

TEL-AVIV UNIVERSITY
RAYMOND AND BEVERLY SACKLER
FACULTY OF EXACT SCIENCES
SCHOOL OF COMPUTER SCIENCE

A Transactional Consistency Clock Defined and Optimized

Dissertation submitted in partial fulfillment of the requirements for the M.Sc.
degree in the School of Computer Science, Tel-Aviv University

by

Hillel Avni

The research work for this thesis has been carried out at Tel-Aviv University
under the supervision of Prof. Nir Shavit

May 2009

Abstract

A crucial property required from software transactional memory systems (STMs) is that transactions, even ones that will eventually abort, will operate on consistent states. A known technique for providing this property is through the introduction of a globally shared version clock whose values are used to tag memory locations. Realizing global clock is the root of significant overhead, this thesis presents TLC^1 , the first thread-local clock mechanism for allowing transactions to operate on consistent states. TLC is the proof that one can devise coherent-state STM systems without a global clock.

Existing optimized global clock implementations: GV4, GV5 and GV6 are examined and a new one, GV7 is proposed.

Another contribution of this thesis is the definition of the abstract data type which has the minimal characteristics that are required to maintain STM consistent states. We name the object transactional consistency verifier (TCV) and prove all the clocking schemes demonstrated are TCV implementations.

A set of benchmarks is presented within the context of the TL2 STM algorithm, using TLC, and is compared to known global clock schemes and our new GV7. Our benchmarks show TLC perform as well as a global clock on small scale machines.

The big promise of the TLC approach is in providing a decentralized solution for future large scale machines. On such machines, a globally coherent clock based solution is most likely infeasible, and TLC promises a way for transactions to operate consistently in a distributed fashion.

¹Hillel Avni, Nir Shavit: Maintaining Consistent Transactional States without a Global Clock. SIROCCO 2008

Contents

1	Introduction	1
2	Background	3
3	Thread Local Clocks (TLC)	7
4	Transactional Consistency Verifier	9
4.1	TL2 Algorithm with Isolated Clock	9
4.1.1	Optimized Clocking Scheme	9
4.1.2	TL2 Algorithm	10
4.2	TCV Definition and Correctness	11
4.2.1	TCV Formal Properties	11
4.2.2	Correctness of the TL2 Algorithm with TCV	12
4.2.3	TLC as a TCV Implementation	15
5	Empirical Performance Evaluation	17
6	Conclusion	23



Chapter 1

Introduction

The goal of multicore programming is to maximize parallelism and use all cores in the system simultaneously. This task was relatively easy when the number of cores was limited, and could be achieved by simple lock based, application specific techniques. As number of cores grows [19][18] and distributed programming becomes more common, larger amounts of code has to be ported to higher number of cores and the task becomes tricky and expensive.

The larger machines memory access is not uniform and they are combined of tightly linked cores in clusters, and slower inter cluster communication. Such architectures make centralized elements usage resource consuming while requiring the option for global cooperation.

STM is a set of techniques to keep data consistency while performing transactions in multi core systems. It is meant to keep multicore programming as simple as possible while producing efficient programs.

To make transactions safe and correct two types of consistency must be kept:

External consistency is making a transaction seem atomic. Missing this characteristic a transaction may see partial affects of other transactions, which will crash the system quickly.

internal consistency means a transaction always work on a coherent snapshot. Neglecting this may cause a transaction, to work on data that is not expectable. Thus, an innocent operation like $(x/(y-z))$ may yield a division by zero. So although only a doomed transaction has done some wrong calculations, the whole system has to deal with complicated exceptions.

The state of the art way for maintaining the above involves the introduction of a *global clock*.

Unfortunately, *global clock* requires frequent remote accesses and introduces invalidation traffic,

which causes a loss of performance even on small scale machines [6].

To overcome this problem, there have been suggestions of distributing the *global clock* (breaking the clock up into a collection of shared clocks) [13, 14], or of providing globally coherent clock support in hardware [15]. The problem with schemes that aim to distribute the *global clock* is that the cost of reading a distributed clock grows with the extent to which it is distributed, so it may solve hot synchronization point but increase invalidation traffic which is bigger performance inhibitive. The problem with globally coherent hardware clocks, even if such hardware modifications were to be introduced, is that they seem to be limited to small scale machines. In practice, these clock's granularity and accuracy will worsen when they will be required to service more processors and transactions, making them unscalable.

This thesis presents *TLC*, the first *thread-local* clock mechanism that allows transactions to operate on consistent states. The breakthrough *TLC* offers is in showing that one can support coherent states without the need for a global notion of time. Rather, one can operate on coherent states by validating memory locations on a *per thread* basis. *TLC* has the same access patterns as prior STMs that operate on inconsistent states [7, 17, 10, 16]: the only shared locations to be read and written are the tags associated with memory locations accessed by a transaction. This makes *TLC* a highly distributed scheme by its very nature.

The thesis looks into algorithms that use a global clock while minimizing accesses to it, and suggests a scheme that reads global clock only upon aborts to see the performance tradeoffs.

TLC and global schemes are leveraged to define *TCV* which accommodates all clock optimizations while maintaining STM correctness. All modes are proved to implement *TCV* and *TCV* is proved to allow STM consistency.

Chapter 2

Background

In 1993 Herlihy and Moss [3] defined a transaction as a serialized code that must be executed atomically. Their vision was to support short critical sections in hardware, by modifying cache coherency protocol. To interact with software they defined machine instructions for transactional load and store, validate and commit. These instructions accumulate to the system we call transactional memory (TM). They also pointed some of the basic problems which arise with the introduction of TM, orphan (doomed) transaction that can make system errors, mixing non transactional code with TM, and privatizing of memory that was accessed by TM. The worst limitation of this proposal was that it all occurred in primary cache which is small and can not answer all programmers' transactional needs.

As early as 1996 Shavit and Touitou [17] saw the need for a software tool that shall facilitate transactions for a multicore programmer and will be hardware independent, i.e will not utilize any specific hardware support. They provided non-blocking implementation of static transactions. They had transactions maintain transaction records with read-write information, access locations in address order, and had transactions help those ahead of them in order to guarantee progress.

In 2003 Herlihy et al developed DSTM [2]. It was obstruction free to avoid both complexity of lock-free algorithm, and known engineering problems of locks, i.e deadlock and convoying. DSTM was a proof of concept for dynamic obstruction free STM. However, two levels of indirection for acquired objects, kept performance as measured by Dice and Shavit, [27] low.

In the same year Harris and Fraser [10] created a lock-free STM implementation which kept the data per-transaction instead of per-object. DSTM had a relatively complex read-write struc-

ture called locator per object, while [10] have only an ownership record (*orec*) which points to a transaction.

ASTM [24] was an attempt to optimize DSTM by making readers release acquired objects, thus reducing their indirection level. Measuring the tradeoffs [24] found the eager acquiring of DSTM yields better performance as doomed transactions abort earlier. That's why it chose DSTM obstruction-free approach over [10] lock-free one.

As found in [27] it is the overhead of the STM implementations (measured, for example, by single thread performance cost) that limits their performance. The root of this is that shorter transactions are simply faster and less exposed to contention. Thus the trend to relax progress condition, which allows simplified algorithms continued when Ennals claimed transactional memory should not be obstruction free [7], as lock-freedom belongs to the distributed world and not to the multicore arena. This is true because when many cores are available, and task scheduler is on-chip, task switching can be suspended during critical sections, without causing any harmful priority inversion. In McRT [16] such suspension is done by a simple function call. Another reason is that users of single chip expect that hardware and software faults will halt their application, and these should be so rare they do not worth worrying about. Regarding convoying he said long transactions have to block to complete also in lock-free implementations, as there is no other way to make them seem atomic.

Non blocking property implies managing public records of transactions which adds a lot of cache traffic and reduces STM bandwidth [27, 7]. Since it was discovered to be unnecessary, later STM algorithms are lock based.

Ennals proposed an algorithm that keeps only the objects themselves in public memory while handling all transactional bureaucracy in private memory. Each object has a handle that can be its version if lowest bit is one, and a pointer to a private write descriptor otherwise.

Another lock based proposal was the STM of McRT [16] which is a part of McRT framework. It uses McRT scheduler to verify a thread in middle of transaction is never swapped out. McRT allocation is used to group objects in blocks according to their size. These blocks have headers, which are used to compute locations for locks. These locks may reside in a separate per-block table or in the data they control. The first approach saves memory and the second is more cache friendly. McRT uses the locking mechanism of STM to implement MCAS. This is a set of CAS operations that can be executed atomically. As every address is written MCAS avoids read-write sets management

and is more efficient when appropriate. It is also compliant with hardware transactional memory (HTM) and thus, when available, HTM can be used to even further optimize MCAS. An important insight of this work [16] is that read versioning performs an order of magnitude better than reader locking.

Until recently, STM algorithms [17, 7, 10, 16] allowed the execution of “zombie” transactions: transactions that have observed an inconsistent read-set but have yet to abort. These solutions ignored internal inconsistency. The reliance on an accumulated read-set that is not a valid snapshot [1] of the shared memory locations accessed can cause unexpected behavior such as infinite loops, illegal memory accesses, and other run-time misbehavior. Overcoming zombie behavior requires specialized compiler and runtime support, and even then cannot fully guarantee transactional termination [6].

Ideally all conflicts could be resolved in hardware by intercepting any external write to a read set as it happens. Alas, this implies large and complicated caches. Thus hybrid TMs [11, 5] are only using hardware for acceleration. Hybrid Transactional memory [5] introduces support for *internal consistency*, however, this approach requires semi visible readers and global conflict counter which cause cache traffic even for read only transactions.

To avoid the extensive verifications, still working only on coherent snapshot Reigel, Felber, and Fetzer [12] and Dice, Shalev, and Shavit [6] introduced a *global clock* mechanism as a means of guaranteeing that transactions operate on consistent states. Reigel et al LSA [22] allow working on any coherent snapshot of read values, and incorporate a contention manager for write-write conflicts, at the price of maintaining a history per location and complicated implementation. TL2 [6] forces snapshot existence at linearization point to demonstrate simplicity at the price of higher abort rate. Very performance oriented version of *global clock* STM is TinySTM [8] which is a synthesis of TL2, Saha et al [16] and Ennals [7]. It is write through, encounter locking, and dynamically tunable. SwissTM [23] uses eager write-write and lazy read-write conflict detection and two-phase contention manager.

Transactions in past STM systems [5, 17, 7, 10, 16] typically updated a tag in the lock-word or object-record associated with a memory location as a means of providing transactional validation. In the new *global clock* based STMs [6, 11, 20] instead of having transactions locally increment the tags of memory locations, they update them with a time stamp from the globally coherent clock.

Transactions use the *global clock* to provide consistency (recently given the name *opacity* [9]) by comparing the tags of memory locations being read to a value read from the *global clock* at the transaction's start, guaranteeing that the collected read-set remains coherent.

Chapter 3

Thread Local Clocks (TLC)

Here is how TLC works. As usual, a *tag* containing a time-stamp (and other information such as a lock bit or HyTM coordination bit) is associated with each transactional memory location. In TLC, the time-stamp is appended with the ID of the thread that wrote it. In addition, each thread has a *thread local clock* which is initially 0, and is incremented by 1 at the start of every new transaction. There is also a *thread local array* of entries, each recording a time-stamp for each other thread in the system. We stress that this array is local to each thread and will never be read by others.

Without getting into the details of a particular STM algorithm, we remind the reader that transactions in coherent-state STMs [6, 20, 11] typically *read* a location by first checking its associated tag. If the tag passes a *check*, the location is consistent with locations read earlier and the transaction continues. If it is not, the transaction is aborted. Transactions *write* a memory location by *updating* its associated tag upon commit.

Here is how TLC's *check* and *update* operations would be used in a transaction by a given thread i :

1. *Update(location)* Write to the location's tag my current transaction's new local clock value together with my ID i .
2. *Check(location)* Read the location's tag, and extract the ID of the thread j that wrote it. If the location's time-stamp is higher than the current time-stamp stored in my local array for the thread j , update entry j and abort my current transaction. If it is less than or equal to the stored value for j , the state is consistent and the current transaction can continue.

This set of operations fits easily into the global-clock-based schemes in many of today’s STM frameworks, among them McRT [20], TinySTM [8], or TL2 [6], as well as hardware supported schemes such as HyTM [5] and SigTM [11].

How does the TLC algorithm guarantee that a transaction operates on a consistent read-set? We argue that a TLC transaction will always fail if it attempts to read a location that was written by some other transaction after it started. For any transaction by thread i , if a location is modified by some thread j after the start of i ’s transaction, the first time the transaction reads the location written by j , it must find the associated time-stamp larger than its last recorded time-stamp for j , causing it to abort.

An interesting property of the TLC scheme is that it provides natural locality. On a large machine, especially NUMA machines, transactions that access a particular region of the machine’s memory will only ever affect time-stamps of transactions accessing the same region. In other words, the interconnect traffic generated by any transaction is limited to the region it accessed and goes no further. This compares favorably to *global clock* schemes where each clock update must be machine-wide.

The advantages of TLC come at a price: it introduces more false-aborts than a *global clock* scheme. This is because a transaction by a thread j may complete a write of some location completely before a given transaction by i reads it, yet i ’s transaction may fail because its array recorded only a much older time-stamp for j .

Chapter 4

Transactional Consistency Verifier

A transactional consistency verifier (TCV) can mark a set of locations. Furthermore, It can test a location to find another TCV marked it after him. When seeing a mark, it can tell if it is the newest mark it has seen.

TCV is the basis for STM as well as other concurrent and distributed problems where a thread needs to verify exclusion for a set of locations. Being a component of a system that may be fault tolerant, TCV is required to be wait free.

4.1 TL2 Algorithm with Isolated Clock

To provide the reader with a better intuition for the more abstract formal definitions presented later, the properties of a TCV are first outlined informally via the example of its usage in the context of the TL2 STM algorithm. We use an optimized version clock scheme implementation. This implementation is isolated to let the reader see it as an encapsulated entity.

4.1.1 Optimized Clocking Scheme

Herein is an implementation of GV5 [6] which is the an optimized clocking scheme for TL2. As a prerequisite we require a shared variable GV which is initialized to zero and holds the current global version. We define RV which is a snapshot of GV and is initialized to zero, and TAG which is just a local variable. We name the functions of GV5 with the names used later for TCV methods to let the reader see the relation between this concrete example and TCV.

-
1. **version-access(x)**: Read the version of x to TAG. If TAG is greater than current RV, increment GV, take a new snapshot and store it in RV, and return FLASE, otherwise return TRUE.
 2. **label-write(x)**: Set the version of x to be greater than GV.

4.1.2 TL2 Algorithm

We now describe how an STM executes a sequential code fragment that was placed within a transaction, using TCV as a versioning technique:

1. **Start (optional¹)**: Call *version-access* for each shared address in memory.
2. **Run through a speculative execution**: Execute a transformation of the transaction code where load and store instructions are mechanically augmented and replaced so that speculative execution does not change the shared memory's state. Locally maintain a *read-set* of addresses loaded and a *write set* of address/value pairs stored. This logging functionality is implemented by augmenting loads with instructions that record the read address and replacing stores with code recording the address and value to-be-written.

The transactional load first checks to see if the load address already appears in the write-set. If so, the transactional load returns the last value written to the address. This provides the illusion of processor consistency and avoids so-called read-after-write hazards.

A *version-access* is inserted after each original load. If the *version-access* fails, possibly the location has been modified after the current thread performed step 1, and the transaction is aborted. ²

3. **Lock the write set**: Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock. In case not all of these locks are successfully acquired, the transaction fails.

¹GV7 is about skipping this step

²Abort can be avoided by calling successful *version-access* for all read set or revert to older version of the address. This is implemented in LSA STM.[22]

-
4. **Validate the read-set:** Call *version-access* for each location in the read-set. In case the validation fails, the transaction is aborted. By re-validating the read-set, we guarantee that its memory locations have not been modified while previous steps were being executed.
 5. **Commit, label, and release the locks:** For each location in the write-set, store to the location the new value from the write-set and *label-write* and release the locations lock.

4.2 TCV Definition and Correctness

The above TL2 usage of a clocking scheme can be used as an intuition to what is TCV. Informally, the global version based TCV is using a shared counter that can be incremented or read by any number of asynchronous processes. If multiple agents try to increment it concurrently, at least one will succeed.

4.2.1 TCV Formal Properties

We proceed to a formal definition of the TCV properties. The following is a formal definition of a TCV for a system of processes numbered $1, \dots, n$. A TCV which permits n concurrent operations has $2n$ operation types, specifically *version-access_i(α)* and *label-write_i(α)*.

A *label-write* operation associates an input value α , taken from any domain D , with a label. We call α the *labeled-value* of operation *label-write*. In an STM the α would be an address of shared data. A *version-access_i(α)* returns a truth value by verifying consistency property defined in this section.

Before defining the TCV properties we present a conceptual framework. Assume that any threads program consists of these two operations, whose execution generates a sequence of elementary operations executions, totally ordered by precedes relation (denoted " \rightarrow "), as defined by Lamport [25]. The following:

$$V_i^1(\alpha_1) \rightarrow V_i^2(\alpha_2) \rightarrow L_i^1(\alpha_3) \rightarrow V_i^3(\alpha_4) \rightarrow L_i^2(\alpha_5) \rightarrow V_i^4(\alpha_6) \rightarrow V_i^5(\alpha_7) \rightarrow L_i^3(\alpha_8) \rightarrow \dots$$

Is an example of such a sequence for thread i , where $V_i^k(\alpha_j)$ denotes thread i k^{th} execution of *version-access* and $L_i^k(\alpha_j)$ denotes thread i k^{th} execution of *label-write*, both called with abstract parameter (α_j) . V and L serialization points create a sequence for all the threads in the system.

Here we use the *not before* ($A \dashrightarrow B$) which includes the states A precedes B and A is concurrent with B.

$$V_i^1(\alpha_1) \dashrightarrow V_j^1(\alpha_2) \dashrightarrow L_k^1(\alpha_3) \dashrightarrow V_l^1(\alpha_4) \dashrightarrow L_m^1(\alpha_5) \dashrightarrow V_n^1(\alpha_6) \dashrightarrow V_o^1(\alpha_7) \dashrightarrow L_p^1(\alpha_8) \dashrightarrow \dots$$

Is a sequence of elementary operations in a system with multiple threads.

Note that if $L_k^j(\alpha)$ is concurrent with $op_m^n(\alpha)$, i.e. either with the *label-write* or *version-access* of another thread, than the results are undefined. It is the role of the algorithm using TCV to prevent such cases.

The elementary operation executions of TCV must have the following properties:

Property 1. Consistency: $V_i^q(\alpha)$ returns false if $L_k^r(\alpha) \rightarrow V_i^q(\alpha)$ and $(i \neq k)$ and there is no $V_i^m(\beta)$ s.t $L_k^r(\alpha) \rightarrow L_k^j(\beta) \rightarrow V_i^m(\beta) \rightarrow V_i^q(\alpha)$.

Property 1 formalizes the idea that TCV must sense that an object was labeled last by a thread other than its local thread and that label is the newest one it saw.

Property 2. Progress: $V_i^q(\alpha)$ returns true if $V_i^r(\alpha) \rightarrow V_i^q(\alpha)$ and there is no $L_k^w(\alpha)$ s.t $V_i^r(\beta) \rightarrow L_k^w(\alpha) \rightarrow V_i^q(\alpha)$ and $(i \neq k)$.

Here we formalize the fact that if an object is validated twice and no labeling occurs between the two instances, the second one must succeed. It is necessary to define this property to avoid infinite validations of the same location i.e. live locks. This property is meant to preserve the solo progress liveness property [29] of TL2 with TCV.

In the proof of TL2 algorithm we will use only these two properties, so it will cover any implementation of TCV and not only the traditional global clock. Specifically it should cover local clock implementation of TCV, i.e. TLC.

4.2.2 Correctness of the TL2 Algorithm with TCV

We base the correctness of the algorithm on simple claims on the linearization of transactions. In the proof, we refer to the algorithm steps as they were defined in Section 4.1.2. Data-modifying transactions are linearized after step 3 of the algorithm, while reading-only transactions are linearized at step 1.

In this section, we show that successful transactions affect each other with respect to the order of their linearization points. We treat concurrently executing modifying transactions separately from

concurrent read-only/modifying ones. We prove the algorithm's correctness by showing that any transaction is not affected by ones linearized later in time, and **fully** aware of earlier transactions.

Write-Write Conflicts

Consider op_1 and op_2 , two successful modifying transactions executed on a shared data structure by separate threads t_1 and t_2 . We assume *w.l.o.g.* that t_1 was the first to execute step 3. We denote the read- and write-set as constructed in step 2 of the algorithm by $rs(op)$ and $ws(op)$.

In Lemma 1, we claim that the more recent transaction, op_2 , sees the other's data modifications as if they happened atomically, and in Lemma 2 we show that op_1 is not aware of the modifications done by op_2 .

If op_1 serialized before op_2 we write $op_1 \rightarrow op_2$.

Lemma 1. *For all addresses $x \in ws(op_1) \cap rs(op_2)$, if there was no successful transaction op' with $op_1 \rightarrow op_3 \rightarrow op_2$ and $x \in ws(op_3)$, then op_2 reads from x the same value op_2 wrote in it.*

Proof. Assume by way of contradiction that op_2 reads a different value. One of the following must be true:

1. op_2 read x before op_1 wrote to it.

Since op_1 needs x locked during steps 4 and 5 in the algorithm and since it was the first to execute step 3, we know that op_2 was in step 5 later than the time x was locked by op_1 while executing step 3. From the success of op_2 's validation in step 4, x must have been released by op_1 after performing $L_1(x)$.

In step 4, op_2 called $V_2(x)$. Therefore, we get that $L_1 \rightarrow V_2$, so V_2 returned false, according to property 1, which means that op_2 failed which is a contradiction.

2. There exists an op_3 that wrote into x later than op_1 and earlier than the time op_2 started step 3 in the algorithm.

It follows that op_3 executed step 4 earlier than op_2 . Since both op_1 and op_3 needed to lock x and op_1 did it first, we have that op_3 executed step 3 later than op_1 . Therefore, $op_1 \rightarrow op_3 \rightarrow op_2$, contradicting the lemma's predicate.

The above contradictions conclude the proof of the lemma. \square

Lemma 2. *For simplicity, we will assume that stored values are unique. If $x \in rs(op_1) \cap ws(op_2)$, then op_1 reads from x a different value than the one op_2 writes there.*

Proof. If op_1 reads the value written to x by op_2 , it does it while in step 2. op_2 cannot possibly execute step 5 earlier than op_1 does step 3, because $op_1 \rightarrow op_2$, which means that op_2 gets to step 4 later than op_1 . \square

Read-Write Conflicts

We now define op_2 as a successful reading only transaction. We define op_1 as taking effect before op_2 if step 3 is executed before op_2 starts step 2. In Lemma 3 we claim that op_2 sees the modifications of op_1 as if they took place atomically.

Lemma 3. *If $x \in ws(op_1) \cap rs(op_2)$ and there does not exist a successful transaction op' with $op' \rightarrow op_1$ that executes step 3 before op_2 executes step 1, then op_2 reads from x the same value op_1 has written to it.*

Proof. Assume by way of contradiction that op_2 returns a different value. One of the following must be true:

1. op_2 read x before op_1 wrote to it.

Impossible since op_2 started when op_1 was in step 4, holding the lock on x at least until the time it wrote to x in step 5. op_2 checks the lock and version before and after reading from x . If it read from x before op_1 wrote there, the first check must have found an occupied lock and the transaction would have failed.

2. There exists a successful transaction op_3 that wrote a value val_3 to x before op_2 read from it. From the lemma, op_3 either executes step 4 after op_2 does step 1, or $op_3 \rightarrow op_1$. In the first case, $L_3(x) \rightarrow V_2(x)$ and op_2 fails.

In the second case, op_3 executes step 3 earlier than op_1 , and since both must lock x , op_1 executes step 5 later. op_2 starts after op_1 did step 3, but since it succeeds, by the time it post-validates x , op_1 has released the lock overriding val_3 .

\square

4.2.3 TLC as a TCV Implementation

It is left to show TLC meets the TCV properties.

Algorithm 1 TLC scheme

Private variables:

CArray : array $[0, \dots, max - thread - id]$ of versions, initialized to 0

TLClock : local thread version, initialized to 0

Tag: Datum tag.

Private methods:

GET-LBL(X) : Get the tag for X.

SET-LBL(X,L) : Set the tag for X.

procedure version-access(α)

Tag := GET-LBL(α)

id := id from Tag

1: version := version from Tag

2: **if** CArray[id] < version

3: CArray[id] := version

4: **return** FALSE

return TRUE

end version-access

procedure label-write(α)

INC(TLClock)

SET-LBL(α , TLClock)

end label-write

Lemma 4. *TLC implementation maintains the TCV consistency property.*

Proof. Assume by way of contradiction that $L_1(\alpha)$ and $V_2(\alpha)$ are consecutive successful TCV-TLC operations in the system history. We define successful as returning true. At the point of $L_1(\alpha)$ Thread 2 held at most the $V = (\text{TLClock of Thread 1})$. Then $L_1(\alpha)$ wrote the α tag with $(V+1)$. Thus $V_2(\alpha)$ read $(V+1)$ at line 1, and compared it to at most V at line 2, returning *false* in line 4. □

Lemma 5. *TLC implementation maintains the TCV progress property.*

Proof. Assume by way of contradiction that $V_1^1(\alpha)$ and $V_1^2(\alpha)$ are consecutive TCV-TLC operations in the system history, and that $V_1^2(\alpha)$ failed. At the line 1 of $V_1^1(\alpha)$ thread 1 either hold a higher

version than α 's or updates to it in line 3. Thus, when $V_1^2(\alpha)$ starts, thread 1 already has an entry in CArray to validate α , a contradiction. \square

Chapter 5

Empirical Performance Evaluation

We will now present a set of benchmarks to gauge the impact of the new clocking schemes on STM performance. In particular we will use the TL2 algorithm. We are reducing global invalidation traffic at the cost of more computations and aborts. Thus we benefit on hardware platforms that keep traffic as local as possible while paying large delays for remote accesses. We foresee that when architectures will accommodate thousands of cores, these will reside in clusters that will be far apart.

We tested our algorithms on 3 different platforms:

1. A 128-way Enterprize T5140 ®Server (Maramba) machine, a 2-chip Niagara system. This machine has relatively high inter-chip coherence costs, which favors TLC. However, it has a problem that will be fixed in future processors: [28] CAS always ignores the L1 and in fact will invalidate the location from the L1 if it's found there. Therefore, locality of transactions is not reducing invalidation traffic dramatically, which makes the benefit of TLC less obvious.
2. A Sun E25KTM system, an older generation NUMA multiprocessor machine with 144 nodes arranged in clusters of 2, each of which sits within a cluster of 4 (so there are 8 cores per cluster, and 18 clusters in the machine), all of which are connected via a large and relatively slow switch. This machine is great to demonstrate TLC potential, but it is not state of the art.
3. A 32-way Sun UltraSPARC T1TM machine (N1). Single chip multi-core machine based on the Niagara architecture that has 8 cores, each supporting 4 multiplexed hardware threads.

Invalidation traffic on the N1 is not an issue and therefore we do not expect TLC to improve performance there.

We ran the following tests:

1. A Randomized work-distribution benchmark in the style of Shavit, Touitou and Dice [17][26]. The benchmark picks random locations to modify, in our case 4 per transaction. We control the locality of the accesses, and the percentage of stores overall. We ran this benchmark on Maramba, using different localities, write loads, and data base sizes. We show graphs of operations and of abort rates to see the interaction of the two factors.
2. A performance comparison of the TLC and GV7 algorithm to the original TL2 GV6 *global clock* on Maramba machine.

Our benchmark is the standard concurrent red-black tree algorithm, written by Dave Dice, taken from the official TL2 release. It was in turn derived from the `java.util.TreeMap` implementation found in the Java 6.0 JDK. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java TreeMap were derived from the Cormen et al [4].

3. Same as the first but with a hint (current reader version) given on read operations. It ran on N1 machine, compared to GV5 flavor of TL2 and more importantly to TLC without a hint.
4. Performance comparison of the TL2C algorithm to the TL2 algorithm on a Sun E25KTM system. Our benchmark is a synthetic work-distribution benchmark in the style of Shavit and Touitou [17]. The benchmark picks random locations to modify, in our case 4 per transaction, and has overwhelming fraction of operations within the cluster and a minute fraction outside it. This is intended to mimic the behavior of future NUMA multicore algorithms that will make use of locality but will nevertheless have some small fraction of global coordination.

The graph of an execution of small 1K nodes and large 100K nodes random array appears in Figure 5.1. To show that a dominant performance factor in terms of TLC is the abort rate, we plot its graph too.

First we show no stores, only loads benchmark. Here there are no aborts and the graph demonstrates two insights:

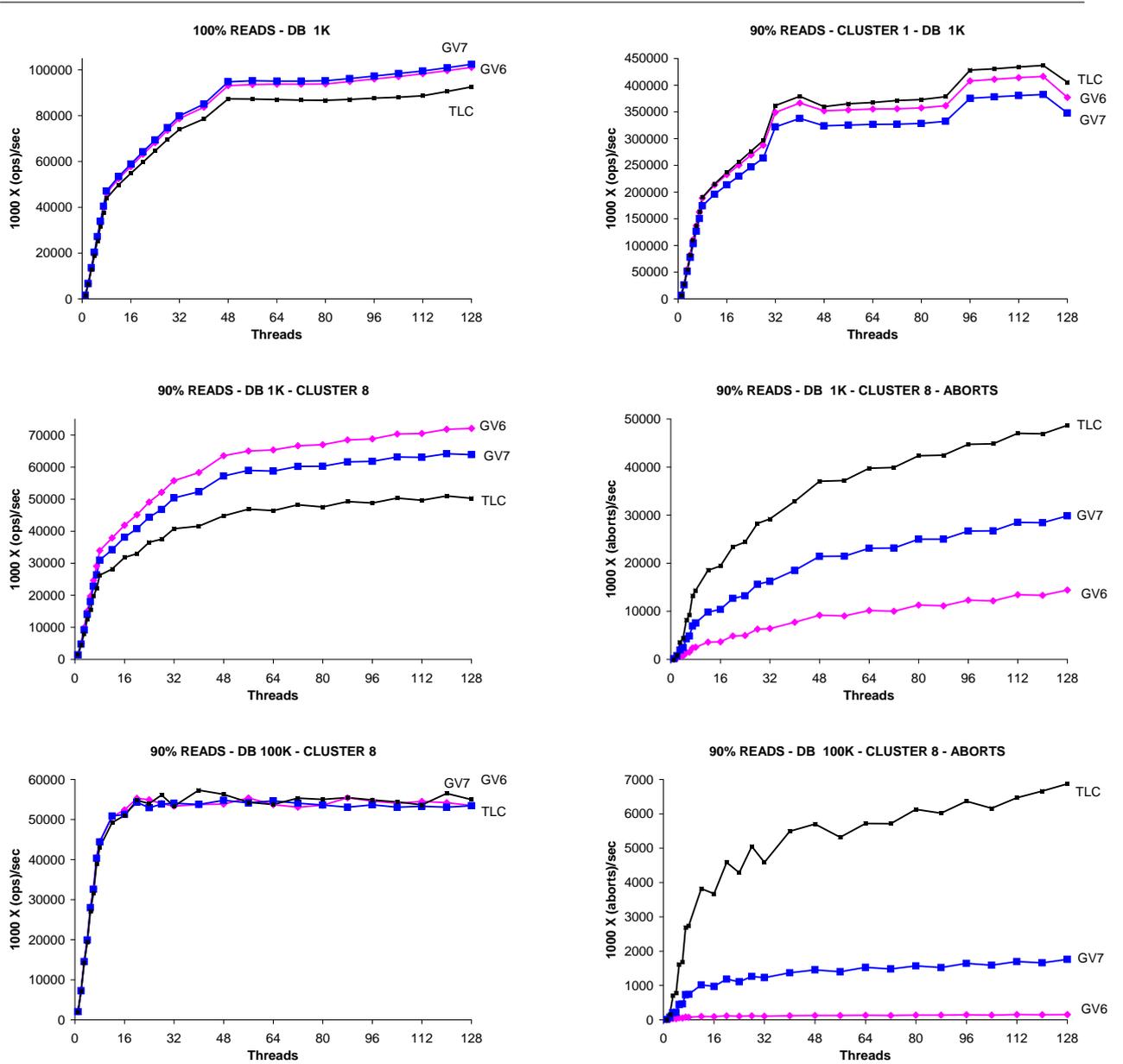


Figure 5.1: Throughput vs. abort rates of GV6, GV7 and TLC on small and large random arrays with different portions of stores.

- TLC is below GV6 and GV7 when there are no aborts. This implies the computational overhead of TLC is keeping it down.
- GV7 is slightly above GV6, which comes from the overhead of unnecessary reads of the global clock.

Then we present a benchmark where most accesses are local to each thread. TLC is optimized

to synchronize self clock in clocks array, thus it has no aborts here while other algorithms do. Detecting self made changes could be done without TLC but not as naturally and with some overhead.

with 10 percent stores, in large 100K nodes RB tree TLC is somewhat better than GV6 and GV7, because there are relatively few aborts comparing to operations number. On small trees (1000 nodes), TLC has poor performance because of high abort rate compared to operations.

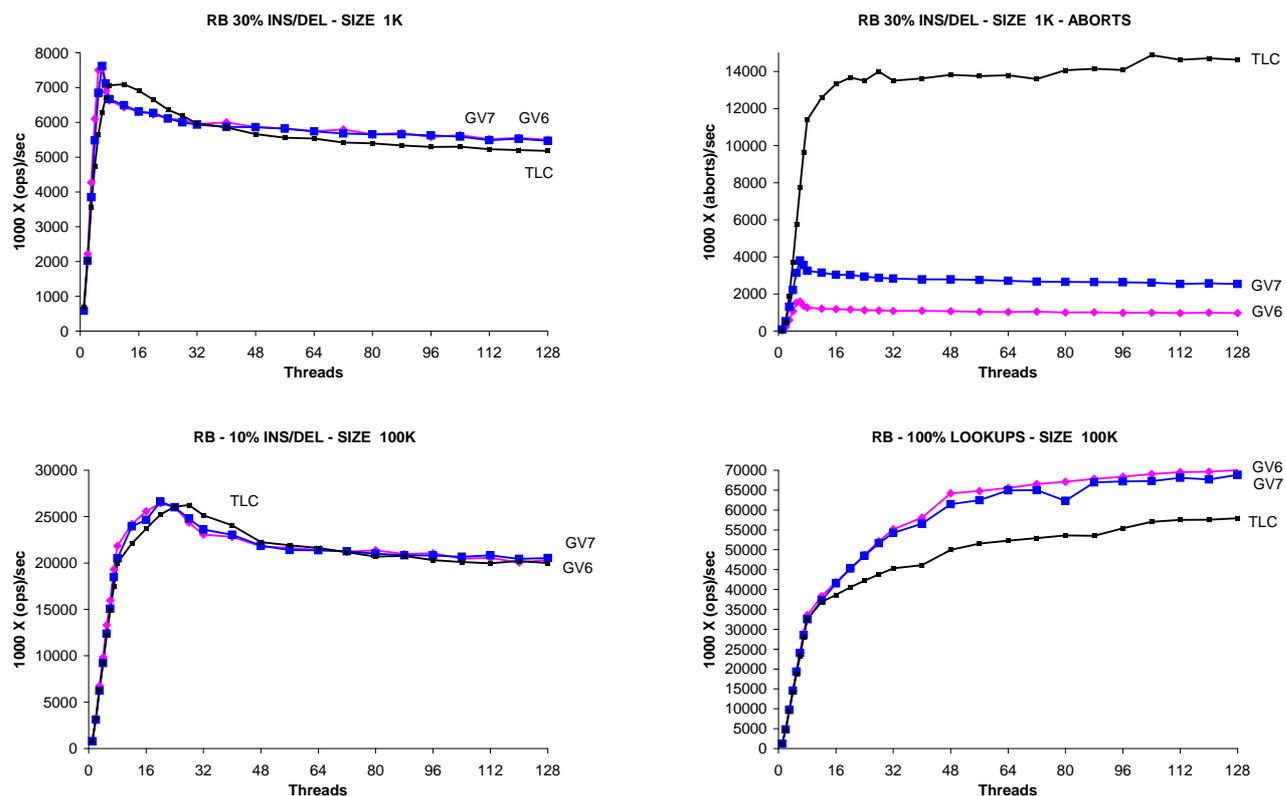


Figure 5.2: Throughput vs. abort rates of GV6, GV7 and TLC on small and large Red-Black Tree with with different portions of inserts/deletes.

Red black tree benchmark in figure 5.2 show the same behavior as random array for read only work load. TLC is down which means again it's computations slow it down. On workloads with inserts and deletes we see TLC, GV7, and GV6 all have about the same bandwidth even though TLC has a much higher abort rates. The only explanation is that the lower invalidation traffic of TLC improved its relative performance.

Figure 5.3 is again for small red black tree, but a hint is added to reduce false abort rate. The hint is an extra version added in the node that is written by each read operation. An aborting

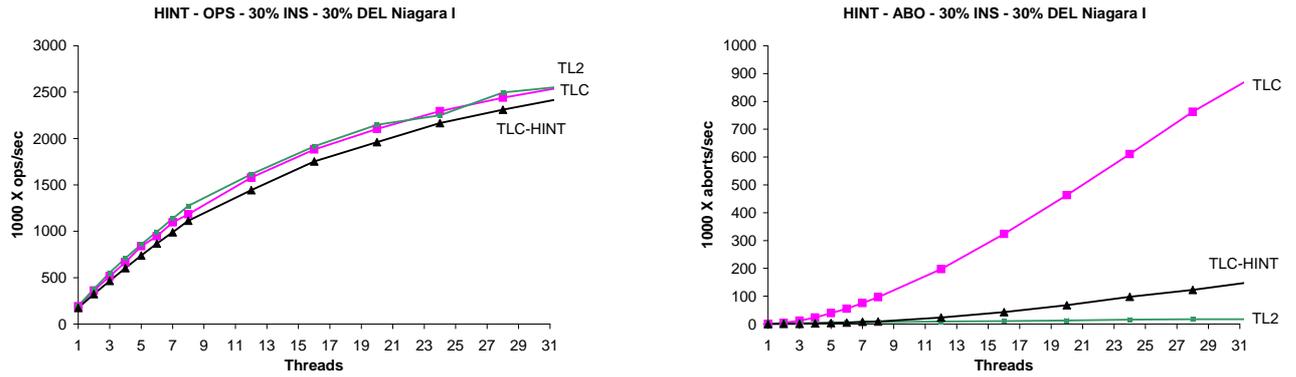


Figure 5.3: Throughput graph and abort rate graph of TL2, TL2C and TL2C with hint on a Red-Black Tree with 30% puts, 30% deletes.

thread is collecting this information and hopefully prevents itself from future false aborts. As can be seen on the right graph, abort rate decremented significantly, but, as can be seen on the left graph, performance slightly dropped. We tried many combinations like getting the hint for every read, writing the hint only on aborts, etc., and this combination is the only one that even though did not improve performance, at least reduced abort rate. The reason we did not see a performance improvement is the overhead for writing and reading the hints which is small by itself but accumulates over all operations in the benchmark.

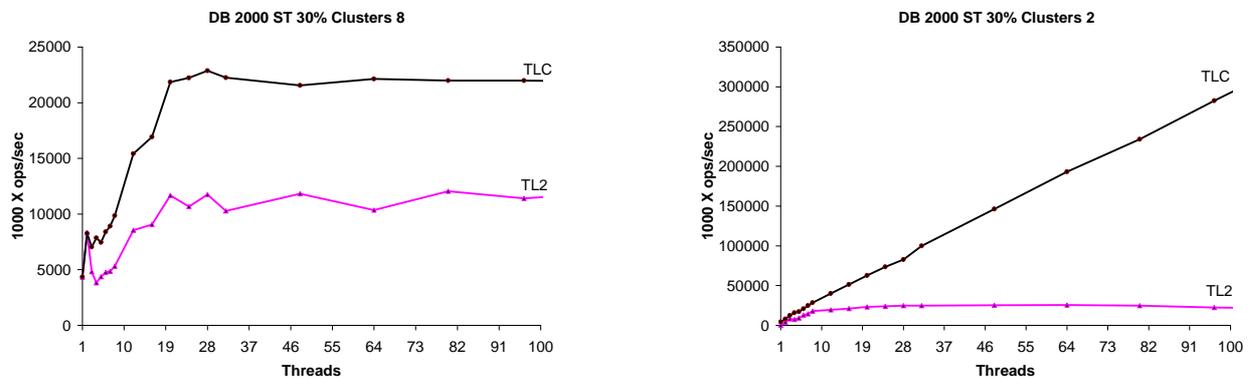


Figure 5.4: Throughput graph of GV6 and TLC on localized benchmark for clusters of 2 and 8 cores.

The graph in Figure 5.4 shows the performance of the artificial work-distribution benchmark where each thread picks a random subset of memory locations out of 2000 to read and write during a transaction, mimicking a pattern of access that has high locality by having an overwhelming

fraction of operations happen within a predefined cluster of nodes and a minute fraction outside it. As can be seen, the TLC algorithm has about twice the throughput of GV6 in clusters of 4 dual cores, despite having a high abort rate (not shown) as in the Niagara I benchmarks. On clusters of single dual core, TL2C scales almost perfectly while TL2 reaches the same performance it got on any other cluster size. The reason is that the cost of accessing the *global clock*, even if it is reduced by in relatively infrequent accesses in TL2's GV5 clock scheme, still dominates performance.

We expect the phenomena which we created in this benchmark to become prevalent as machine size increases. Algorithms, even if they are distributed across a machine, will have higher locality, and the price of accessing the *global clock* will become a dominant performance bottleneck.

Chapter 6

Conclusion

Maintaining transactional consistency with invisible readers requires time stamping all written data. This assignment until this thesis required centralized global data. We presented a novel decentralized local clock based implementation of the coherence scheme used in the TL2 STM. The scheme is simple, and can greatly reduce the overheads of accessing a shared location. It did however significantly increase the abort rate in the micro benchmarks we tested. Variations of the algorithm that we tried, for example, having threads give other threads hints, proved too expensive given the simplicity of the basic TLC mechanism: they reduced the abort rate but increased the overhead. The hope is that in the future, on larger distributed machines, the cost of the higher abort rate will be offset by the reduction in the cost that would have been incurred by using a shared *global clock*.

Another approach to reducing invalidation traffic is by avoiding as much as possible accessing the global clock. Removing redundant CAS operations was done in GV5 and GV6 [6]. We introduced GV7 which avoids redundant reads and improves performance when there is an overwhelming majority of transactional loads. GV7 concept is to reduce unnecessary reads, which are always the majority. It may become beneficial if we know how to reduce unnecessary reads in the general case.

To facilitate the development of new more efficient clocking schemes, the TCV abstract data type was defined with the minimal set of characteristics necessary to keep transactional consistency.

Bibliography

- [1] AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. Atomic snapshots of shared memory. *J. ACM* 40, 4 (1993), 873–890.
- [2] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. *PODC*, (2003), 92–101.
- [3] HERLIHY, M., MOSS, J.E.B. Transactional Memory: Architectural Support for Lock-Free Data Structures. *ISCA*, (1993), 289–300.
- [4] CORMEN, T. H., LEISERSON, CHARLES, E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press, 1990. COR th 01:1 1.Ex.
- [5] DAMRON, P., FEDOROVA, A., LEV, Y., LUCHANGCO, V., MOIR, M., AND NUSSBAUM, D. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM, pp. 336–346.
- [6] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking ii. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)* (2006), pp. 194–208.
- [7] ENNALS, R. Software transactional memory should not be obstruction-free. www.cambridge.intel-research.net/rennals/notlockfree.pdf, 2005.
- [8] FELBER, P., FETZER, C., AND RIEGEL, T. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2008).
- [9] GUERRAOU, R., AND KAPALKA, M. On the correctness of transactional memory, 2007.
- [10] HARRIS, T., AND FRASER, K. Language support for lightweight transactions. *SIGPLAN Not.* 38, 11 (2003), 388–402.
- [11] MINH, C. C., TRAUTMANN, M., CHUNG, J., McDONALD, A., BRONSON, N., CASPER, J., KOZYRAKIS, C., AND OLUKOTUN, K. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture* (New York, NY, USA, 2007), ACM, pp. 69–80.
- [12] RIEGEL, T., FETZER, C., AND FELBER, P. Snapshot isolation for software transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.
- [13] Dice, D., Moir, M., Lev, Y.: Personal communication (2007)

-
- [14] Felber, P.: Personal communication (2007)
- [15] RIEGEL, T., FETZER, C., AND FELBER, P. Time-based transactional memory with scalable time bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (Jun 2007).
- [16] SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., MINH, C. C., AND HERTZBERG, B. Mct-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2006), ACM, pp. 187–197.
- [17] SHAVIT, N., AND TOUITOU, D. Software transactional memory. *Distributed Computing* 10, 2 (February 1997), 99–116.
- [18] SUN MICROSYSTEMS, AND ADVANCED MICRO DEVICES. Tokyo institute of technology (tokyo tech) suprecomputer, Nov 2005.
- [19] SYSTEMS, A. Azul 7240 and 7280 systems, Jun 2007.
- [20] WANG, C., CHEN, W., WU, Y., SAHA, B., AND ADL-TABATABAI, A. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 34–48.
- [21] SHAVIT, N., AND DOLEV, D. Bounded Concurrent Time-Stamp Systems Are Constructible. *STOC*, (1989), 454–466.
- [22] RIEGEL, T., FETZER, C., AND FELBER, P. A Lazy Snapshot Algorithm with Eager Validation. *DISC*, (2006.)
- [23] A. D. RACHID GUERRAOUI, M. KAPALKA., AND FELBER, P. Stretching transactional memory. *PLDI*, (2009.)
- [24] MARATHE, V. J., SCHERER, W. N., AND SCOTT, M. L. Adaptive software transactional memory. *DISC*, (2005)
- [25] LESLIE LAMPORT The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, (1994), 872–923.
- [26] DICE, D., AND SHAVIT, N. TLRW: Return of the Read-Write Lock TRANSACT (2009)
- [27] DICE, D., AND SHAVIT, N. What Really Makes Transactions Faster? TRANSACT (2006)
- [28] Dice, D.: Personal communication (2009)
- [29] RACHID GUERRAOUI MICHAL KAPALKA How Live Can a Transactional Memory Be? POPL (2010)