Software transactional memory

Transactional locking II (Dice et. al, DISC'06)
Time-based STM (Felber et. al, TPDS'08)

Ioana Giurgiu

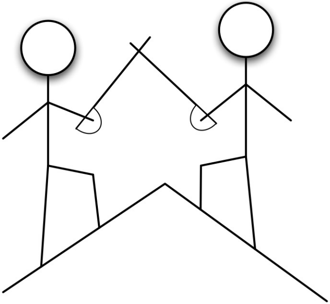
Mentor: Johannes Schneider

March 16^{th}, 2011

Motivation

- Multiprocessor systems
  - Speed up time-sharing applications
  - How to parallelize easily and efficiently?

  - Concurrent access to shared memory



T1

```
int arr[n];
i = 0;
...
while (i < n) {
        i++;
        ...
        arr[i] = i;
}
```

T2

```
...

i = 2*i;
```

Locking

T1

```
int arr[n];

lock.acquire();
i = 0;
...
while (i < n) {
      i++;
      ...
      arr[i] = i;
}
lock.release();
```

T2

```
lock.acquire();
      i = 2*i;
lock.release();
```

- Coarse-grained or fine-grained
- Difficult to program when # of locks is large
- Problems: deadlocks, priority inversions, convoying
- Race conditions
- **Other solutions?**

Transactional memory (Herlihy et. al, ISCA'93)

- Architectural support for lock-free data structures

- Based on LL/SC
- Changes cache coherence protocols
- Instructions
  - LT, LTX, ST
  - COMMIT, ABORT
  - VALIDATE

- NOT HERE YET

```
LoadLinked(r : Register, a : WordAddress)
  r ← *a
  Linked(a) ← True

StoreConditional(r : Registr, a : WordAddress)
  if Linked(a) = True
     then *a ← r
          r ← 1
     else r ← 0
```

Software transactional memory (Shavit et. al, PODC'95)

Software method to support flexible transactional programming of synchronization operations

- *Transactional model*
  - Transaction = atomic sequence of steps
  - Protect access to shared objects
- Only for static transactions
- Easier to program
- Higher performance than locks?

Transactional locking II Dice et. al, DISC'06

"If we implemented fine-grained locking with the same simplicity of coarse-grained, we would never think of building a transactional memory"

- Fits any memory lifecycle (GC, malloc/free)
- Safe → consistent memory states
- Performance
  - 10x faster than single locks on RBTs
  - Better than all lock-based & non-blocking STMs

Transactional locking II (cont.)

1. Use *commit-time locking* instead of encounter-time locking

- Encounter-time locking (Ennals, Saha): quick reads of freshly written values in memory by the read-only transaction
- Commit-time locking: memory locked only during the commit phase
  - Under high loads, better performance
  - Works with malloc/free

Transactional locking II (cont.)

2. Use *global version clock* for validation

- Why a clock?
  - Lock STMs – inconsistent states
  - Need validation periodically
- Shared global version clock
  - Incremented by write transactions
  - Read by all transactions
  - State always consistent

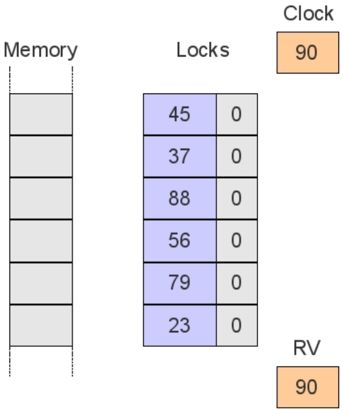
Transactional locking II (cont.)

3. Locks to shared data?

- PO (per object): lock per shared object
  - Insertion of lock fields
- PS (per stripe): array of locks per memory stripe
  - Each transactable location is mapped to a stripe
  - No changes to data structure

Transactional locking II (cont.)

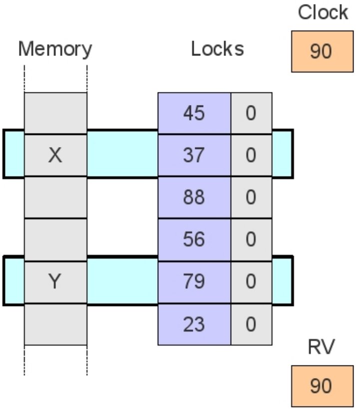
Read-only transactions



- RV $\leftarrow$ Clock
- Speculative execution
  - Write lock is free
  - If lock value $\leq$ RV $\rightarrow$ commit
  - If lock value $>$ RV $\rightarrow$ abort

Transactional locking II (cont.)

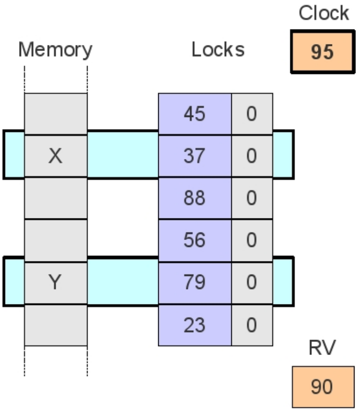
Write transactions



- RV $\leftarrow$ Clock
- Speculative execution
  - Write lock is free
  - Lock value $\leq$ RV
  - Maintain read/write set
- Lock write set

Transactional locking II (cont.)

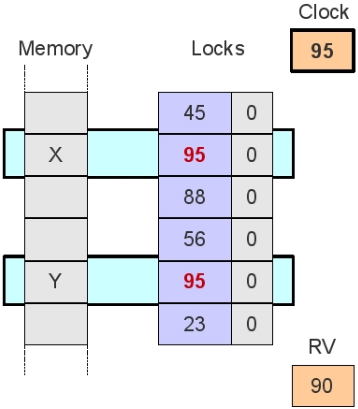
Write transactions



- RV $\leftarrow$ Clock
- Speculative execution
  - Write lock is free
  - Lock value $\leq$ RV
  - Maintain read/write set
- Lock write set
- WV $\leftarrow$ increment(Clock)
- Validate each lock value $\leq$ RV

Transactional locking II (cont.)

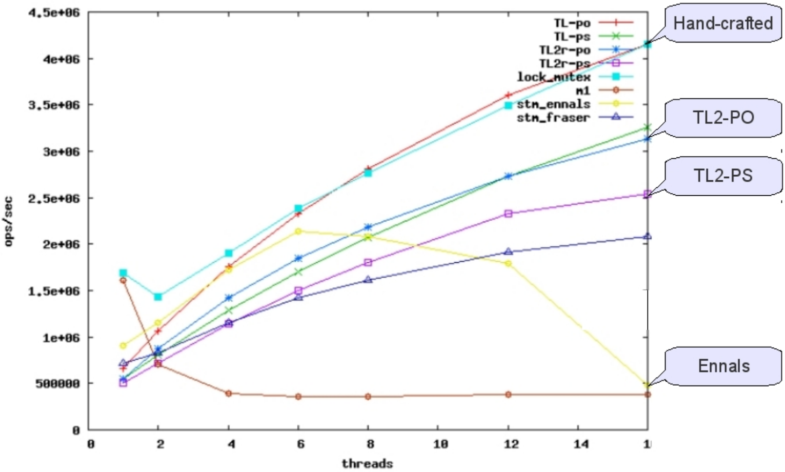
Write transactions



- RV ← Clock
- Speculative execution
  - Write lock is free
  - Lock value ≤ RV
  - Maintain read/write set
- Lock write set
- WV ← increment(Clock)
- Validate each lock value ≤ RV
- Release locks with value ← WV

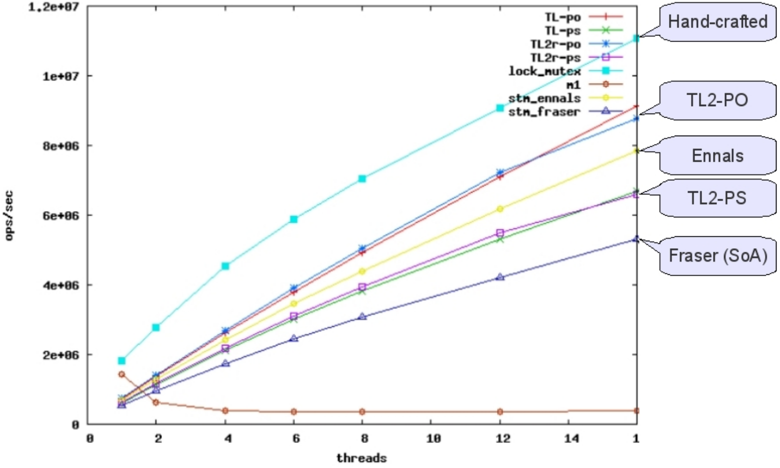
Transactional locking II (cont.)

Small RBT: 30% put, 30% delete, 40% get/16-proc SunV890



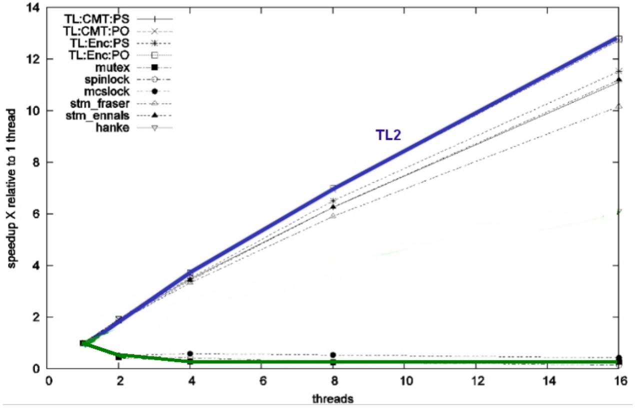
Transactional locking II (cont.)

Large RBT: 5% put, 5% delete, 90% get/16-proc SunV890



Transactional locking II (cont.)

Speedup – Large RBT – 5% puts, 5% deletes, 90% gets



Transactional locking II (cont.)

Conclusions

- STM scalability is comparable with hand-crafted, but overheads are much higher
- Read set and validation cost affect performance

- No meltdown under contention
- Seamless operation with malloc/free

LSA-STM Felber et. al, TPDS'08

Current trade-off between consistency and performance

LSA-STM (cont.)

Lazy snapshot algorithm speeds up transactions for large data sets, while reducing the overhead of incremental validation

How? Time-based algorithms allow to keep multiple versions of objects for RO transactions

LSA-STM (cont.)

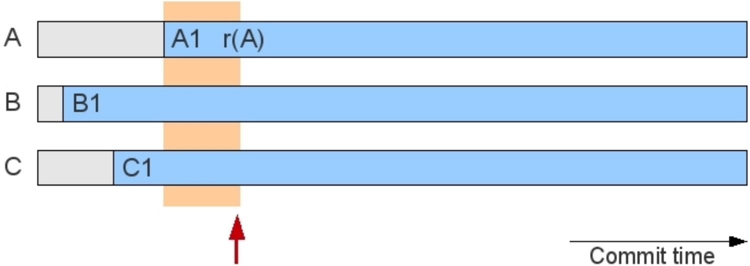
- Global clock CT counts # of commits
- STM objects (A,B,C) have multiple versions
  - Each version has a validity range R_v relative to CT
  - Most recent version has upper bound ∞

LSA-STM (cont.)

- Every transaction maintains a snapshot with a validity range R_T
  - Snapshot = $\bigcap$ of the accessed versions' validity range
  - Initialized to $[S_T, \infty]$
  - If snapshot == **nonempty** $\rightarrow$ commit

LSA-STM (cont.)

- When a transaction T reads an object
  - The version's validity range must $\bigcap$ T's snapshot
  - Snapshot bounds are adjusted to the $\bigcap$
  - Validity range ends at time of the read

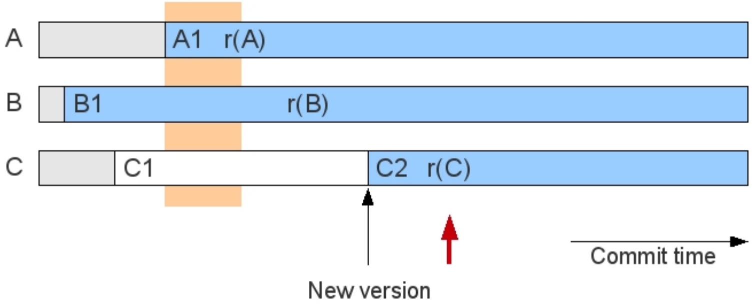


LSA-STM (cont.)

- If T's snapshot $\bigcap$ with the latest version's validity range
  - No need to update the snapshot

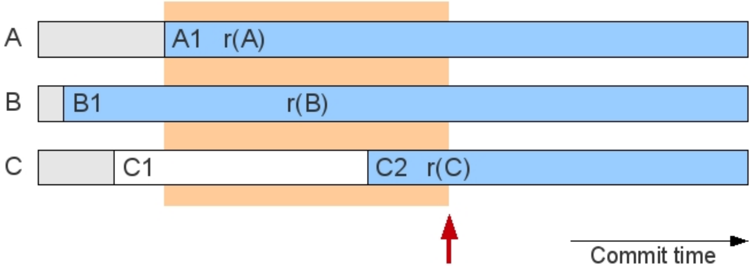
LSA-STM (cont.)

- If T's snapshot does not $\bigcap$ with the latest version's validity range
  - Extend snapshot (may fail)
- Read-only transactions can use old versions



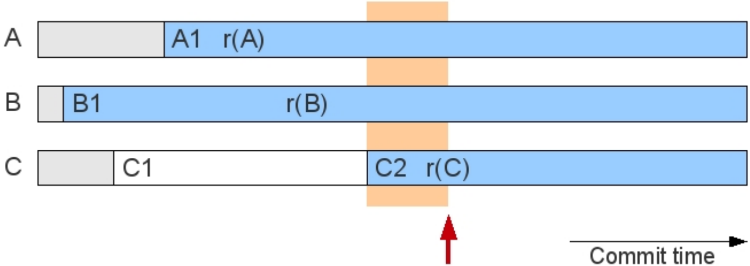
LSA-STM (cont.)

- Extension tries to increase the upper bound of the snapshot
  - Check if all read versions are valid
  - If yes, snapshot's upper bound = CT (now)



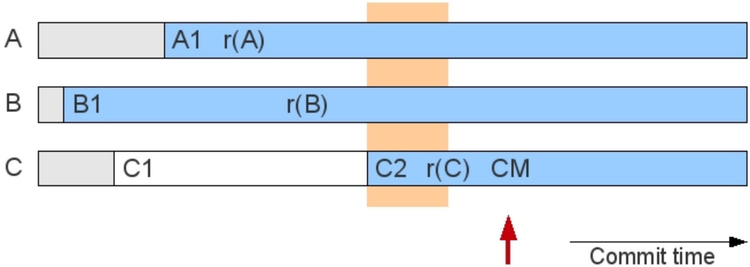
LSA-STM (cont.)

- Extension may increase the lower bound of the snapshot
  - = largest lower bound among the validity ranges of accessed versions



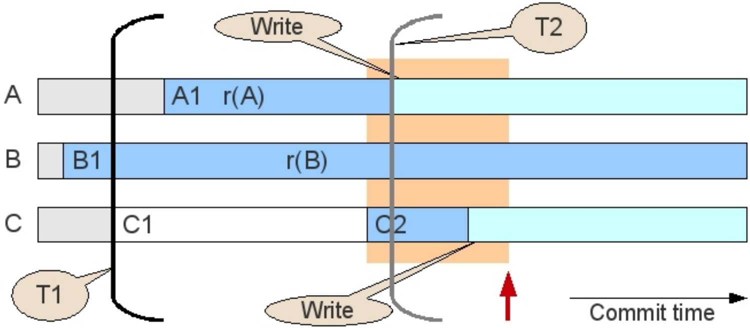
LSA-STM (cont.)

- Read-only transactions can commit if snapshot is not $\emptyset$
  - No need to extend range to CT



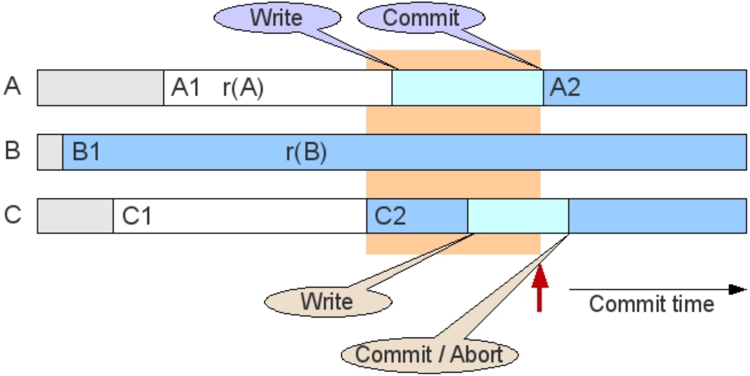
LSA-STM (cont.)

- Update transactions create new versions of modified objects when commiting at C_T
  - Validity range of new objects starts at C_T



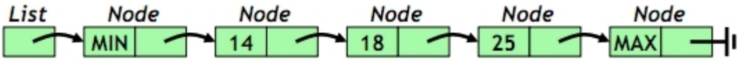
LSA-STM (cont.)

- Upon commit, an update transaction tries to acquire a new, unique commit timestamp at C_T
  - Transaction can commit iff the snapshot can be extended to C_T - 1



LSA-STM (cont.)

How to program?



Non-transactional

```
public class Node {

    public int getValue()
    public Node getNext()
    public void setValue(v)
    public void setNext(n)

}
```

Transactional

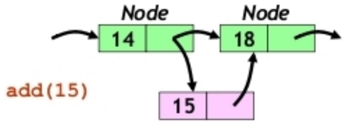
```
@Transactional
public class Node {

    @ReadOnly
    public int getValue()
    @ReadOnly
    public Node getNext()
    public void setValue(v)
    public void setNext(n)

}
```

LSA-STM (cont.)

How to program?



Non-transactional

```
public boolean add(v) {
    …
}
```

Transactional

```
@Atomic
public boolean add(v) {
    …
}
```

LSA-STM (cont.)

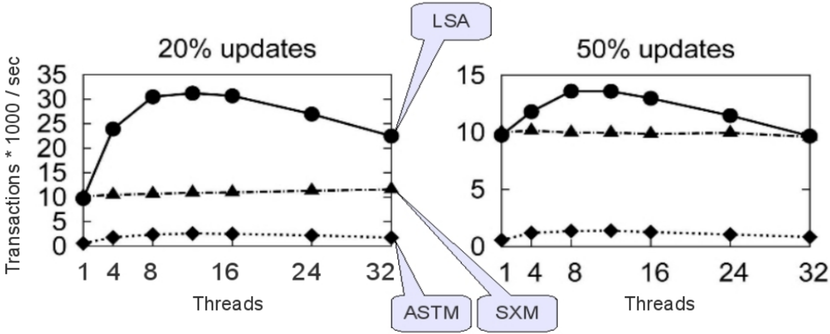
Performance evaluation

- Java implementation
- Sun Fire T2000 8 core UltraSparc T1 processor (8-core Niagara)

- SXM: visible reads
- ASTM: invisible reads, incremental validation
- LSA: time-based invisible reads

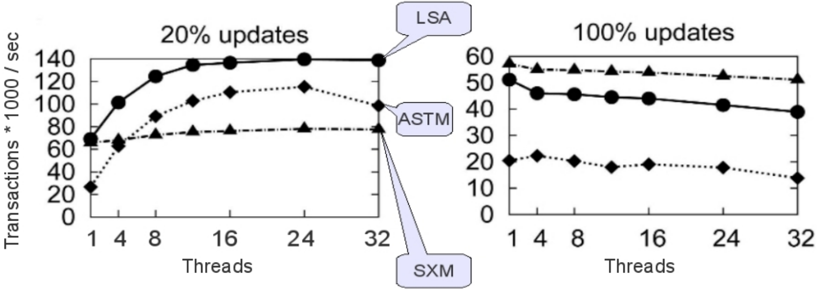
LSA-STM (cont.)

Linked list: 256 elements



LSA-STM (cont.)

RBT: 65536 elements



LSA-STM (cont.)

Conclusions

- High performance and consistency
- Obstruction-free implementation in Java
  - Weakest guarantee for a system
  - At least one thread makes progress

Multiplayer gaming

Multiplayer gaming (cont.)

Parallelization of SynQuake (Lupei et. al, Eurosys'10)

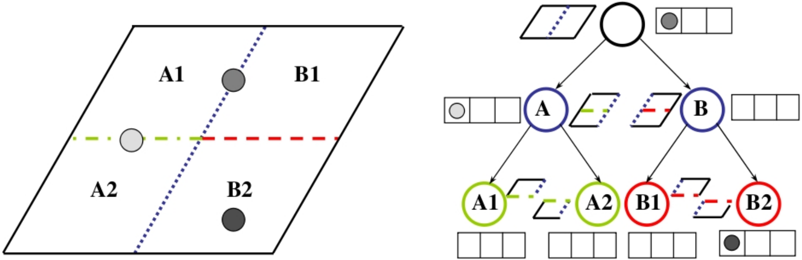
Why? Scale the game server

- SynQuake
  - Extracts data structures and features of Q3
  - Driven with synthetic workload (game actions, hot-spot scenarios)
- libTM (Lupei et al., Interact'09)
- C/C++ support

Multiplayer gaming (cont.)

Areanode tree

- Binary space partitioning tree (each node = specific map region)
- Efficient searching for all entities that a player interacts with
- By recursively dividing the map into submaps (median on X and Y)

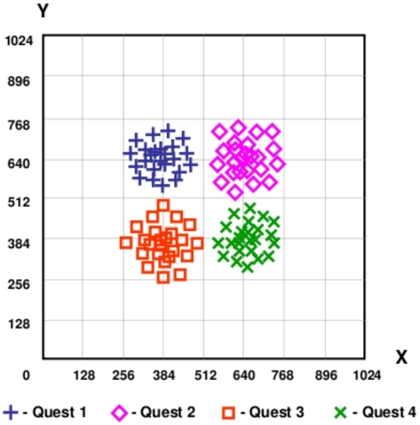


Multiplayer gaming (cont.)

- 8 cores $\rightarrow$ 2 Xeon Quad-Core @ 3GHz
- 600 to 2000 players
- 1000 server frames on a 1024 x 1024 map
- areanode tree depth = 8

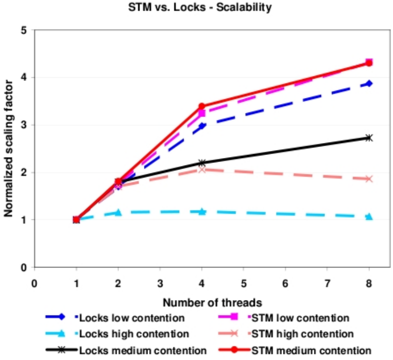
Multiplayer gaming (cont.)

Default setting: 4 quests with low/medium/high contention overload

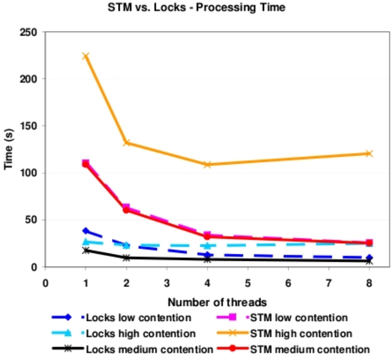


Multiplayer gaming (cont.)

Scalability vs. processing time



(a) Scalability



(b) Processing Time

Conclusions

- STMs are a viable alternative to locks
- Different flavors: TL2, LSA
- SynQuake

- Easier to program than locks
- Better performance for higher degree of concurrency
- Higher overheads
- Integration with existing languages

- **Support in hardware...?**