McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime

B. Saha, A-R. Adl-Tabatabai, R. Hudson, C.C. Minh, B. Hertzberg

PPoPP 2006
Introductory TM Sales Pitch™

- “Two legs baaaad, four legs gooood”
Introductory TM Sales Pitch™

- Oops ! I meant: “Locks are baaaaaad, transactions are gooood”
- HTM is either limited, or very complicated (and the HW doesn’t exist anyway), so STM is the way to go for sure!
Overview

- **STM Design Tradeoffs**
- Variable granularity conflict detection
- Novel design (blocking, but leverages other McRT components – scheduler)
- Novel software MCAS implementation
- **Detailed performance results** on microbenchmarks and sendmail’s spam filter
STM Design Tradeoffs
STM Design Tradeoffs 1: Non-blocking guarantee

- Strict two-phase locking protocol
- Each memory location is mapped to a lock
  - All locks must be acquired before committing a transaction
- McRT-STM has blocking sequences
  - However, if T1 is waiting for T2 to release a lock, T1 may abort T2 in certain cases
STM Design Tradeoffs 1: Non-blocking guarantee

- Reasons to allow blocking:
  - Non-blocking properties are enforced through the scheduler, by using cooperative preemption
  - Scheduler runs at user level -> low overhead
  - Emerging multi-core architectures make preemption less of an issue
STM Design Tradeoffs 1: Non-blocking guarantee

- Benefits from allowing blocking:
  - Reduced number of transaction aborts
  - Lock-based implementation simpler and more efficient
  - Expose optimization opportunities (e.g., early detection that a Tx will commit)
  - Simplified memory management
  - Better integration with transaction monitors, and support for other features that may require locks (RPC)
STM Design Tradeoffs 1: Non-blocking guarantee

- **Deadlock avoidance**
  - One way: maintain a graph of waiting Txs and detect cycles
    - Con: lots of overhead!
  - Second way: wait a finite amount of time for a lock to be released, then abort
    - Con: may give false positives
    - Pro: very cheap: no conflicts -> no overhead!
STM Design Tradeoffs 1: Non-blocking guarantee

- Data conflicts and avoidance: in McRT-STM, data conflicts = lock contention
- Basic idea: try to maximize throughput by making readers/writers wait for a lock
  - T1 will abort T2 (which holds a lock that T1 is waiting for) only when T2 yields the CPU
    • This is done with the assistance of the McRT scheduler
STM Design Tradeoffs 2: Locking mechanism

- Reader-writer locking
- Readers need to acquire a read lock before reading a speculative value
- Writers need to acquire a write lock for a speculative update
- Multiple readers or a writer can acquire one lock at any time
STM Design Tradeoffs 2: Locking mechanism

- Acquiring a read lock prevents writers from updating
- Acquiring a write lock prevents readers from reading
- Problem: cannot use conventional reader-writer locks in STM
  - Because a reader Tx may later become writer
STM Design Tradeoffs 2: Locking mechanism

- Solution: use 32-bit integers for locks
  - Lowest 3 bits are state bits:
    - Notify (N) – list of waiting readers (to be notified)
    - Upgrade (U) – when a reader wants to upgrade
    - Reader (R) – 0 if a writer owns the lock

- Key point: when a reader wants to upgrade, it waits for all readers to release the lock
STM Design Tradeoffs 2: Locking mechanism

- Read versioning and write locking
- A writer takes a lock before changing memory, but readers use versioning to detect data conflicts
- 32 bit integer lock word: can be owned by a writer, or it can contain a version number
- Readers add this to their read set, and keep track of the version number
STM Design Tradeoffs 2: Locking mechanism

Figure 1: Comparison of reader-version with reader-lock
STM Design Tradeoffs 2: Locking mechanism

- Reader versioning outperforms reader-writer locks by an order of magnitude
- Two main reasons:
  - Readers no longer have to atomically write to the lock word (improves cache effects)
  - Dynamic reader to writer upgrades are expensive because reader has to wait for all other readers to release their read lock
    - This may also trigger aborts in the other readers
STM Design Tradeoffs 3: Write buffering vs. undo logging

- **Undo advantages:**
  - Fast commits
  - RAW cases are straightforward: reads occur from the actual memory, which holds the most recent value
  - More amenable to certain compiler optimizations: CSE, code hoisting, read-and-write barriers ([1])
STM Design Tradeoffs 3: Write buffering vs. undo logging

- **Write buffering advantages:**
  - Allows for lazy acquire of locks (i.e., until commit), which reduces lock holding time
  - Allows for lock acquiring in a canonical order (e.g., address order), which eliminates deadlocks

- **Write buffering disadvantages:**
  - Reads are expensive! (need to search memory)
  - Even more so for nested transactions
STM Design Tradeoffs 3: Write buffering vs. undo logging

Figure 2: Comparison of undo logging versus write buffering
STM Design Tradeoffs 3: Write buffering vs. undo logging

- Conclusions: undo logging performs better than write buffering
- Main reason: expensive reads (search for most recent speculative value)
- Hashtable: large read set -> write buffering fares really poorly
- This is without compiler optimizations or nested transactions – undo would be even better in that case!
STM Design Tradeoffs 4: Object vs. cache-line conflict detection

- **Object-level detection**
  - More intuitive for the user
  - Allows for aggressive compiler optimizations (e.g., acquire ownership of object once for several consecutive updates of the object)

- Easy to implement in Java, more difficult in unmanaged environments (C/C++)
STM Design Tradeoffs 4: Object vs. cache-line conflict detection

- Cache-line based conflict detection:
  - Array of locks indexed with the cache line address of the memory location
  - Easy to implement: bit masking and shifting operations
STM Design Tradeoffs 4: Object vs. cache-line conflict detection

- Object based conflict detection in C/C++
  - Uses the McRT memory manager (McRT-Malloc)
  - Segregated heap (divided in small blocks) to store small objects
  - Objects larger than 8k are allocated from a different (not size-segregated) heap
STM Design Tradeoffs 4: Object vs. cache-line conflict detection

- Object-based conflict detection
  - Object-based locking is provided only for objects in the size–segregated heap
  - Large objects, stack objects, and global variables use cache-line locking
  - This is good because object-based locking for large objects would coarsen granularity of conflict detection
STM Design Tradeoffs 4: Object vs. cache-line conflict detection

Figure 3: Comparison of object based and cache line locking
STM Design Tradeoffs 4: Object vs. cache-line conflict detection

- Conclusion: object-level conflict detection performs better for trees and hashtable (low contention)
- Cache-line detection does better for linked list (high contention), because of ping-pong effect
- We expect object-level detection to improve performance by using compiler optimizations
STM Performance
STM Performance: STM vs. Locking

Figure 6: STM versus locking on hashtable
STM Performance: STM vs. Locking

- Same programming effort to use STM or fine-grained locking
- At 16 CPUs, STM is about 1.8x slower than fine-grained locking
STM Performance: STM vs. Locking

Figure 7: STM versus locking on binary search tree with balancing
STM Performance: STM vs. Locking

- Both with and without balancing, STM outperforms locking for BSTs
- Fewer updates -> STM does even better (because the number of aborts is reduced)
- For list (sorted/unsorted), and B-Trees: STM has similar performance with coarse-grained locking
STM Performance: STM vs. Locking

- Is comparison vs. naïve coarse-grained locking fair?
- For all previous experiments, coarse-grained locking was used
- Authors claim fine-grained locking was implemented for BSTs and list, and is an order of magnitude slower than STM or coarse-grained locking
STM Performance: Overhead breakdown

Most overhead comes from the read barrier and validation costs

July 5, 2006
STM Performance on a non-synthetic workload

![Graph showing total time for mail/spam delivery with sendmail](image)

**Figure 16:** STM and lock behavior for sendmail spam filter
Conclusions

- McRT-STM provides a quantitative analysis of STM design tradeoffs
- It shows how to leverage parts of the runtime (scheduler, memory manager) to increase efficiency
- Has similar performance with lock-based applications (but lesser programming cost!)