Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

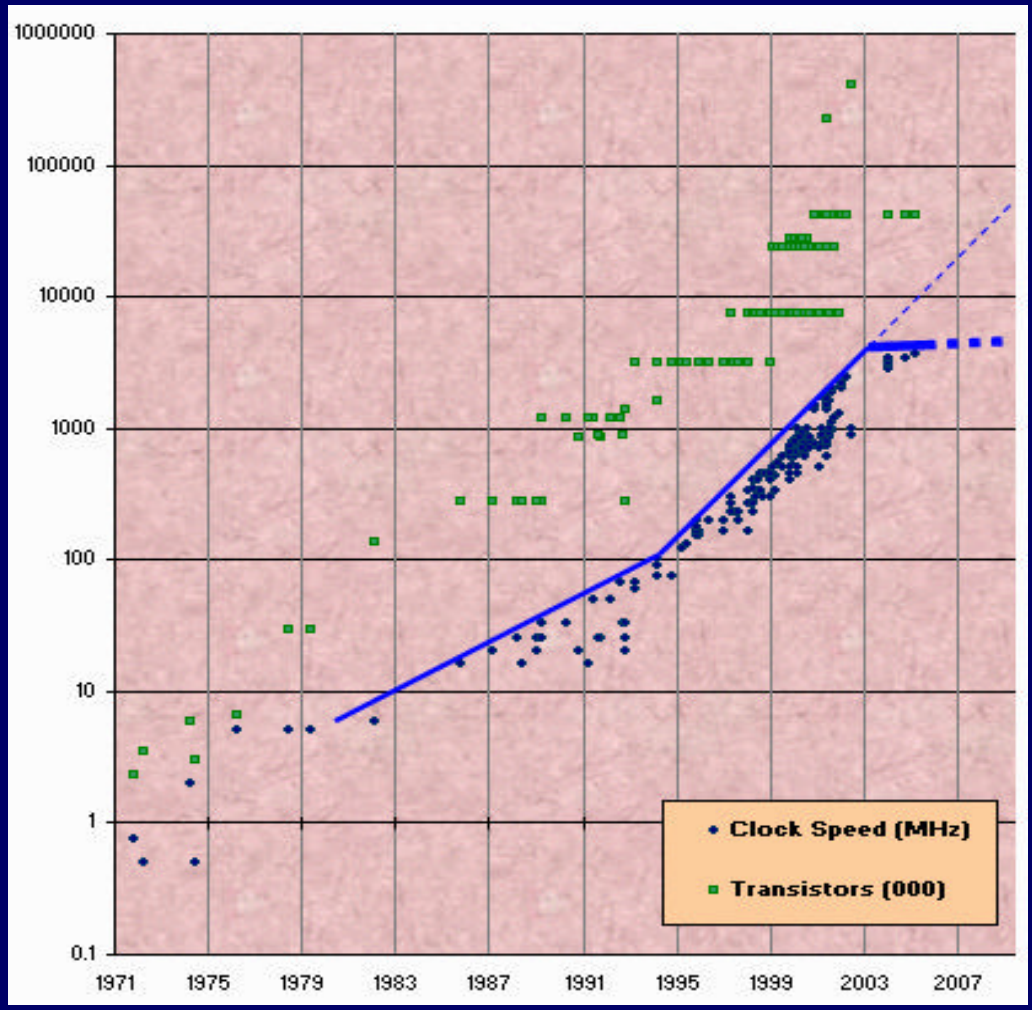
Testing Patterns for Software Transactional Memory Engines

João Lourenço
Joao.Lourenco@di.fct.unl.pt

Gonçalo Cunha
Goncalo.Cunha@gmail.com

Motivation [1]

- Limits of Moore's law
  - Clock speed stopped increasing (3 GHz for the last 5 years)
- Future (present) PC's are multi-core
  [Borkar, Dubey, Kahn, et al. "Platform 2015." Intel White Paper, 2005]
  - Parallelism is the "way to go"



Sutter. "A Fundamental Turn Toward Concurrency." Dr. Dobb's Journal, 2005.

Motivation [2]

- Concurrent programming is difficult
  - Deal with many issues
  - e.g., concurrency control
- Locks “lead the market”
  - Coarse-grained: lock the whole big routine (e.g., java “synchronized”)
    - Easier to use but limits concurrency
  - Fine-grained: lock only the needed item
    - Allows more concurrency but error-prone

Motivation [3]

- Problems with locking
  - Races: forgotten locks
  - Priority inversion: low-priority job holds a lock waited by a higher-priority one
  - Deadlocks: locks acquired in “inconsistent order”, no progress at all
  - Livelocks: permanent “do/undo”, no effective progress
  - Convoying: lock-older descheduled, no others may proceed
  - Starvation: a process never runs
  - Tricky error handling: need to restore invariants and release locks in exception handler
  - Simplicity vs. scalability tension
  - But the worst of all is… **locks do not compose!**

Motivation [4]

- Is composition that important?
  - Composition helps dealing with complexity (scalability)
    - Build large programs from small working pieces
- Example:
  - A.withdraw(m) / A.deposit(m)
    - Withdraw / deposit money from / to account A
    - Use lock to block access to account within the methods
  - A.transfer(B,5)
    - A.withdraw(5); B.deposit(5)
    - There is a period where the money is not in A neither B
      - Remove lock / unlock from primitives and expose locking
      - lock(A); lock(B); A.withdraw(3); B.deposit(3); unlock(A); unlock(B)

Motivation [5]

- The alternative…
  (Software) Transactional Memory – STM
- Inspired in DB folk
  - ACID properties
    Atomicity, Consistency,
    Isolation, Durability
- Drop CD and keep AI
- Ensure speed, more speed and… oh yes!
  Speed!

```
atomic {

A.withdraw(3)
    B.deposit(3)
}
```

Not in scope of this talk

Agenda

- ~~Motivation~~
- STM Design and Implementation Issues
- Testing an STM implementation
- Conclusions

STM Design & Implementation Issues

- Synchronization
  - May use blocking or non-blocking techniques
- Recovery strategy
  - Undo-log (in-place) / Redo-log (out-of-place)
- Transactional granularity
  - Object-level / block-level (word or cache-line size)
- Lock placement
  - Adjacent to data / separate lock table

Testing an STM Implementation

- Synchronization
  - May use blocking or non-blocking techniques
- Recovery strategy
  - Undo-log / Redo-log
- Transactional granularity
  - Object-level / block-level (word or cache-line size)
- Lock placement
  - Adjacent to data / separate lock table

Based in TL2 from Dice, Shalev and Shavit

Terminology – Transactional level

Symbol	Meaning
TxStart()	Start a transaction
TxCommit()	Commit a transaction
TxAbort()	Abort a transaction
TxLoad(x)	Transactional operation to read the value of variable “x”
TxStore(x,a)	Transactional operation to store the value “a” in variable “x”
TxSterilize(x)	Prevents running transactions to do any further read / write to the variable “x”

Terminology – Lock & Data level

Symbol	Meaning
a = RV(x)	Read the value of transactional variable “x”
v = RL(x)	Read the lock version of transactional variable “x”
SL(x,v)	Store “v” as the lock version of transactional variable “x”
WV(x,a)	Write the value “a” to transactional variable “x”
Acq(x)	Acquire the lock of transactional variable “x”

Simplified decomposition of TL to RL

Some internal operations were omitted

Symbol	Redo-log STM	Undo-log STM
TxStart()	ts = clock;	ts = clock;
TxLoad(y)	v1 = RL(y); a = RV(y); v2 = RL(y);	v1 = RL(y); a = RV(y); v2 = RL(y)
TxStore(x,a)		Acq(x); WV(x,a)
TxCommit(x)	Acq(x); RL(y); WV(x,a); Rel(x);	RL(y); Rel(x)
TxAbort(x,v)		WV(x,old); Rel(x)
TxSterilize(x)	SL(x,clock)	SL(x,clock)

Sample of Bugs Found [1]

Reference to non-transactional memory (undo/redo-log mode)

T1	T2	Description
y=TxLoad(x.n)		Get the pointer to node “y”
	y=TxLoad(x.n) z=TxLoad(y.n) TxStore(x.n,z) TxCommit() TxSterilize(y) free(y)	Remove node “y” from linked list
a=TxLoad(y.v)		Access to free’d memory in “y”

Original implementation only limited new writes (allowing reads)

Sample of Bugs Found [2]

Lost update with a small lock table (redo-log mode)

T1
TxLoad(x)
TxStore(y,a)
TxStore(x,a)
TxCommit()
-Acq(y)
-Acq(x)
-RL(x)

```
[…]
for each v in write-set {
  if (v is not locked) {
    if (v is also in the read-set)
      // read/write variable
      if (get_lock_version (v) >
tx_timestamp)
        abort (); // variable has been
changed
      else
        lock (v);
    else
      // write
      lock (v)
  }
}
```

"y" is hashed into the same position in lock table as "x"

Sample of Bugs Found [3]

Dirty-read not invalidated when transaction aborts (undo-log mode)

T1	T2	Description
TxLoad(x) - RL(x)		T1 is loading variable “x”
	TxStore(x,a) - Acq(x); WV(x,a)	T2 writes a new value into “x”
- RV(x)		T1 rea
	TxAbort() - WV(x,old); Rel(x)	T2 aborts and restores previous (old) “x” value
- RL(x) TxCommit()		Lock version revalidation OK

T2 did not commit
The lock for “x” was not incremented

Sample of Bugs Found [4]

Lost update on lock upgrade
(undo-log mode)

T1	T2	Description
v=TxLoad(x)		
	TxStore(x, a) TxCommit()	
TxStore(x, v+1)		Upgrade from read-only to cess

Acquired lock for “x”
No validation of previous reads was done

Harmful interleavings

- Improperly read and/or modify a shared variable
- Only occur while the transactional space has been changed by a transaction
  - During read, write/update, commit, abort, and adding or removing variables to/from the transitional space
- Frequently are uncommon, allowing test

Testing patterns [1]

- Aim at identifying patterns that increment the probability of generating harmful interleavings
- Target concurrency control errors, but also specific implementation options (such as bug [2: *lost update with small lock table*])
- Maximize $f_n = S_i C_i / T_i$
  - C_i → time transaction runs with shared state changed
  - T_i → transaction total runtime

Testing patterns [2]

- Very short transactions with Read & Write operations
  - Aims at maximizing interleavings between the main transactional operations (read, write, commit, abort)
  - Also aims at maximizing the frequency of commits
  - Adequate to redo-log based STMs
    - Only change the shared state at commit time
  - Useful for bug [1: *reference to non-transactional memory*]

Testing patterns [3]

- High frequency of variables entering and leaving the transactional space
  - Aims at stressing the variability of the transactional space
  - Targets bugs related to transactions holding pointers to variables being simultaneously released by other transactions
  - Useful for bug [1: *reference to non-transactional memory*]

Testing patterns [4]

- High number of updates on a small number of variables
  - Aims at generating a very high frequency of collisions between transactional read and write (frequent aborts)
  - Adequate to undo-log based STMs
    - Change the shared state on writes (data and locks), commit (only locks) and aborts (data and locks)
  - Useful for bugs [2: *lost update with a small lock table*] and [3: *dirty-read not invalidated when transaction aborts*]s
  - Overall was one of the most effective

Testing patterns [5]

- Small lock table
  - Lock table stores object/data locks
  - Hash function map objects/data to its lock (within table)
  - Hash function may map several objects to same table position (lock collision)
  - Lock collisions may cause improper validation of the lock state
  - Useful for bug [4: *lost update on lock upgrade*]

Testing patterns [6]

- More concurrent transactions than CPUs
  - If number of transactions < number of CPUs, any transaction willing to run will be scheduled immediately
  - Transactions will never be stalled waiting for CPU
  - Some interleavings depend on transactions being preempted and stalled for some time
  - Useful for bugs [3: *dirty-read not invalidated when transaction aborts*]
    - This bug depends on transaction T1 being preempted while execution a TxLoad() operation

Conclusions [1]

- Our experiments focused mainly on a TL2 variant
  - Testing cross-referencing with LibLTX (Ennals)
- Identifying testing patterns
  - A testing pattern may be instantiated by different test routines
- The identified patterns, proved to be very effective on testing two completely different STM implementations
- Reasoning in terms of testing patterns (behavior)

Conclusions [2]

- Fine tuning of testing patterns may lead to quite different results
- Experiments suggest that…
  - Execution environment has strong implications on STM engine stability
    - Tests executed in multi-core computers may behave differently when execution in multiprocessors
    - Multi-core share cache, multiprocessors don't → high frequency of out-of-order executions
  - Some errors are directly or indirectly related to out-of-order processor instruction execution hazard
    - Tests that could run for hours or days may fail in seconds

Future work [1]

- Identify other harmful interleavings (we already have some more…) and synthesize testing routines (and patterns) that trigger those interleavings
- Develop a visualization/display interface relating transactional events at…
  - Application perspective (transactional level)
  - STM engine (lock- & data-level)
  - Processor perspective (machine code instructions)

Future work [2]

- Augmenting state space coverage
  - e.g, “noise” generators
  - Higher probability of generating harmful interleavings
- Debugging STM based computations
  - Debugging within the context of a memory transaction
- Visualizing STM based computations

The end…

Thank you!

Questions?