Designing Transactional Memory Systems

Part III: Lock-based STMs

Pascal Felber
University of Neuchatel
Pascal.Felber@unine.ch

Based on joint work with Christof Fetzer & Torvald Riegel
with slides borrowed from several other people

Agenda

- **Part I:** Introduction
- **Part II:** Obstruction-free STMs
- **Part III:** Lock-based STMs
  - WSTM: a lock-based STM design
  - TINYSTM: a time-based lock-based STM design

Why not obstruction-free? [Ennals]

- OF prevents a long-running transaction blocking others
  - Not true: neither OF nor LF guarantee that!
- OF prevents the system locking up if a thread is switched part-way through a transaction
  - But this is rare and temporary event…
- OF prevents the system locking up if a thread fails
  - But failure would break the original, non-STM program as well…

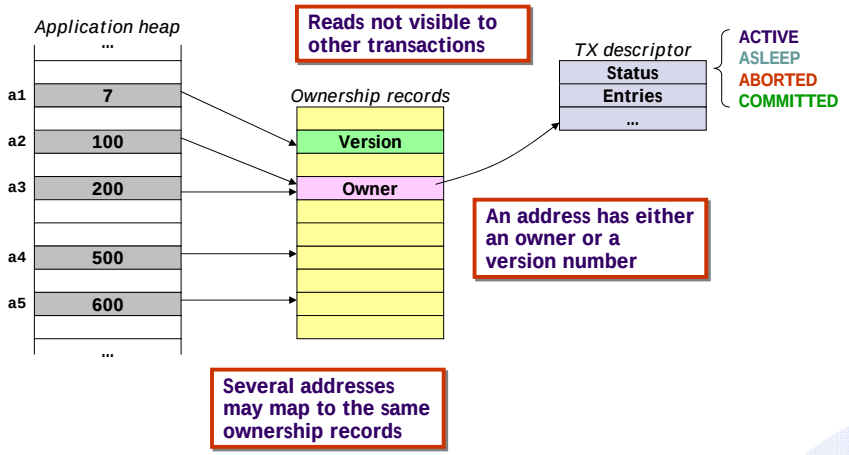
Why lock-based? [Ennals]

- More efficient implementation
  - No extra indirections for data accesses
  - Better cache locality
  - Simpler (and often faster) algorithm
- Typical implementation
  - Revocable two phase locking for writes (blocking)
    - Preventive abort to avoid deadlocks
  - Optimistic concurrency control for reads
    - Lazy conflict detection (if data has changed)

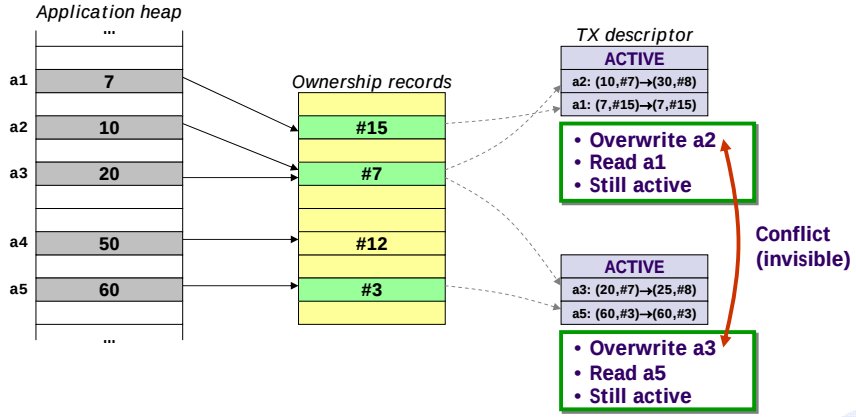
WSTM [Harris & Fraser, 2003]

- Word-based STM
  - Granularity of conflict detection is (roughly) memory location
- Basic principle
  - For every word, there exists a version number
  - Transactions don't update memory or version numbers (i.e., no synchronization) until commit
    - They update (and consult) thread-local log
    - Modified words are only "locked" during commit
  - Parallel reads don't cause aborts

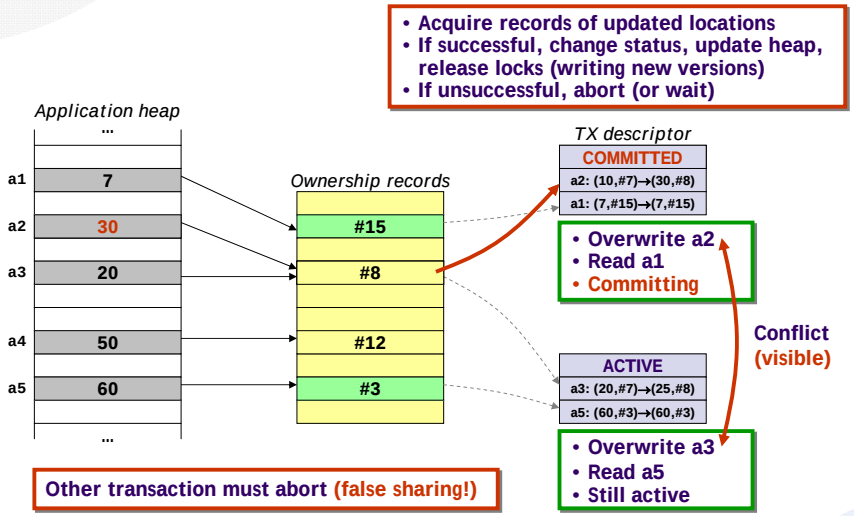
WSTM: data structures



WSTM: load and store



WSTM: commit



STM design choices

- We have already seen a number of designs
- There are many more **design choices**:
  - Obstruction-free vs. lock-based (blocking)
  - Object-based vs. word-based
  - Visible vs. invisible reads
  - Encounter-time vs. commit-time locking
  - Write-through vs. write-back
  - Etc.

STM design choices

- The "right" design depends on the **workload**
  - E.g., ratio of update to read-only transactions, number of locations read or written, contention on shared memory locations, etc.

There is no "one-size-fits-all" STM

Object-based vs. word-based

Object-based

- Granularity of conflict detection is object
- Store metadata in object (or proxy)
- Need language support

• Good with small objects (cloning cost)
• Coarse-grained "lock"

Word-based

- Granularity of conflict detection is memory location
- Separate metadata
- Load/store API

• Good for low-level, unmanaged languages
• Fine-grained "lock"

Visible vs. invisible reads

Visible reads

- Let writers detect conflicts with readers
- Need to maintain shared reader lists
- Pessimistic design

• Abort early may save useless processing
• Better progress upon high contention

Invisible reads

- Writers do not detect read-write conflicts
- Incremental validation costly (use time-based)
- Optimistic design

• Much faster when there is no conflict
• No (costly) reader list
• May fall back to visible

ETL vs. CTL

Encounter-time locking

- Acquire locks when memory is written
- Detect conflicts early

Commit-time locking

- Acquire locks at commit time
- Detects conflicts late

- Avoids executing doomed transactions
- Fast RW-after-write

- May reduce conflicts with some workloads

Write-through vs. write-back

Write-through (ETL)

- Writes to memory (undo log)
- Upon abort, copy back old values

Write-back (ETL)

- Buffered writes (redo log)
- Upon commit, copy new values

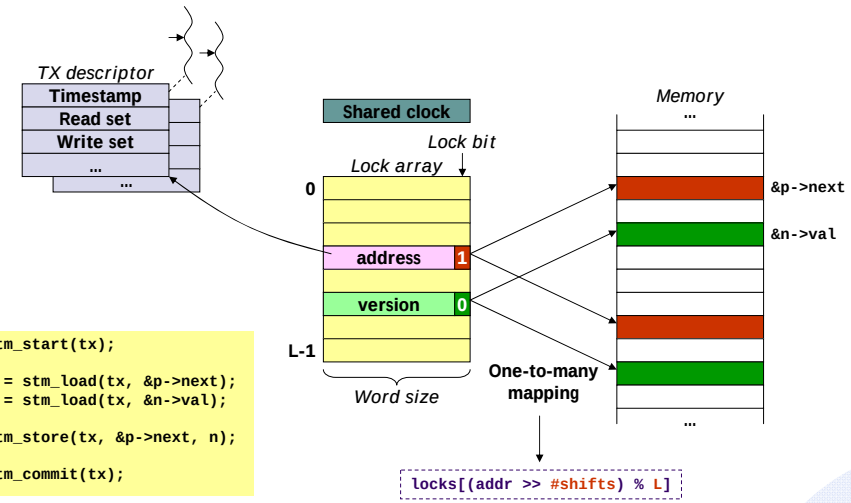
- Faster commit
- Faster RW-after-write, enables compiler optimizations

- Faster abort
- Version numbers don't change on abort (no ABA problem)

TinySTM: a lightweight design

- Word-based lock-based STM implementation
  - Written in portable C, 32/64-bit
  - Small code base (<1,000 LOC), GPL
  - Memory management operations
- Time-based algorithm like LSA & TL2
  - Versioned locks used to build consistent snapshot
- "Classical" word-based STM design
  - Per-stripe locks, encounter-time locking (ETL)
  - Write-through and write-back versions
- Used as underlying STM in TANGER

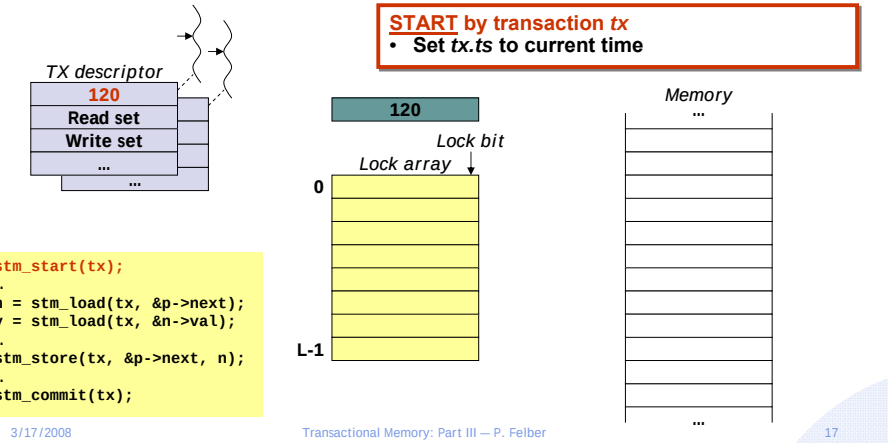
TinySTM: basic data structures




```
stm_start(tx);
…
n = stm_load(tx, &p->next);
v = stm_load(tx, &n->val);
…
stm_store(tx, &p->next, n);
…
stm_commit(tx);
```

TinySTM: start

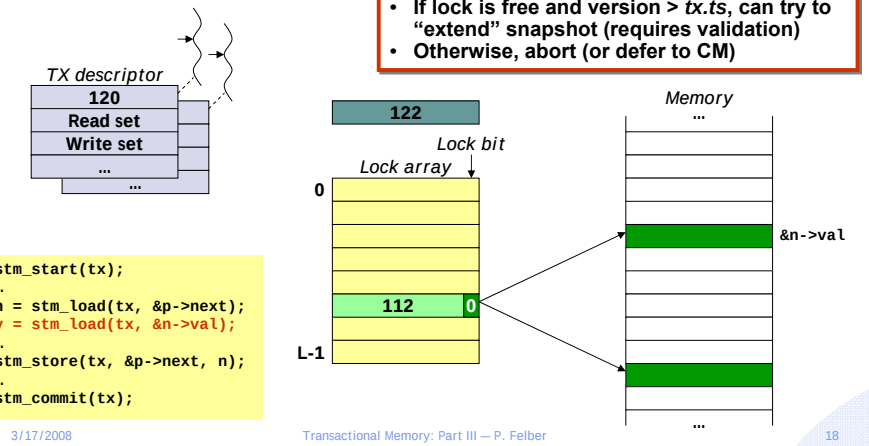
START by transaction *tx*
- Set *tx.ts* to current time




```
stm_start(tx);
…
n = stm_load(tx, &p->next);
v = stm_load(tx, &n->val);
…
stm_store(tx, &p->next, n);
…
stm_commit(tx);
```


TinySTM: load

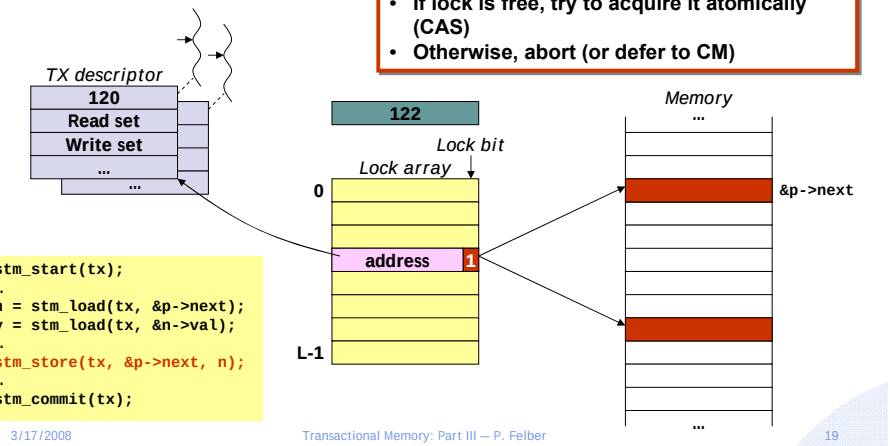
LOAD(addr) by transaction *tx*
- Find lock for *addr* and read lock, value, lock
- If lock is owned by *tx*, return latest value
- If lock is free and version ≤ *tx.ts*, return latest value
- If lock is free and version > *tx.ts*, can try to "extend" snapshot (requires validation)
- Otherwise, abort (or defer to CM)




```
stm_start(tx);
…
n = stm_load(tx, &p->next);
v = stm_load(tx, &n->val);
…
stm_store(tx, &p->next, n);
…
stm_commit(tx);
```


TinySTM: store

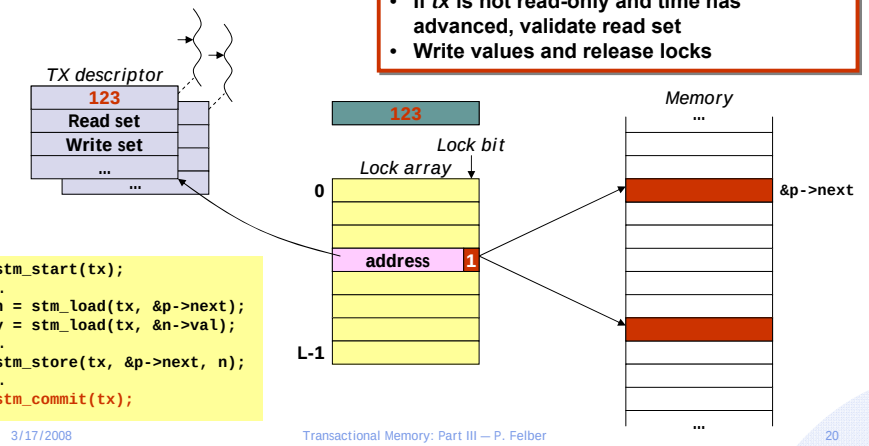
STORE(addr) by transaction *tx*
- Find lock for *addr* and read lock
- If lock is owned by *tx*, write new value
- If lock is free, try to acquire it atomically (CAS)
- Otherwise, abort (or defer to CM)




```
stm_start(tx);
…
n = stm_load(tx, &p->next);
v = stm_load(tx, &n->val);
…
stm_store(tx, &p->next, n);
…
stm_commit(tx);
```


TinySTM: commit

COMMIT by transaction *tx*
- Acquire unique timestamp from clock
- If *tx* is not read-only and time has advanced, validate read set
- Write values and release locks




```
stm_start(tx);
…
n = stm_load(tx, &p->next);
v = stm_load(tx, &n->val);
…
stm_store(tx, &p->next, n);
…
stm_commit(tx);
```

Implementation notes

- Implementing concurrent algorithms **efficiently** is challenging
  - Need to understand how MP systems and multicore processors work
  - Need to use the cheapest primitives that are sufficient (for safety)
  - Need to understand the memory models (when defined!)

If this scares you,
leave it to "experts" and use STM!

Memory models

Initially

```
int x, y, i, j;
x = y = 0;
```

Thread 1

```
x = 1;
j = y;
```

Thread 2

```
y = 1;
i = x;
```

Value of `i` and `j`?

Can we have?
`i == 0 && j == 0`

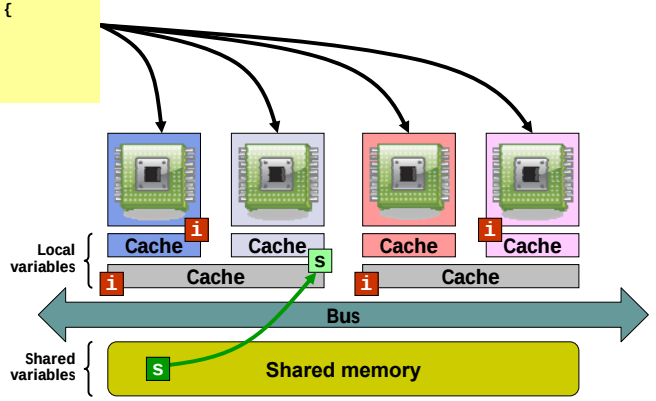
This is not wrong!

- We are assuming sequentially consistent behavior
  - But time is a relative dimension!
- Computers don't care about your intuition regarding time across multiple threads
  - Compilers, processors, caches can reorder instructions
- Preventing reordering must be explicitly requested
  - Using synchronization operations (lock/unlock)
  - Using memory barriers

Based on slide by Herlihy & Shavit

Where things fit

```
class SharedObject {
  int s;

  void f() {
    int i;
    …
  }
}
```



Memory barriers

- A processor can execute hundreds of instructions during a memory access
  - Why delay on every memory write?
  - Instead, keep value in register or cache

```
for (i = 0; i < 10000; i++)
  x += i;
```

Typically: `x` and `i` not written to memory at each iteration

- Memory barrier instruction (**expensive**)
  - Flush unwritten caches
  - Bring caches up to date
  - Added by compiler (synchronization, **`volatile`**)

Based on documentation by Doug Lea

Memory barriers

- **LD1 L/L LD2**

  LD1's data loaded before data accessed by LD2 and all subsequent loads are loaded
- **ST1 S/S ST2**

  ST1's data visible to other processors before data associated with ST2
- **LD1 L/S ST2**

  LD1's data loaded before data associated with ST2 and all subsequent store instructions are flushed
- **ST1 S/L LD2**

  ST1's data visible to other processors before data accessed by LD2 and all subsequent loads are loaded

Based on documentation by Doug Lea

Memory barriers example (Java)

```
class X {
  int a, b;
  volatile int v, u;
  void f() {
    int i, j;

    i = a;
    j = b;
    i = v;
    // L/L
    j = u;
    // L/S
    a = i;
    b = j;
    // S/S
    v = i;
    // S/S
    u = j;
    // S/L
    i = u;
    j = b;
    a = i;
  }
}
```

The keyword `volatile` asks compiler to keep variable up-to-date and inhibits reordering & other optimizations

The compiler inserts memory barriers where necessary

Based on documentation by Doug Lea

Memory barriers in TinySTM

```
stm_word_t stm_load(stm_tx_t *tx, stm_word_t *addr)
{
  stm_word_t l1, l2, val;
  …
  // Read lock status and value
  l1 = *lock;
  L/L;
  val = *addr;
  L/L;
  l2 = *lock;
  …
}

int stm_commit(stm_tx_t *tx)
{
  stm_write_entry_t *w;
  …
  // Write value and release lock
  *w->addr = w->val;
  S/S;
  *w->lock = tx->timestamp;
  S/L;
  …
}
```

Memory barriers must be inserted explicitly

An ABA problem in TinySTM

- Consider write-through implementation
  - Is “read lock, read address, read lock” safe?

```
// Read lock status and value
l1 = *lock;
val = *addr;
l2 = *lock;
if (l1 == l2) {
  // OK…
```

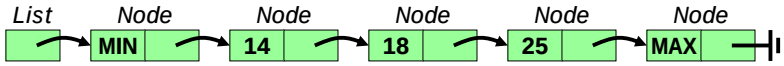
Requires changing the lock each time the value is updated!

We can increment the clock upon abort (costly)

We use “incarnation numbers” in timestamps (updated locally)

```
// Read lock status and value
l1 = *lock;
                    // WRITE
                    // Acquire lock
                    CAS(lock, …);
                    // Write value
                    *addr = val;
val = *addr;
                    // ABORT
                    // Write back old value
                    *w->addr = w->val;
                    // Release lock
                    *w->lock = w->l;
l2 = *lock;
if (l1 == l2) {
  // OK…
```


TinySTM: programming example



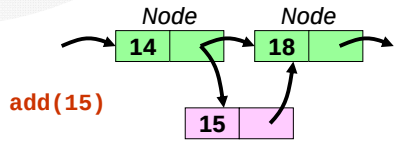
Non-transactional

```
typedef struct node {
  int val;
  struct node *next;
} node_t;

typedef struct intset {
  node_t *head;
} intset_t;
```

Transactional

```
typedef struct node {
  int val;
  struct node *next;
} node_t;

typedef struct intset {
  node_t *head;
} intset_t;
```


TinySTM: programming example



Non-transactional

```
int set_add(intset_t *set, int val) {
 int result;
 node_t *prev, *next;

 prev = set->head;
 next = prev->next;
 while (next->val < val) {
  prev = next;
  next = prev->next;
 }
 result = (next->val != val);
 if (result)
  prev->next = new_node(val, next);
 return result;
}
```

Transactional

```
int set_add(intset_t *set, int val) {
 int result, v;
 node_t *prev, *next;

 START;
 prev = (node_t *)LOAD(&set->head);
 next = (node_t *)LOAD(&prev->next);
 while ((v = (int)LOAD(&next->val)) < val) {
  prev = next;
  next = (node_t *)LOAD(&prev->next);
 }
 result = (v != val);
 if (result)
  STORE(&prev->next, new_node(val, next));
 COMMIT;
 return result;
}
```


TinySTM: programming example

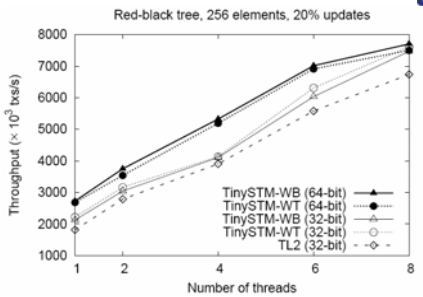
- Some useful macros to save a few keystrokes (ah… the joy of C!)

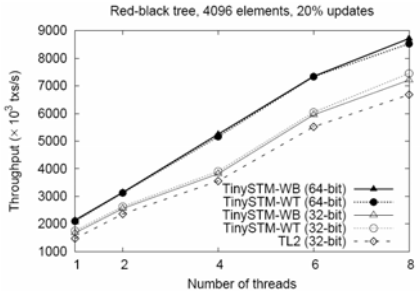
```
/* Transactional helpers */
#define START                { \\
                              stm_tx_t *tx = stm_new(NULL); \\
                              sigjmp_buf *_e = stm_get_env(tx); \\
                              sigsetjmp(*_e, 0); \\
                              stm_start(tx, _e, NULL)

#define START_RO              /* Read-only variant (omitted) */

#define LOAD(addr)            stm_load(tx, (stm_word_t *)addr)

#define STORE(addr, value)    stm_store(tx, (stm_word_t *)addr, (stm_word_t)value)

#define COMMIT                stm_commit(tx); \\
                              stm_delete(tx); \\
                             }

#define MALLOC(size)         stm_malloc(tx, size)

#define FREE(addr, size)     stm_free(tx, addr, size)
```

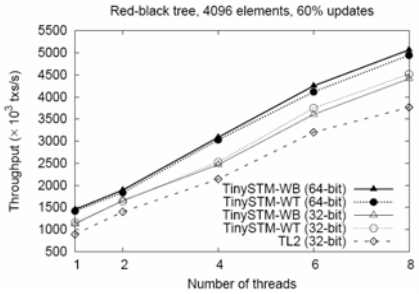
Throughput (red-black tree)

8-core Intel Xeon at 2 GHz, Linux 2.6.18-4 (64-bit)

**All designs scale well.
64-bit version noticeably faster.
Performance of CTL and ETL is comparable (little contention).**



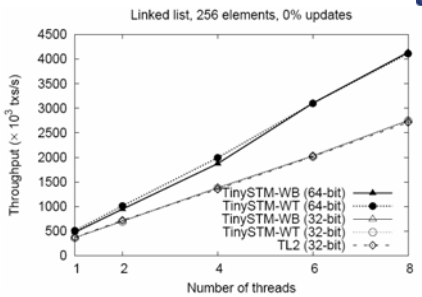


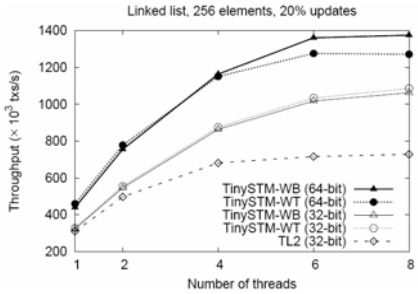


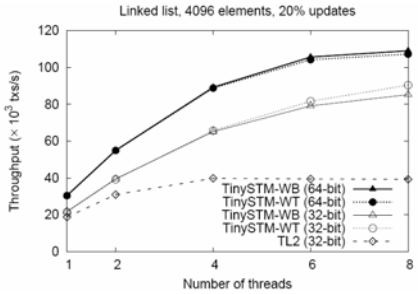
Throughput (linked list)

8-core Intel Xeon at 2 GHz, Linux 2.6.18-4 (64-bit)

**All designs scale well.
64-bit version noticeably faster.
CTL suffers more from long transaction (no CM).**



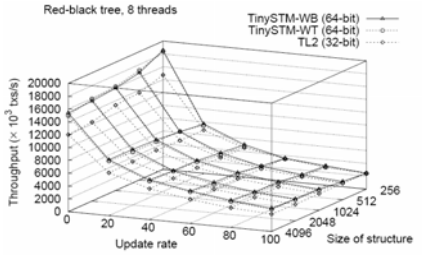


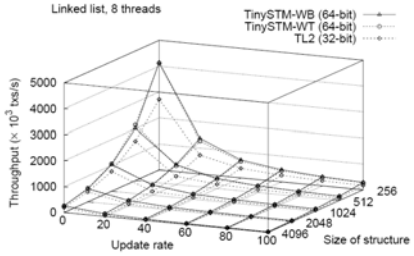


Size and update rates

8-core Intel Xeon at 2 GHz, Linux 2.6.18-4 (64-bit)

**Linked list more sensitive to size than red-black tree (linear vs. logarithmic).
Read-only *much* faster.**

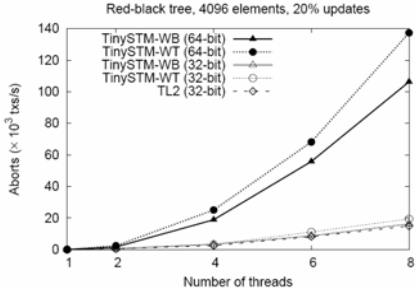


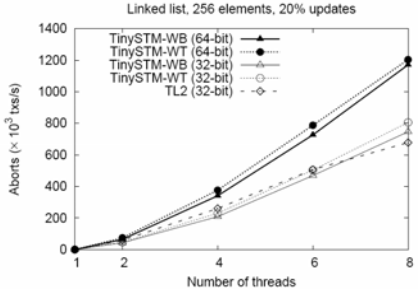


Abort rates

8-core Intel Xeon at 2 GHz, Linux 2.6.18-4 (64-bit)

**Abort rates increase upon contention, as expected.
64-bit has higher abort rate.
CTL has slightly less aborts.**





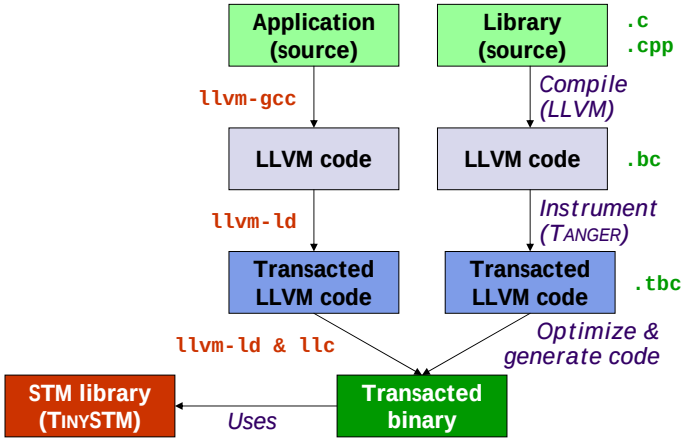
TANGER: transactional C compiler

- **Problem:** explicit load/store instructions are cumbersome and error-prone
- **TANGER:** open source transactional compiler
  - Application code with transaction boundaries transformed into STM transactional code
  - Uses LLVM's open-source compiler framework
    - Intermediate representation for a load/store architecture (no stack)
    - Supports aggressive optimizations at compile-, link-, or run-time
    - Easily extensible by "compiler passes"

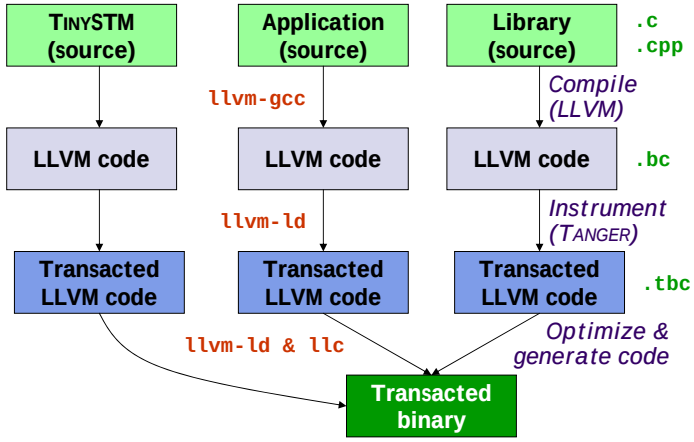
What TANGER does

- Application code uses minimal API
  - TX begin/commit
  - Language syntax unchanged
- Transform code to use word-based STM API
  - Create transactional version of functions
  - Within TX begin/commit, redirect:
    - (Heap) memory accesses to STM load/store
    - Calls to transactional versions of functions
    - Dynamic memory management (malloc, free)

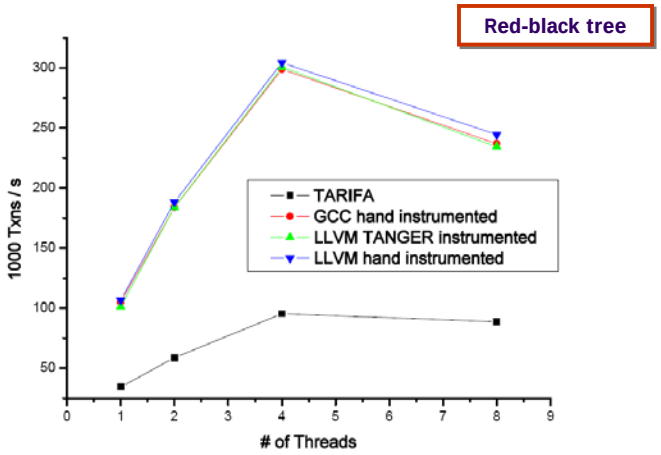
TANGER: build cycle



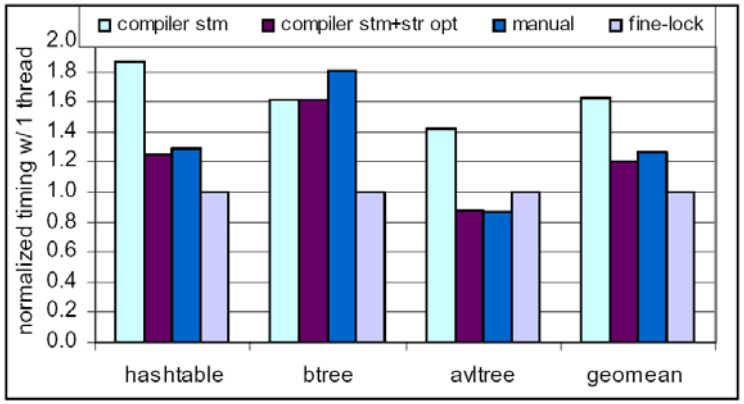
TANGER: build cycle



TANGER: performance



Compiler: overhead [Intel]



Compiler: scalability [Intel]

Conclusion (Part III)

- **Lock-based** STM designs are very efficient
  - No progress guarantee, but sufficient from a pragmatic perspective
- **Word-based** designs good for OS, low-level languages, and compiler integration
  - Fine conflict detection granularity
  - Can be used to build object-based STMs
- The **design space** is very large
  - No "one-size-fits-all" STM
  - The right design depends on the workload!

The one slide to remember

Concurrent programming is hard
... but necessary to exploit multicore CPUs

TM is a simple programming paradigm
... safe and scalable

You can (usually) trust TM designers

Our TM implementations: **www.tinystm.org**

Thank you!