

# *Efficient Object-Based Software Transactions*

**C. Scott Ananian and Martin Rinard**

**Computer Science and Artificial Intelligence Laboratory  
Massachusetts Institute of Technology  
{cananian, rinard}@csail.mit.edu**

**SCOOD 2005**

**This research supported by DARPA/AFRL Contract F33615-00-C-1692.**

# 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

# *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**
- Performance benefits for long-running transactions
  - Pay cloning costs up-front, then **run at full-speed** in own copy of the object graph

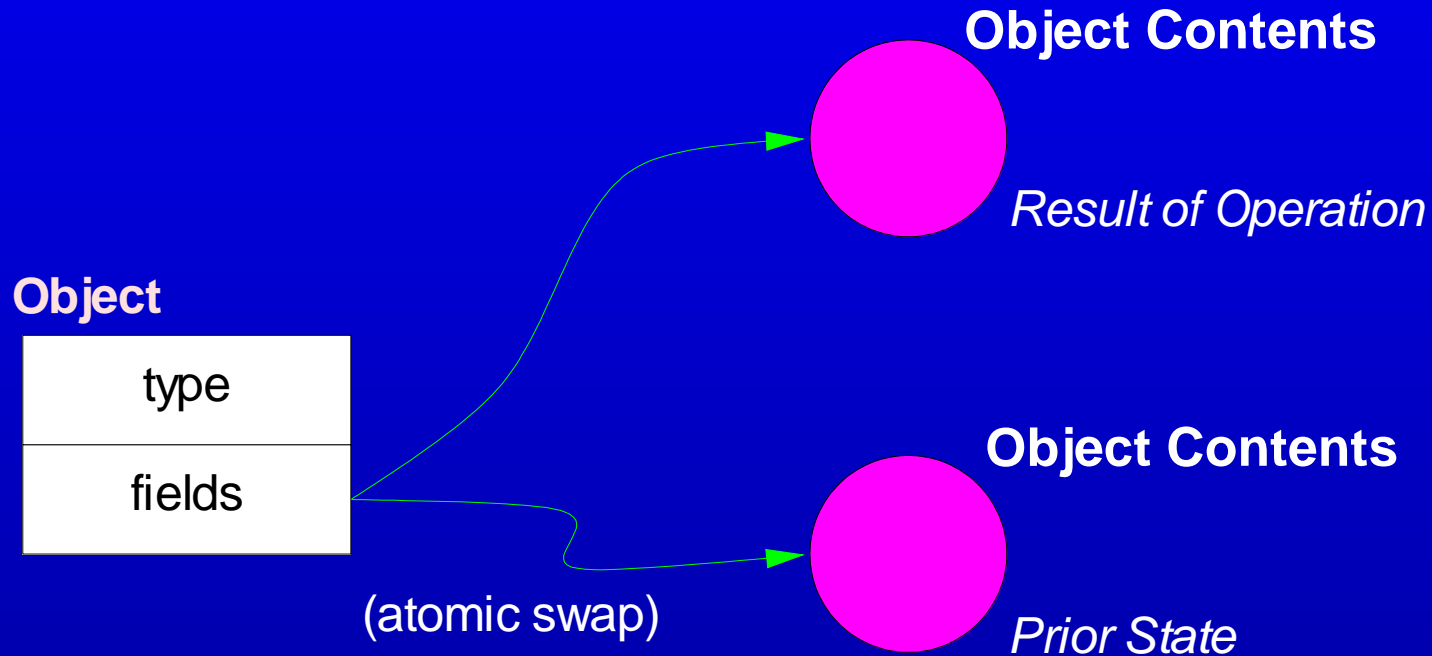
# *Three Big Ideas*

- **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

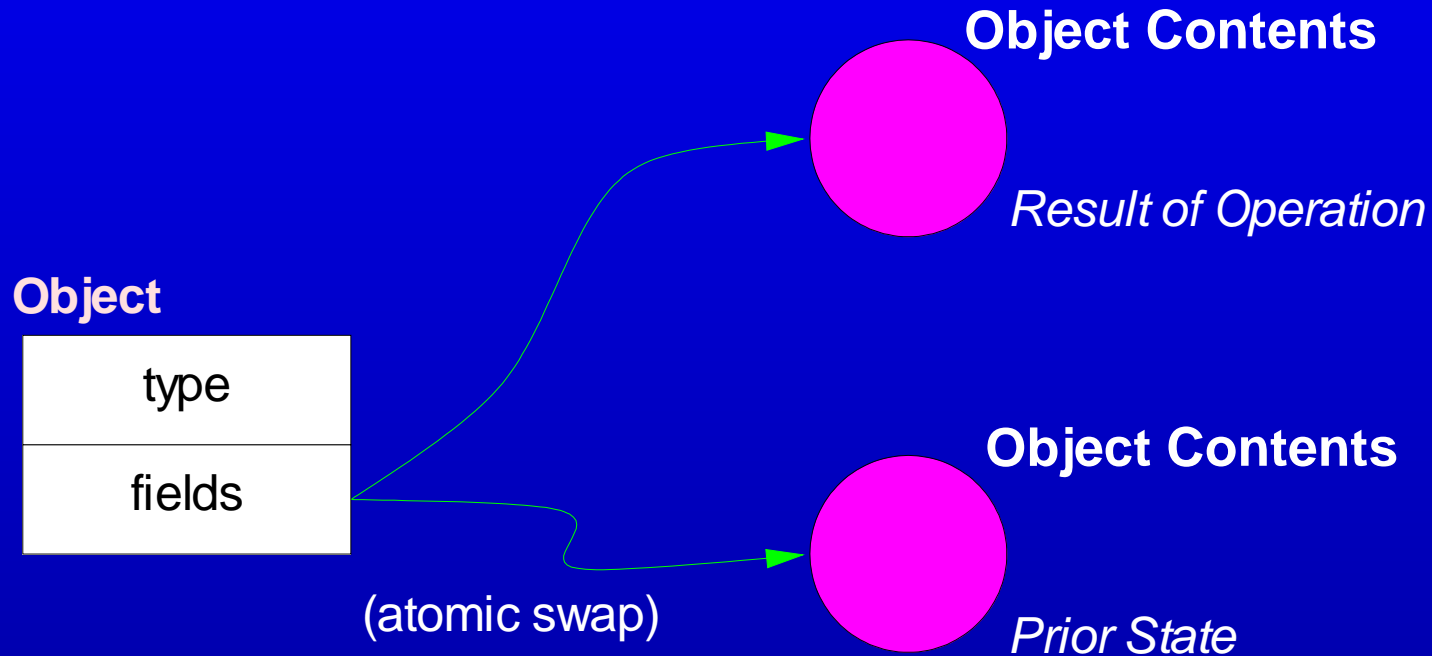
Valid for operations on a single object only.



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

# Single-Object Protocol

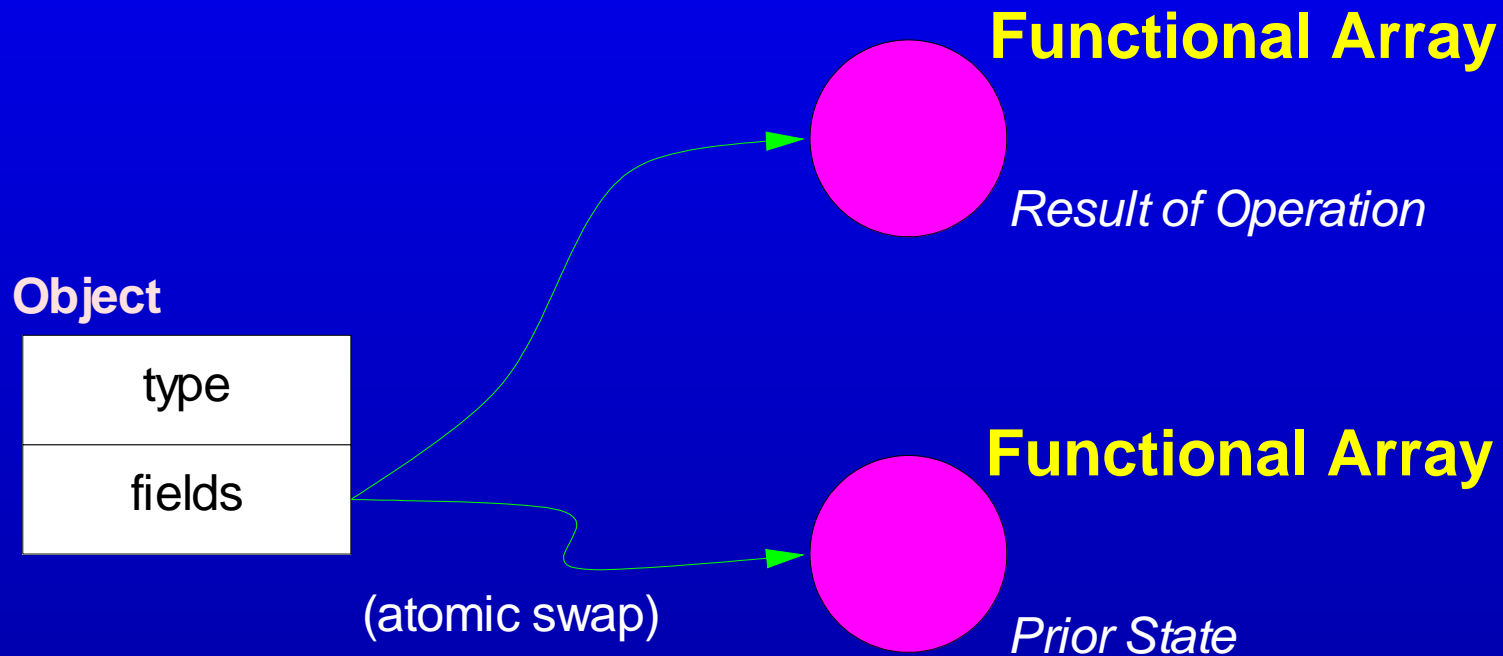
Valid for operations on a single object only.



- 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.

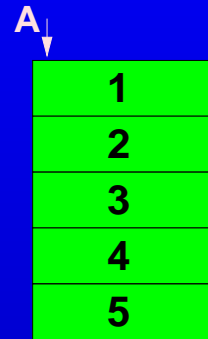


- **Large Object Problem:** cloning *prior state* for *result of operation* is  $O(\text{object size})$
- **Solution:** use a data structure where cloning is cheap –  $O(1)$  would be nice!

# Functional arrays

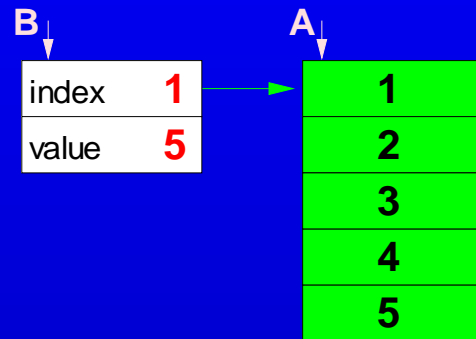
- Functional arrays are **persistent**: after an element is updated both the new and the old contents of the array are available for use.
- **Fundamental operation:**  
$$\text{Update}(A, i, v): A \rightarrow N_0 \rightarrow V \rightarrow A$$
- **Arrays are just mappings from integer to value; any persistent 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



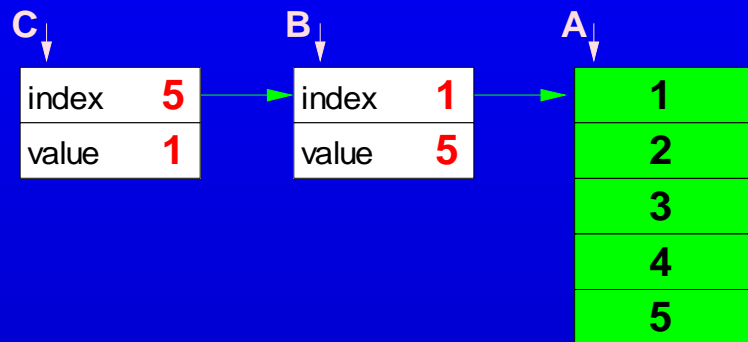
- A functional array is either a **cache node**...

# Functional Arrays using Shallow Binding



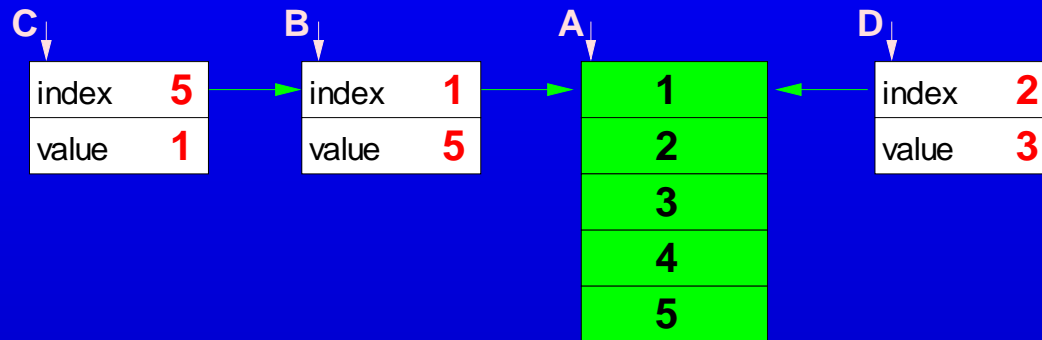
- A functional array is either a **cache node** or a **difference node**.
- $A[1]=1$  but  $B[1]=5$

# Functional Arrays using Shallow Binding



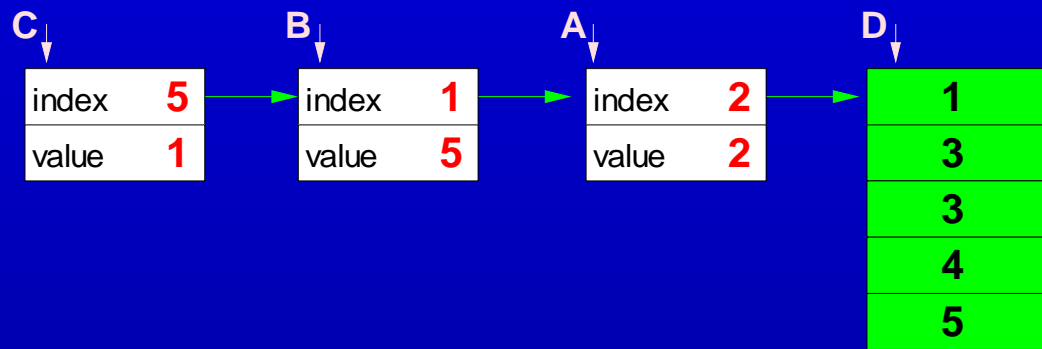
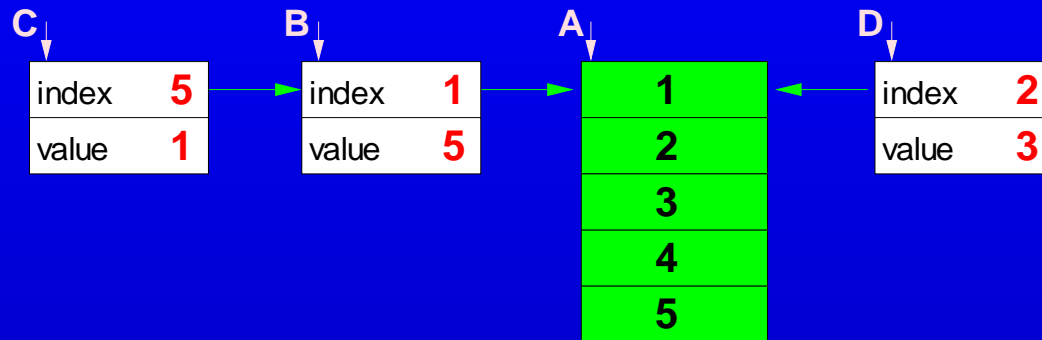
- Changing one element is  $O(1)$

# Functional Arrays using Shallow Binding



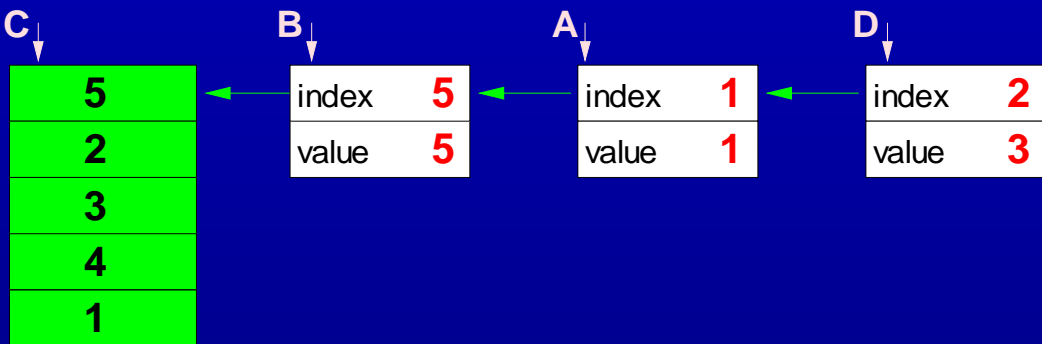
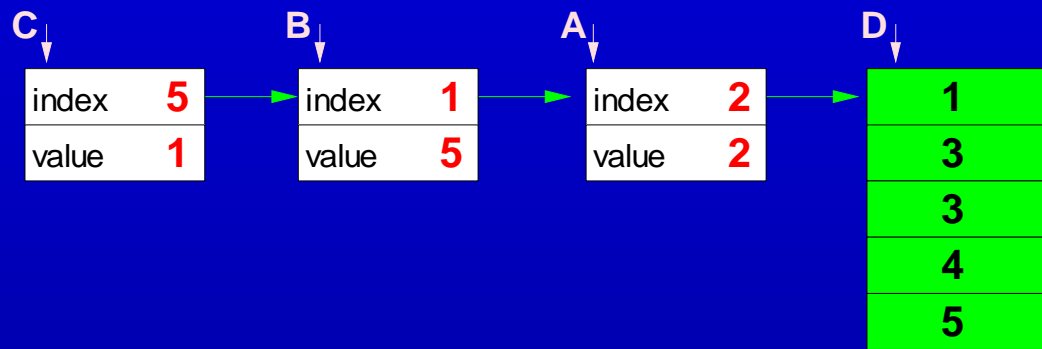
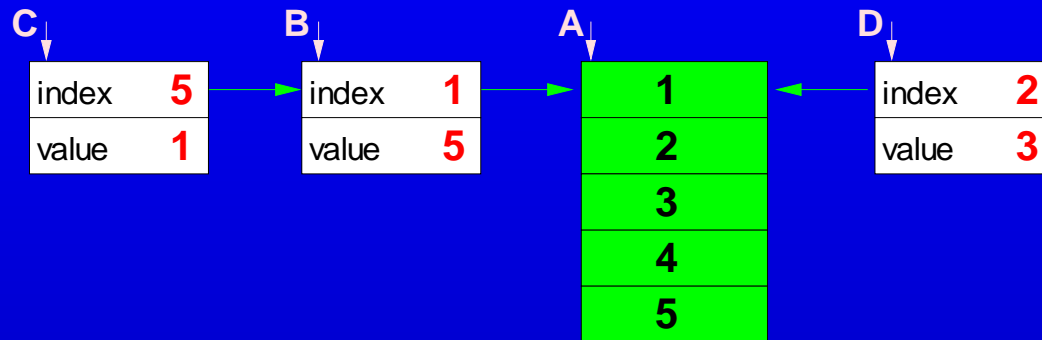
- $A[1] = D[1] = 1$        $C[1] = B[1] = 5$
- $C[5] = 1$        $D[2] = 3$

# Functional Arrays using Shallow Binding



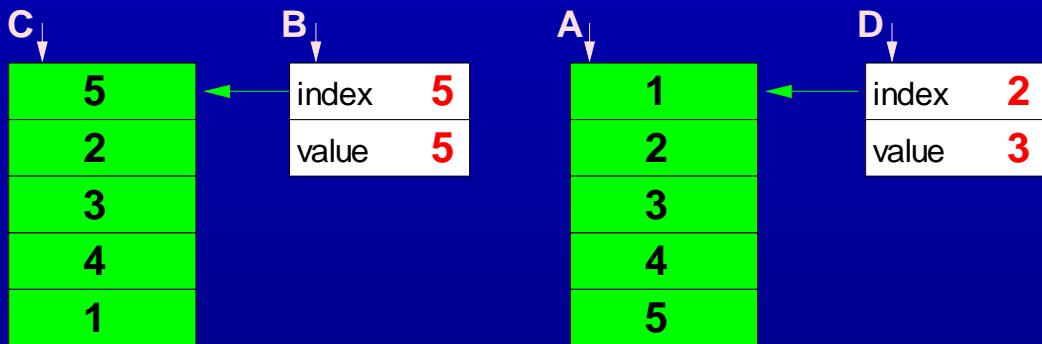
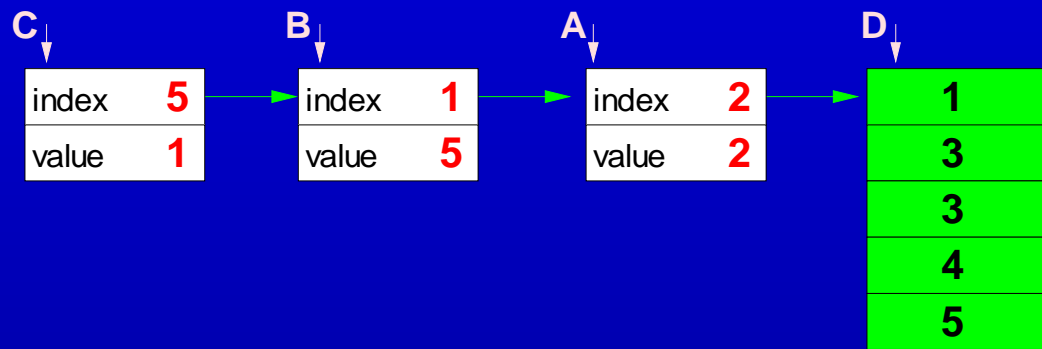
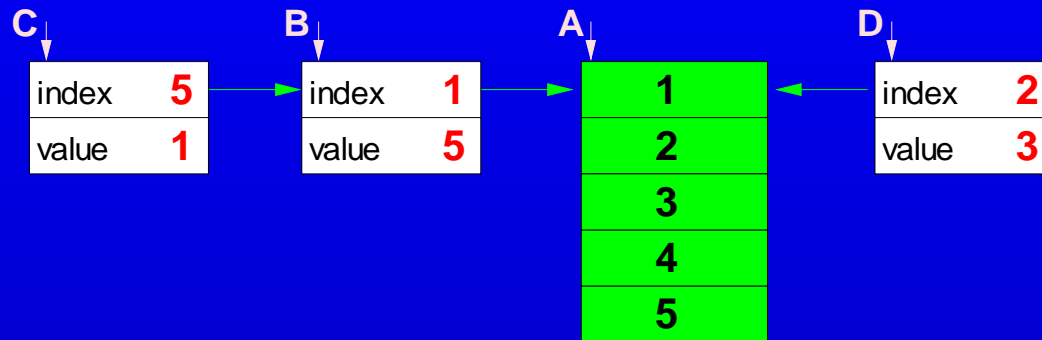
- We **rotate** the cache node on reads to keep access times fast.
- The bottom shows the graph after D is read.

# Functional Arrays using Shallow Binding



- **C is read.**
- **Ping-pong danger!**

# Functional Arrays using Shallow Binding



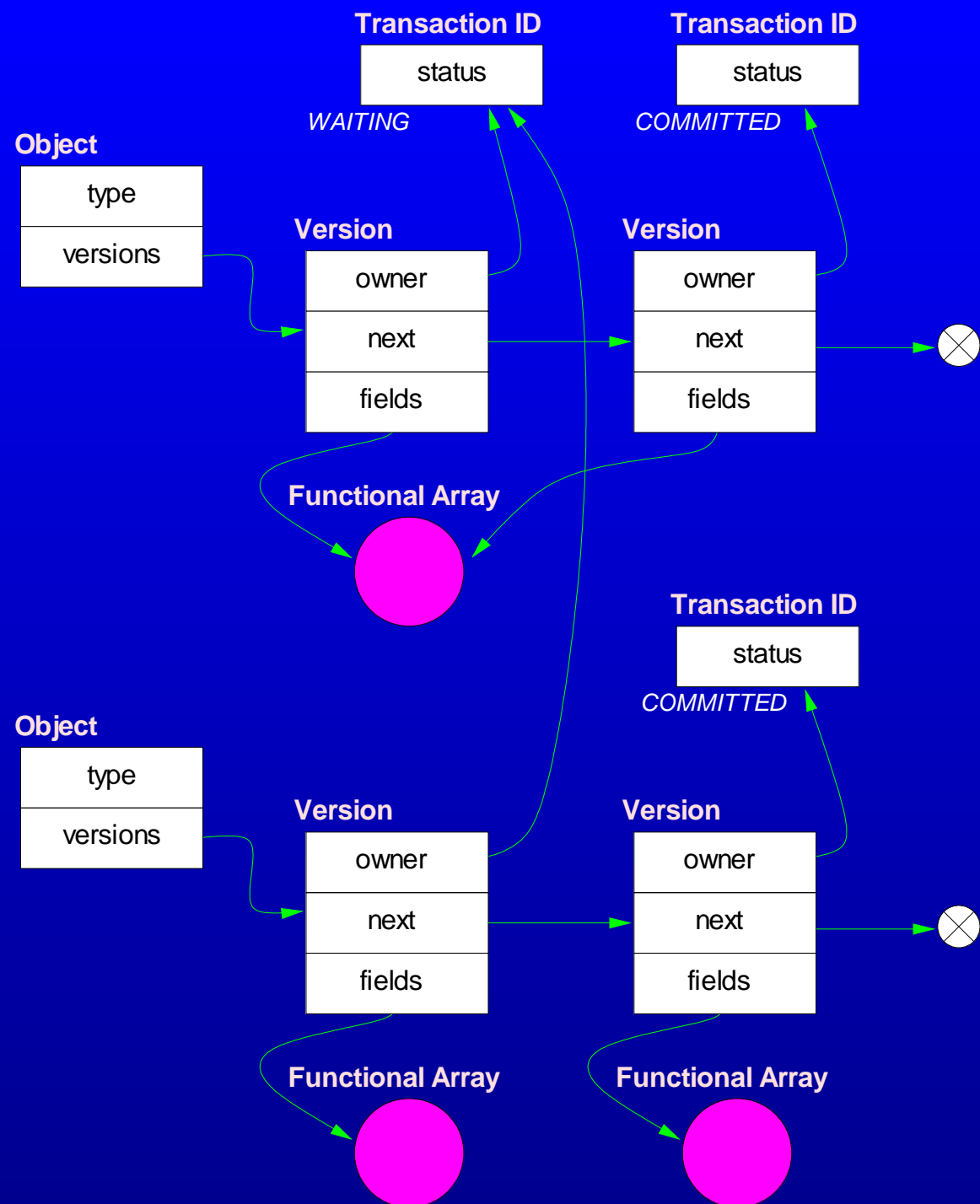
- **Split** with  $1/N$  chance.

# *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.

# Multiple-Object Protocol

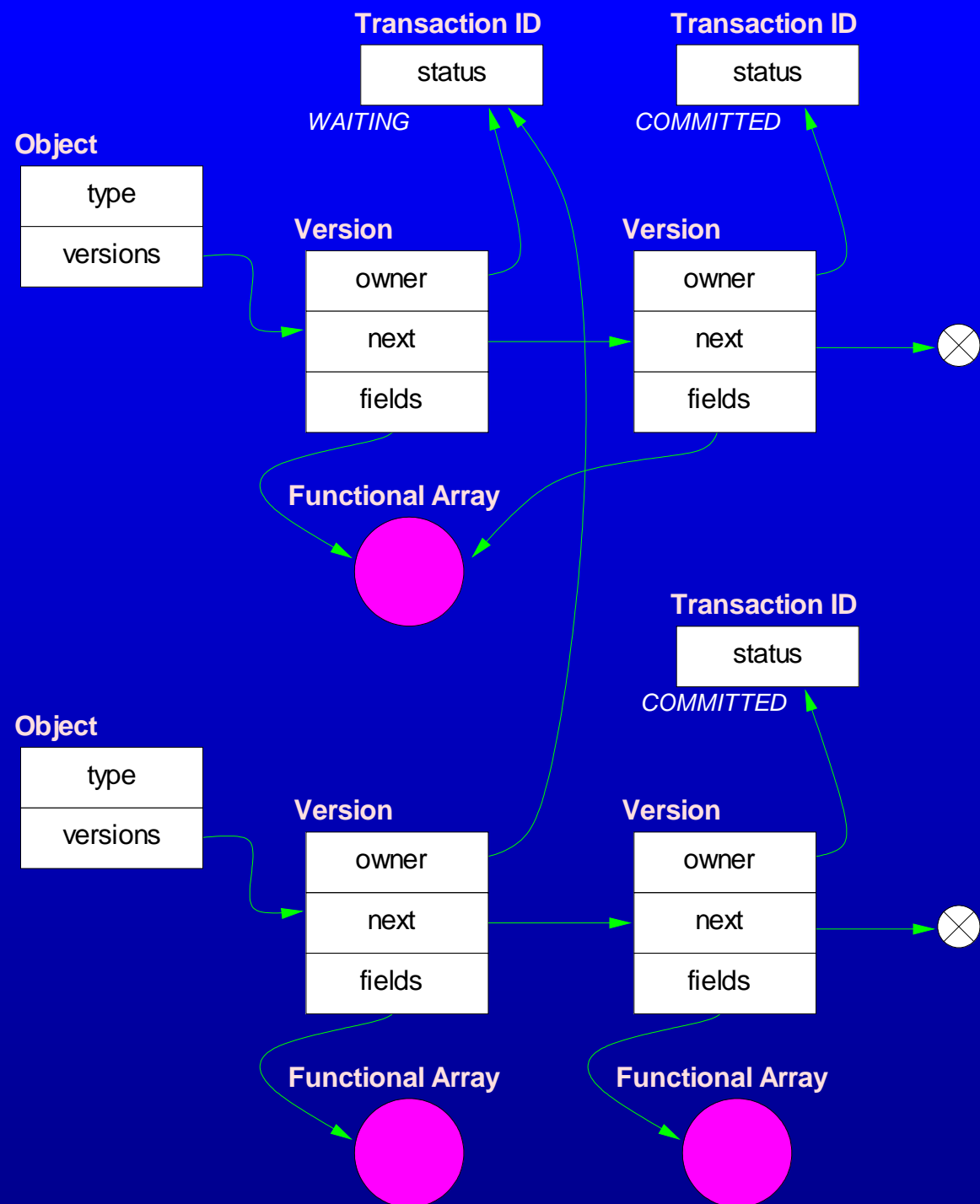
- Objects point to lists of versions.
- Each version has an associated **Transaction ID** and field array reference.
- Transaction IDs are initialized to **WAITING** and are **changed exactly once** to **COMMITTED** or **ABORTED**.





# Multiple-Object Protocol

- 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**.



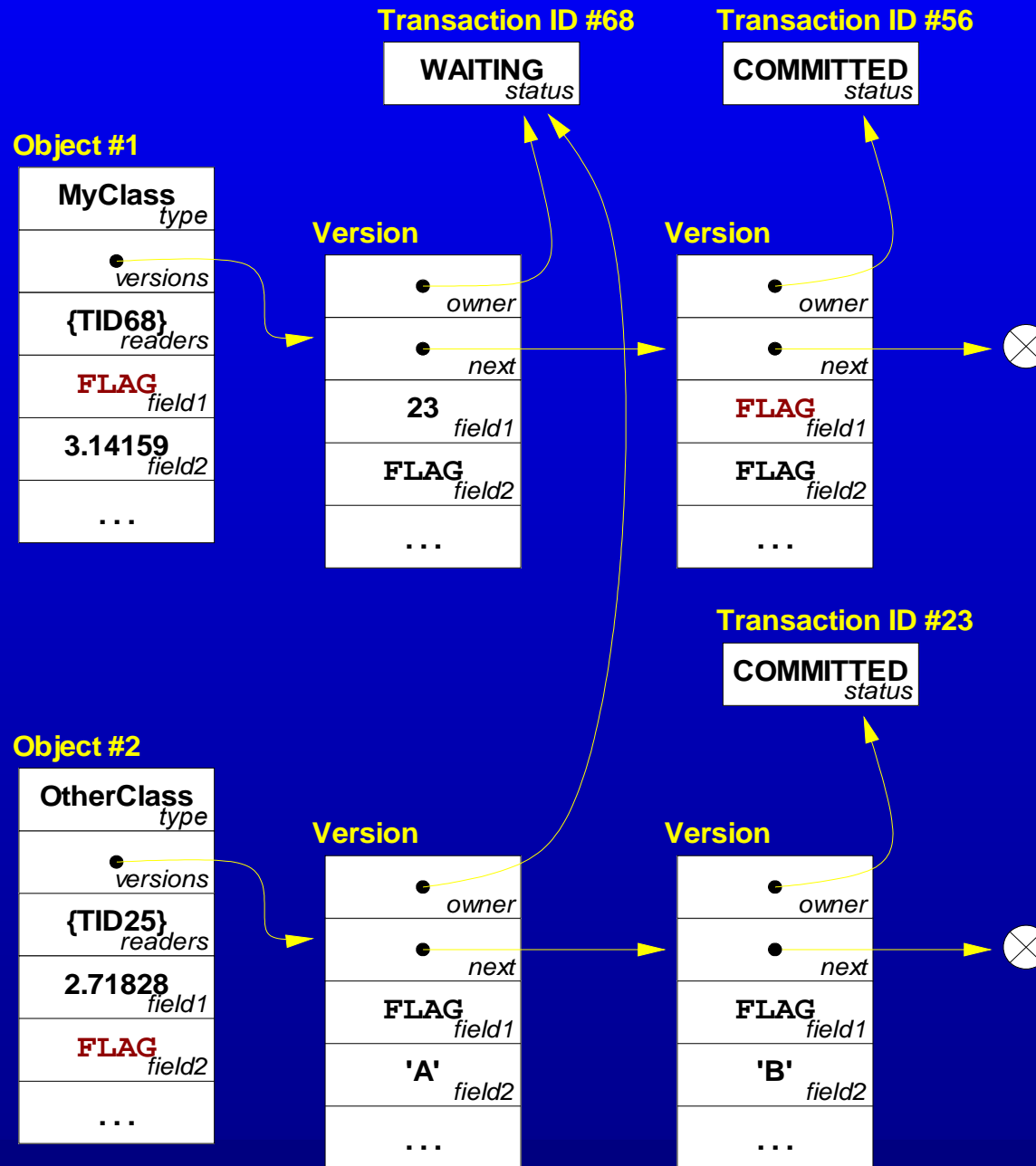
# *Making things practical: Things to keep in mind*

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

# Software Transaction Implementation

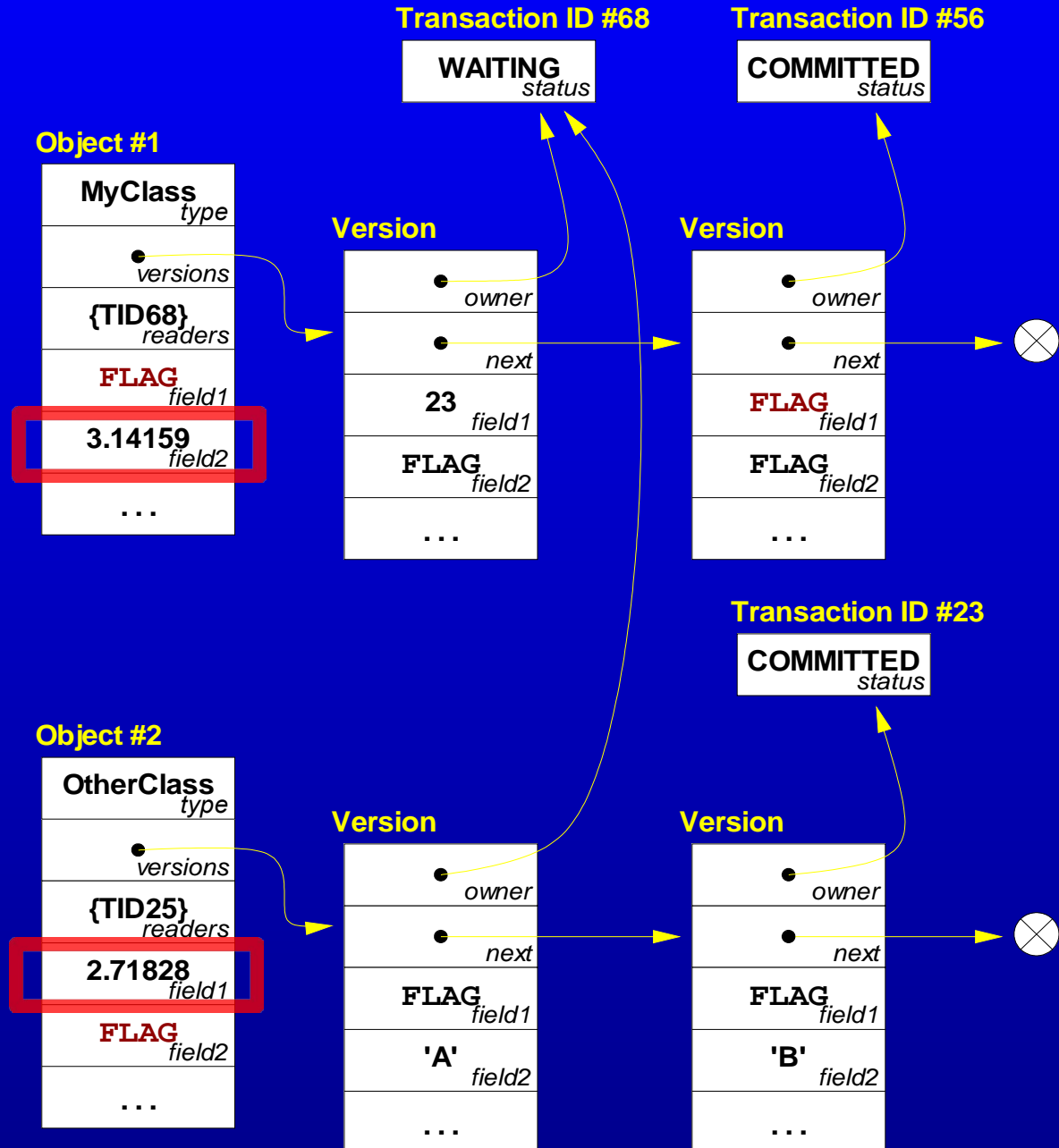
- **Goals:**
  - Non-transactional operations should be fast
  - Reads should be faster than writes
  - Minimal amount of object bloat
- **Solution:**
  - Use special `FLAG` value to indicate “location involved in a transaction”
  - Object points to a linked list of **versions**, containing values written 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”**)

# Transactions using version lists



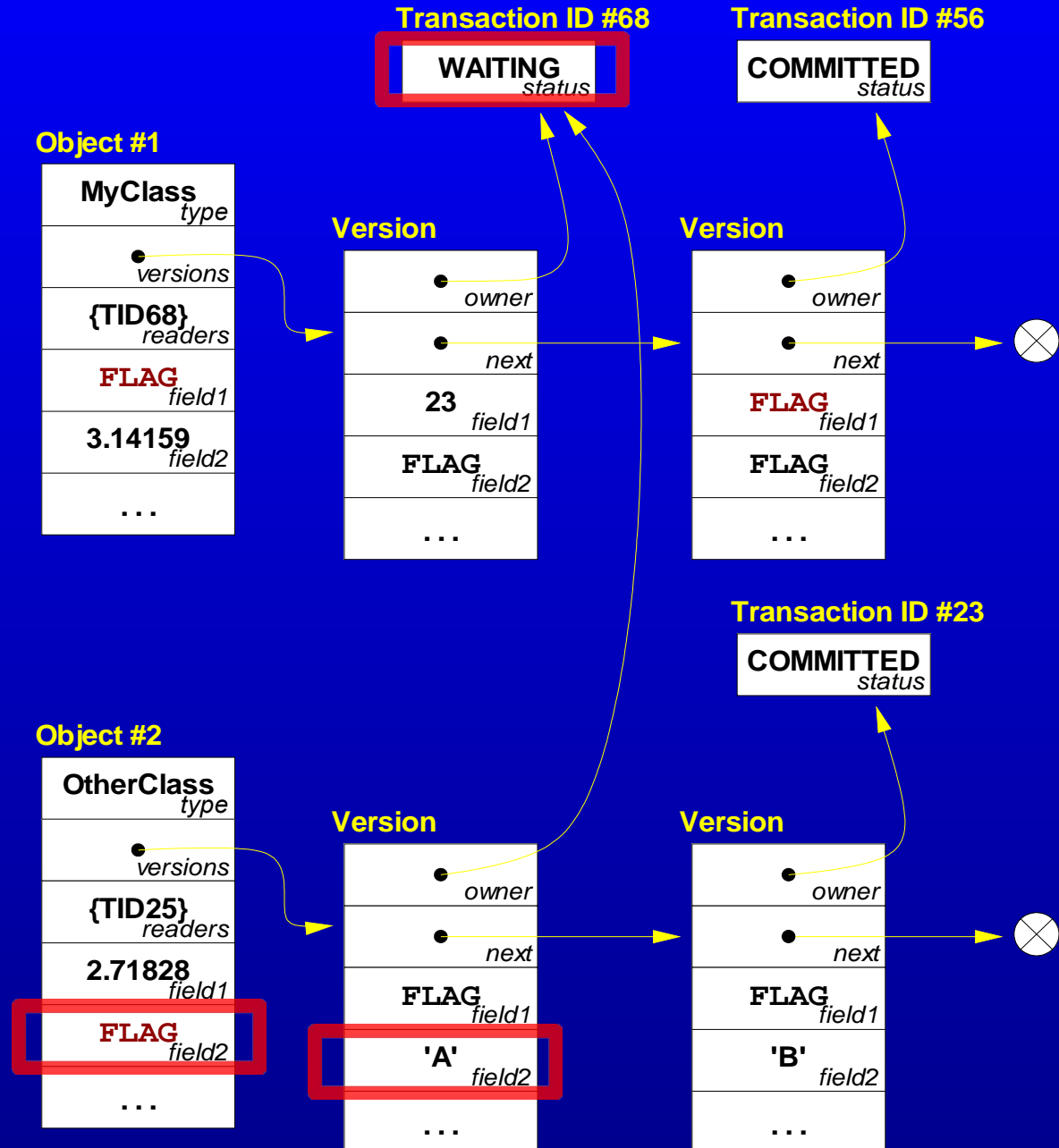
# Non-transactional Read (ReadNT)

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



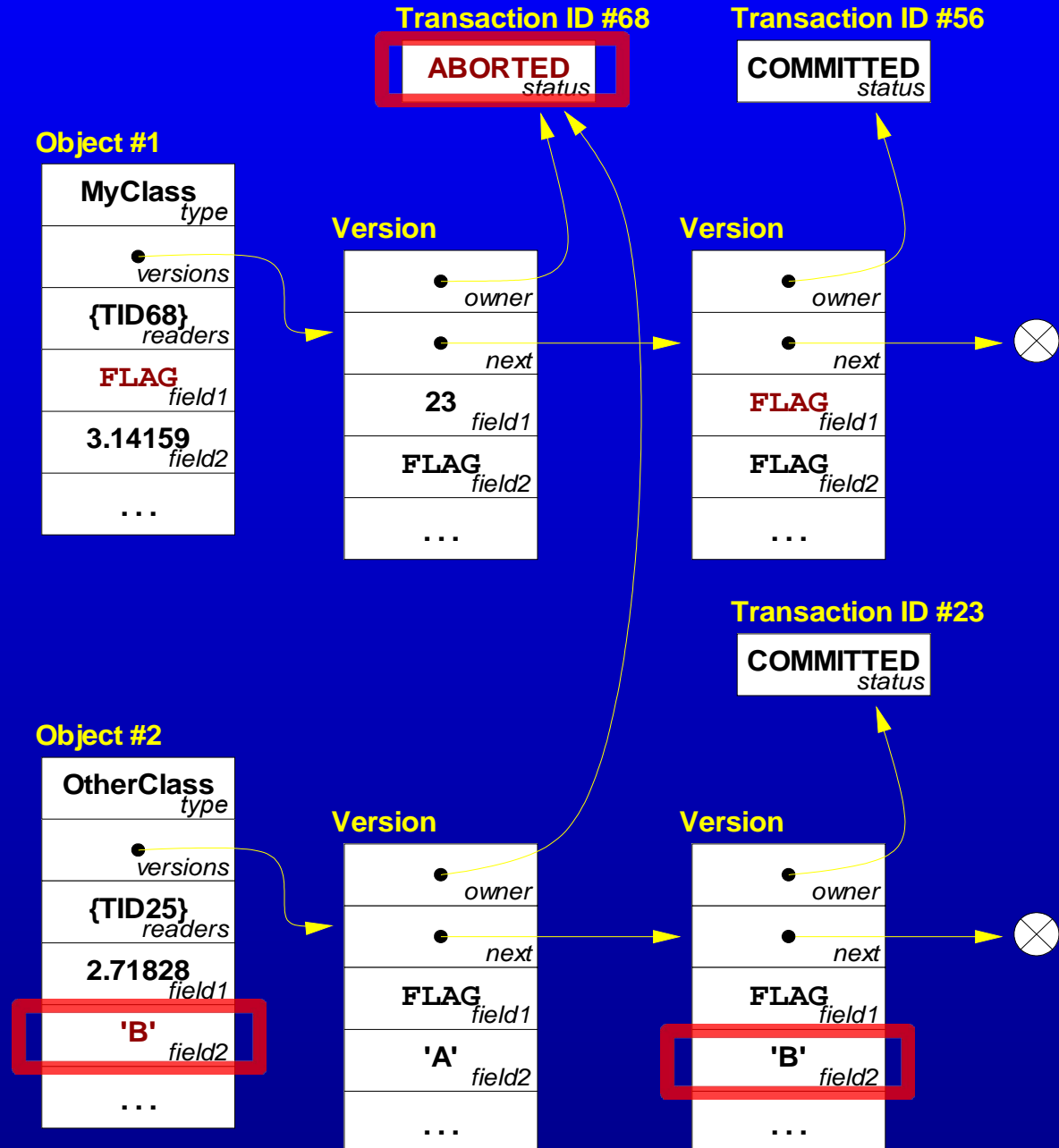
# Non-transactional Read (ReadNT)

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



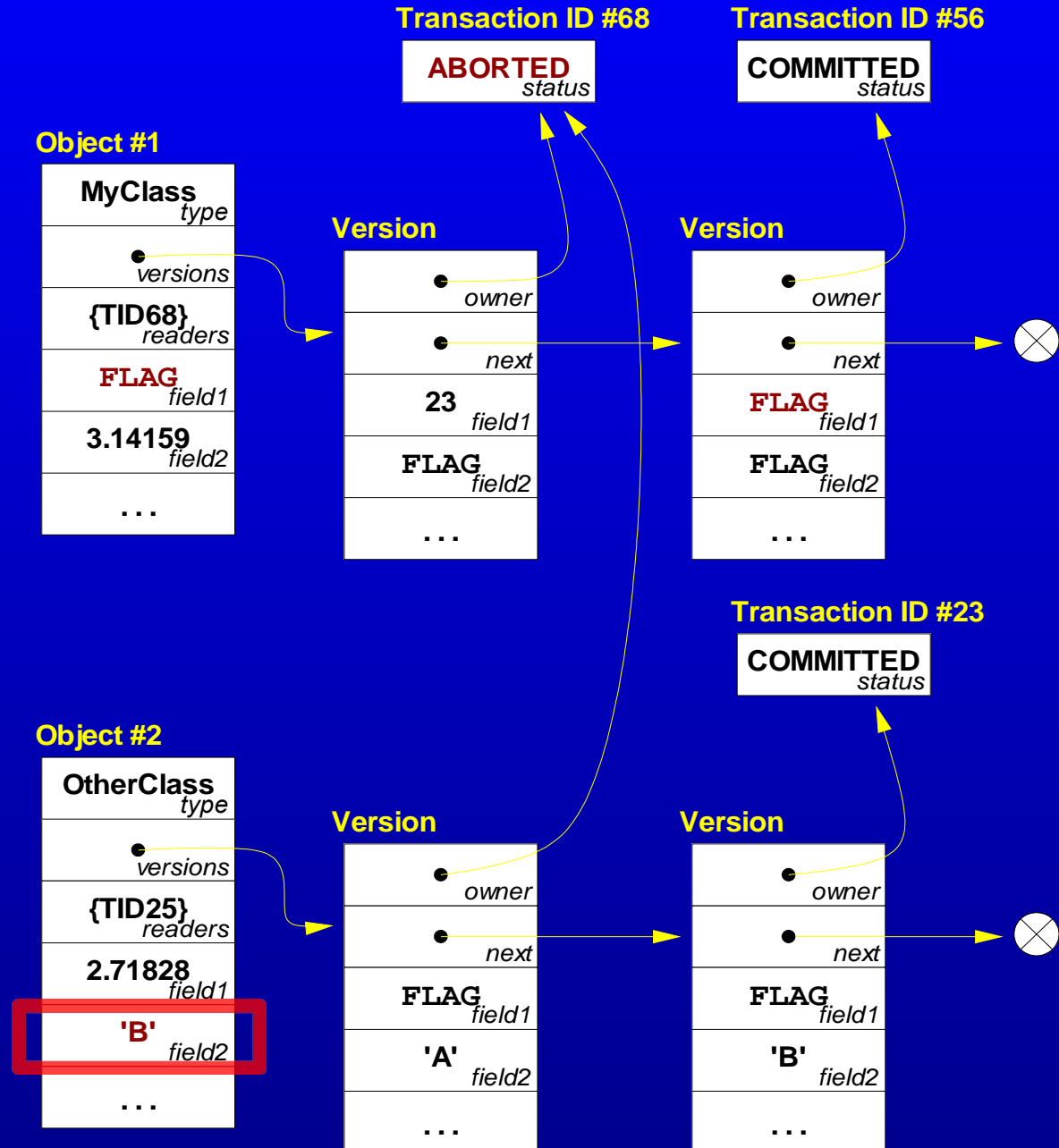
# Non-transactional Read (ReadNT)

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



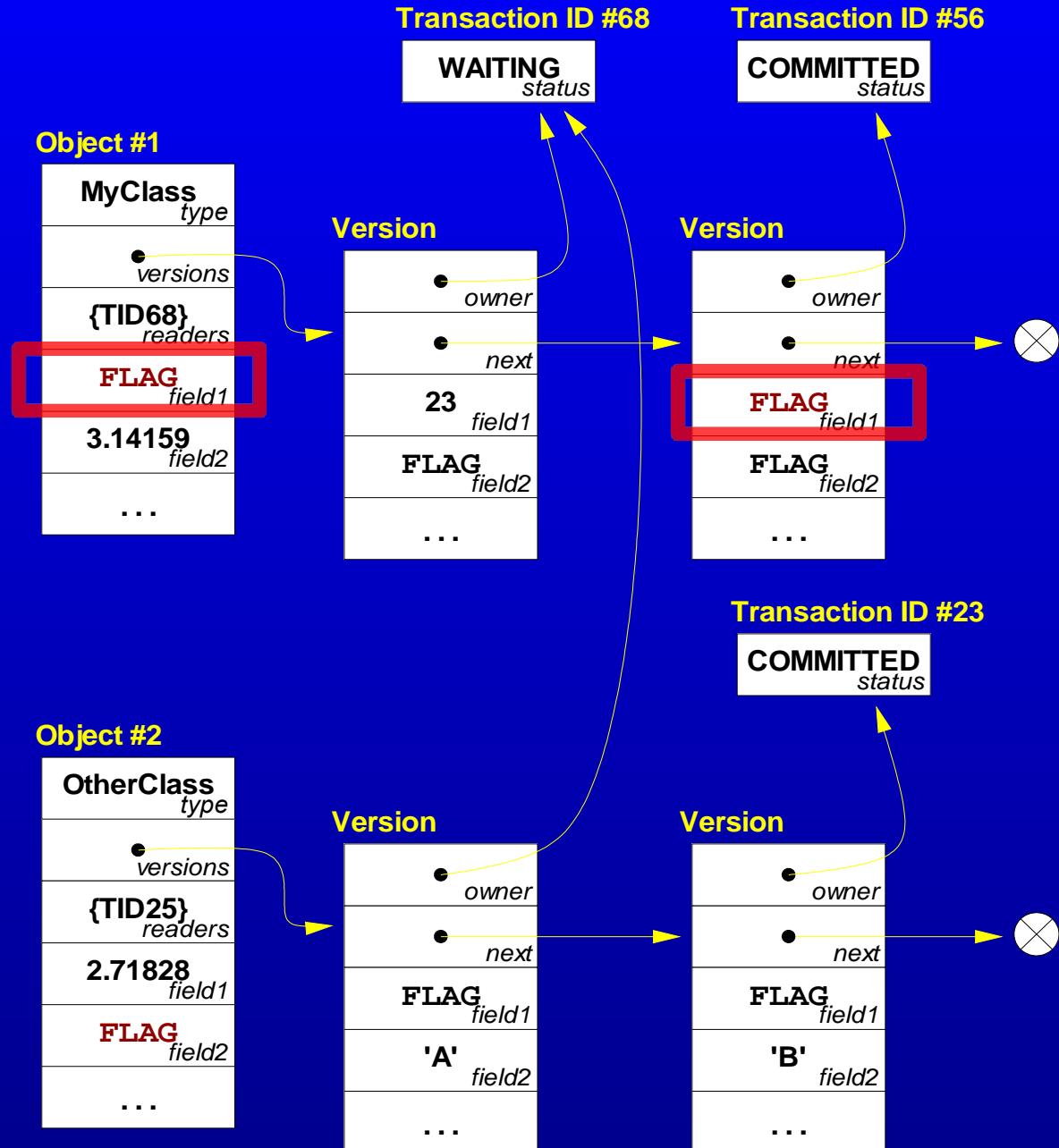
# Non-transactional Read (ReadNT)

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



# Non-transactional Read (ReadNT)

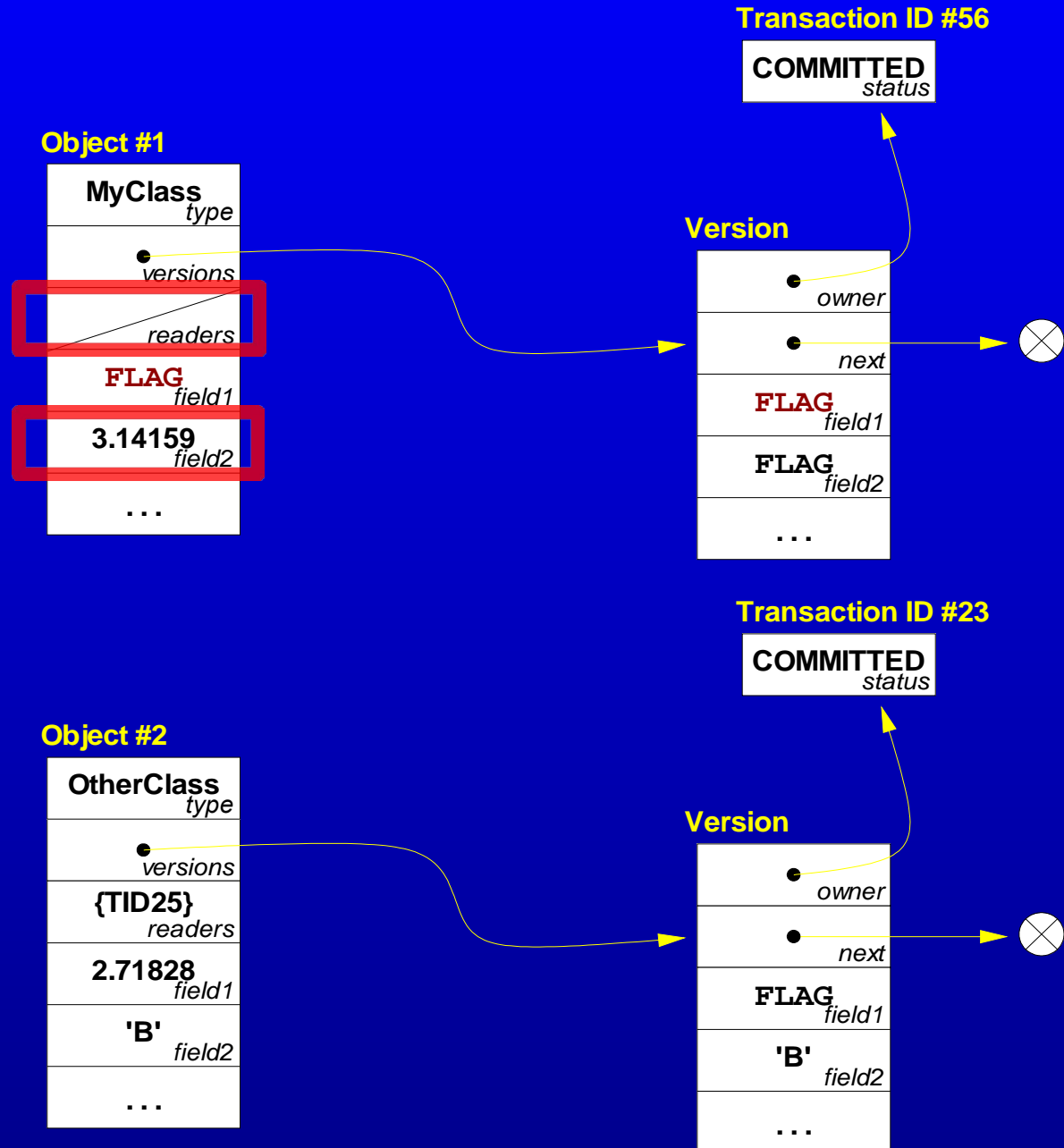
- Begins with a normal read of the field...
- “False flags” are discovered during copy-back; the read value is FLAG in this case.



# 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)

- If unsuccessful

- kill readers and writers
- repeat

Object #1



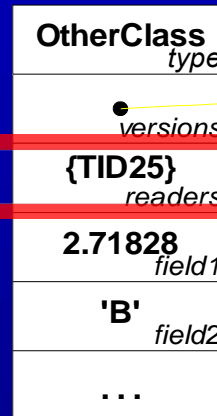
Version



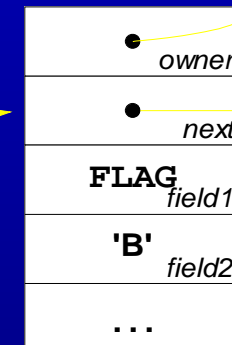
Transaction ID #56

**COMMITTED**  
*status*

Object #2



Version

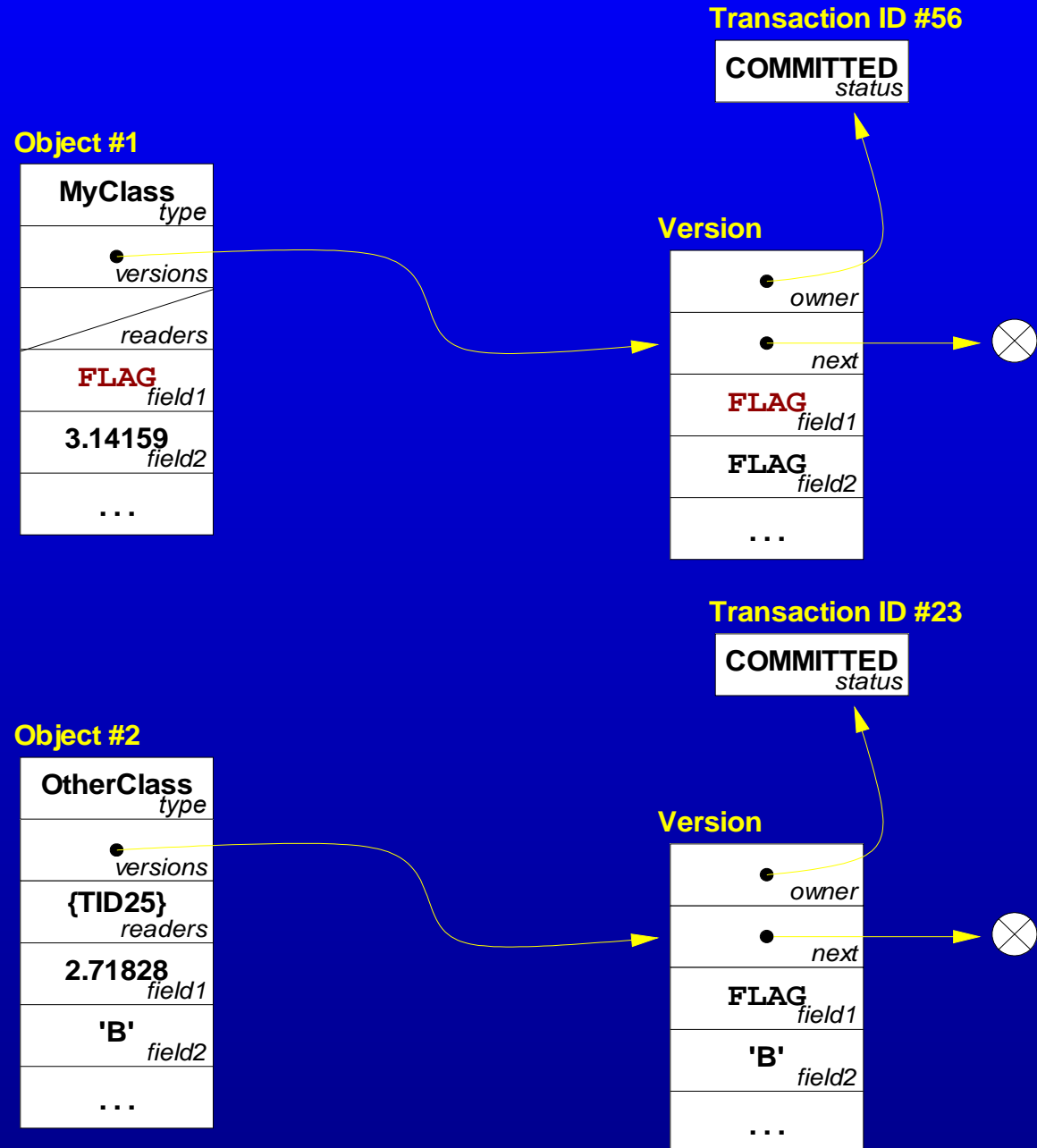


Transaction ID #23

**COMMITTED**  
*status*

# Non-transactional Write (WriteNT)

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



# Transactional Write (WriteT)

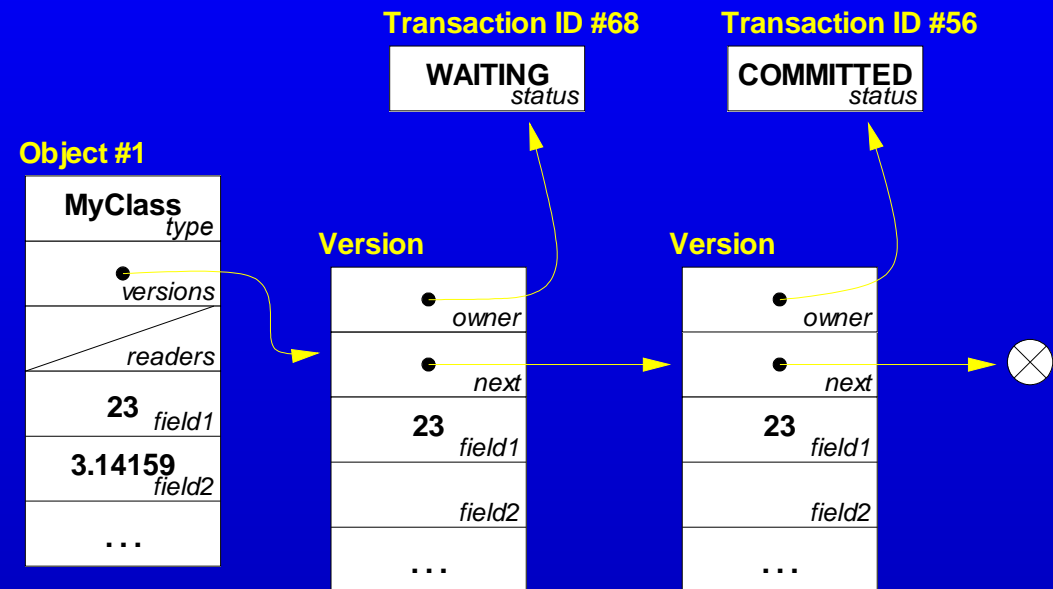
- **Once** per object written in this transaction:
  - find writable version
  - create (by cloning) if necessary
- Analysis and rewriting can offer big wins

Object #1

<b>MyClass</b> <i>type</i>
<i>versions</i>
<i>readers</i>
<b>23</b> <i>field1</i>
<b>3.14159</b> <i>field2</i>
...

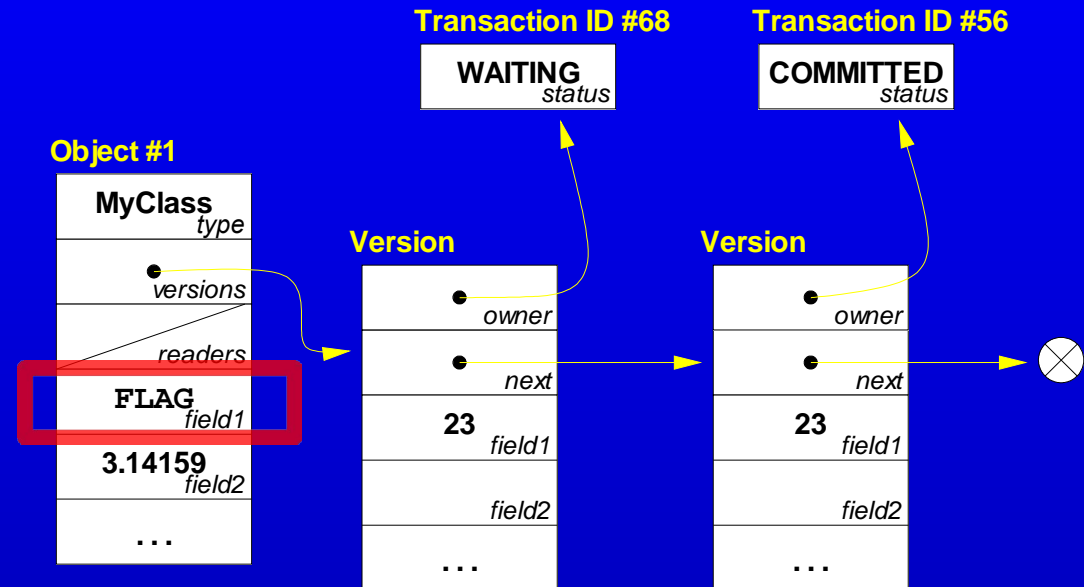
# Transactional Write (WriteT)

- **Once** per object written in this transaction:
  - find writable version
  - create (by cloning) if necessary
- Analysis and rewriting can offer big wins



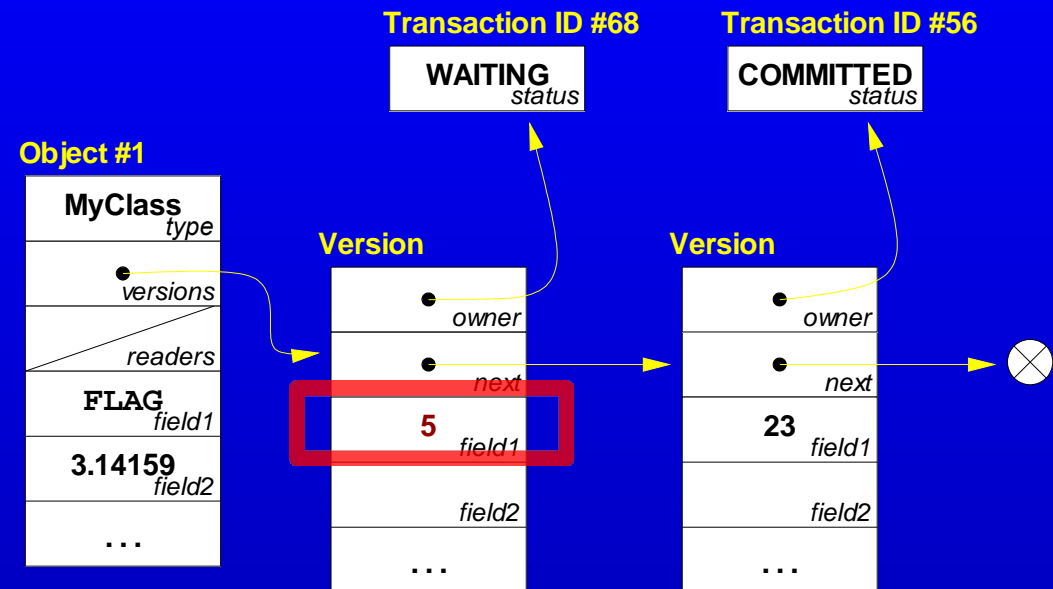
# Transactional Write (WriteT)

- **Once** per object written in this transaction:
  - find writable version
  - create (by cloning) if necessary
- Analysis and rewriting can offer big wins



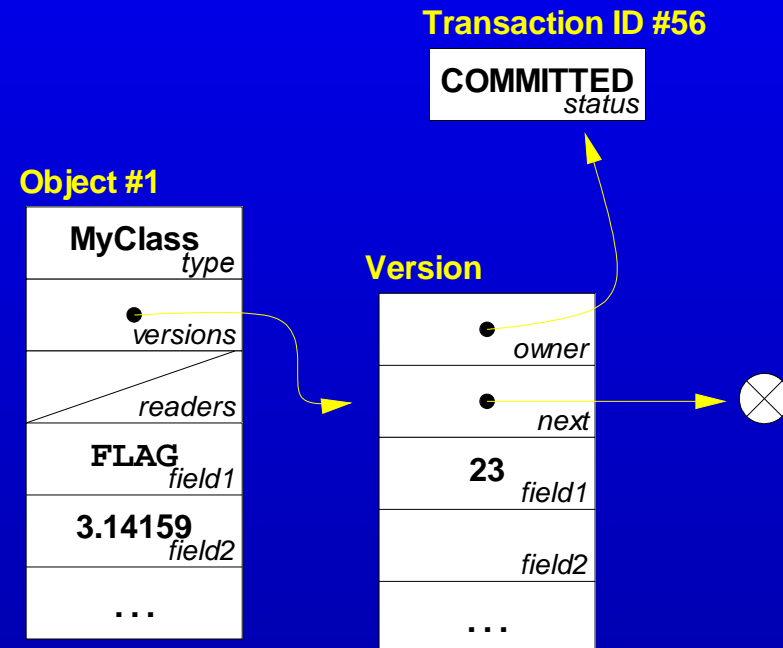
# Transactional Write (WriteT)

- **Once** per object written in this transaction:
  - find writable version
  - create (by cloning) if necessary
- Analysis and rewriting can offer big wins
- Then, **just write to the version.**



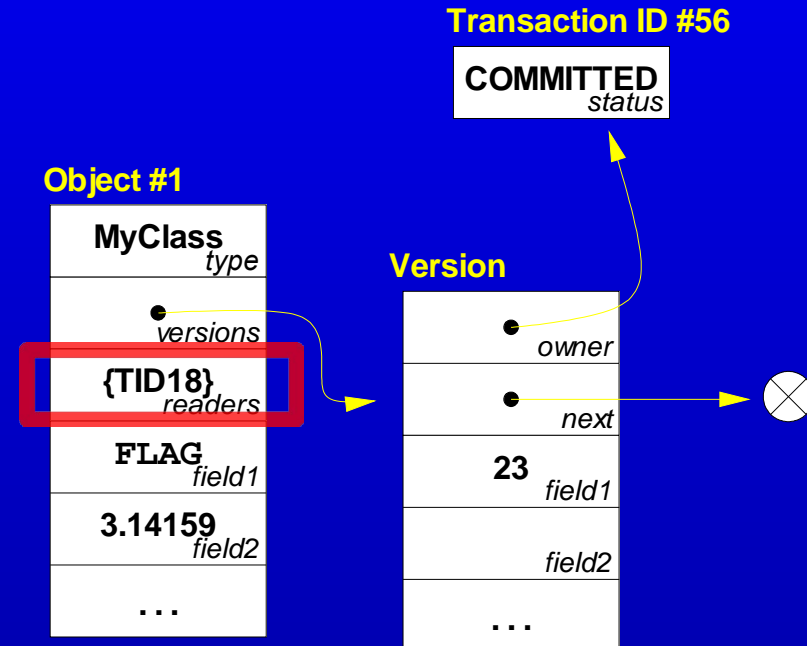
# Transactional Read (ReadT)

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



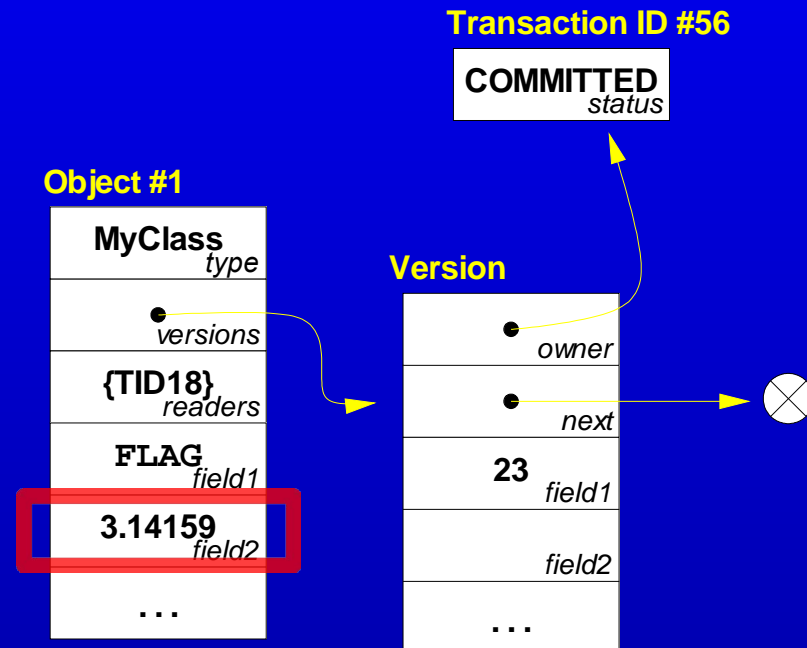
# Transactional Read (ReadT)

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



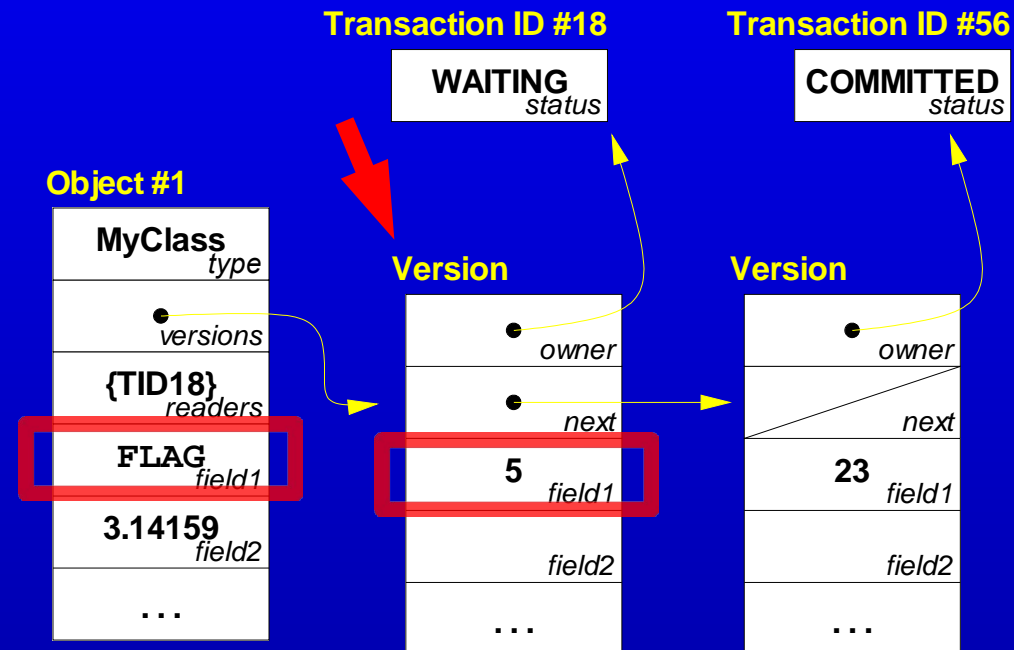
# Transactional Read (ReadT)

- **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
  - This gives only a few % slowdown
- **Once FLAGged, transactional code operates directly on the object's "version"**
- **Creating versions can be an issue for large arrays; use "functional array" techniques**

# *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!)

# *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.

# 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

# *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)

# Hybrid Hardware/Software Implementation

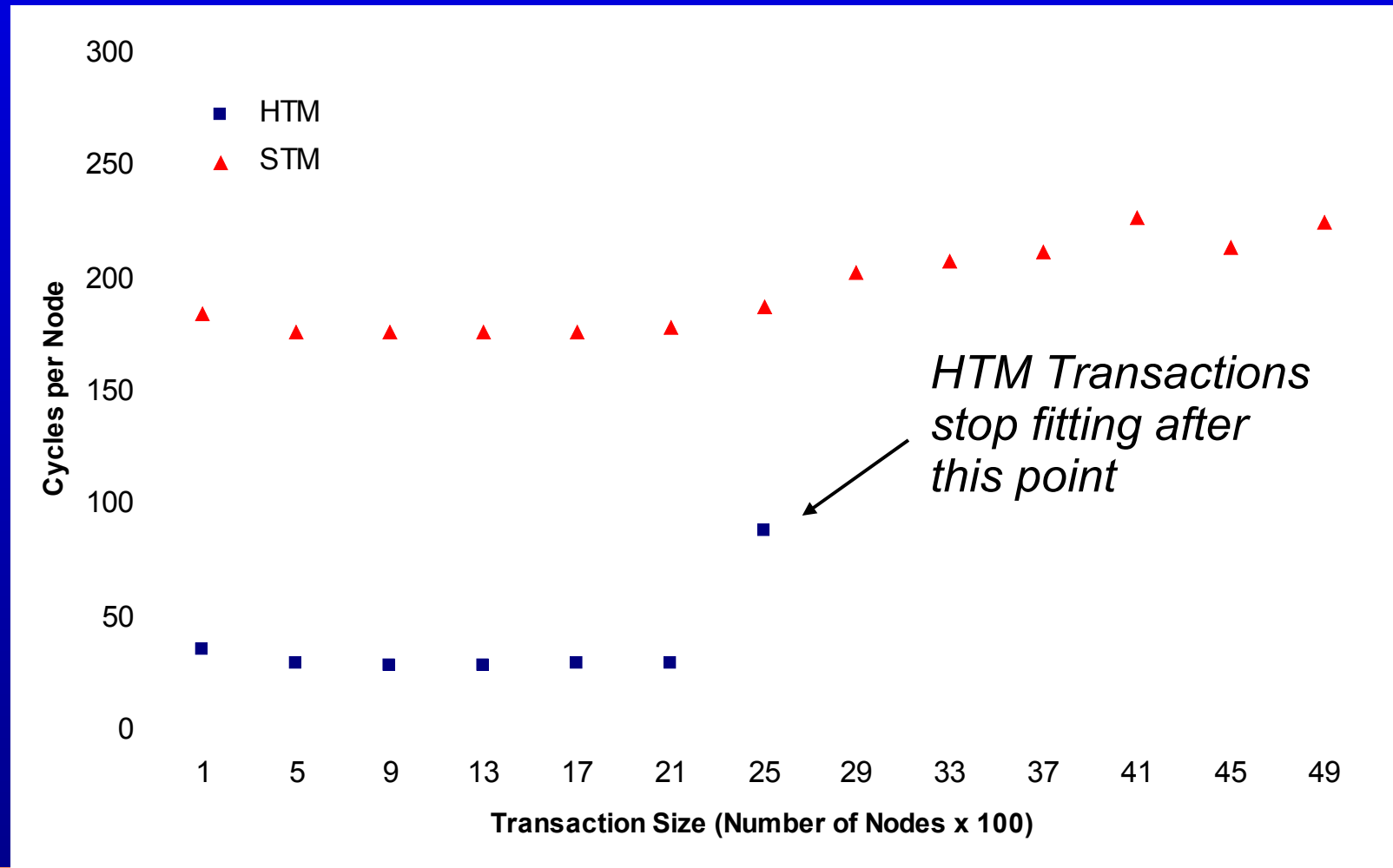
- Hardware transaction implementation is very fast!  
But it is limited:
  - Slow once you exceed cache capacity
  - Transaction lifetime limits (context switches)
  - Limited semantic flexibility (nesting, etc)
- Software transaction implementation is unlimited and very flexible!
  - But transactions 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 conflicting HTM
- HTM 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

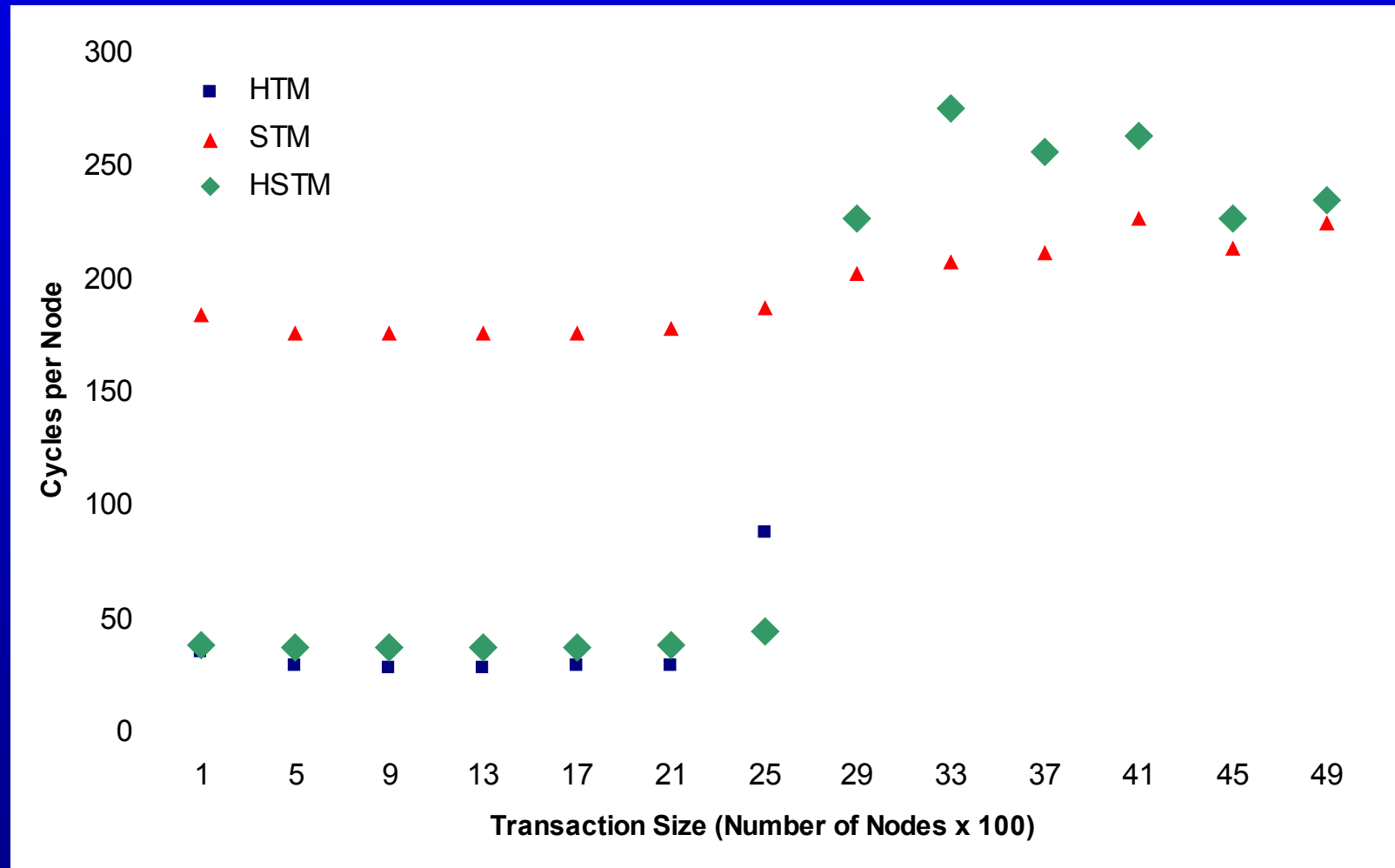
# Leveraging hardware for speed

- 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!**



# *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

***Thank you!***

***(p.s. I'm graduating soon!)***