Transactional Memory: Myths and Limits

R. Guerraoui, EPFL
This tutorial is about

Principles of transactional memory
Transactional memory

1. Why do we care?

2. What should we expect?

3. What might we expect?
Transactional memory

1. Why do we care?
From the New York Times
San Francisco, May 7, 2004

Intel announces a drastic change in its business strategy:

« Multicore is THE way to boost performance »
Moore’s Law and CPU speed

- Transistor count still rising according to Moore’s Law
- Clock speed flattening
Multicores

Multicores are the only way to increase performance

Indeed single-thread performance doesn’t improve...
... but we can put more cores on a chip
Multicores everywhere

- **Dual-core** commonplace in laptops
- **Quad-core** in desktops
- **Dual quad-core** in servers
- All major chip manufacturers produce multicore CPUs
  - **SUN Niagara** (8 cores, 32 concurrent threads)
  - **Intel Xeon** (4 cores)
  - **AMD Opteron** (4 cores)
SUN’s Niagara CPU2 (8 cores)
AMD Opteron (4 cores)
Basically

Two fundamental components that *fall apart*: *processors* and *memory*

The Interconnect links the processors with the memory:
- *SMP* (symmetric): bus (a tiny Ethernet)
- *NUMA* (network): point-to-point network
Cycles

- The basic unit of time is the cycle: time to execute an instruction.
- This changes with technology but the relative cost of instructions (local vs memory) does not.
Hardware synchronization objects

The basic unit of communication is the *read* and *write* to the memory (through the cache).

More sophisticated objects are sometimes provided: *C&S, T&S, LL/SC*
The free ride is over
The free ride is over

- Cannot rely on CPUs getting faster

- Utilizing more than one CPU core requires thread-level parallelism (TLP)
Everyone will need to fork threads

Travailler plus pour gagner plus
Forking threads is easy

Handling their conflicts is hard
**Traditional scaling**

- **Speedup**
  - 1x
  - 2x
  - 4x

- **User code**

- **Traditional CPU**

**Time: Moore's Law**
Ideal multicore scaling

- **Speedup**
  - 1x
  - 2x
  - 4x

- **User code**
  - Single core
  - Two cores
  - Four cores

- **Multicore CPU**
  - Single core
  - Two cores
  - Four cores

**Time: Moore’s Law**
Real-world scaling

Parallelization & synchronization require great care!

Speedup

1x  1.4x  2.2x

User code

Multicore CPU

Time: Moore’s Law
The problem
Sharing
public class Counter

private long value;

public Counter(int i) { value = i;}

public long getAndIncrement()
{
    return value++;
}
How to synchronize?

Concurrent processes

Shared object
Locking (mutual exclusion)

Locked object
Locking with compare&swap()

- A Compare&Swap object maintains a value x, init to ⊥, and y;
- It provides one operation: c&s(v,w);

✓ Sequential spec:
  - c&s(old,new)
    {y := x; if x = old then x := new; return(y)}
Locking with compare&swap()

lock() {
    repeat until
    unlocked = this.c&s(unlocked,locked)
}

unlock() {
    this.c&s(locked,unlocked)
}
Locking with compare\&swap()

```java
lock() {
    while (true) {
        repeat until (unlocked = this.getState());
        if unlocked = (this.c\&s()) return(true);
    }
}

unlock() {
    this.setState(0);
}
```
Explicit use of a lock

```
Lock l = ...;
    l.lock();
    try {
        // access the resource protected by this lock
    } finally {
        l.unlock();
    }
```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void getAndincrement() {
        c++; return c;
    }
    public synchronized int value() {
        return c;
    }
}
Locking is the current state of concurrency affairs
The use of locks is dangerous

50% of the bugs reported in Java come from the mis-use of « synchronized »
Coarse grained locks => slow

Fine grained locks => errors
Double-ended queue
Fine-grained locking

It took two years for the Java Standards Committee to approve (in Java 5) a fine-grained locking-based implementation of a hash-table.
Locks do not compose
Lock-free computing?

Every lock-free data structure
⇒ podc/spaa/disc
Wanted

A concurrency control abstraction that is simple and efficient
Transactions
Historical perspective

- Eswaran et al (CACM’76) Databases
- Papadimitriou (JACM’79) Theory
- Liskov/Sheifler (TOPLAS’82) Language
- Knight (ICFP’86) Architecture
- Herlihy/Moss (ISCA’93) Hardware
- Shavit/Touitou (PODC’95) Software
- Herlihy et al (PODC’03) Software - Dynamic

Now: DISC/PODC/POPL/PLDI/ECOOP/OOPSLA-SPLASH/CAV...Transact
Back to the undergraduate level

accessing object 1;
accessing object 2;
Back to the undergraduate level

atomic {
    accessing object 1;
    accessing object 2;
}
class Queue {
    QNode head;
    QNode tail;
    public enq(Object x) {
        atomic {
            QNode q = new QNode(x);
            q.next = head;
            head = q;
        }
    }
    ...
}
Simple example
(consistency invariant)

\[ 0 < x < y \]
Simple example (transaction)

\[ T: x := x+1 ; y := y+1 \]
The illusion of a critical section

atomic {
  accessing object 1;
  accessing object 2;
}
How to provide that illusion?

Software (STM) or Hardware (HTM)?
The garbage-collection analogy

In the early times, the programmers had to take care of allocating and de-allocating memory

The GC gives the illusion of infinite memory

A hardware support was initially expected, but now software solutions are very effective
Behind the scenes
Two-phase locking (2PL)

- To **write** O, T requires a **lock** on O; T **waits** if some T’ acquired a **lock** on O

- To **read** O, T requires a **lock** on O; T **waits** if some T’ acquired a **lock** on O

- Before committing, T **releases** all its locks
Two-phase locking (2PL)

- To **write** O, T waits to for a **lock** on O;

- To **read** O, T waits to for a **lock** on O;

- Before committing, T **releases** all its locks
Two-phase locking
(more details)

Every object O, with state \( s(O) \) (a *register*), is protected by a lock \( l(O) \) (a *c&s*)

Every transaction has local variables \( wSet \) and \( wLog \)

Initially: \( l(O) = \) unlocked, \( wSet = wLog = \) empty
Two-phase locking

Upon op = \texttt{read()} or \texttt{write(v)} on object O
if O outside wSet then
  wait until unlocked = $l(O).c&s(\text{unlocked, locked})$

wSet = wSet U O
wLog = wLog U S(O).\text{read()}

if op = \texttt{read()} then return $S(O).\text{read()}$
S(O).write(v)
return ok
Two-phase locking (cont’d)

Upon `commit()`
cleanup()
return ok

Upon `abort()`
rollback()
cleanup()
return ok
Two-phase locking (cont’d)

Upon *rollback()*
for all O in wSet do S(O).write(wLog(O))
wLog = empty

Upon *cleanup()*
for all O in wSet do l(O).c&s(locked,unlocked)
wSet = empty
Why two phases? (what if?)

To **write** or **read** O, T requires a **lock** on O; T **waits** if some T’ acquired a **lock** on O

T **releases** the lock on O when it is done with O
Why two phases?

T1

read(0) | write(1)
------|--------
O1     | O2

T2

read(0) | write(1)
------|--------
O2     | O1
No STM implements 2PL

All implement a variant of it
Two-phase locking
(read-write lock)

- To *write* O, T requires a *write-lock* on O; T *waits* if some T’ acquired a *lock* on O

- To *read* O, T requires a *read-lock* on O; T *waits* if some T’ acquired a *write-lock* on O

- Before committing, T *releases* all its locks
Two-phase locking
- better dead than wait -

- To write O, T requires a write-lock on O;
- T aborts if some T’ acquired a lock on O

- To read O, T requires a read-lock on O;
- T aborts if some T’ acquired a lock on O

- Before committing, T releases all its locks
Two-phase locking
- better kill than wait -

To **write** O, T requires a **write-lock on** O; T **aborts T’** if some T’ acquired a **lock** on O

To **read** O, T requires a **read-lock on** O; T **waits** if some T’ acquired a **write-lock** on O

Before committing, T releases all its locks
A transaction that is aborted restarts again
Visible Read
(SXM; RSTM)

Write is mega killer: to write an object, a transaction aborts any live one which has read or written the object

Read is visible: when a transaction reads an object, it says so
Visible Read

- A visible read invalidates cache lines

- This reduces the throughput of read-dominated workloads, by inducing a lot of traffic on the bus
Two-phase locking
- invisible reads – DSTM -

To **write** O, T requires a **write-lock on** O; T aborts T’ if some T’ acquired a **write-lock** on O

To **read** O, T checks if **all objects read remain valid** - else T aborts

Before committing, T checks if **all objects read remain valid** and releases all its locks
Invisible reads
(more details)

- Every object $O$, with state $s(O)$ (register), is protected by a lock $l(O)$ (c&s).

- Every transaction maintains, besides $wSet$ and $wLog$:
  - a local variable $rset(O)$ for every object.
Invisible reads

Upon \texttt{write}(v) on object O
if O outside wSet then
wait until unlocked = l(O).c&s(unlocked,locked)
wSet = wSet U O
wLog = wLog U S(O).read()
(*,ts) = S(O).read()
S(O).write(v,ts)
return ok
Invisible reads

Upon \texttt{read()} on object O
(v,ts) = S(O).read()
if O in wSet then return v
if l(O) = locked or not validate() then abort()
if rset(O) = 0 then rset(O) = ts
return v
Invisible reads

Upon `validate()`
for all O s.t. rset(O) > 0 do
  `(v,ts) = S(O).read()`
  if ts not rset(O) or
      (O outside wset and I(O) = locked)
  then return false
else return true
Invisible reads

Upon `commit()`

s := validate()

for all O in wset do
  (v,ts) = S(O).read()
  S(O).write(v,ts+1)
  cleanup()

if s then commit() else abort()
Invisible reads

Upon `rollback()`
for all O in wSet do S(O).write(wLog(O))
wLog = empty

Upon `cleanup()`
for all O in wset do l(O).c&s(locked,unlocked)
wset = empty
rset(O) = 0 for all O
DSTM

- Killer write (ownership)
- Careful read (validation)
Performance figures look good
“It is better for Intel to get involved in this [Transactional Memory] now so when we get to the point of having ...tons... of cores we will have the answers”

Justin Rattner, Intel Chief Technology Officer
“...we need to explore new techniques like transactional memory that will allow us to get the full benefit of all those transistors and map that into higher and higher performance.”

*Bill Gates, Businessman*
“...manual synchronization is intractable... transactions are the only plausible solution....”

Tim Sweeney, Epic Games
The TM Topic is VERY HOT

- Sun, Intel, AMD, IBM, MSR, ...
- Fortress (Sun); X10 (IBM); Chapel (Cray)
All set?

Hmmm....
Tests

Micro-Benchmarks
- Linked-lists; red-black trees, etc.
- Consider specific loads: typically focus on read-only transactions
Challenging TMs

STMBench7 (GKV’07)

- **Large data structure**: challenge memory overhead
- **Short and long operations**: kills non-linear algorithms
- **Complex access patterns**
Performance figures were not that good

All TMs eventually collapsed because of memory usage (except X)
A new generation

SwissTM,
TL2,
TinySTM,...
Real-world scaling

Parallelization & synchronization require great care!

Speedup

User code

Multicore CPU

Time: Moore’s Law
Software Transactional Memory: Why is it only a Research Toy (CACM 2009)

C. Cascaval, C. Blundell, M. Michael, H. Cain, P. Wu, S. Chiras, S. Chatterjee
Why STM can be more than a Research Toy (CACM 2010)

A. Dragojević, P. Felber, V. Gramoli, R. Guerraoui
Wanted

Some principles
Transactional memory

1. Why do we care?
   Simplicity

2. What should we expect?
   What safety property?

3. What might we expect?
Transactional memory

Program

TM

Hardware
Safety of a TM

Let’s recall the old good atomicity property

Gray, Papadimitriou, Weihl, ..
Transactions and objects

Transactions invoke operations on shared objects

Every operation invocation is expected to return a reply

Every transaction is expected either to abort or commit
Transactions and objects
Transactions and objects

T1

operation

O1

commit

T2

operation

O2

abort

T3

operation

O1

operation

O2

commit
Transactions

Transactions are *sequential* units of computations

Transactions are *asynchronous*

(pre-emption, page faults, crashes)
The execution of a set of transactions on a set of objects is modeled by a **history**

A history is a **total order** of operation, commit and abort events

\[ H = (S, <) \]

The history depicts what the user sees
History H1

T1

read(0)

write(1)

commit

O1

O2

T2

read(0)

write(1)

commit

O2

O1
Histories

Two **transactions** are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise.

A **history** is **sequential** if it has only sequential transactions; it is **concurrent** otherwise.

Two histories are **equivalent** if they have the **same** transactions.
Sequential history \( H_2 \Leftrightarrow H_1 \)

\[
\begin{array}{c|c}
T_1 & \\
\hline
\text{read}(0) & \text{write}(1) \\
O_1 & O_2 \\
\end{array}
\]

commit

\[
\begin{array}{c|c}
T_2 & \\
\hline
\text{read}(0) & \text{write}(1) \\
O_2 & O_1 \\
\end{array}
\]

commit
The old theory (Pap 79)

A history is atomic if its restriction to committed transactions is serializable
A history $H$ of *committed* transactions is *serializable* if there is a history $S(H)$ that is

(1) equivalent to $H$
(2) sequential
(3) has every *read* returns the last value written
Atomic history?

T1

read(0)

O1

write(1)

O2

commit

T2

read(0)

O2

write(1)

O1

commit
Sequential history?

T1

read(0) write(1)

O1 O2

T2

read(0) write(1)

O2 O1
Sequential history?

T1

read(0) write(1)

O1 O2

T2

read(0) write(1)

O2 O1
Atomic history?

T1

```
read(0)
```

```
write(0)
```

```
O1
```

```
O2
```

commit

T2

```
read(0)
```

```
write(1)
```

```
O2
```

```
O1
```

commit
Sequential history

T1

read(0)  write(0)

O1  O2

T2

read(0)  write(1)

O2  O1
A history $H$ of *committed* transactions is serializable if there is a history $S(H)$ that is (1) equivalent to $H$ (2) sequential (3) has every read returns the last value written
Atomic history

T1

read(0)  write(1)

O1  O2

commit

T2

read(0)  write(1)

O2  O1

abort
A history $H$ of committed transactions is serializable if there is a history $S(H)$ that is
(1) equivalent to $H$
(2) sequential
(3) has every read return the last value written
There is more to shared objects than read/write registers, e.g., queues, compare&swap, counters, etc.

All these objects have a sequential specification (Weihl)
Sequential specification of a register

Sequential specification

read()

return(x)

write(v)

x <- v;

return(ok)
A queue has two operations: enqueue() and dequeue().

A queue internally maintains a list $x$ which exports operation appends() to put an item at the end of the list and remove() to remove an element from the head of the list.
Sequential specification

**dequeue()**
- if(x=0) then return(nil);
- else return(x.remove());

**enqueue(v)**
- x.append(v);
- return(ok)
A sequential history is **legal** if each restriction to an object belongs to its **sequential specification**
Legal history

read(0)  write(0)
O1       O2

read(0)  write(1)
O2       O1
Legal history

\[
\begin{array}{c|c}
\text{enq}(a) & \text{deq}(\text{nil}) \\
O1 & O2 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{enq}(b) & \text{deq}(a) \\
O2 & O1 \\
\end{array}
\]
A history $H$ of committed transactions is serializable if there is a history $S(H)$ that is
(1) equivalent to $H$
(2) sequential
(3) legal
Real-time

T1

write(1)

O1

commit

T2

read(0)

O1

commit
Histories

Two histories are *equivalent* if they have the *same* transactions

Two histories are *strictly equivalent* if they have the same transactions in the same order
Atomicity

A history $H$ of committed transactions is strictly serializable if there is a history $S(H)$ that is

1. strictly equivalent to $H$
2. sequential
3. legal
Is classical atomicity enough?
ARREST MEMORABLE DU
Parlement de Tolosse:

Contenant,
Une histoire prodigieuse, de nostre temps, avec
cent & onze belles, & doctes annotations,
dont les onze ont ete nouvellement adjou-
stees, sur le procès de l’exécution drud. Arret.

Par Monsieur M. Jean de Coras, Conseiller en lad-
ite Cour, & rapporteur du procès.

Prononcé et arrêté Generaux, le 111.
Septembre M. D. L. X.

ITEM,
LES DOUZE REIGLES
du Seigneur Iean Pic, de la Mirandol-
le, lesquelles adressent l’hôme au com-
bat spirituel : traduites de Latin en
Français par ledit de Coras.

Lyon,
PAR ANTOINE VINCENT,
M. D. L. X.

Avec privilege du Roy.
**DSTM**

- To **write** O, T requires a **write-lock on** O; T aborts T’ if some T’ acquired a write-lock on O.

- To **read** O, T checks if all objects read remain valid - else abort.

- At commit time, T checks if all objects read remain valid and releases all its locks.
DSTM

- Killer write (ownership)
- Careful read (validation)
More efficient algorithm

Apologizing versus asking permission

- Killer write
- Optimistic read
  - validity check only at commit time
Example

Invariant: $0 < x < y$
Initially: $x := 1; y := 2$
Division by zero

T1: x := x+1 ; y:= y+1

T2: z := 1 / (y - x)
Infinite loop

T1: x := 3; y := 6

T2: a := y; b := x;
repeat b := b + 1 until a = b
The old theory restricts committed transactions

We need a theory that restricts all transactions: this is what critical sections give us
Requirement: every operation sees a consistent state

How can we capture that precisely?
Histories

Let H be any history (made of committed, aborted and pending transactions)

$\textbf{Complete}(H)$ is the history made of all transactions of H by removing all pending and aborted ones, except the last one, completed with a commit event
A history $H$ is opaque if every prefix $H'$ of $H$ has a complete($H'$) which is strictly serialisable.
Opacity?

T1

read(0)
O2

write(1)
O1

commit

T2

write(1)
O2

read(0)
O1

abort
Illegal

T1
read(0)write(1)
O2 O1

commit

T2
write(1)read(0)
O2 O1
Illegal

T1

commit

T2

write(1)

read(0)

O2

O1

write(1)

read(0)

O2

O1
Recoverable (no dirty reads)

T1

read(0)

O2

write(1)

write(1)

O2

O1

T2

read(0)

O1

commit

abort
Opacity < rigorous scheduling

T1

write(0)
O2

commit

write(0)
O1

T2

write(1)
O2

write(1)
O1

abort
Most TMs ensure Opacity
Simple algorithm (DSTM)

- Killer write (ownership)
- Careful read (validation)
**Visible Read**
(SXM; RSTM)

- **Write is super killer:** to write an object, a transaction aborts any live one which has read or written the object.

- **Visible but not so careful read:** when a transaction reads an object, it says so.
Visible Read

A visible read invalidates cache lines

For read-dominated workloads, this means a lot of traffic on the bus between processors

This would reduce the throughput
Theorem (GK’08)

The read is either visible or careful

NB. Modulo a weak progress property and the assumption of a single version system
Intuition of the proof

T1

\[ \text{read()} \]
\[ O_1, O_2, \ldots, O_n \]

T2

\[ \text{write()} \]
\[ I_1, I_2, \ldots, I_m \]

commit

T1

\[ \text{read()} \]
\[ I_k \]
The fact that the read is invisible means T1 cannot inform T2, which would in turn abort T1 if it accessed similar objects (SXM, RSTM)
The theorem does not hold for classical atomicity

i.e., the theorem does not hold for database transactions
ARRÊT MEMORABLE DU
Parlement de Toulouse:
Contenant
Une histoire prodigieuse, de notre temps, avec cent & onze belles, & dökès annotations, dont les onze ont été nouvellement adjointes, sur le procès de l'exécution dud. Arrêt.
Par Monsieur M. Jean de Coras, Conseiller en ladis Cour, et rapporteur du procès.
Prononcé et arrêté Generaux, le 111.
Septembre, M. D. L. X.

LES DOUZE REIGLES
du Seigneur Jean Pic; de la Miramonde, lesquelles adressent l'hôte au combat spirituel; traduites de Latin en François par le dit de Coras.

À LYON,
PAR ANTOINE VINCENT,
M. D. LV.
Avec privilege du Roy.
Costes Durne (1565)
How can we verify the opacity of a TM?

Check that the conflict graph is \textit{acyclic}

- Number of nodes is unbounded
- NP-Complete problem
Reduce the verification space

*Uniform* system

- All transactions are treated equally
- All variables are treated equally
TM verification theorem (GHS’08)

A TM either violates opacity with 2 transactions and 3 variables or satisfies it with any number of variables and transactions.
A finite-state transition system (12,500 states) generates all opaque histories for 2 transactions and 3 variables.

A TM is correct if its histories could be generated by the reference implementation.

Simulation relation between the TM (e.g., TL2 4500 states) and the reference implementation.
Examples

- It takes 15mn to check the correctness of TL2 and DSTM

- Reverse two lines in TL2: bug found in 10mn - a history not permitted by the reference implementation
Transactional memory

1. Why do we care?
   Simplicity

2. What should we expect?
   Opacity

3. What might we expect?
   What progress?
What might we expect?

Program
T1/T2/..../Tn

Block

Abort

TM
We want progress

Operations return

Transactions commit
Nevertheless

We cannot require from a TM that it commits transactions:

- from a **dead** process; i.e., a dead transaction
- that infinitely **loop**
Progress?

T1: read(0) read(0) read(0) read(0)
   O2   O2   O2   O2

T2: read(0)
   O2

? crash
We can only hope progress for *correct* transactions

But what is a *correct* transaction exactly?
Correctness depends on the scheduler and the application

- Application: R/W/C/A
- Scheduler: TM
- TM: R/W/C&S/T&S/LL&SC/C/A
History

A history (as seen by the user) does not say what the scheduler does.

We need a refined notion of history.
Low-level history

A low-level history depicts the events of the implementation

It is also a total order of invocation, reply, and termination events

\[ H = (S, <) \]
Low-level history

The invocations and replies include also low-level objects used in the implementation.

The low-level history is a refinement of the high-level one (seen by the user).
Low-level history

Well-formed (low-level) history:
Every transaction that aborts is immediately repeated until it commits, i.e.,:

Every process executes:
T1:op1; T1.op2; ..; T1:Commit?; T1:Abort; T1:op1;.. ...
A transaction T is **correct** if

(a) *commit* is invoked after a finite number of invocation/reply events of T and

(b) either T *commits* or T performs an infinite number of (low-level) steps

(a) depends on the **application**

(b) depends on the **scheduler**
Ideally

Every *correct* transaction *commits*
Aborting is a fatality

T1

read()

O1

write()

O2

T2

write()

O1

read()

O2

commit

abort
Eventual progress
- wait-freedom -

- Every *correct* transaction *eventually commits*

- NB. We allow the possibility for a transaction to abort a finite number of times as long as it eventually commits
Eventual progress

T1
- read()
- O1
- write()
- O2

T2
- write()
- O1
- read()
- O2

commit

abort
Eventual progress

**Impossible** in an asynchronous system

NB. This impossibility is fundamentally different from FLP: It holds for any underlying object
Conditional progress - obstruction-freedom -

A correct transaction that eventually does not encounter *contention* eventually commits

*Obstruction-freedom* is indeed possible
DSTM

- To **write** O, T requires a **write-lock on** O (use C&S); T aborts T’ if some T’ acquired a write-lock on O (use C&S)

- To **read** O, T checks if all objects read remain valid - else abort (use C&S)

- Before committing, T releases all its locks (use C&S)
DSTM uses C&S

*C&S* is the strongest synchronization primitive

Is OF-TM possible with less than C&S? e.g., R/W objects
OF-TM

Program
R/W/TC/A

Scheduler

TM

Low-level objects?
Consensus number of OF-TM?

<table>
<thead>
<tr>
<th></th>
<th>Compare&amp;Swap</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>..</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Queue</td>
<td>Test&amp;Set</td>
</tr>
<tr>
<td>1</td>
<td>Register</td>
<td>Snapshot</td>
</tr>
</tbody>
</table>
FO-consensus

A process can decide or *abort*

- No two different values can be decided
- A value decided was proposed

If *abort* is returned from propose(v) then (1) there is contention and (2) v cannot be returned
OF-TM <=> FO-consensus

- From OF-TM to FO-consensus: \textit{propose()} is performed within a transaction

- From FO-consensus to OF-TM: slightly more tricky - as for DSTM but using a one shot object instead of C&S
Consensus

propose(vi) returns a value vj (no abort)

- No two different values can be decided
- A value decided was proposed
OF-consensus vs consensus

OF-consensus can implement consensus among exactly 2 processes

**Algorithm**

P1 writes its value and keeps proposing until it decides a value

P2 either decides or reads the value
Computability

The consensus number of OF-TM is 2

- OF-TM cannot be implemented with R/W
- OF-TM does not need C&S
Transactional memory

1. Why do we care?
   Simplicity

2. What should we expect?
   Opacity

3. What might we expect?
   Obstruction-freedom
Those are my principles
If you don’t like them
I have others

G. Marx
What opacity in the jungle?
Two ways compatibility (GHKS10)
What progress beyond OF?
Boosting obstruction-freedom

OF-TM

CM
Contention managers

- **Aggressive**: always aborts the victim

- **Backoff**: wait for some time (exponential backoff) and then abort the victim

- **Karma**: priority = cumulative number of shared objects accessed – work estimate. Abort the victim when number of retries exceeds difference in priorities.

- **Polka**: Karma + backoff waiting
Greedy contention manager

State
- Priority (based on start time)
- Waiting flag (set while waiting)

**Wait** if other has
- Higher priority AND not waiting

**Abort** other if
- Lower priority OR waiting
Off-line scheduler (GHP’95)

Compare the TM protocol with an off-line scheduler that knows:

- The starting time of transactions
- Which objects are accessed (i.e., conflicts)
Let $s$ be the number of objects accessed by all transactions.

Compare time to *commit all transactions*.

Greedy is $O(s)$-competitive with the off-line scheduler.
- GHP’05 $O(s^2)$
- AEST’06 $O(s)$
What progress beyond OF?
The weakest CM-FD to implement WF-TM (GKK’06)
Eventual global progress - lock-freedom -

Some *correct* transaction *eventually commits*

NB. OSTM ensures eventual global progress

Eventual global progress is the strongest liveness property that can be ensured by an STM
**Permissiveness (GHS’08)**

A TM is permissive if it never aborts when it should not.
Permissiveness

Let $P$ be any safety property and $H$ any $P$-safe history prefix of a deterministic TM.

We say that a TM is permissive w.r.t $P$ if

- Whenever $<H;\text{commit}>$ satisfies $P$
- $<H;\text{commit}>$ can be generated by the TM
Permissiveness

No TM can be *permissive* with respect to *opacity* (or *serializability*)
Let P be any safety property and H any history generated by a TM.

The TM is **probabilistic permissive** with respect to P if:
- Whenever \( <H;\text{commit}> \) satisfies P:
- \( <H;\text{commit}> \) can be generated by TM with a **positive probability**.
There is a probabilistically permissive TM with respect to opacity: **AVSTM**

AVSTM should *outperform* all TMs

In theory...
Probabilistic permissiveness

**AVSTM** indeed outperforms all TMs under very *high contention*

**AVSTM** does not perform well under *low contention*

**AVSTM** combined with a pragmatic TM:
- **TL2** under normal mode and then fallback to **AVSTM**
Transactions are conquering the parallel programming world

They sound familiar and thus make the programmer happy

Getting them correct is in fact tricky and that should make YOU happy
ARRESTATE BEMORABLE
du Parlement de Tolose:

Contenue

Une histoire prodigieuse, de notre temps, avec cent onze belles, & doctes annotation, dont les onze ont esté nouvellement adjou-

stées, sur le procès de l’exécution dudit Arret.

Par Monsieur M. Jean de Coras, Conseiller en ladis-

ite Cour, & rapporteur du procès.

Prononcé & arrêté Generaux, le xii.

Septembre M. D.L.X.

LES DOUZE REIGLES

du Seigneur Jean Pic, de la Mirandole, lesquelles adressent l’homme au combat spiritual : traduites de Latin en

François par ledit de Coras.

A Lyon,

PAR ANTOINE VINCENT.

M. D. L.XV.

Avec privilege du Roy.

Coste Durand [Signature].
Biblio

lpdwww.epfl.ch

Transactions@epfl