**

# *Performance/Limits of UTM*

- **Limits:**
  - **More-complicated** implementation
    - Best way to create xstate from LTM state?
  - **Performance** impact of swapping.
    - When should we abort rather than swap?
- **Benefits:**
  - Unlimited **footprint**
  - Unlimited **duration**
  - **Migration** and **paging** possible
  - **Performance** may be as fast as LTM in the common case

# Hybrid Hardware/Software Implementation

- **Hardware transaction implementation is very fast!**  
**But it is limited:**
  - Slow once you exceed cache capacity
  - Transaction lifetime limits (context switches)
  - Limited semantic flexibility (nesting, etc)
- **Software transaction implementation is unlimited and very flexible!**
  - But transactions may be slow
- **Solution: failover from hardware to software**
  - Simplest mechanism: after first hardware abort, execute transaction in software
  - Need to ensure that the two algorithms play nicely with each other (consistent views)  
→ see next slide...

# Cooperation

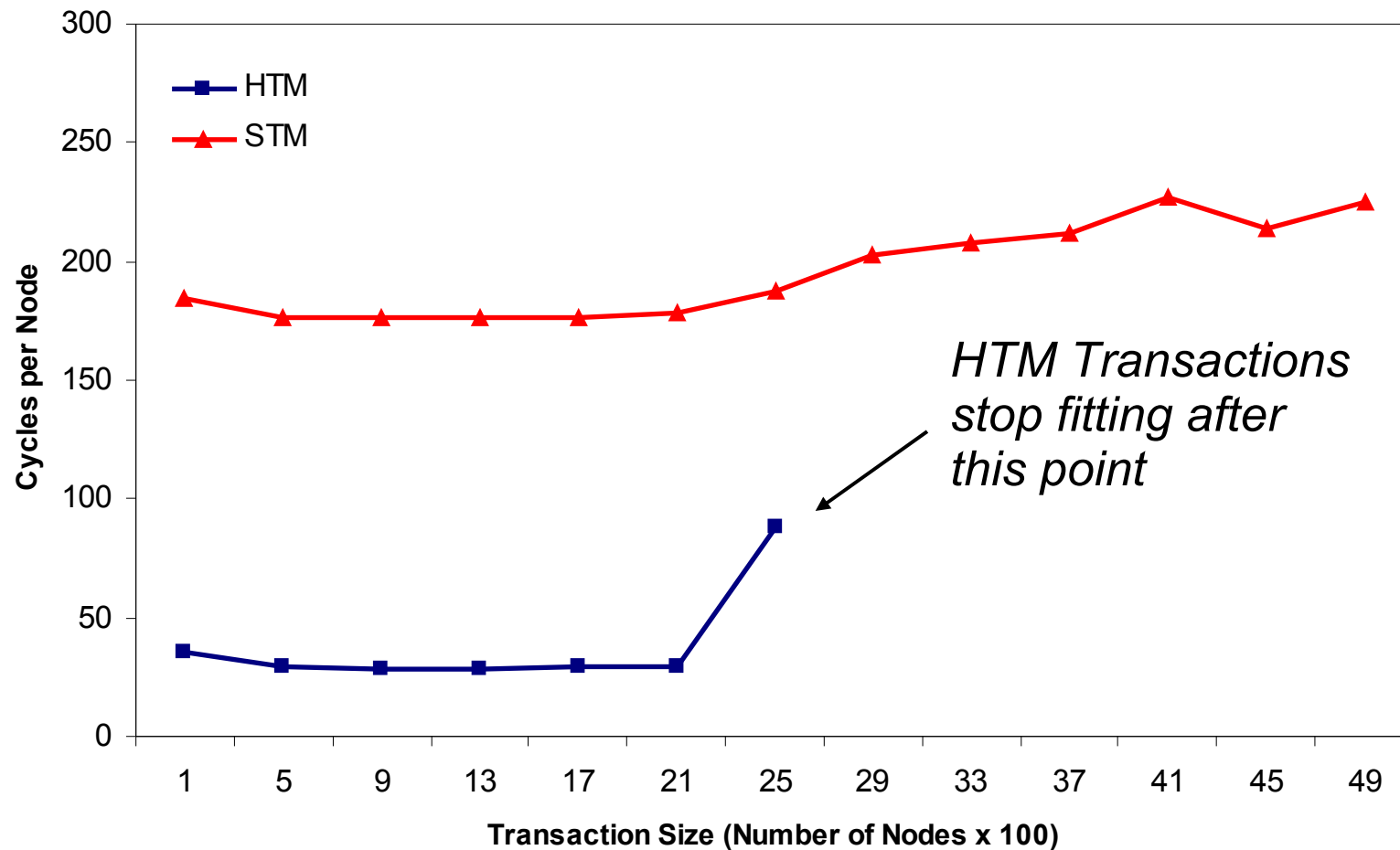
- Software transaction mechanism writing FLAG over object fields is sufficient to abort conflict transaction in LTM
- LTM must execute ReadNT/WriteNT algorithms (**read barrier**) to cooperate with the software mechanism
  - no extra silicon needed!
  - can still leverage compiler analysis
- Other synergies:
  - non-blocking functional array implementation
  - LL/SC sequences

# *Hardware/Software Implementation*

- **Hardware transaction implementation is very fast! But it is limited:**
  - **Slow once you exceed cache capacity**
  - **Transaction lifetime limits (context switches)**
  - **Limited semantic flexibility (nesting, etc)**
- **Software transaction implementation is unlimited and very flexible!**
  - **But transactions may be slow**
- **Solution: failover from hardware to software**
  - **Simplest mechanism: after first hardware abort, execute transaction in software**
  - **Need to ensure that the two algorithms play nicely with each other (consistent views)**

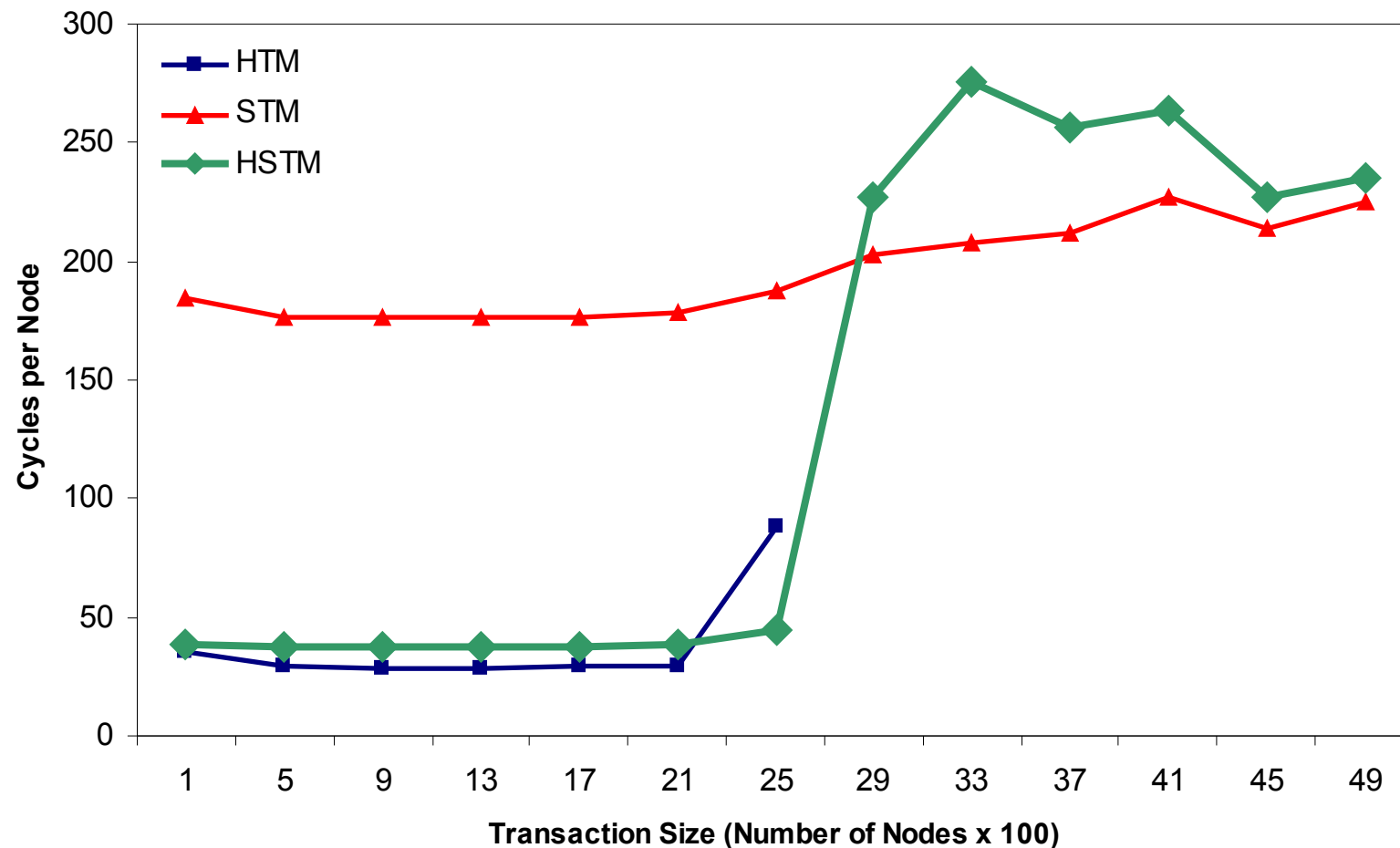
# Leveraging hardware for speed

- Simple node-push benchmark [Lie '04]
- As xaction size increases, we eventually run out of cache space in the HW transaction scheme



# Leveraging hardware for speed

- Simple node-push benchmark [Lie '04]
- **Hybrid scheme best of both worlds!**



# *Related Work*

- **HTM work**
  - Knight, Herlihy&Moss, BBN Pluribus
  - Oklahoma Update (Stone et al)
- **Speculative Lock Elision/Transactional Lock Removal (Rajwar & Goodman)**
  - Use “locks” as the API, dynamically translate to transactional regions
- **Speculative Synchronization (Martinez & Torrellas)**
  - Speculatively execute locking code
- **TM Coherency and Consistency (Hammond et al)**
  - Relies on broadcast for large transactions
- **Software Transactional Memory**
  - Harris&Fraser, Shavit&Touitou, Herlihy et al

# *Conclusions*

- **Transactional/non-transactional cooperation is really a lot like STM/HTM cooperation**
  - same mechanism can be used!
- **The Large Object Problem can be solved!**
  - Good news for object-based transactions
  - Compiler and analysis benefits to reap
- **Writing correct transaction protocols is hard**
  - Model checking can help

# *Conclusions*

- **Transactional Memory systems should support unbounded transactions in hardware**
- **Both fully-scalable (UTM) and easily-implemented (LTM) systems are possible**
- **Big step towards making parallel computing practical and ubiquitous!**

# Conclusions

- **First look at xaction properties of Linux:**
  - 99.9% of xactions touch  $\leq 54$  cache lines
  - but may touch  $> 8000$  cache lines
  - 4x concurrency?
- **Unbounded, scalable, and efficient**  
**Transactional Memory systems can be built.**
  - Support large, frequent, and concurrent xactions
  - What *could* software for these look like?
    - Allow programmers to (finally!) use our parallel systems!
- **Two implementable architectures:**
  - LTM: easy to realize, almost unbounded
  - UTM: truly unbounded

# *Open questions*

- **I/O interface?**
- **Transaction ordering?**
  - **Sequential threads provide inherent ordering**
- **Programming model?**
- **Conflict resolution strategies**

# *Graveyard of Unused slides*

## *Multi-object atomic update*

- Programmer's mental model of locks can be faulty
- **Monitor synchronization**: associates locks with objects
- Promises modularity: locking code stays with encapsulated object implementation
- Often breaks down for multiple-object scenarios
- End result: **unreliable software, broken modularity**

# *A problem with multiple objects*

```

public final class StringBuffer ... {
    private char value[ ];
    private int count;
    ...
    public synchronized StringBuffer append(StringBuffer sb) {
        ...
A:int len = sb.length();
        int newcount = count + len;
        if (newcount > value.length)
            expandCapacity(newcount);
        // next statement may use state len
B:sb.getChars(0, len, value, count);
        count = newcount;
        return this;
    }
    public synchronized int length() { return count; }
    public synchronized void getChars(...) { ... }
}

```

# *Why Transactions?*

- **Concurrency control**
  - Locking disciplines are awkward, error-prone, and limit concurrency
    - Especially with multiple objects!
  - Nonblocking transaction primitives can express optimistic concurrency more simply
    - Focus on “performance” instead of “correctness”
- **Fault-tolerance**
  - Locks are irreversible; semantics for exceptions/crashes unclear
    - Also: “priority inversion”
  - Programming languages in general are irreversible
  - Transactions allow clean “undo”

# *Conventional Locking: Ordering*

- **When more than one object is involved in a critical region, **deadlocks may occur!****
  - Thread 1 grabs A then tries to grab B
  - Thread 2 grabs B then tries to grab A
  - No progress possible!
- **Solution: all locks ordered**
  - A before B
  - Thread 1 grabs A then B
  - Thread 2 grabs A then B
  - No deadlock

# Conventional Locking: Ordering

- Maintaining lock order is a lot of work!
- Programmer must choose, document, and rigorously adhere to a **global** locking protocol for each object type
  - **development overhead!**
- All symmetric locked objects must include lock order field, which must be assigned uniquely
  - **space overhead!**
- Every multi-object lock operation must include proper conditionals
  - which lock do I take first? which do I take next?
  - **execution-time overhead!**
- ***No exceptions!***

# *Fault-tolerance*

- Locks are **irreversible**
- When a thread fails holding a lock, the system will crash
  - it's only a matter of time before someone else attempts to grab that lock
- What are the proper semantics for exceptions thrown within a critical region?
  - data structure consistency not guaranteed
- Asynchronous exceptions?

# Priority Inversion

- Well-known problem with locks
- Described by Lampson/Redell in 1980 (Mesa)
- Mars Pathfinder in 1997, etc, etc, etc
- Low-priority task takes a lock needed by a high-priority task -> the **high priority task must wait!**
- Clumsy solution: the low priority task must become high priority
- What if the low priority task takes a long time?

# *Invisible transactions?*

- **Rajwar & Goodman: Speculative Lock Elision and Transactional Lock Removal**
  - speculatively identify locks; make xactions
- **Martinez & Torrellas: Speculative Synchronization**
  - guarantee fwd progress w/ non-speculative thread



*Keep  
transactions  
visible*

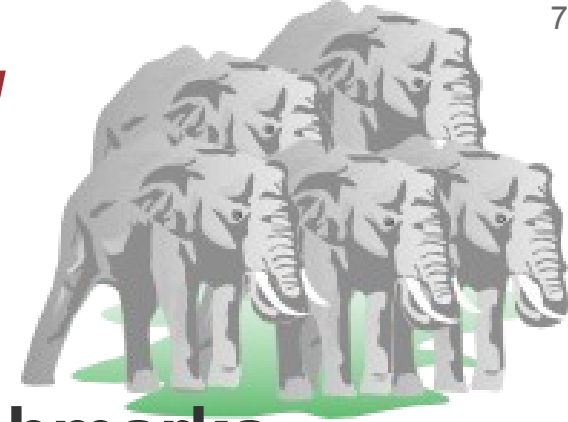
# *Infrequent, Small, Mostly-Serial?*

To date, transactions assumed to be:

- **Small**
  - BBN Pluribus (~1975): 16 clock-cycle bus-locked “transaction”
  - Knight; Herlihy & Moss: transactions which fit in cache
- **Infrequent**
  - Software Transactional Memory (Shavit & Touitou; Harris & Fraser; Herlihy et al)
- **Mostly-serial**
  - Transactional Coherence & Consistency (Hammond, Wong, et al)



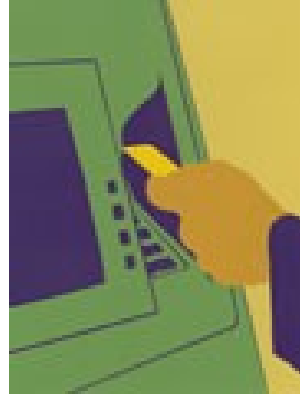
# May Be Large, Frequent, and Concurrent



- Lots of small xactions
  - Millions of xactions in these benchmarks
  - **Problem for software-only schemes**
- Significant tail: large xactions are few, but very large
  - Thousands of cache lines touched
  - **Problem for bounded transactional schemes**
- Potential for additional concurrency
  - Distribution of hot cache lines suggest that **4x more concurrency** may be possible on our Linux benchmarks

*Programmers want unbounded xactions...*

# Transactional Programming



- Locks: the devil we know
- Complex sync techniques: library-only
  - Nonblocking synchronization
  - Bounded transactions
    - **Compilers don't expose memory references**  
(Indirect dispatch, optimizations, constants)
    - Not portable! Changing cache-size breaks apps.
- Unbounded Transactions:
  - Can be thought about at **high-level**
  - Match programmer's **intuition** about atomicity
  - Allow black box code to be **composed safely**
  - Promise future excitement!
    - **Fault-tolerance** / exception-handling
    - Speculation / search

# *Transactional Memory Systems*

- **Hardware Transactional Memory (HTM)**
  - Knight, Herlihy & Moss, BBN Pluribus
  - atomicity through architectural means
- **Software Transactional Memory (STM)**
  - atomicity through languages, compiler, libraries
- **Traditionally assume:**
  - Transactions are “small” and thus it is reasonable to bound their size (esp. HTM)
  - Transactions are “infrequent” and thus overhead is acceptable (esp. STM)

# *Transaction Size Distribution*

- **Lots of small xactions**
  - Millions of xactions in these benchmarks
  - **Use hardware support to make these fast**
- **Significant tail: large xactions are few, but very large**
  - Thousands of cache lines touched
  - **Unbounded Transactional Memory makes these possible**

*Free the compiler/programmer/ISA from arbitrary limits on transaction size*

# *Our Thesis*

*Transactional memory should support transactions of **arbitrary size and duration**. Such support should be provided with **hardware assistance**, and it should be made **visible to the software** through the machine's instruction-set architecture (ISA).*

An **unbounded TM** can handle transactions of arbitrary duration with footprints comparable to its virtual memory space

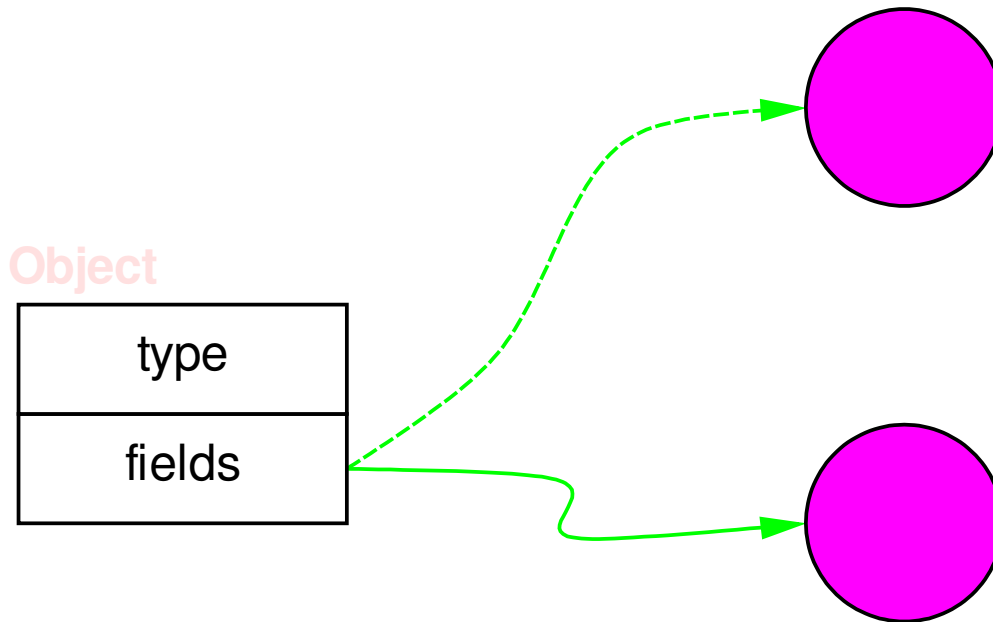
# *Three Big Ideas*

- **Functional Arrays: A solution to the Large Object Problem**
- **Cooperating with FLAGS**
  - Non-transactional code interacting with transactions
  - Software transactions interacting with a Hardware Transactional Memory
- **Model-checking Software Transactions**

# *The Large Object Problem*

# *Single-Object Protocol*

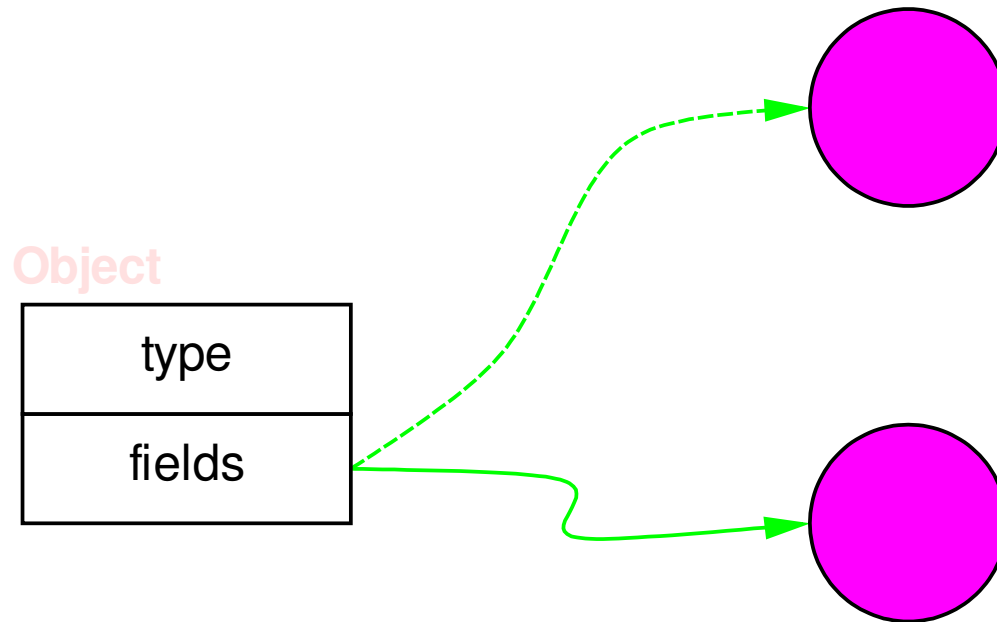
Valid for operations on a single object only.



- **Object representation contains a pointer to object contents.**
- **Object mutation inside transaction creates new object contents.**

# Single-Object Protocol

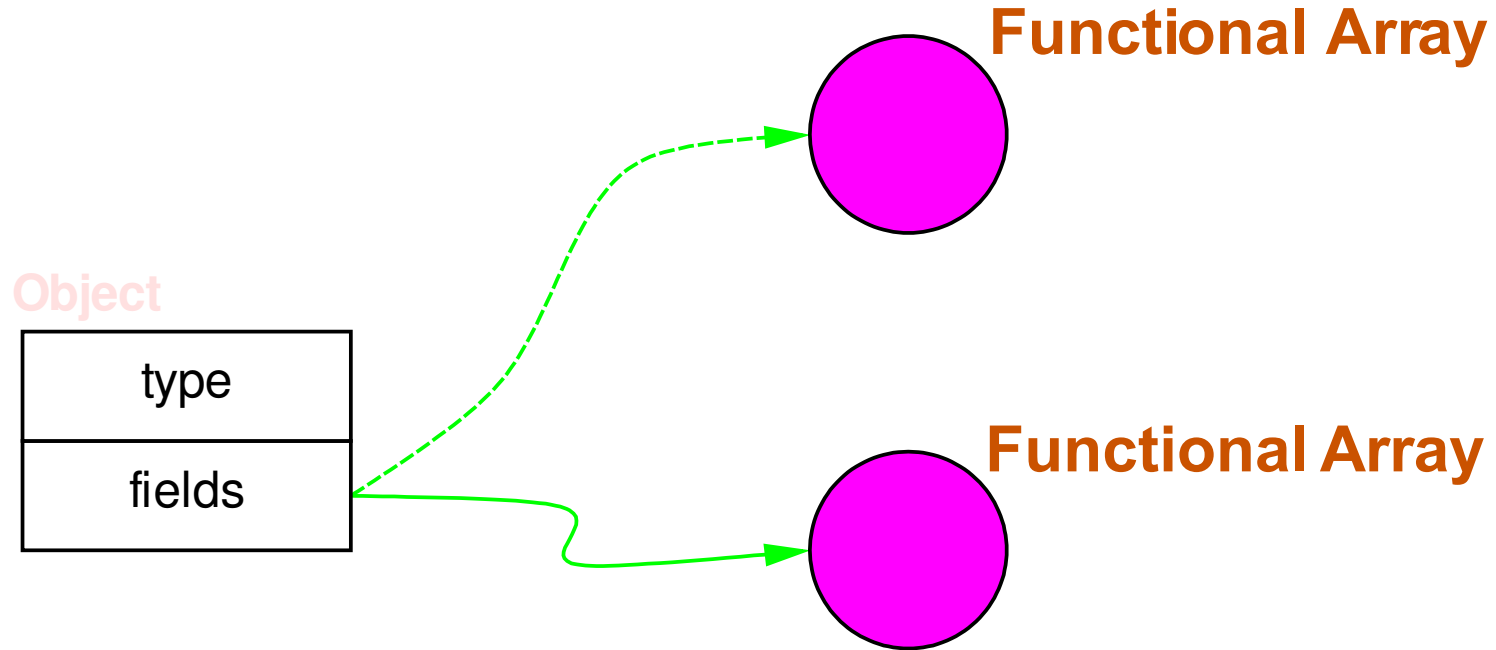
Valid for operations on a single object only.



- At start of transaction, load and remember `fields` pointer as *prior state*.
- To commit, **compare-and-swap** the *result of operation* for *prior state*.

# Single-Object Protocol

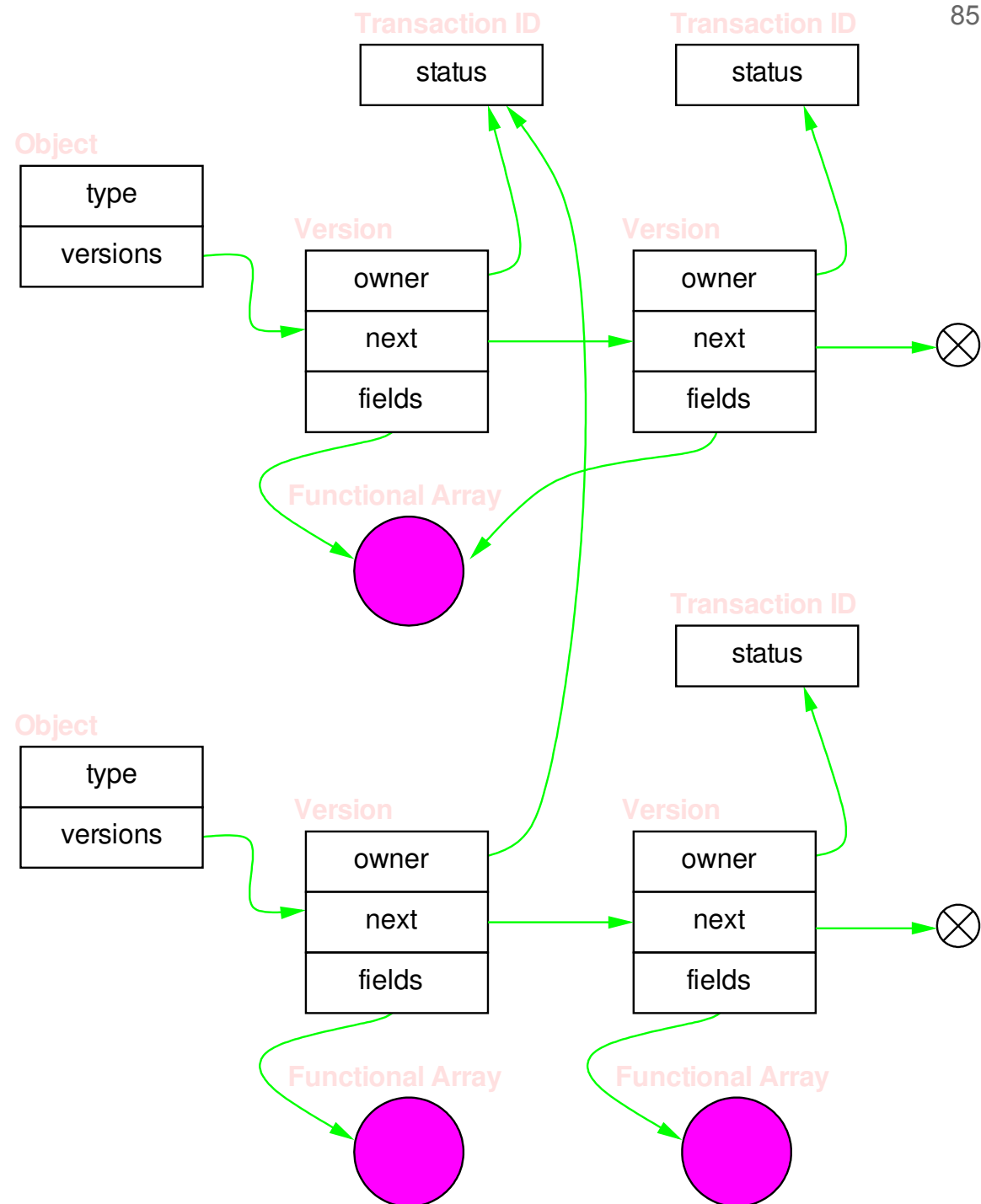
Valid for operations on a single object only.



- **Large Object Problem:** cloning *prior state* for *result of operation* is  $O(\text{object size})$
- **Solution:** use a data structure where cloning is cheap –  $O(1)$  would be nice!

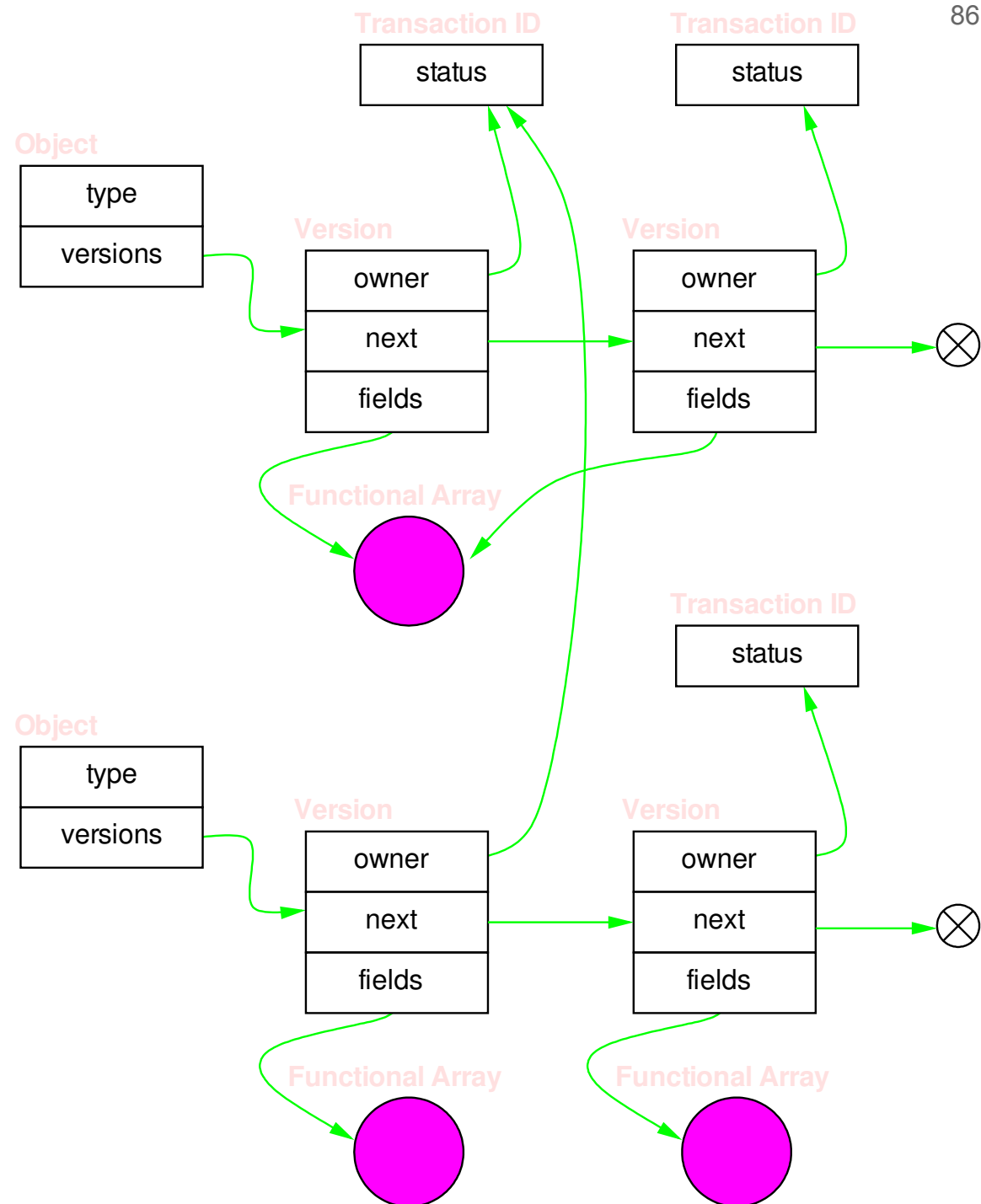
# Multiple-Object Protocol

- Objects point to lists of versions.
- Each version has an associated **Transaction ID** and field array reference.
- Transaction IDs are initialized to **WAITING** and are **changed exactly once** to **COMMITTED** or **ABORTED**.



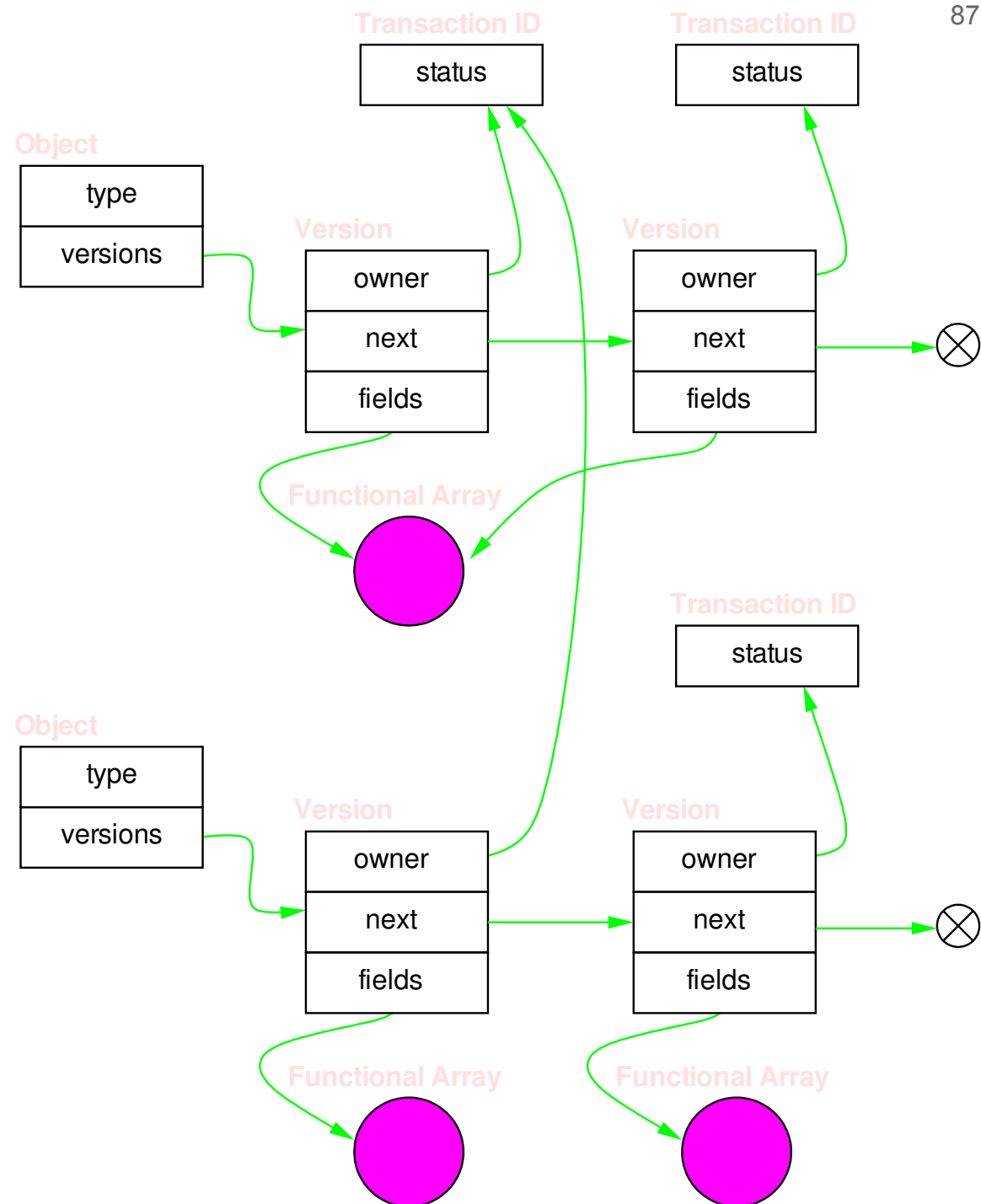
# Multiple-Object Protocol

- At end of transaction, attempt to set Transaction ID to **COMMITTED**.
- Value of object is the **value of the first committed version**.
- **ABORTED** versions can be collected.



# Multiple-Object Protocol

- Only one **WAITING** version allowed on versions list, and it must be at the head.
- Before we can link a new version onto the versions list, we must ensure that every other version is either **COMMITTED** or **ABORTED**.



# *Non-blocking concurrent algorithms are hard!*

- In published work on Synthesis, a non-blocking operating system implementation, three separate races were found:
  - One **ABA problem** in LIFO stack
  - One **likely race** in MP-SC FIFO queue
  - One **interesting corner case** in quaject callback handling
- It's hard to get these right! Ad hoc reasoning doesn't cut it.
- Non-blocking algorithms are too hard for the programmer
- Let's get it right **once** (and verify this!)

# *The Spin Model Checker*

- Spin is a **model checker** for communicating concurrent processes. It checks:
  - Safety/termination properties
  - Liveness/deadlock properties
  - Path assertions (requirements/never claims)
- It works on **finite** models, written the Promela language, which describe **infinite** executions.
- Explores the **entire state space** of the model, including all possible concurrent executions, verifying that Bad Things don't happen.
- Not an absolute proof – pretty useful in practice
- **Make systems reliable by concentrating complexity in a verifiable component**

# Spin theory

- Generates a **Büchi Automaton** from the Promela specification.
  - Finite-state machine w/ special acceptance conditions
  - Transitions correspond to executability of statements
- **Depth-first search of state space**, with each state stored in a hashtable to detect cycles and prevent duplication of work
  - If  $x$  followed by  $y$  leads to the same state as  $y$  followed by  $x$ , will not re-traverse the succeeding steps
- If memory is not sufficient to hold all states, may **ignore hashtable collisions**: requires one bit per entry. # collisions provides approximate coverage metric

# *Verification with Spin*

- **Modeled the software transaction implementation in Promela**
- **Low-level model – every memory operation represented**
  - details in the paper
- **Spin used 16G of memory to check the implementation within a 6-version 2-object scope.**

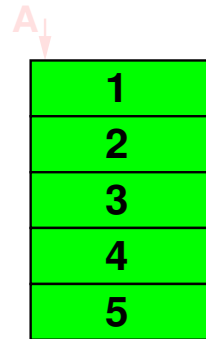
# *Bugs Found*

- **Memory management**
  - reference counting, object recycling
- **Read caching**
  - check freshness of copies in local variables
- **“Big” bug**
  - missing abort of readers during a non-transactional write (field copy back)

## *Functional arrays*

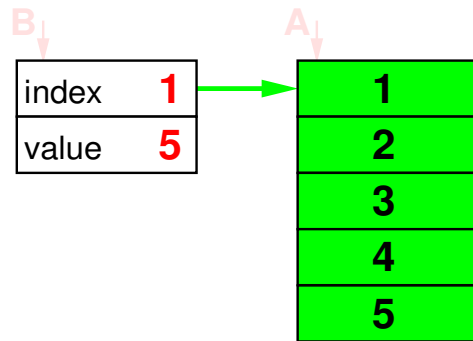
- Functional arrays are **persistent**: after an element is updated both the new and the old contents of the array are available for use.
- Fundamental operation:
 
$$\text{Update}(A, i, v) : A \rightarrow N_0 \rightarrow V \rightarrow A$$
- Arrays are just mappings from integer to value; any persistent map can be used as a functional array.
- A *fast* functional array will have  **$O(1)$  access and update** for the common cases.
  - Variant of shallow binding due to [Chuang '94]

# Functional Arrays using Shallow Binding



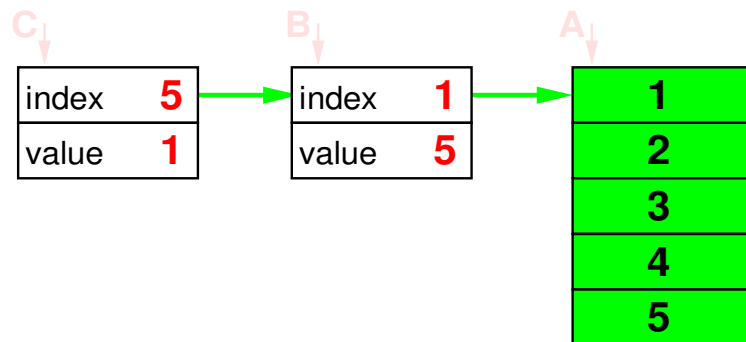
- A functional array is either a **cache node**...

# Functional Arrays using Shallow Binding



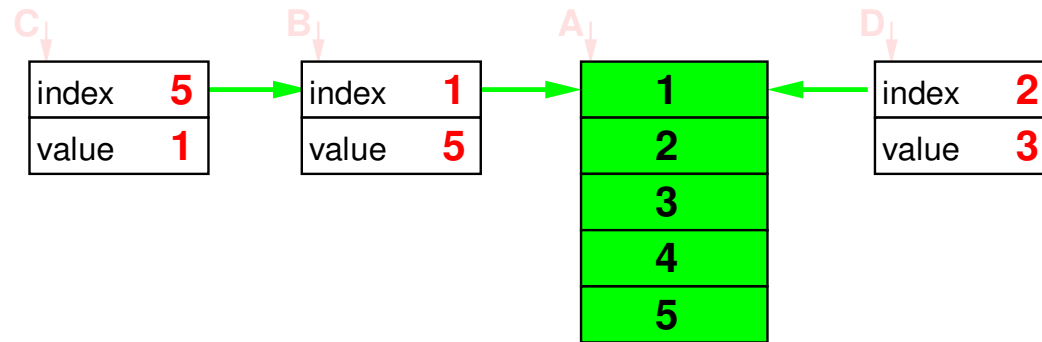
- A functional array is either a *cache node* or a *difference node*.
- $A[1]=1$  but  $B[1]=5$

# Functional Arrays using Shallow Binding



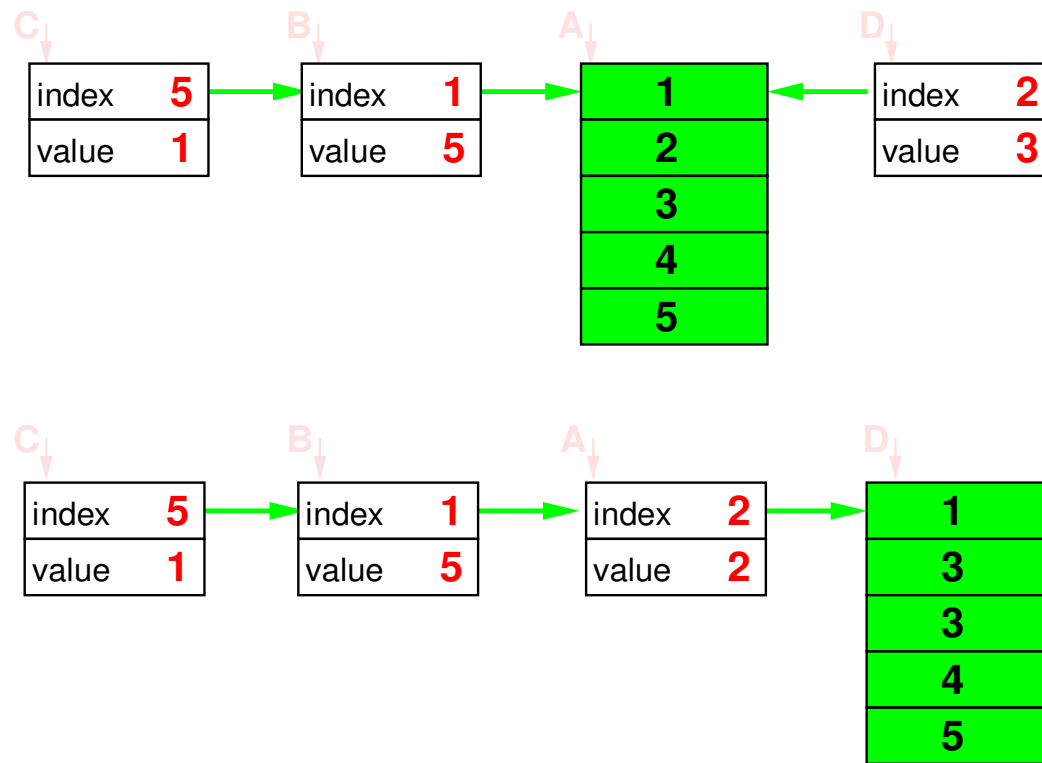
- Changing one element is  $O(1)$

# Functional Arrays using Shallow Binding



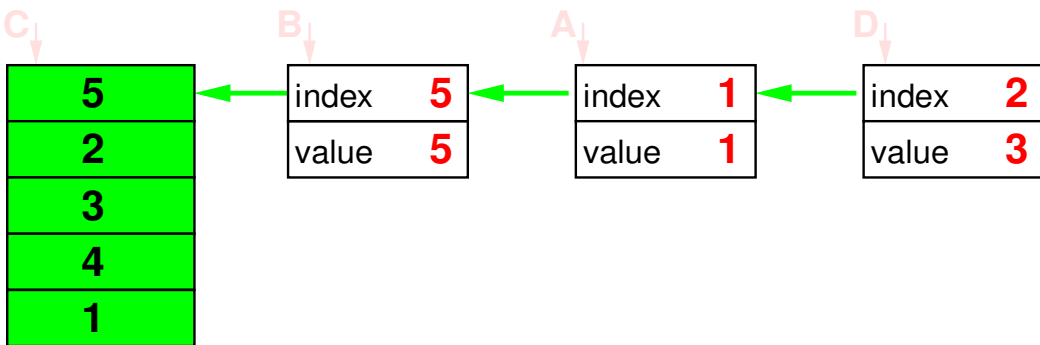
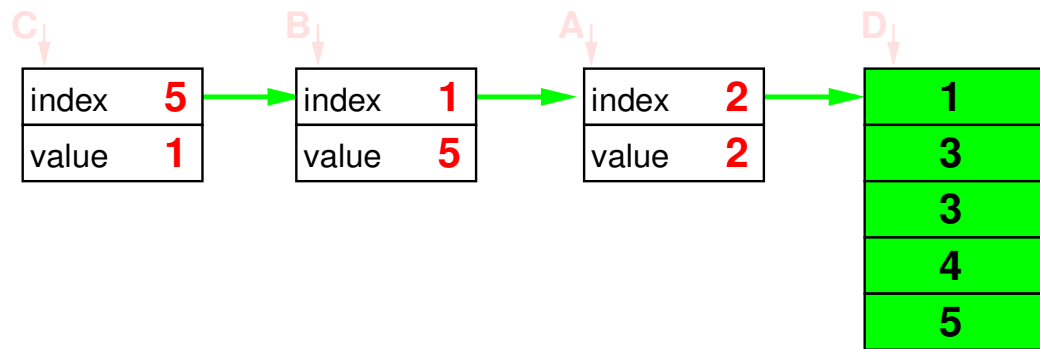
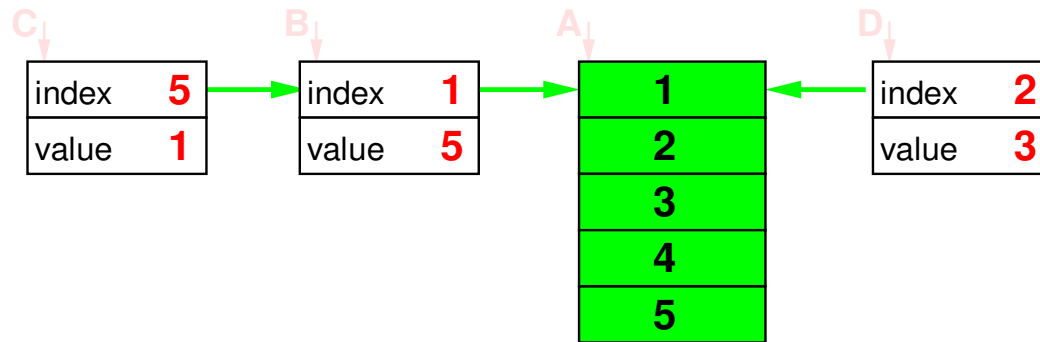
- $A[1] = D[1] = 1$        $C[1] = B[1] = 5$
- $C[5] = 1$                $D[2] = 3$

# Functional Arrays using Shallow Binding



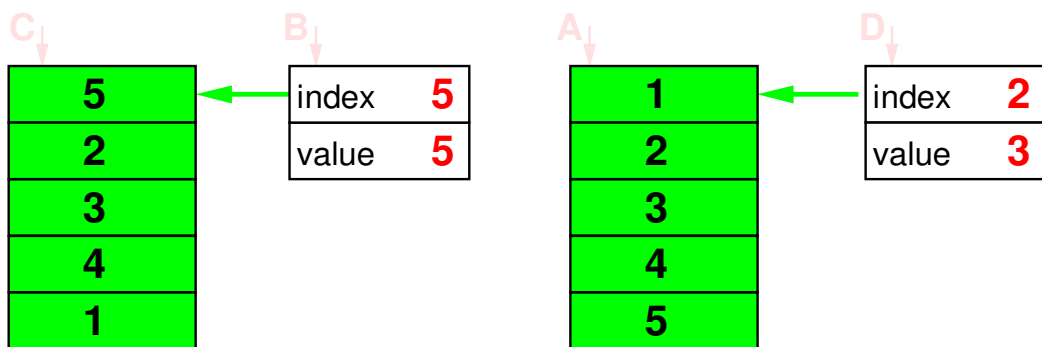
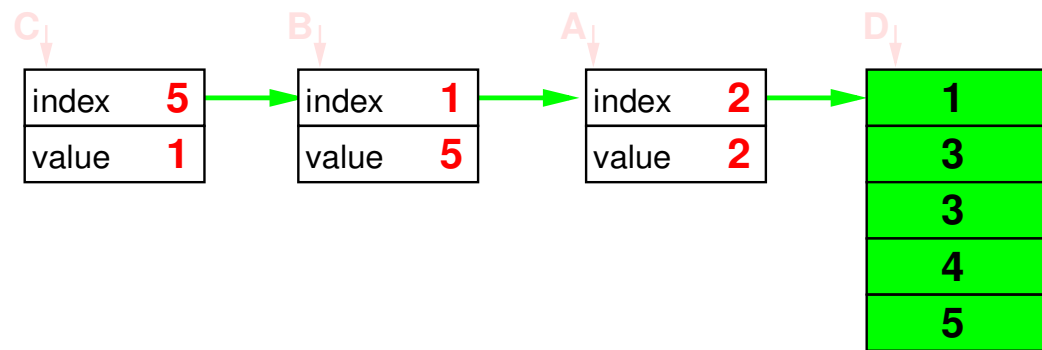
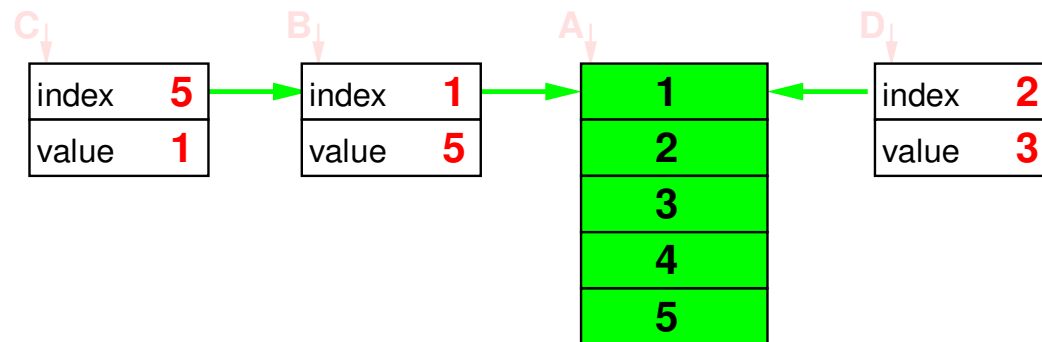
- We *rotate* the cache node on reads to keep access times fast.
- The bottom shows the graph after D is read.

# Functional Arrays using Shallow Binding



- C is read.
- Ping-pong danger!

# Functional Arrays using Shallow Binding

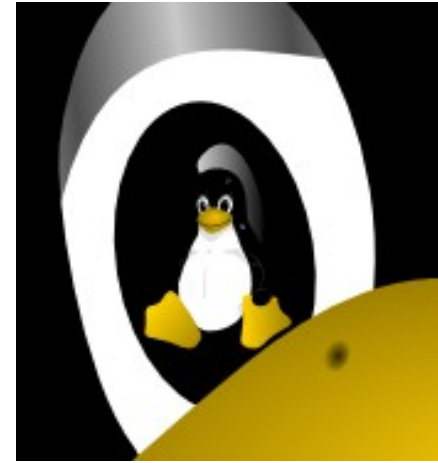


- **Split** with  $1/N$  chance.

# ***Making a non-blocking algorithm***

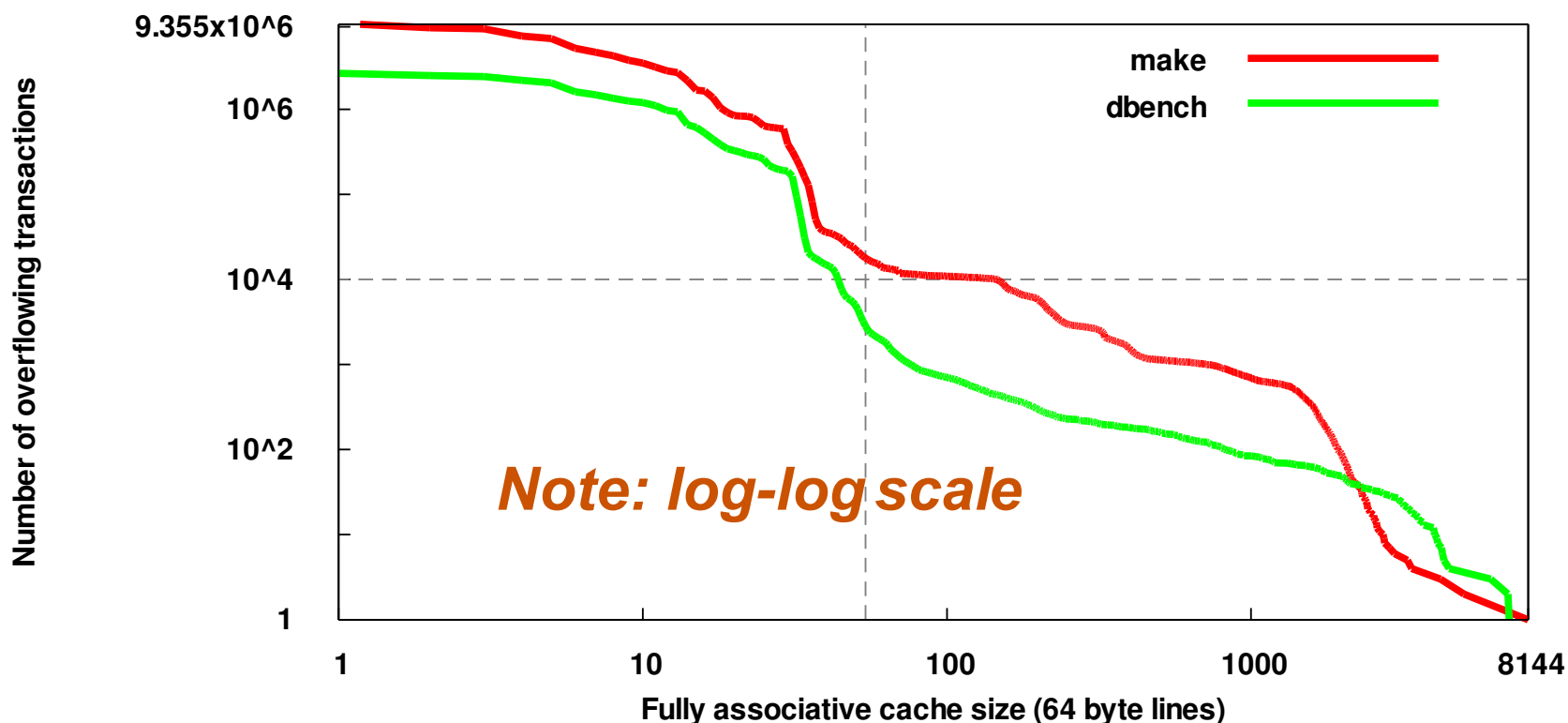
- **Adding difference nodes is easy.**
- **Two hard operations:**
  - **Rotation**
  - **Splitting**
- **These can be made non-blocking [Ananian '03]**
- **Can also use a small Hardware Transactional Memory to implement these operations.**

# Transact-ifying Linux



- Experiment to discover xaction properties of large real-world app.
  - First complete OS investigated!
- **User-Mode Linux 2.4.19**
  - instrumented every load and store, all locks
  - locks → xactions; locks not held over I/O!
  - run 2-way SMP (two processes; two processors)
- Two workloads
  - Parallel make of Linux kernel ('**make linux**')
  - **dbench** running three clients
- Run program to get a trace; run trace through **Transactional Memory simulator**
  - 1MB 4-way set-associative 64-byte-line cache
  - Paper also has simulation runs for SpecJVM98

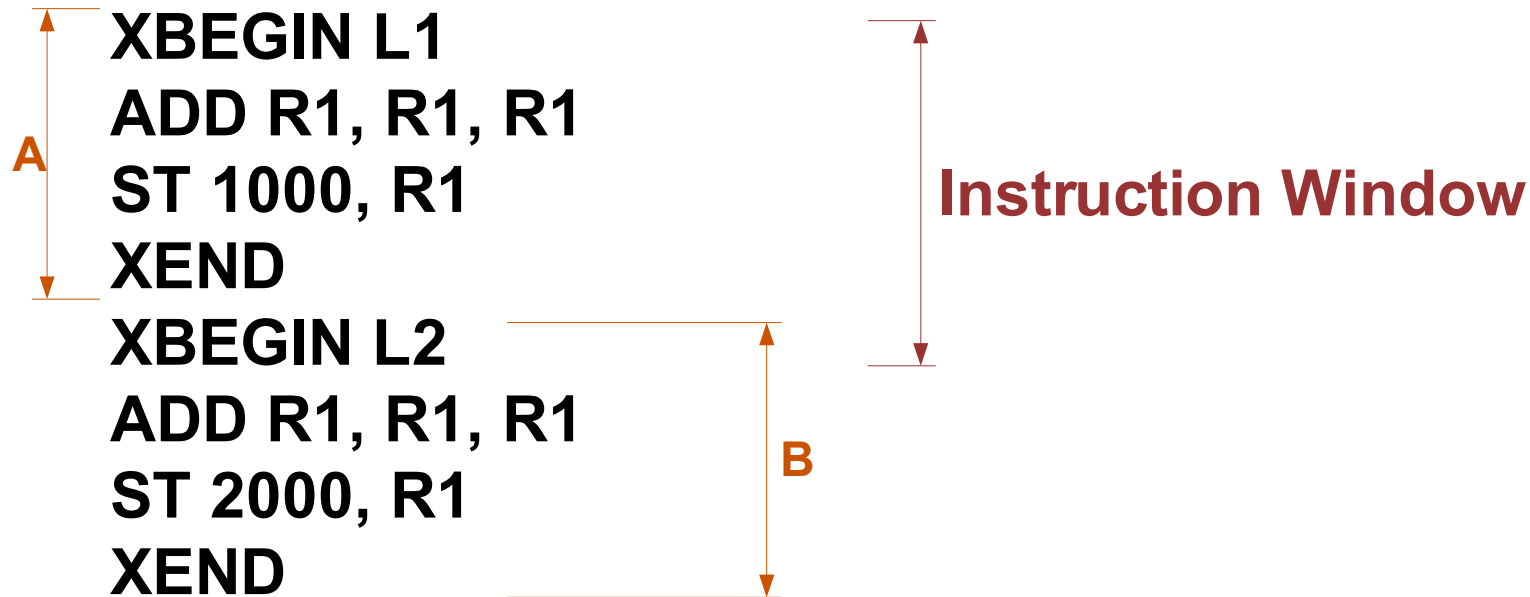
# TM Cache-size requirements (Linux)



- # of overflowing transactions as a function of (fully-associative) cache size for `make_linux` & `dbench`
- Almost all of the transactions require  $< 100$  cache lines
  - **99.9% need fewer than 54 cache lines**
- There are, however, some very large transactions!
  - **$>500$ k-byte fully-associative cache required**

# Multiple in-flight transactions

## Original



- This example has two transactions, with abort handlers at L1 and L2
- Assume instruction window of length 5
  - allows us to speculate into next transaction(s)

graduate →

# Multiple in-flight transactions

decode →

<u>Original</u>	<u>Rename Table</u>	<u>Saved set</u>
XBEGIN L1	R1→P1, ...	{ P1, ... }
ADD R1, R1, R1		
ST 1000, R1		
XEND		
XBEGIN L2		
ADD R1, R1, R1		
ST 2000, R1		
XEND		

- **During instruction decode:**
  - Maintain rename table and “saved” bits
  - “Saved” bits track registers mentioned in current rename table
    - Constant # of set bits: every time a register is added to “saved” set we also remove one

# Multiple in-flight transactions

<u>Original</u>	<u>Rename Table</u>	<u>Saved set</u>
XBEGIN L1	R1→P1, ...	{ P1, ... }
ADD P2, P1, P1	R1→P2, ...	{ P2, ... }
ST 1000, R1		
XEND		
XBEGIN L2		
ADD R1, R1, R1		
ST 2000, R1		
XEND		

- **When XBEGIN is decoded:**
  - Snapshots taken of current Rename table and S-bits.
  - This snapshot is not active until XBEGIN graduates

# Multiple in-flight transactions

graduate →

Original

XBEGIN L1

ADD P2, P1, P1

decode →

ST 1000, P2

XEND

XBEGIN L2

ADD R1, R1, R1

ST 2000, R1

XEND

Rename Table

R1→P1, ...

Saved set

{ P1, ... }

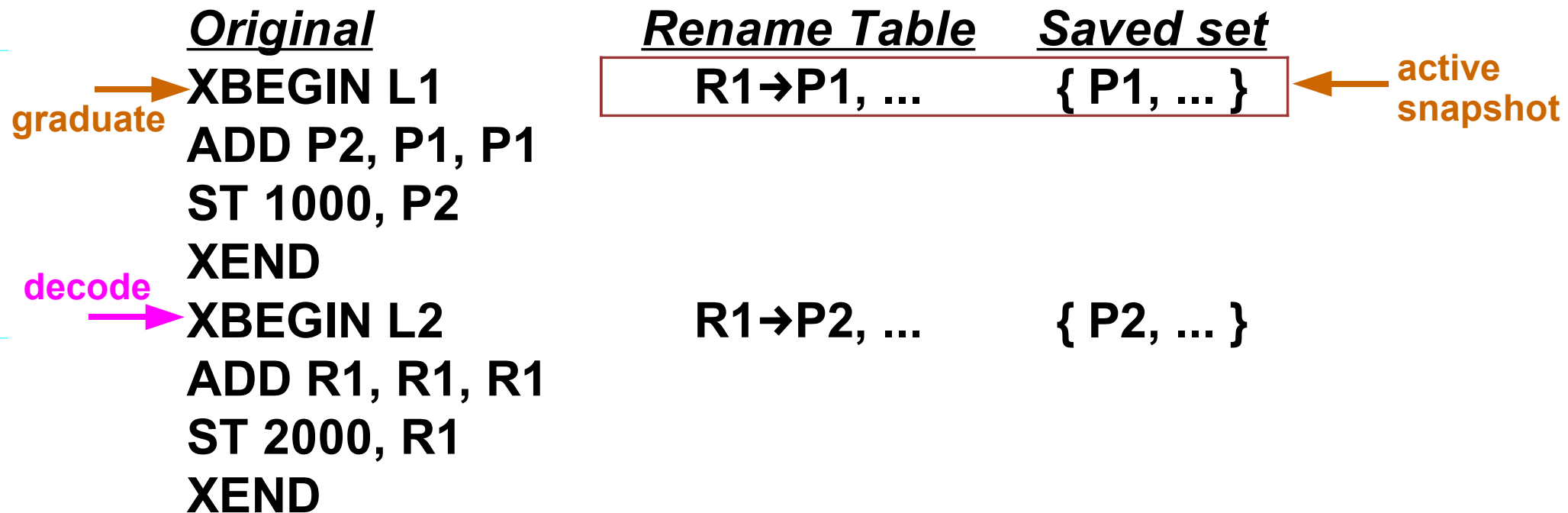
R1→P2, ...

{ P2, ... }

# Multiple in-flight transactions

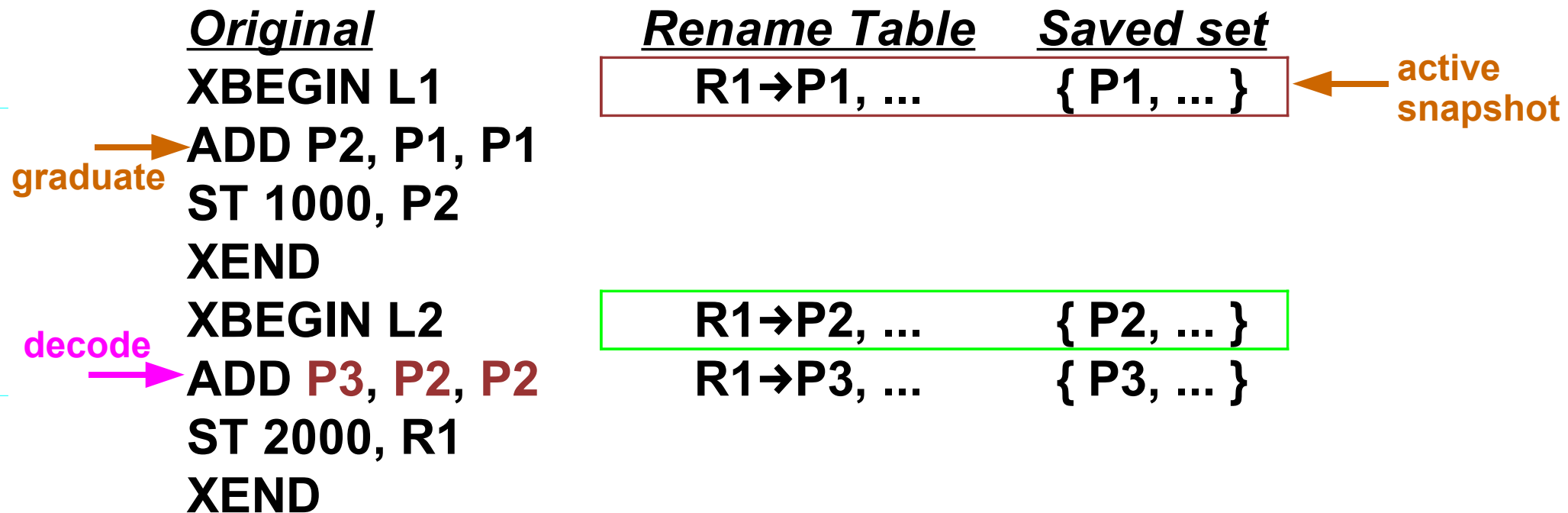
	<u>Original</u>	<u>Rename Table</u>	<u>Saved set</u>
graduate →	XBEGIN L1	R1→P1, ...	{ P1, ... }
	ADD P2, P1, P1		
	ST 1000, P2		
decode →	XEND	R1→P2, ...	{ P2, ... }
	XBEGIN L2		
	ADD R1, R1, R1		
	ST 2000, R1		
	XEND		

# Multiple in-flight transactions



- **When XBEGIN graduates:**
  - Snapshot taken at decode becomes **active**, which will prevent P1 from being reused
  - 1<sup>st</sup> transaction queued to become active in memory
  - To abort, we just restore the active snapshot's rename table

# Multiple in-flight transactions



- We're only reserving registers in the active set
  - This implies that exactly #AR registers are saved
  - This number is strictly limited, even as we speculatively execute through multiple xactions

# Multiple in-flight transactions

	<u>Original</u>	<u>Rename Table</u>	<u>Saved set</u>	
	XBEGIN L1	R1→P1, ...	{ P1, ... }	← active snapshot
	ADD P2, P1, P1			
graduate	ST 1000, P2			
	XEND			
	XBEGIN L2	R1→P2, ...	{ P2, ... }	
	ADD P3, P2, P2			
decode	ST 2000, P3	R1→P3, ...	{ P3, ... }	
	XEND			

- Normally, P1 would be freed here
- Since it's in the active snapshot's “saved” set, we put it on the register reserved list instead

# Multiple in-flight transactions

<u>Original</u>	<u>Rename Table</u>	<u>Saved set</u>
-----------------	---------------------	------------------

XBEGIN L1  
 ADD P2, P1, P1  
 ST 1000, P2

graduate → XEND

XBEGIN L2  
 ADD P3, P2, P2  
 ST 2000, P3

decode → XEND

R1→P2, ...	{ P2, ... }
------------	-------------

R1→P3, ...	{ P3, ... }
------------	-------------

- **When XEND graduates:**
  - Reserved physical registers (P1) are freed, and active snapshot is cleared.
  - Store queue is empty

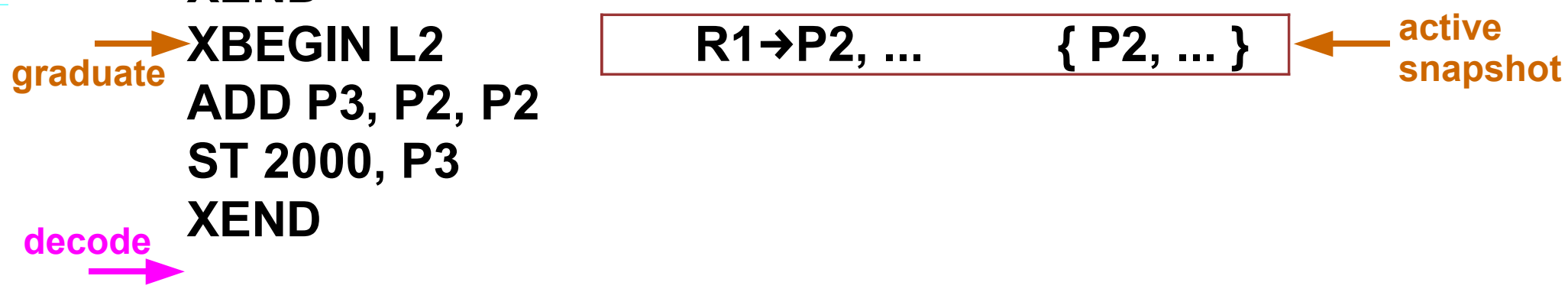
# Multiple in-flight transactions

Original

XBEGIN L1  
 ADD P2, P1, P1  
 ST 1000, P2  
 XEND

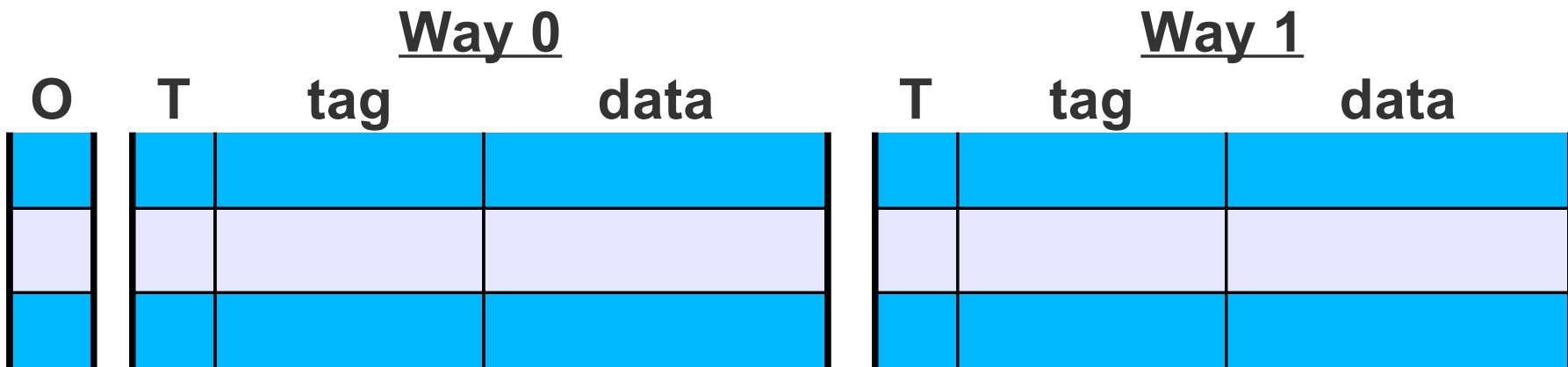
Rename Table

Saved set



- **Second transaction becomes active in memory.**

# Cache overflow mechanism



## Overflow hashtable

key	data

ST 1000, 55  
 XBEGIN L1  
 LD R1, 1000  
 ST 2000, 66  
 ST 3000, 77  
 LD R1, 1000  
 XEND

- Need to keep “current” values as well as “rollback” values
  - Common-case is commit, so keep “current” in cache
  - What if uncommitted “current” values don't all fit in cache?
- Use overflow hashtable as extension of cache
  - Avoid looking here if we can!

# Cache overflow: miss handling

<u>Way 0</u>				<u>Way 1</u>		
O	T	tag	data	T	tag	data
O	T	1000	55	T	2000	66

## Overflow hashtable

key	data
3000	77

- Miss to an overflowed line checks overflow table
- If found, swap overflow and cache line; proceed as hit
- Else, proceed as miss.

ST 1000, 55  
 XBEGIN L1  
 LD R1, 1000  
 ST 2000, 66  
 ST 3000, 77  
 LD R1, 1000  
 XEND



# Cache overflow: commit/abort

<u>Way 0</u>				<u>Way 1</u>		
O	T	tag	data	T	tag	data
O	T	1000	55	T	2000	66

## Overflow hashtable

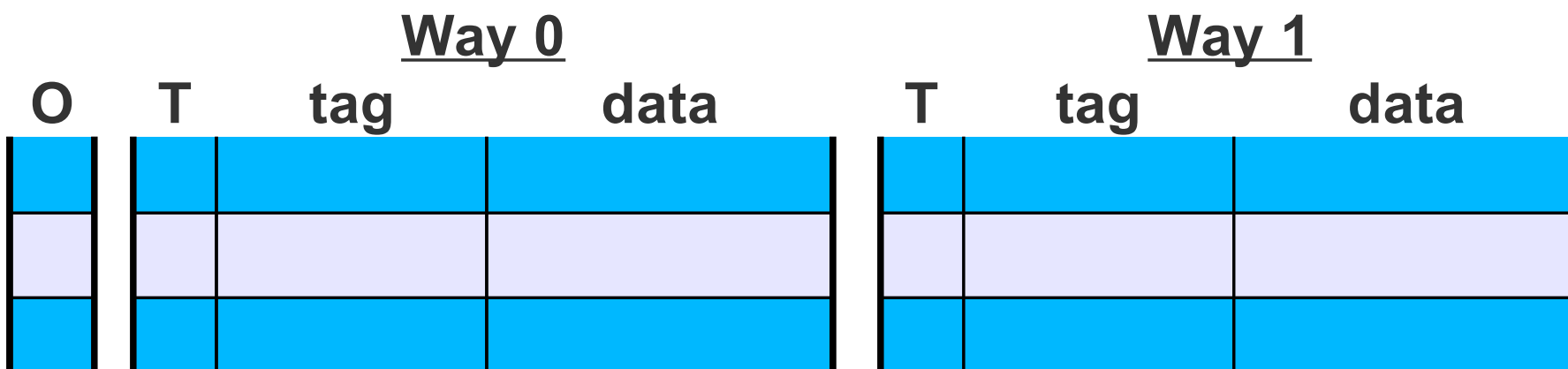
key	data
3000	77

ST 1000, 55  
 XBEGIN L1  
 LD R1, 1000  
 ST 2000, 66  
 ST 3000, 77  
 LD R1, 1000

XEND

- **Abort:**
  - invalidate all lines with T set
  - discard overflow hashtable
  - clear O and T bits
- **Commit:**
  - write back hashtable; **NACK** interventions during this
  - clear O and T bits

# Cache overflow mechanism



## Overflow hashtable

key	data

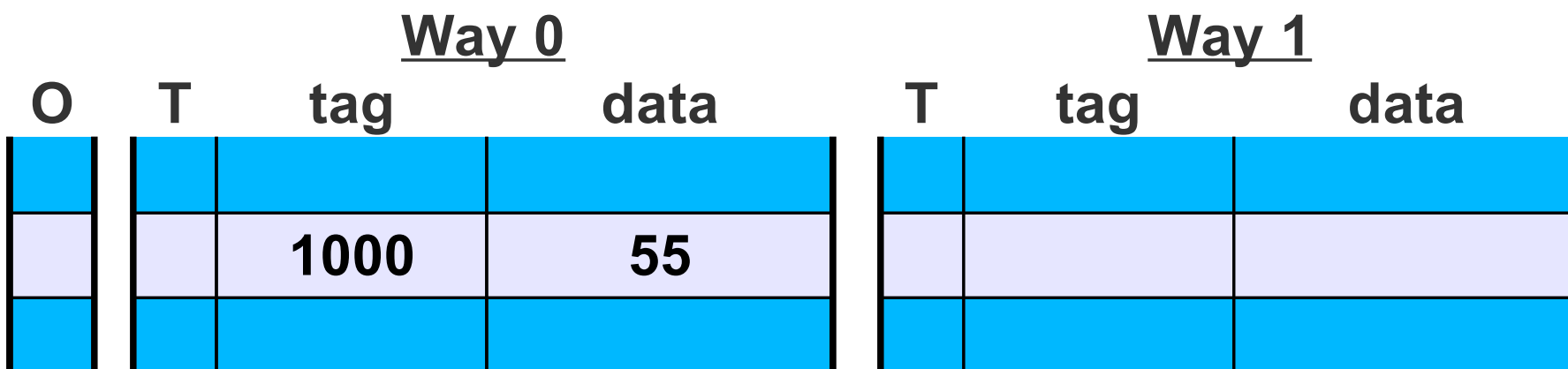
```

ST 1000, 55
XBEGIN L1
LD R1, 1000
ST 2000, 66
ST 3000, 77
LD R1, 1000
XEND

```

- **T bit per cache line**
  - set if accessed during xaction
- **O bit per cache set**
  - indicates set overflow
- **Overflow storage in physical DRAM**
  - allocated/resized by OS
  - probe/miss: complexity of search  $\approx$  page table walk

# Cache overflow mechanism



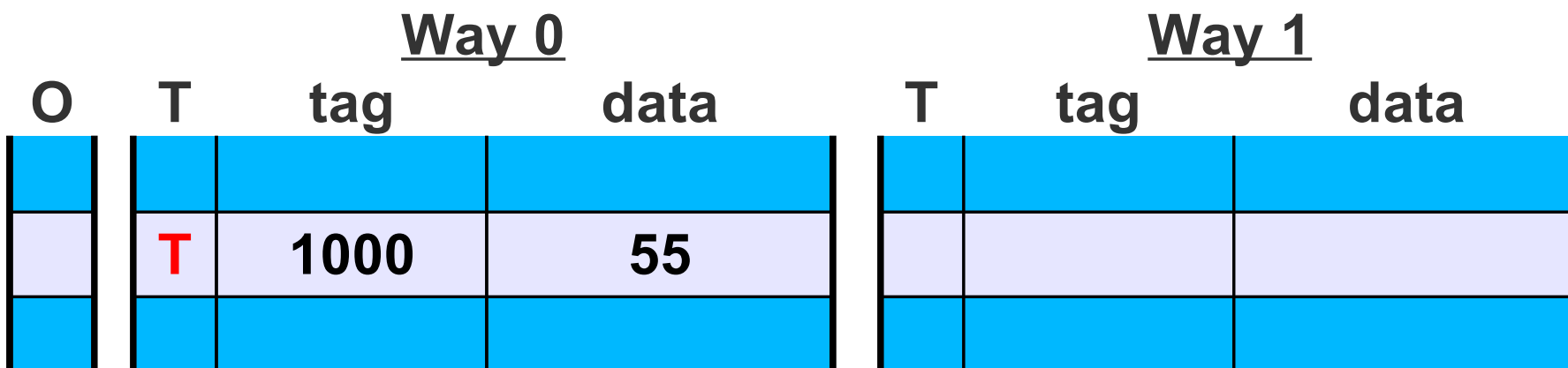
## Overflow hashtable

key	data

- Start with non-transactional data in the cache

ST 1000, 55  
 XBEGIN L1  
 → LD R1, 1000  
 ST 2000, 66  
 ST 3000, 77  
 LD R1, 1000  
 XEND

# Cache overflow: recording reads



## Overflow hashtable

key	data

- Transactional read sets the T bit.

```

ST 1000, 55
XBEGIN L1
LD R1, 1000
ST 2000, 66
ST 3000, 77
LD R1, 1000
XEND

```



# Cache overflow: recording writes

<u>Way 0</u>				<u>Way 1</u>		
O	T	tag	data	T	tag	data
	T	1000	55	T	2000	66

## Overflow hashtable

key	data

- Most transactional writes fit in the cache.

ST 1000, 55  
 XBEGIN L1  
 LD R1, 1000  
 → ST 2000, 66  
 ST 3000, 77  
 LD R1, 1000  
 XEND

# Cache overflow: spilling

<u>Way 0</u>				<u>Way 1</u>		
O	T	tag	data	T	tag	data
<b>O</b>	<b>T</b>	<b>3000</b>	<b>77</b>	<b>T</b>	<b>2000</b>	<b>66</b>

## Overflow hashtable

key	data
<b>1000</b>	<b>55</b>

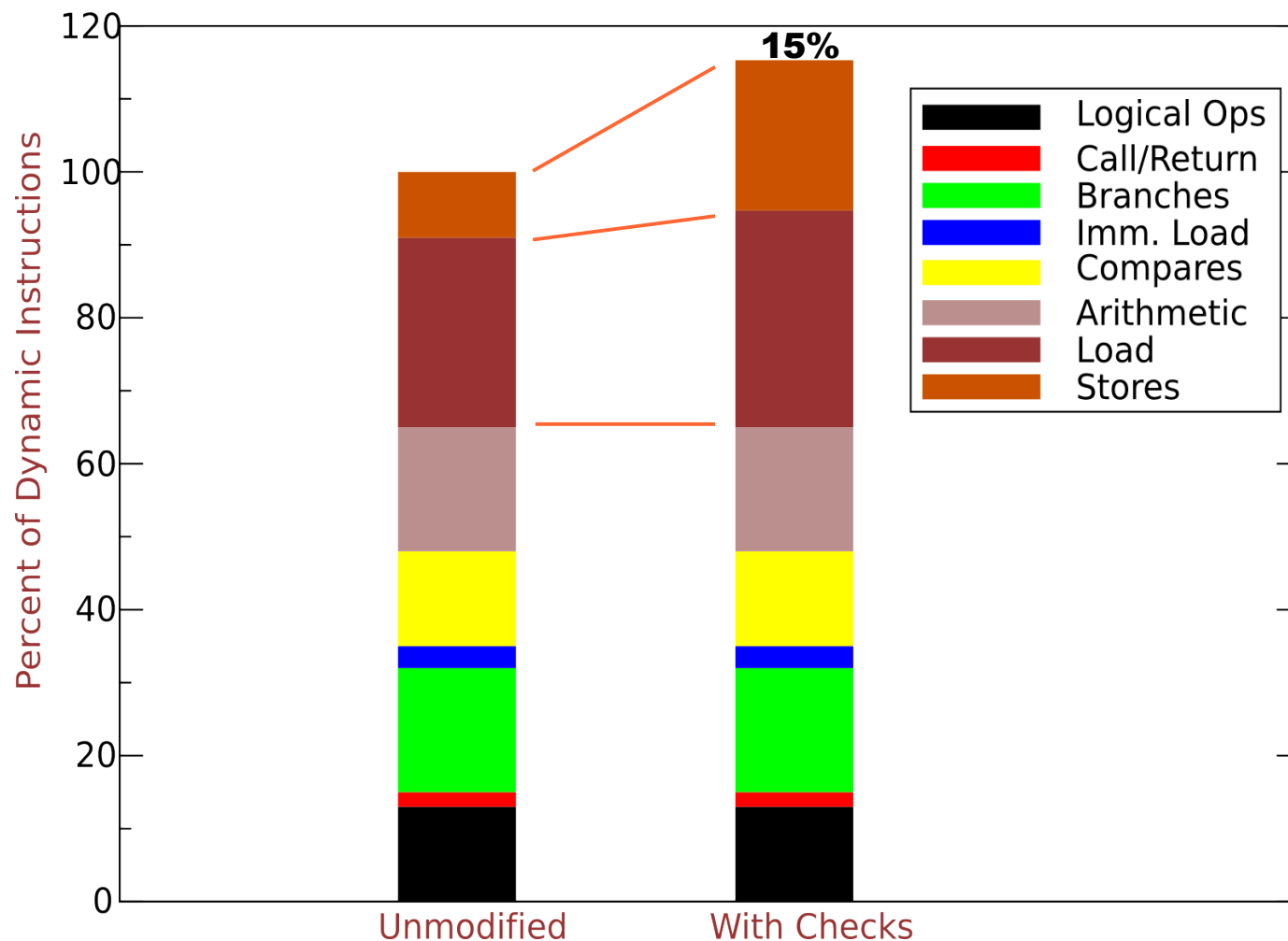
- Overflow sets O bit
- New data replaces LRU
- Old data spilled to DRAM

ST 1000, 55  
 XBEGIN L1  
 LD R1, 1000  
 ST 2000, 66  
**ST 3000, 77**  
 LD R1, 1000  
 XEND



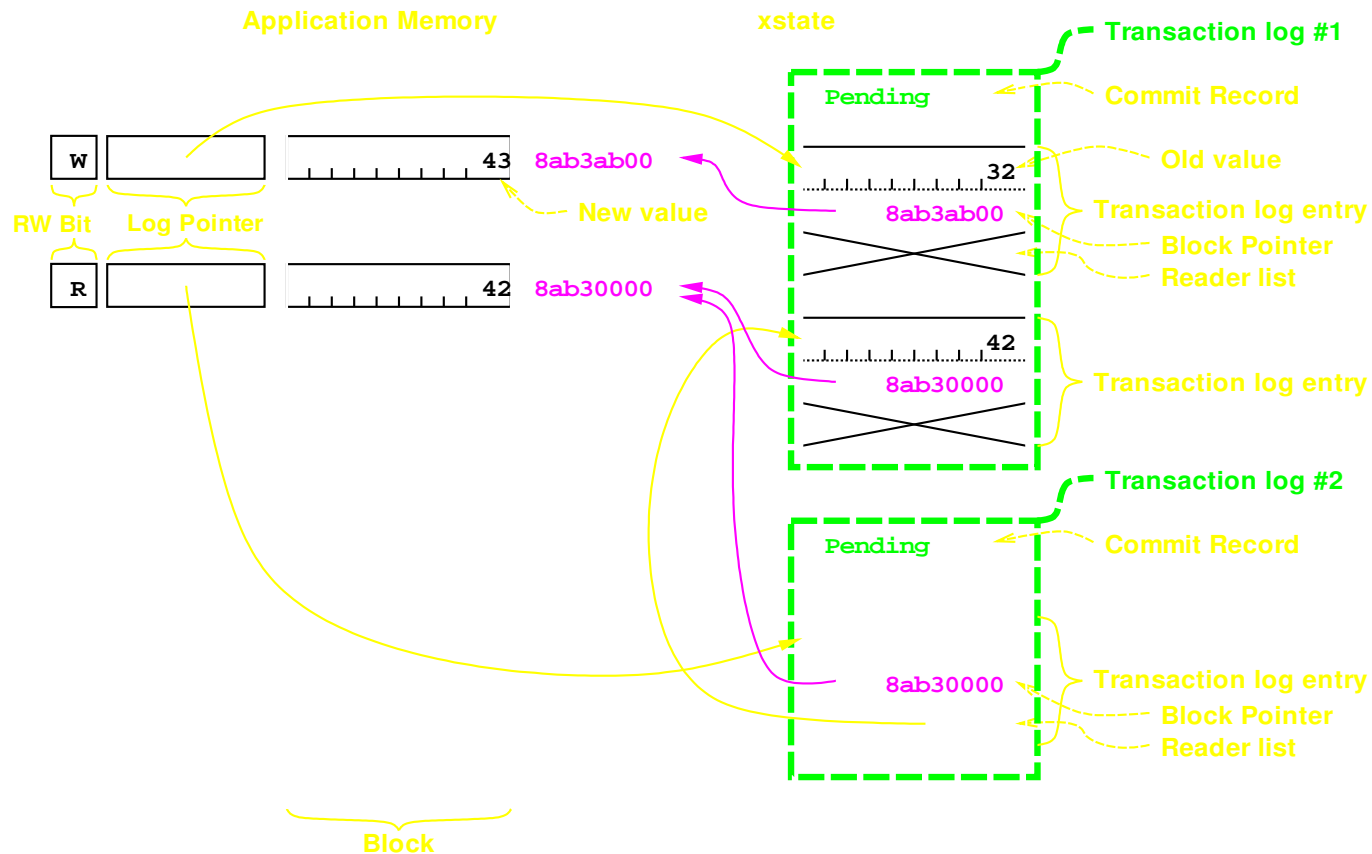
# Check Overhead

as a component of the overall instruction mix



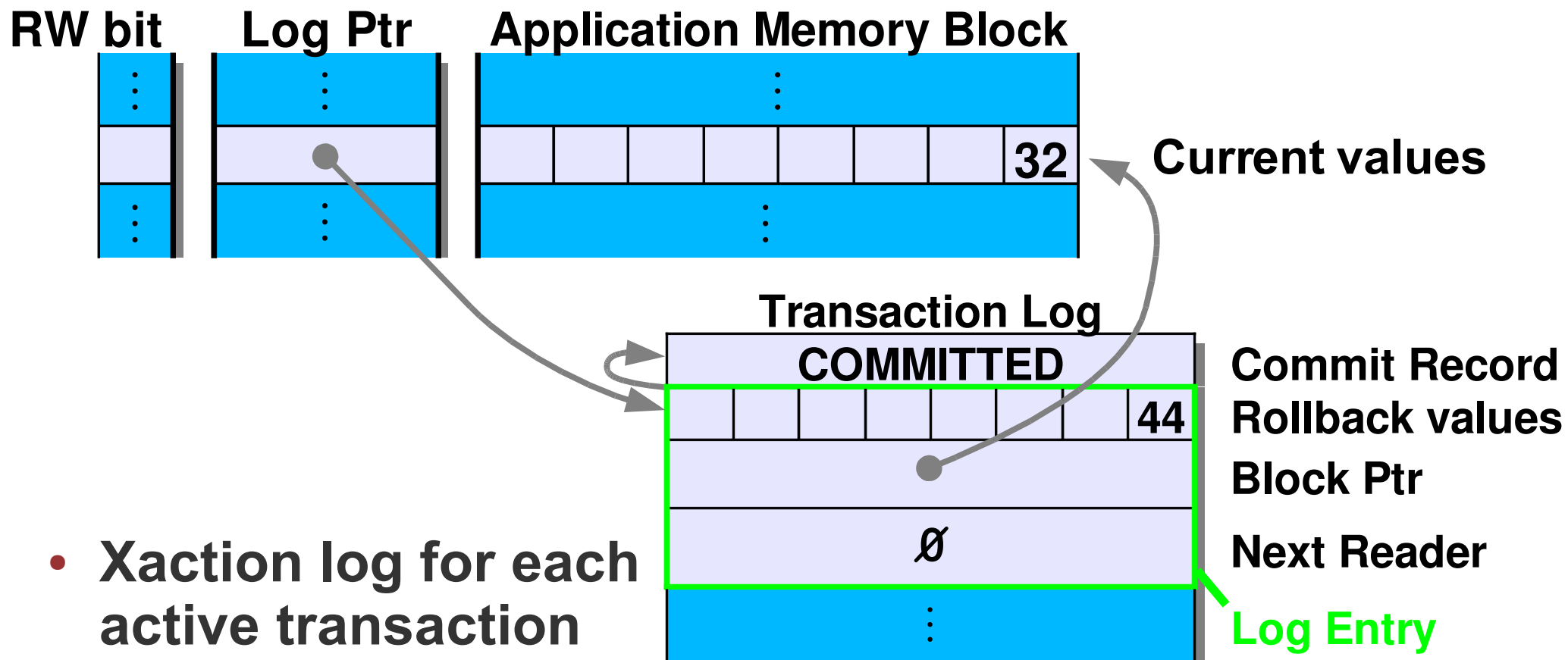
- **Back of the envelope calculation: 26% reads and 9% writes = 15% slowdown**

# xstate data structure



- **Xaction log for each active transaction**
  - **commit record:** PENDING, COMMITTED, ABORTED
  - **vector of log entries** w/ “old” values
    - each corresponds to a block in main memory
- **Log ptr and RW bit for each memory block**
  - **linked list of entries for each block**

# xstate data structure



- Xaction log for each active transaction
  - **commit record:** PENDING, COMMITTED, ABORTED
  - vector of **log entries** w/ “rollback” values
    - each corresponds to a block in main memory
- Log ptr and **RW bit** for each memory block
  - **Log ptr/next reader** form linked list of all log entries for a given block