Ensuring Irrevocability in Wait-free Transactional Memory

Jan Kończak
Institute of Computing Science,
Poznań University of Technology
Poznań, Poland
jan.konczak@cs.put.edu.pl

Paweł T. Wojciechowski
Institute of Computing Science,
Poznań University of Technology
Poznań, Poland
pawel.t.wojciechowski@cs.put.edu.pl

Rachid Guerraoui
EPFL
Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Abstract

Transactional Memory (TM) aims to be a general purpose concurrency control mechanism. But some operations are forbidden inside transactions, as they cause effects that the TM system cannot manage. Networking, I/O and some system calls cannot be executed in a transaction that may abort and restart. Thus, many TM systems let transactions become irrevocable, that is guaranteed to commit. Although support for irrevocability is a challenge, there exist TM systems that are fast, highly parallel and support irrevocability. However, no such system so far provides guarantees that all transactional operations finish in a finite time. In this paper, we show that support for irrevocability does not entail inherent waiting. We present an algorithm that guarantees wait-freedom for each transactional operation. The TM algorithm is based on the weakest synchronization primitive possible (test-and-set), and guarantees opacity and strong progressiveness. We develop upon it a TM system, and use it to experimentally evaluate our algorithm with the STMBench7 benchmark.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Software transactional memory, irrevocable operations support, operation-level wait-freedom

1. Introduction

Transactional memory (TM) [10, 14] is a concurrency control mechanism that has been proposed to simplify concurrent programming by allowing a sequence of read and write instructions, a *transaction*, to execute atomically. There has been a lot of interest in TM recently, and now TM is slowly entering the industry. For example, modern CPUs are equipped with instructions for executing short transactions (hardware TM). There also exist industry-strength implementations of software TM for popular programming languages, and standardizing committees work on appropriate standards. However, a lot more has to be done before TM becomes a versatile tool for programmers.

The advocated advantages of TM over explicit usage of locks include good support for multicore processors, since transactions can freely execute in parallel, and, in case of any conflicts, the conflicting transaction is simply rolled back and its code is executed again. But this behavior is also one of the weaknesses, since transactions that may potentially abort cannot contain arbitrary code. Thus, the designers of TM systems must take care of *irrevocable operations*, that is operations which cannot be undone, such as system calls, I/O actions, and networking operations.

Initially, the developers of TM systems simply forbid the use of irrevocable operations in transactional code at all. With rising need for a practical TM the irrevocability support began to emerge. Now, there exists a wide range of mechanisms to support irrevocable operations: from the ability to switch to sequential execution to sophisticated algorithms allowing transactions to run and commit alongside the irrevocable one. However, all of these approaches have a common disadvantage – in some cases one transaction, to complete its operation, must wait for another transaction to finish.

In this paper, we investigate the possibility of supporting irrevocable operations within transactions in *operation-level wait-free transactional memory* [5, 6] – a system in which no transaction should ever wait for another transaction. Implementation of a TM that supports irrevocability and requires no waiting is considered typically as not very realistic. We disprove the myth and show that a system like this in fact can be built with little cost. And, since no thread blocks other threads, by analogy to locks, an operation-level wait-free TM can be used instead of a traditional TM in similar cases when the `trylock` primitive is preferred over `lock`.

We focus on software TM since hardware TM alone typically does not offer any guarantees for programmers. Actually, we propose a wait-free TM algorithm, which is a rework of well-known ideas extended with routines providing seamless support for irrevocability. Then, we analyze the progress of our algorithm – the property that makes TM algorithms useful in practice. In general, a *progress property* asserts that it is always the case that some transaction eventually commits. In particular, we show that our TM system is *strongly progressive* [5], which means that two conditions hold: 1) a transaction that encounters no conflict must be able to commit, and 2) if a number of transactions conflict only on a single transactional variable, then at least one of them must be able to commit. Next, we present the results of experimental evaluation of a prototype wait-free TM system that we developed using the algorithm. To our best knowledge this is the first operation-level wait-free TM system that supports irrevocability.

1.1 Blocking transactions in TM

For a TM system to run fast, transactions should execute in parallel. But for correctness and simplicity, many TM algorithms involve periods when a transaction has exclusive access to particular data (e.g., during transaction commit) and other transactions are forced to wait until it completes. As this limits parallelism, some researchers investigated obstruction-free TM [11] – a class of TM that

TRANSACT '16, March 12, 2016, Barcelona, Spain.

guarantees completion of a transaction in a finite number of steps whenever no other transaction makes progress concurrently. While the idea is promising, when facing real-world problems a practical TM ensuring obstruction freedom is impossible. Programmers require the possibility to put arbitrary code within transactions, that is to support irrevocable transactions. However, irrevocability is incompatible with obstruction-freedom. A transaction running in isolation must commit, regardless if it conflicts with another transaction that temporarily stalled. If the latter is irrevocable, then neither of the conflicting transactions can abort.

Thus we focus on another property, which also introduces time constraints – wait-freedom [9]. It is stronger in terms of liveness than obstruction-freedom, since the latter leaves aside any guarantees for transactions that do not execute in isolation. According to the original definition, a method is wait-free if it guarantees that every call finishes in a finite number of steps. In the area of TM, wait-freedom is understood twofold: by some, as an impossible property that guarantees committing transactions in a finite time [13], by others as a property which guarantees finishing transactions in a finite time [5, 6]. We accept the latter, and to prevent confusion, call it *operation-level wait-freedom* [5]. The idea of operation-level wait-freedom is to limit the number of steps in which any transactional operation finishes. Thus a finite transaction finishes, either by aborting or by committing, in a finite number of its steps. Note that ensuring this alone is trivial – a TM system that aborts all transactions is operation-level wait-free. Therefore, for a practical TM, appropriate progress property must also hold.

1.2 Contribution

In the paper, we show that operation-level wait-freedom is achievable in a strongly progressive TM system with support for irrevocable transactions. To show this, we propose an operation-level wait-free TM algorithm that requires no extra assumptions and adds little overhead to support irrevocable transactions. Our algorithm uses only registers and *trylocks* (a test-and-set equivalent), which are known to be the weakest primitives that suffice for building opaque and strongly progressive TMs [5].

Next, we point out that common progress properties are infeasible in any system that both supports irrevocability and is operation-level wait-free. This is because the progress properties do not take into account irrevocability, which may introduce a conflict between transactions that share no data. Therefore, we propose a method of adapting progress properties to TM systems that support irrevocability and are operation-level wait-free. Once adapted, strong progressiveness is ensured by our algorithm.

To examine the usefulness of our approach, we developed an implementation of proposed algorithm, which we used to evaluated the behavior and performance of our algorithm. For this, we used the STMBench7 benchmark [7].

1.3 Paper structure

We discuss related work in Section 2. Then, we explain how to extend TM model to support irrevocable operations in Section 3. Next, we describe the design of our algorithm in Section 4. Then, we discuss the correctness and progress properties of the algorithm. Finally, we describe the implementation of our wait-free TM system and present the results of experimental evaluation in Section 6, followed by conclusions.

2. Related work

Baugh and Zilles [1] analyzed the use of irrevocable operations in critical sections of real-world applications (Firefox and MySQL). In particular, they took into account a subset of possibly irrevocable actions – I/O and system calls. They classified some of these operations as revocable, since they produce compensable side effects, and investigated the possibility of moving others outside transactions. For file system operations, the authors proposed extending operating systems by transactional I/O semantics. Nevertheless, they stated that these workarounds do not cover all irrevocable operations, and there still remains a significant number of truly irrevocable actions within the critical regions. This result is of a major importance for TM system designers. Since programmers do use irrevocable operations within critical regions, a versatile TM system must support them as well.

Zyulkyarov *et al.* [18] reimplemented Quake game server using TM as the main synchronization primitive. They stated that supporting irrevocable operations was necessary. Their TM of choice was Intel C++ STM Compiler [12], which allows for irrevocability by entering a serial mode – safe but inefficient idea.

Of course, there are more efficient solutions for supporting irrevocability. For example, all state-of-the-art TM systems that support irrevocably allow read-only transactions to run in parallel with an irrevocable transaction, as it is easy to ensure that they cause no harm. As for update transactions, less permissive approaches let the update transactions perform reads and writes, but disallow commit as long as any irrevocable transaction is running. Most advanced solutions allow all parallel transactions to progress and commit, thereby guaranteeing no aborts of the transaction running in the irrevocable mode. Below we discuss example approaches and TM systems of this sort. However, all of them have one disadvantage: in certain conditions, some transaction must wait for other transactions to progress. So, a slow transaction can postpone completion of other ones.

Single-Owner Read Locks (SORL) [17] is a highly-efficient optimized locking approach, in which each data item is guarded by a lock. While ordinary locks are designed to represent two states (locked and unlocked), SORLs let the transaction lock the data in one of three modes: normal, irrevocable read, and irrevocable write. This allows for both high level of parallelism and the support for irrevocability, but as in the traditional locking approach, to access any data item a transaction must wait for the item to become unlocked.

Inevitable Read Locks (IRL) [15, 16] exploit the same idea as SORL. In this approach, when irrevocable transaction T_i detects that its next operation would conflict with T_k, then T_i can cause T_k to eventually abort (while in SORL T_i waits until T_k finishes). Spear *et al.* did not specify whether such operation is blocking. Since making it non-blocking is a challenging problem not references in the papers, we assume that waiting is a must in IRL.
Spear *et al.* proposed also *Inevitable Read Filter* [15, 16], which aims for increasing parallelism. In this approach, a single global bloom filter is used to store read locks. This improves over IRL by alleviating any read-read concurrency problems, which may arise in the latter. However, accesses to the filter itself must be sequential, so when a single transaction updates or lookups the filter, other transactions must wait.

The work on hardware transactional memory is also relevant, as dealing with I/O operations is obviously an issue for hardware as well. As in software TM, hardware TM systems must abort transactions if a conflict is detected. Most of these systems simply disallow irrevocable operations. In state-of-the-art hardware TMs each invocation of an irrevocable operation within a transaction ends up with a forceful abort, just to prevent any upcoming problems. Blundell *et al.* [2] made an attempt to alleviate any limits on the operations used within a hardware transaction. To this end, they introduced a new mode for running transactions in hardware (called the unrestricted mode) that guarantees no aborts. This allows using unsafe operations within transactions, making the unrestricted mode analogue to irrevocable transactions in software TM. As for wait-freedom, there is no way to observe any hardware waiting from a program-

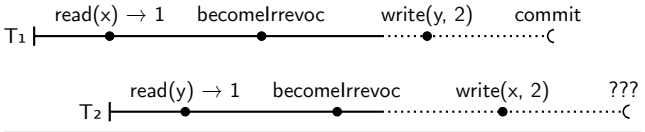

Figure 1. Two irrevocable trans. result in an unsolvable conflict

mers point of view. However, the implementation of a hardware TM, as described in [2], forces threads to stall upon concurrent access to common locations.

3. Irrevocability in TM

A typical transactional memory system defines the following operations: *begin transaction*, *read*, *write*, *abort* and *commit*. All these operations can succeed or fail by aborting the transaction. To support irrevocability, this set of operations must be extended. One of the ideas introducing support for irrevocable operations is to tag transactions using them (*irrevocable transactions*) upon start, thus extending the *begin transaction* operation by a parameter. We find this approach not flexible enough. While it allows for creating simpler TM algorithms, it requires from programmers to explicitly start an irrevocable transaction. Also, prior to the first irrevocable operation, the transaction is idly considered as irrevocable. Thus, we adapt a more popular idea on how to introduce support for irrevocability in TM, and let the transaction decide at any point of its execution that it wants to transit to an *irrevocable state*. For this purpose a new operation is defined: *become irrevocable*. The transition from revocable to irrevocable state can fail, that is a transaction can be forcibly aborted while executing the *become irrevocable* operation. However, if the operation succeeds, then no subsequent operation can end up with an abort.

In general, there can be at most one irrevocable transaction at a time. Otherwise, conflicts are unavoidable. As depicted in Figure 1, if any algorithm would let two arbitrary transactions transit to the irrevocable mode, and would let each of them read a (separate) variable, each transaction could request an update to the variable read by the other one, resulting in an unsolvable conflict. It is theoretically possible to let a transaction become irrevocable and stall it before its first read, but this effectively still lets only one irrevocable transactions progress.

Supporting irrevocability to some extent restricts the design of a TM system. Without irrevocability, read operations can be either *visible* or *invisible* – that is, transactions either can or cannot tell which transactions read a certain variable. When supporting irrevocability, no transaction must update a variable read by the irrevocable transaction until the latter finishes. Thus, either irrevocable transactions use visible reads, or extra synchronization is required.

4. Algorithm

To show that a TM supporting irrevocable operations can be operation-level wait-free, we propose an algorithm satisfying the following properties:

- operation-level wait-freedom
- support for irrevocability
- opacity (a safety property)
- strong progressiveness (a progress property)

The algorithm uses only registers and *trylocks*. A trylock is an object having two methods: *unlock* and *trylock*, with the usual locking semantics. The trylock can be trivially implemented using a test-and-set instruction, and there exists no weaker synchronization primitive that can be used to develop an opaque and strongly progressive TM system [5, 9]. We chose object-based approach (used e.g., in [8]).

Algorithm 1: Data structures

```
Global data
    maxThreadNum;
    irrTransactionLock ← unlocked;
Thread local data
    ct ← ⊥ ;                          // current transaction
    threadId;
class Transaction:
    rsetBuffs ← ∅;
    wsetBuffs ← ∅;
    hijackedBuffs ← ∅ ;               // used by irrevoc. to take over var.
    locksHeld ← ∅ ;                   // list of acquired locks
    cleanRsetL ← unlocked ;           // lets other trans. abort this one
    commitL ← unlocked ;              // lets irrevoc. trans. abort this one
    aborted ← false ;                 // tells if trans. has been aborted
    amIIrrevocable ← false;
class Variable:
    valuePtr ;                        // points to the global copy of the variable
    usedByIrr ← false ;               // prevents revocable trans. from writes
    dirty ← false ;                   // if set, a revoc. trans. updates the var.
    dirtyIrr ← false ;                // if set, an irrevoc. trans. updates the var.
    readers ← [⊥,⊥,...,⊥] ;           // trans. that read the variable
    lock ← unlocked ;
    mostRecentLockOwner ← ⊥;          //written to just after locking lock
```

The algorithm is presented in Algorithm 1, 2, 3 and 4. Since it relies upon indirection level, we use the following notation to express clearly the intended operations:

`CLONE`(ptr) makes a copy of an object pointed by ptr
$ptrA \leftarrow ptrB$ sets the value of pointer $ptrA$
$*objptr$ accesses the pointed object

For brevity of the algorithms, we use the following notation:

$c \cup\!\leftarrow o$ adds o to the set/map c (i.e., $c \leftarrow c \cup \{o\}$)
$c \setminus\!\leftarrow o$ removes o from the set/map c (i.e., $c \leftarrow c \setminus \{o\}$)
$\emptyset$ is an empty set/map
$\perp$ is a null (empty) value
$c[k]$ accesses object at given index/key k of array/map c

Without irrevocable transactions, our algorithm is a simple lock-based algorithm. It locks a variable upon write (line 22) and releases it at commit or abort (line 94 or 69). If, during write, try-locking the variable fails, the transaction aborts (line 22). Transactions work on local copies of the shared variables (line 10 and 34), and overwrite the global copy on commit (line 90). We assume that one thread can execute one transaction at a time, and there is a well-known upper bound on the number of threads. Under these assumptions visible reads are implemented in a lightweight fashion, by keeping a list of readers in a constant-sized array. Whenever a transaction is going to update the global copy on commit, it marks the variable as dirty (line 73). Then it prevents all transactions in the readers list from completing any subsequent read or commit (line 74-77). To enable the latter, on commit each transaction must lock one additional lock, which guarantees a consistent read set (line 79). Such algorithm is operation-level wait free, opaque and strongly progressive. It is not especially permissive (i.e., it aborts some transactions with read-write conflicts that potentially could commit), albeit is easily extensible for irrevocability support.

To support irrevocability, it must be ensured that a) no transaction can abort the irrevocable transaction, b) the irrevocable transaction must always successfully execute any TM operation.

Algorithm 2: Transactional read and write

```
   read(var)
1  |  if var ∈ (ct.rsetBuffs ∪ ct.wsetBuffs) then
2  |  |  return *(ct.rsetBuffs ∪ ct.wsetBuffs)[var];
3  |  if ct.amIIrrevocable then
4  |  |  irrAcquire(var, true);
5  |  |  return *(ct.rsetBuffs ∪ ct.wsetBuffs)[var];
6  |  var.readers[threadId] ← ct;
7  |  if var.dirty ∨ var.dirtyIrr then Abort;
8  |  ct.rsetBuffs ∪← (var, CLONE(var.valuePtr));
9  |  if ct.aborted then Abort;
10 |  return *ct.rsetBuffs[var];
   write(var, value)
11 |  if var ∈ ct.wsetBuffs then
12 |  |  *ct.wsetBuffs[var] ← value;
13 |  |  return *ct.wsetBuffs[var];
14 |  if ct.amIIrrevocable then
15 |  |  if var ∈ ct.rsetBuffs then
16 |  |  |  ct.wsetBuffs ∪← (var, ct.rsetBuffs[var]);
17 |  |  |  ct.rsetBuffs \← var;
18 |  |  else irrAcquire(var, false);
19 |  |  *ct.wsetBuffs[var] ← value;
20 |  |  return *ct.wsetBuffs[var];
21 |  if var.usedByIrr then Abort;
22 |  if ¬ var.lock.trylock() then Abort;
23 |  var.mostRecentLockOwner ← ct;
24 |  if var.usedByIrr then
25 |  |  var.lock.unlock();
26 |  |  Abort;
27 |  local buffer ← CLONE(var.valuePtr);
28 |  if ct.aborted then
29 |  |  var.lock.unlock();
30 |  |  Abort;
31 |  ct.wsetBuffs ∪← (var, buffer);
32 |  ct.rsetBuffs \← var;
33 |  ct.locksHeld ∪← var.lock;
34 |  *ct.wsetBuffs[buffer] ← value;
35 |  return *ct.wsetBuffs[buffer];
   irrAcquire(var, ro)
36 |  var.usedByIrr ← true;
37 |  if var.lock.trylock() then ct.locksHeld ∪← var.lock;
38 |  else
39 |  |  local lo ← var.mostRecentLockOwner;
40 |  |  if lo.commitL.trylock() then lo.aborted ← true;
41 |  |  else if ¬ lo.aborted then
42 |  |  |  ct.wsetBuffs ∪← (var, CLONE(lo.wsetBuffs[var]));
43 |  |  |  ct.hijackedBuffs ∪← (var, lo.wsetBuffs[var]);
44 |  |  |  return;
45 |  if ro then ct.rsetBuffs ∪← (var, CLONE(var.valuePtr));
46 |  else ct.wsetBuffs ∪← (var, CLONE(var.valuePtr));
```

Algorithm 3: Transiting to irrevocable state

```
   become irrevocable()
47 |  if ¬ irrTransactionLock.trylock() then Abort;
48 |  if ¬ acquireReadset() then
49 |  |  irrTransactionLock.unlock();
50 |  |  Abort;
51 |  if ¬ (ct.cleanRsetL.trylock() ∧ ct.commitL.trylock()) then
52 |  |  foreach v ∈ ct.rsetBuffs do v.usedByIrr ← false;
53 |  |  irrTransactionLock.unlock();
54 |  |  Abort;
55 |  ct.amIIrrevocable ← true;
   acquireReadset()
56 |  local acquired;
57 |  foreach v ∈ ct.rsetBuffs do
58 |  |  v.usedByIrr ← true;
59 |  |  if v.lock.trylock() then acquired ∪← v;
60 |  |  else
61 |  |  |  foreach v′ ∈ acquired do
62 |  |  |  |  v′.usedByIrr ← false;
63 |  |  |  |  v′.lock.unlock();
64 |  |  |  return False;
65 |  ct.locksHeld ∪← acquired;
66 |  return true;
```

To guarantee no aborts, the irrevocable transaction must never abort itself, as well as must never be aborted by others. For the latter, notice that revocable transactions can abort others only on commit (line 76). So, while transiting to the irrevocable state, a transaction must acquire the lock used to abort other transactions (line 51). However, extra care is required here: since the lock guaranteed consistent reads, just acquiring it would break consistency. Thus, on becoming irrevocable, the transaction locks variables it read before (line 59). Later on, upon any read, the variables are also locked to prevent concurrent writes (line 37). Once the variables accessed by the irrevocable transaction are locked, no concurrent writes may happen. Of course, locking the variable can fail. If the transaction is already irrevocable, we need to take over the variable. By taking over we understand either aborting the lock's owner or – if the lock's owner is commit-pending[1] – using the value it produced.

Taking over a variable by the irrevocable transaction regardless of the current system state is a challenge in an operation-level wait-free TM. If the variable is not currently locked, the irrevocable transaction simply locks it, regardless of the intended operation (read or write). It is worth pointing out that in our system reading a variable by a revocable transaction does not require it to be unlocked. Since the global copy is updated at commit, one transaction can perform a read between write (or read) and commit operation of an other transaction. Thus, locking the variable upon read by the irrevocable transaction introduces no read-read conflicts.

Regardless if the variable x is locked, as the first step of accessing it the irrevocable transaction T_i marks x as in use by the irrevocable transaction (line 36). This causes any new transaction to abort upon a write to x (lines 21, 24). This limits the number of transaction competing on x to at most two – T_i and T_k. It is easy to guarantee that either T_i correctly identifies T_k as the lock's owner (line 39), or T_k aborts (lines 23, 24)[2]. Then, T_i tries to force the supposed lock's owner to abort on any subsequent transactional operation. If either T_i successfully forces the abort or notices that T_k already aborted (line 40, 41), then T_i knows that it has exclusive access to x and that the global copy has the correct value.

There is, however, one case when T_i and T_k compete on the variable x and T_i is unable to abort T_k and T_k did not abort. This can happen iff T_k already started its commit and acquired all needed locks (that is, passed line 79). In such case T_k is commit-pending, and T_i can use values produced by it. For read operations, it is sufficient to return the value from buffer of T_k. As for writes, the solution is not so simple: T_k can at any time write to the global copy of x, as part of the commit procedure (line 83). Due to indirection level used by us, T_i can precisely tell what address T_i will write to the global copy – the address of its buffer (line 83). This makes it possible for T_i to use the buffer of T_k as its own (*hijack* it, lines 42, 43). Now, T_i at the end of its commit updates

[1] A transaction T_k is called *commit-pending* if T_k invoked commit and in all possible continuations of the current history T_k eventually commits.

[2] Or T_k stalls until T_i finishes, what is indistinguishable from a case when T_k issues the write after commit of T_i.

Algorithm 4: Start, abort and commit procedures

```
   begin transaction()
67 |  ct ← new Transaction
   abort()
68 |  ct.aborted ← true;
69 |  foreach lock ∈ ct.locksHeld do lock.unlock();
   commit()
70 |  if ct.aborted then Abort;
71 |  foreach v ∈ ct.wsetBuffs do
72 |  |  if ct.amIIrrevocable then v.dirtyIrr ← true;
73 |  |  else v.dirty ← true;
74 |  foreach v ∈ ct.wsetBuffs do
75 |  |  foreach i ∈ [1,2,3,...,maxThreadNum] \threadId do
76 |  |  |  if v.readers[i].cleanRsetL.trylock() then
77 |  |  |  |  v.readers[i].aborted ← true;
78 |  if ¬ ct.amIIrrevocable then
79 |  |  if ¬ (ct.cleanRsetL.trylock() ∧ ct.commitL.trylock()) then
80 |  |  |  foreach v ∈ ct.wsetBuffs do v.dirty ← false;
81 |  |  |  Abort;
82 |  |  foreach v ∈ ct.wsetBuffs do
83 |  |  |  v.valuePtr ← ct.wsetBuffs[v];
84 |  |  |  v.dirty ← false;
85 |  else
86 |  |  foreach v ∈ ct.wsetBuffs do
87 |  |  |  if v ∈ ct.hijackedBuffs then
88 |  |  |  |  *ct.hijackedBuffs[v] ← *ct.wsetBuffs[v];
89 |  |  |  |  v.valuePtr ← ct.hijackedBuffs[v];
90 |  |  |  else v.valuePtr ← ct.wsetBuffs[v];
91 |  |  |  v.dirtyIrr ← false;
92 |  |  foreach v ∈ (ct.rsetBuffs ∪ ct.wsetBuffs) do
93 |  |  |  v.usedByIrr ← false;
94 |  foreach m ∈ ct.locksHeld do m.unlock();
95 |  if ct.amIIrrevocable then irrTransactionLock.unlock();
```

the buffer of T_k to the value produced by T_i, and writes the buffer address to the global copy. T_k, on the other hand, writes the same buffer address to the global copy. Since both T_i and T_k want to perform the same write, no conflict can occur. If T_k finishes before T_i starts its commit, read operations on x can succeed, yielding the value produced by T_k. Once T_i finishes, it overwrites the buffer of T_k with the new value (line 88) and points the global copy to it (line 89). From that moment on, all reads will return the value of x as produced by T_i.

Since at most one transaction at a time can be irrevocable, we use a lock – $irrTransactionLock$ – to limit the number of irrevocable transactions (lines 47, 95).

5. Properties

5.1 Correctness

The algorithm we present is opaque [4]. In brief, the opacity is achieved by careful maintenance of the reader list and proper commit procedure. At commit of a transaction T_k the TM system aborts all revocable transactions that read variables updated by T_k. No revocable transaction is allowed to read a variable x while another transaction that wrote to x is in progress of commit. These rules guarantee that all reads of revocable transactions are consistent. As all reads are performed on the global copy and all updates are written to the global copy, real-time order is preserved. The irrevocable transactions protect themselves form inconsistent reads by locking all read variables, just as described in the previous section.

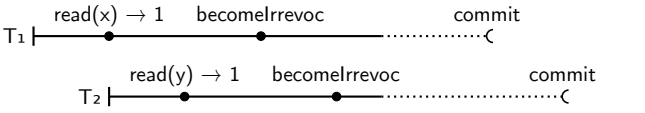

Figure 2. Disallowed execution of disjoint transactions

5.2 Progress

5.2.1 Existing properties versus irrevocability

It is easy to show that in an operation-level wait-free TM system with support for irrevocability some properties are impossible to achieve. Interestingly, in such a TM system no reasonable progress properties can hold. The definitions of the properties are not taking into account irrevocability, what renders applying them impossible. For example, it is not possible to guarantee that a transaction with no conflicts will always commit. Consider two concurrent transactions that do not share any variables, and they both try to become irrevocable (see Figure 2). While one of them can safely become irrevocable, the other one cannot finish the transition to irrevocable state in a wait-free manner, as at most one transaction can be irrevocable at the same time (see Section 3).

Intuitively, in the scenario above aborting one of the transactions should be allowed by progress properties. Thus, to be able to discuss the progress properties in presence of irrevocable transactions, we propose to alter some definitions. In particular, the definition of conflict must be extended. Traditionally, conflicts are defined only with regard to operations on shared variables. However, when supporting irrevocability in a TM system, the transitions to the irrevocable state can also be a legitimate reason of conflict. To minimize changes to the traditional definitions, we propose to model the conflicts on irrevocability by introducing a virtual transactional variable x_{irr} shared by all transactions. A transaction that intends to become irrevocable should execute a read operation immediately followed by a write operation on x_{irr}. This introduces a conflict between transactions that try to become irrevocable.

With the conflicts introduced by variable x_{irr}, the properties like strong progressiveness are achievable. However, this also makes properties like disjoint-access parallelism achievable – a property that intuitively should not apply to TM systems with irrevocability support. Thus, we should question ourselves whether the progress properties retain their intended meaning. In our opinion, after adding virtual variable x_{irr}, strong progressiveness remains the same for both TM users and developers. Moreover it is applicable to a wait-free TM with irrevocable transactions.

5.2.2 Progress of the algorithm

The property guaranteed by our algorithm is strong progressiveness (as defined in [5]) modified to take into account conflicts among the transactions that attempt to become irrevocable (as proposed in previous Section). Strong progressiveness is, despite its name, not very strong, but still practical property. It guarantees that a transaction without conflicts succeeds, and that whenever a group of transactions conflict on at most one variable, then at least one of them will succeed. It leaves out conflicts on multiple variables.

If in a group of conflicting transactions there is an irrevocable transaction, then strong progressiveness holds trivially – the irrevocable transaction is guaranteed to commit. So, we need to take into consideration only revocable transactions and transactions that fail to become irrevocable. In our algorithm, due to lack of global metadata, transactions learn about variables and peer transactions only upon read and write operations. Thus, without competing on a common variable transactions cannot impact each other. Moreover, despite we use visible reads, no read-read conflicts occur. So, a transaction with no read-write or write-write conflict (including

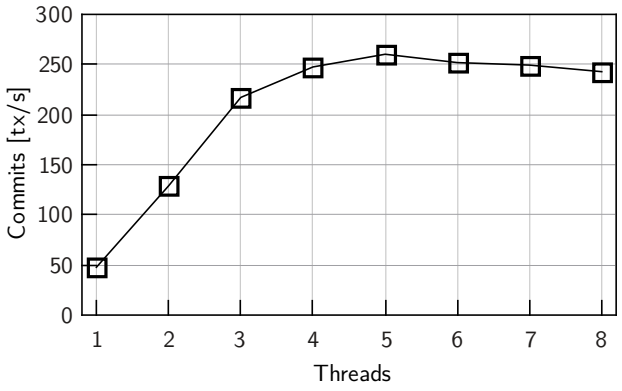

Figure 3. Invisible reads anomaly – STMBench7, readWrite, with traversals, 4 cores

conflicts on x_{irr}) trivially commits. Whenever there is a single conflict, on variable x, there must exist a transaction T_k that successfully locked x as part of its write operation. Now, for T_k to abort, either one of its reads has to be invalidated, or T_k must encounter another locked variable, or T_k must try to become irrevocable and fail. For any of these cases to occur, conflict on a second variable is required (in the latter case, conflict occurs on x_{irr}). Since we care for groups of transactions that conflict on a single variable only, none of these can happen. So, strong progressiveness holds.

5.2.3 Guarantees of becoming irrevocable

Progress properties hold regardless whether the transactions are revocable or not. Therefore, well-known progress properties provide also some guarantees on successful transiting to the irrevocable state. Strong progressiveness, the property which holds for our algorithm, guarantees that a transaction without conflicts must not be forcibly aborted. Thus, if no irrevocable transaction is live, and a single transaction T_k tries to become irrevocable, then T_k must become irrevocable if T_k has no conflicts. Moreover, in a system with no running irrevocable transaction, if multiple transactions with no conflicts so far try to become irrevocable, then one of them must succeed.

6. Experimental evaluation

6.1 Implementation

To evaluate our algorithm, we implemented it as a C++ library suitable for use both in real applications and object-based TM benchmarks. We are restricted to object-based TM benchmarks, since the algorithm relies on indirection level and operates on local buffers, thus there is no constant address of the transactional variables required by word-based TM benchmarks. We use plain C++11, which has the classes necessary to implement registers (`atomic<bool>` and `atomic<void*>` classes, featuring `store` and `load` methods) and trylocks (`atomic_flag` class with `test_and_set` and `clear` methods).

6.2 Injecting irrevocability into benchmarks

Currently none of the existing benchmarks uses irrevocable transactions. Thus, to evaluate the behavior of our algorithm, we needed to inject artificially transitions to the irrevocable state. While the irrevocable transactions are more expensive, they can reduce the number of restarts. Welc *et al.* suggest that the irrevocable transactions should be considered not only for supporting the irrevocable operations, but also for helping transactions that are likely to be aborted and repeated multiple times [17]. In order to test how the

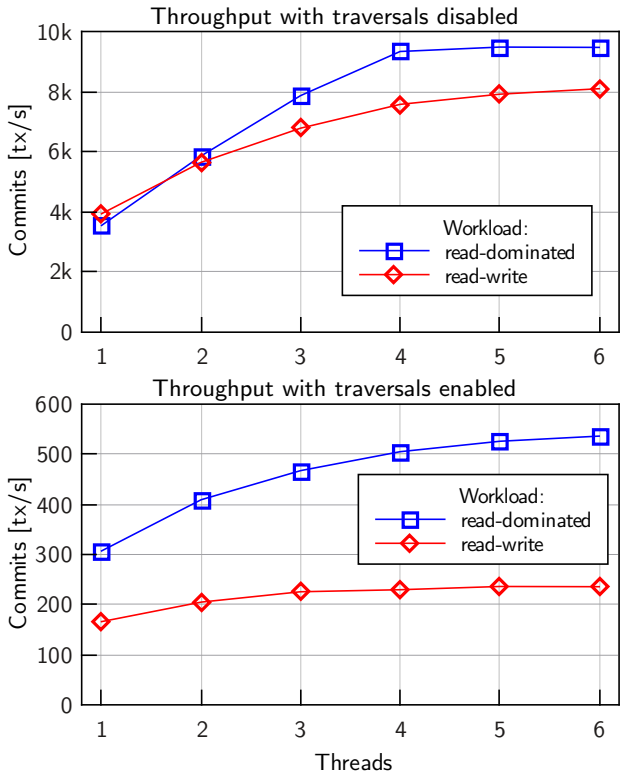

Figure 4. STMBench7 – commit rate / threads

irrevocable transactions perform in ordinary benchmarks, we introduced them as a fallback. We decided that any transaction forcefully aborted, except from waiting for a short time period (backoff), will try with some probability to become irrevocable on restart. This way we both introduce the irrevocable transactions and apply them to potentially problematic transactions in a generic way.

6.3 Invisible reads

The algorithm described in this paper uses visible reads. However, in the initial phase of the development we planned to use invisible reads instead. While invisible reads for ordinary transactions are allowed, the irrevocable ones must use visible reads (or a blocking synchronisation) as long as revocable update transactions are allowed to commit in parallel with a live irrevocable transaction. Otherwise, a revocable transaction cannot tell whether it may commit without overwriting the read set of an irrevocable transaction.

The preliminary results of the version using invisible reads are presented in Figure 3. While the shape of the curves may seem correct at first glance, it displays a severe anomaly: our system scales super-linearly. That is, a run with three threads has been over three times faster than a run with one thread. After investigating the issue it turned out that the design decision of using invisible reads was to blame. As mentioned before, aborted transactions were restarted in the irrevocable mode. Transactions in this mode never need to check their read sets, while revocable transactions (using invisible reads) must do so. With opacity as the correctness property, on every read of a previously unseen variable the read set has to be checked. So, the revocable transactions spent a lot of time validating reads, while the irrevocable transactions simply skipped that step. As a result, the irrevocable transactions performed better than normal. Because of that, with more threads (thus more conflicts) more “fast” irrevocable transactions appeared and boosted the per-

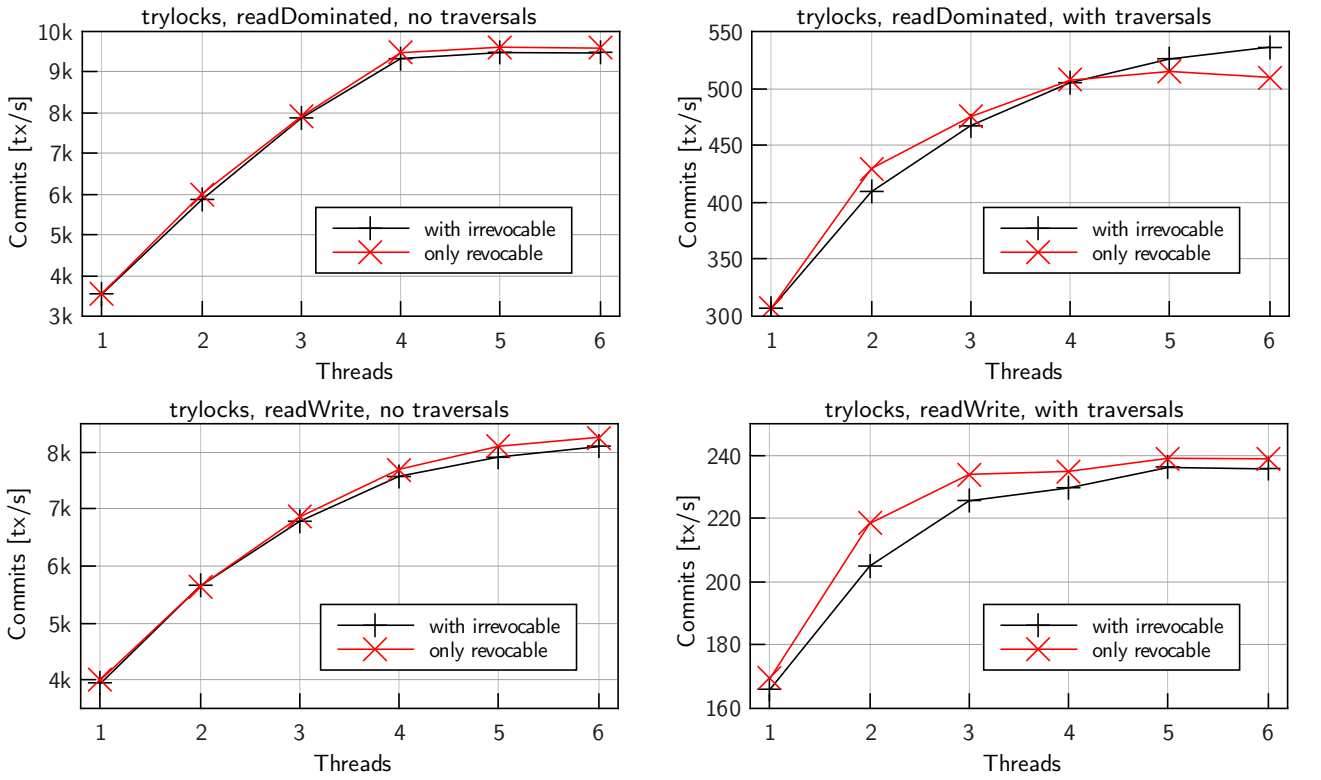

Figure 5. STMBench7 – impact of enabling irrevocability

formance beyond scale. This result clearly shows that extending any existing TM which uses invisible reads by support for the irrevocable operations can introduce a similar anomaly.

6.4 Irrevocable transaction overhead

With visible reads, regular and irrevocable transactions are on par in regard to speed. To calculate what is the exact impact of irrevocability on performance, we measured the time it takes to execute a certain sequence of transactions in a single thread, first as regular transactions, next, forcing each transaction to become irrevocable. As the transactions executed sequentially, we could calculate the average execution time of a revocable transaction and the average execution time of an irrevocable transaction. It turns out that typically irrevocable transactions take 16.23% more time than revocable when tested on machines with Intel® Xeon® X3230, 8MB L2 cache. Surprisingly, results of the same binary on identical OS, but on different CPU (Intel® Xeon® L3360, 12MB L2 cache) show that the irrevocable transactions run by 14.16% slower. Since the measured standard error was respectively 0.14% and 0.31%, we conclude that the slowdown rate noticeably depends on hardware.

The results obtained this way give a good estimate; however, they do not apply directly to normal operation. With contention the irrevocable transactions must take over data items accessed by concurrent transactions, and this may require extra actions, for instance repeating (finitely) certain operations. In general, it is not possible to measure the exact duration of single transactions without impacting the TM system too much to get reliable results. We expect that in normal runs the irrevocable transactions are, compared to normal, about 20% slower.

6.5 STMBench7

For evaluating the algorithms we chose the STMBench7 benchmark [7], that aims at providing workloads which are both realistic and non-trivial to implement in a scalable way. STMBench7 has a word-based API, however internally it uses object-based approach. Thus, it was possible to modify the STMBench7 API to suit our object-based API.

The main aim of evaluating our algorithms with STMBench7 was to see how it behaves in various workloads and how it scales with increasing number of threads. While comparing absolute performance with other TM systems is possible, it is important to notice that our implementation has not been throughly optimized.

For the benchmarks we used 4-core Intel® Xeon® L3360 processor (4 threads, 12 MB L2, 2.83 GHz), with sufficient physical memory, and compiled the programs using gcc 4.8.4, all under openSUSE 13.1.

6.5.1 Scaling trends

In Figure 4 we present the scaling and performance results of our algorithm. The scaling factor from one to four threads varies depending on the workload, from 2.8 in read-dominated scenario without traversals to 1.4 in write-intensive scenario with traversals. While low-contention workloads scale up to the number of cores, some high-contention workloads get better performance with the number of threads higher than the number of cores. This is a result of using a backoff contention manager, which, upon a forceful abort, waits a short time before restarting the transaction. On one hand, the backoff time reduces the likelihood that a transaction will run into the same conflict again. On the other, it also introduces pauses in CPU usage. Thus, extra threads can improve the overall performance, as long as they introduce more commits than conflicts. Results of the benchmark follow the common pattern for all transactional memory systems: read-dominated workloads scale fast and smoothly, introducing more writes reduces scalability. Also, enabling long traversals introduces large number of conflicts and thus hinders scaling.

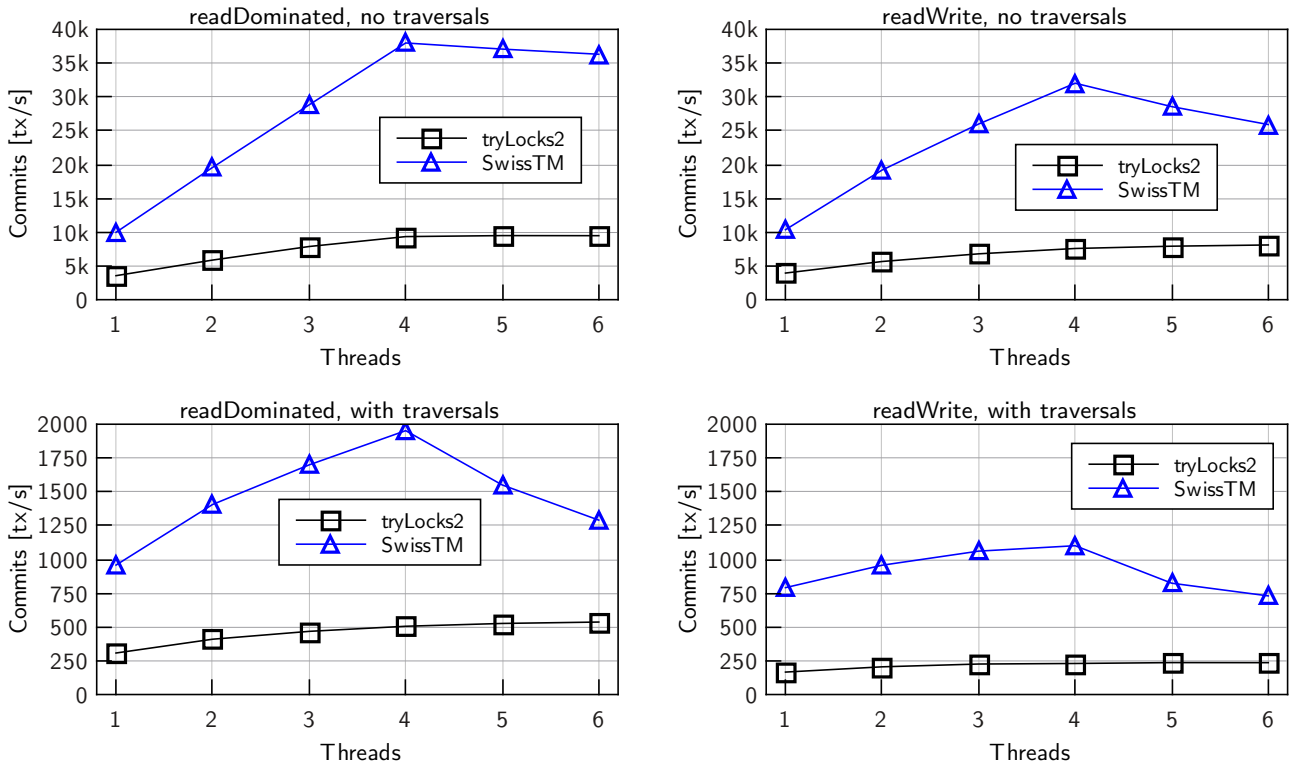

Figure 6. STMBench7 – commit rate / threads

6.5.2 Impact of irrevocability

An important question regarding irrevocability support is its impact on overall performance. Adding support for irrevocability requires revocable transactions to do some extra work – like, in our algorithm, checking for *dirtyIrr* and *irrUsing* as well as locking *commitL* (see Section 4 for details). Removing these operations is possible, but the resulting code would still bear all design requirements needed for irrevocability. We could use another TM system to compare performance, but we fear that influence of the coding style and optimizations level would have higher impact on the performance than sole support for irrevocability. Thus, the overhead is hard to measure. To give an insight into the irrevocability overhead, we turned off the use of irrevocable transactions. To achieve this, we slightly altered the contention manager: while still using the same backoff time, we no longer switch the transactions to the irrevocable state. The results of this test are presented in Figure 5, where we compare runs with and without irrevocable transactions.

The results show that our algorithm generally performs slightly faster without the irrevocable transactions. The slowdown introduced by adding irrevocable transactions in most sample points did not exceed 2%. To remind, the irrevocable transactions are by one fifth slower than the revocable. Impact on the overall performance is lower, since the irrevocable transactions always succeed. It is worth noticing that in the read-dominated workload with traversal transactions present, irrevocability noticeably improved the scaling trend. When using revocable transactions only, the performance dropped as the number of threads rose, since traversal transactions introduced lots of conflicts. With irrevocable transactions enabled, traversal transactions became irrevocable after few restarts and thus finished sooner. This confirms the observation that irrevocability can help in executing problematic transactions.

6.5.3 Comparison with SwissTM

We also compared the results with SwissTM [3] – see Figure 6. SwissTM, being fully optimized, is clearly better in speed and permissiveness. Our main aim was to compare the scaling trends to see if supporting irrevocability causes any anomalies, rather than comparing raw throughput. For runs with no traversals, up to four threads the trends are identical. For runs with traversals, our TM scales worse. Beyond four threads, that is when the number of threads is higher than the number of cores, SwissTM performance drops more rapidly. These results display no anomalies, thus we draw a conclusion that combining operation-level wait-freedom and support for irrevocability brought no unexpected behavior.

7. Conclusions

We showed that a TM system can support irrevocability and be strongly progressive, while at the same time guarantee that each transactional operation finishes in a finite number of steps. To show this, we proposed an algorithm built upon the weakest synchronization primitives that suffice to build a reasonable TM. To discuss properties such as strong progressiveness in presence of irrevocable transactions, we proposed a simple method for representing conflicts among transactions that try to become irrevocable. While irrevocable transactions are considered a necessity by TM users, their support is often associated with a performance penalty. Our results show that the performance overhead is low, and confirm that with increasing contention, the use of irrevocability can help with executing transactions that frequently conflict.

Acknowledgments

The project was funded from National Science Centre funds granted by decision No. DEC-2012/06/M/ST6/00463.

References

[1] L. Baugh and C. Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *IEEE International Symposium on Performance Analysis of Systems and Software*, 2008.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, 2006.

[3] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a Research Toy. *Communications of the ACM*, 2011.

[4] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. In *Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[] R. Guerraoui and M. Kapałka. *Principles of Transactional Memory*. Morgan & Claypool, 2010.

[5] R. Guerraoui and M. Kapałka. The semantics of progress in lock-based transactional memory. *ACM SIGPLAN Notices*, 2009.

[6] R. Guerraoui and Michał Kapałka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[7] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. *ACM SIGOPS Operating Systems Review*, 2007.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 2003.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 1991.

[10] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture*, 1993.

[11] M. Herlihy, V. Luchangco, M. Moir, and I. W. N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of PODC'03: the 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2003.

[12] Intel® Corporation. Intel® Transactional Memory Compiler and Runtime Application Binary Interface. *https://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition*, 2008.

[13] P. Kuznetsov and S. Ravi. On partial wait-freedom in transactional memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, 2015.

[14] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1995.

[15] M. Spear, M. Michael, and M. Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[16] M. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and exploiting inevitability in software transactional memory. In *37th International Conference on Parallel Processing*, 2008.

[17] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2008.

[18] F. Zyulkyarov, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *ACM Sigplan Notices*, 2009.

A. Proofs

Apart from notation used earlier in the text, we use the following:

$a \prec_H b$ means a precedes b in history H

dirty(Irr) variable $dirty$ or variable $dirtyIrr$

op_a^b a single low-level operation (step) of transaction T_b; a is used to indicate a specific operation.

$read^b$ is a read operation made by transaction T_b

A.1 opacity

Our algorithm provides *opacity*, which was defined in [4**?**] as:

> Opacity is simple to express and verify for sequential TM histories in which every transaction, except possibly the last one, is committed. Basically, if S is such a history, then S is considered correct, and called *legal*, if, for every t-object x, the sub-history $S|x$ respects the semantics of x, i.e., $S|x$ conforms to the sequential specification of x. (...)
>
> ***Definition 7.1 (Final-state opacity).*** A finite TM history H is *final-state opaque* if there exists a sequential TM history S equivalent to any completion of H, such that (1) S preserves the real-time order of H, and (2) every transaction T_i in S is legal in S. The sequential history S in the above definition is called a *witness history* of TM history H.
>
> ***Definition 7.2 (Opacity).*** A TM history H is *opaque* if every finite prefix of H (including H itself if H is finite) is final-state opaque.
>
> ***Definition 7.3 (Opacity of a TM).*** A TM object M is *opaque* if every history of M is opaque.

To prove opacity of our algorithm, we show how to construct a witness history S for an arbitrary execution history H of our algorithm. The witness history we show preserves real-time order of transactions and aborts transactions on the same operations as H does. Then, we show that all reads in H are legal in S.

A.1.1 Building witness history

For simplicity of our algorithm, when a transaction T_i commits, it attempts to abort all transactions that read values updated by T_i. This makes it easy to give a recipe of building a witness history for an arbitrary execution.

First, we sort all transactions by a certain real time value. For committed transactions, it is the time of reaching a specific step of the commit procedure:

- for revocable transactions, the time when a transaction grabs locks required to commit (locking *commitL* in line 79),
- for irrevocable transactions, we use the moment just before updating global copies of the variables (line 78).

For aborted transactions, the sort key is the time of a specific step during last successful variable access: for reads line 8, for writes line 27. If none of these occurred, we take the transaction start time.

To get the witness history S, we simply put all transactions in the order we just created. If multiple transactions are assigned to the same time point, any order is fine. When a transaction T_k aborts in H on operation op, we require T_k to abort on the same operation op in S. By using as the sorting key for any transaction a real time of a step between its start and commit, we naturally preserve the real-time order.

A.1.2 Proving read legality

To start, we name and explain the important for our proof events: op_r, op_c^p and op_u^p. The first, op_r, is a single machine operation (step) within $read^k$ (a read operation of T_k), which reads the value of a variable. Typically, op_r reads from the global buffer, however if T_k is irrevocable and it reads from a commit-pending transaction T_i, then op_r reads from a local buffer of T_i. During the commit of a transaction T_p, two operations are performed in order: op_c^p and op_u^p. The first, op_c^p, is used to place T_p in S. The second, op_u^p, updates the global copy of x to v_p.

LEMMA 1. *In any execution history H each read operation is legal in corresponding witness history S (built as described above).*

Let's take an arbitrary read operation $read^k$ of a variable x made by transaction T_k. First, notice that whenever the $read^k$ ends with abort in H, so it does in S (what is guaranteed by the way in which we construct S). Abort is always legal (unless T_k is irrevocable), so we need to consider only the case when $read^k$ succeeds.

To prove the lemma, we first define previous updating transaction. Then we consider two special cases when the irrevocable transaction T_k reads from a commit-pending transaction T_p: one, when T_k reads a value written by T_p, and the other, when T_k reads a value written by an irrevocable T_b which hijacked x from T_p. Finally, we consider the general case when a transaction reads from a finished transaction.

Previous updating transaction For every read $read^k$ of a variable x made by a transaction T_k in a history H, there must[3] exist a *previous updating transaction* T_p, that:

a) performed write operation on x (and updated its local buffers),
b) is commit-pending or committed in H,
c) $op_c^p \prec_H op_r$,
d) there is no other transaction T_b that wrote to x such that $op_c^p \prec_H op_c^b \prec_H op_r$

The above definition is tailored for our TM – we do not require global copy update (op_u^p) to precede read of the value of x (op_r). Notice, that unless $read^k$ aborts, then $T_p \prec_S T_k$. Proof: T_p is previous update transaction of $read^k$, so $op_c^p \prec_H op_r$. When constructing S, T_p is ordered by op_c^p. If $read^k$ does not abort, then T_k is ordered in S by op_r or an event following it, thus $T_p \prec_S T_k$.

Formulating proof by contradiction Next, we show that if T_p is the previous updating transaction of x for the $read^k$ operation of T_k, then $read^k$ returns last value written by T_p to x.

We show it by contradiction, thus we assume that in H:

- the read operation $read^k$ (of T_k) succeeds and returns value v_k,
- T_p sets its local buffer of x to v_p (prior to commit),
- T_p performs (during commit) an operation op_u^p which updates the global copy of x so that it points to the local buffer,
- $v_p \neq v_k$.

Reading x before the global copy is up to date First, we show what happens if T_p did not update global copy before the read operation ($op_u^p \not\prec_H op_r$). It is obvious, that if T_k is revocable, then T_k aborts (see A.4.1). If T_k is irrevocable, then (1) T_p has lock on x (2) T_p is *mostRecentLockOwner* (3) T_p has final value of x in its local buffers (4) T_k fails to abort T_p (5) T_k reads from local buffers of T_p. Thus, $v_p = v_k$, what is in contradiction with our assumptions. So, for the rest of the proof, we can safely assume that $op_u^p \prec_H op_r$

Reading a previously hijacked variable Next, we cut off a special case when an irrevocable transaction reads from local buffers of a

[3] We assume that there exists transaction T_0 that initializes all variables and makes no reads, so T_p always exists.

peer transaction. If T_k is irrevocable and fails to win lock on x and reads that a transaction T_b ($\neq T_p$) is *mostRecentLockOwner* (acquired lock on x), then either:

(1) T_k aborts T_b, or T_b is already aborted, or

(2) $T_b \prec_S T_p \prec_S T_k$, and T_p is irrevocable (and obviously finished before T_k became irrevocable and issued the read on x).

Proof: Notice, that before T_k performs op_r, it either aborts T_b or observes it as either aborted, or committed, or commit-pending (line 40, 41). Case (1) gathers all scenarios where T_b is aborted. When T_b is committed or commit pending at op_r, then $op_c^b \prec_H op_r$. If T_p is the previous update transaction, then there cannot exist such a transaction T_b that commits between T_p and op_r So, $op_c^p \not\prec_H op_c^b$; op_c^p and op_c^b obviously cannot happen in parallel if T_p and T_b both write to x (see A.4.2), thus $op_c^b \prec_H op_c^p$ and $T_b \prec_S T_p$. Since we know that T_k reads T_b as lock's owner, then T_p must have been an irrevocable transaction that hijacked x, which is exactly the case (2). Thus no further cases exist.

In case (1) T_b cannot update x. To update x, a revocable transaction has to hold lock on x and commit. If T_b acquired the lock and did not update *mostRecentLockOwner* before T_k accessed x, then T_b aborts (as it notices *irrUsing* set). If T_b updated *mostRecentLockOwner*, then T_k aborts it (if T_k fails, then it is the case (2)). Whenever T_b is aborted, T_k reads from the global copy of x, what is discussed later on as reading x from global copy.

In (2) T_k is irrevocable and hijacks x from T_b, reading value out of its buffers. However, the value in buffers of T_b has been overwritten by T_p with value v_p during commit of T_p (line 88). Thus, T_k reads $v_k = v_p$, what stands in contradiction with assumptions.

Reading x from global copy In all remaining cases T_p releases x before T_k starts $read^k$. Thus, we know that $op_u^p \prec_H op_r$ and op_r reads from the global copy.

We know that $op_c^p \prec_H op_u^p \prec_H op_r$. Since op_u^p writes v_p to the same location from which op_r reads v_k and we assume that $v_p \neq v_k$, then there must exist an operation op_u^b such that $op_c^p \prec_H op_u^p \prec_H op_u^b \prec_H op_r$ (and obviously $op_c^b \prec_H op_u^b$). Because T_p is the previous update transaction, T_b must commit either before T_p or after $read^k$. The latter is not possible, as we know that $op_u^b \prec_H op_r$. So, $op_c^b \prec_H op_c^p \prec_H op_u^p \prec_H op_u^b$. If T_p is revocable, this is trivially impossible (see A.4.3). If T_p is irrevocable, then the sequence of events is possible (and happens whenever T_p issues a write after op_c^b). In such case, before executing op_u^p the transaction T_p overwrites the buffer of T_b with value v_p (line 88). As a result, the operation op_u^b updates global value of x to point the value v_p (not v_b). So, value read by T_k is equal to value written by T_p, what stands in contradiction to our assumptions.

We showed that in all possible cases our assumptions lead to a contradiction, so either processes do not follow the pseudocode, or all reads in H are legal in corresponding witness history S, what proves that the algorithm is opaque.

A.2 Operation-level wait-freedom

First, notice that all steps in the algorithm are wait-free. Stores and loads are obviously wait-free, as well as are operations *trylock()* and *unlock()* on locks. *readers* is a constant-size array of registers, and thus stores and loads to its cells are wait-free. Besides, we use set and map operations (add, remove, look up, traverse). *rsetBuffs*, *hijackedBuffs* and *locksHeld* are local to a single transaction, thus require no synchronization and do not restrict wait-freedom. *wsetBuffs*, the map containing write set – the local buffers of a transaction – is possibly accessed by multiple transactions. However, notice that a map of T_k is accessed prior to commit of T_k by T_k only. Once T_k issues commit, it no longer modifies the map. Also, all other transactions that access the map perform only lookups. Thus, no concurrent modification can occur once T_k issued commit, and no synchronization is needed for *wsetBuffs*. Next, all loops in the pseudocode are finite – they loop only over finite sets or arrays. Since the pseudocode contains only wait-free steps, all loops are finite and no recursion occurs, the algorithm is trivially wait-free.

A.3 Strong progressiveness

The algorithm is strongly progressive, as defined in [5]:

> More precisely, let H be any history of a TM and T_k be any transaction in H. We say that T_k is forcefully aborted in H, if T_k is aborted in H and there is no invocation of operation $tryA_k$ in H. We denote by $WSet_H(T_k)$ and $RSet_H(T_k)$ the sets of t-variables on which T_k executed, respectively, a write or a read operation in H. We denote by $RWSet_H(T_k)$ the union of sets $RSet_H(T_k)$ and $WSet_H(T_k)$, i.e., the set of t-variables accessed (read or written) by T_k in history H. We say that two transactions T_i and T_k in H conflict on a t-variable x, if (1) T_i and T_k are concurrent in H, and (2) either x is in $WSet_H(T_k)$ and in $RWSet_H(T_i)$, or x is in $WSet_H(T_i)$ and in $RWSet_H(T_k)$. We say that T_k conflicts with a transaction T_i in H if T_i and T_k conflict in H on some t-variable. Let H be any history, and T_i be any transaction in H. We denote by $CVar_H(T_i)$ the set of t-variables on which T_i conflicts with any other transaction in history H. That is, a t-variable x is in $CVar_H(T_i)$ if there exists a transaction $T_k \in H, k \neq i$, such that T_i conflicts with T_k on t-variable x. Let Q be any subset of the set of transactions in a history H. We denote by $CVar_H(Q)$ the union of sets $CVar_H(T_i)$ for all $T_i \in Q$.
>
> Let $CTrans(H)$ be the set of subsets of transactions in a history H, such that a set Q is in $CTrans(H)$ if no transaction in Q conflicts with a transaction not in Q. In particular, if T_i is a transaction in a history H and T_i does not conflict with any other transaction in H, then $\{Ti\} \in CTrans(H)$.
>
> DEFINITION 3. A TM implementation M is strongly progressive, if in every history H of M the following property is satisfied: for every set $Q \in CTrans(H)$, if $|CVar_H(Q)| \leq 1$, then some transaction in Q is not forcefully aborted in H.

We say that two transactions conflict on x if they are parallel, access x and at least one of the accesses is an update. By a *conflict group* Q we call a group of transactions, such that no transaction in Q conflicts with a transaction not in Q. We say that a conflict group Q conflicts on some variables, if each of these variables causes a conflict between two transactions from Q. We say that a transaction forcibly aborts, if the abort is raised from a procedure other than `abort`. A TM is *strongly progressive*, if at least one transaction is not forcibly aborted in any conflict group Q that conflicts on at most one variable.

LEMMA 2. *If a transaction T_i acquires cleanRsetL of a transaction T_k, then then both T_k and T_i access a variable x, and T_i updates x.*

Proof A transaction T_i can lock *cleanRsetL* of a transaction T_k only in line 76. To do so, first T_i must read T_k from $readers$ array of a variable x. T_k writes itself to $readers$ on `read` of x. T_i executes line 76 only for variables it updates, so it must update x.

LEMMA 3. *If a transaction T_i acquires commitL of a transaction T_k, then both T_k and T_i access a variable x, and T_k updates x.*

Proof The lock $commitL$ can be acquired by T_i in line 40 only. For a transaction to reach line 40, it needs to issue an access to

x and read T_k as *mostRecentLockOwner* of x (in line 39). For T_k to set itself as *mostRecentLockOwner*, T_k must perform an update operation on x.

LEMMA 4. *If a transaction T_k reads its aborted as true, then either cleanRsetL or commitL of T_k is locked, and T_k is in conflict with some T_i.*

Proof T_k itself sets *aborted* (in line 68) iff it invokes `abort`. T_k makes no steps after abort, so for it to read *aborted* as true, some other transaction T_i must set it. *aborted* can be set by T_i in lines 40 and 77. Line 40 is executed just after T_i acquires *commitL*, and line 77 is executed just after T_i acquires *cleanRsetL*. Neither of the locks is ever unlocked, so whenever T_k reads *aborted* as true, one of the locks is acquired by T_i. From Lemma 2 and Lemma 3 we know that whenever T_i acquires one of the locks, then both T_i and T_k access a common variable and one of the accesses is an update, so there is a conflict between T_i and T_k.

LEMMA 5. *In our algorithm, a transaction T_k can be forced to abort by a transaction T_i iff T_i and T_k are in conflict*

Proof In our algorithm a transaction T_k can forcibly abort only in the following lines:

7 upon read of x, when T_k notices a transaction T_i updating x,
21, 26 upon updating x, when T_k notices an irrevocable transaction T_i accessing x,
22 upon updating x, when T_k notices a transaction T_i updating x,
47 upon becoming irrevocable, when an irrevocable transaction T_i is live; in this case, T_k and T_i conflict on x_{irr},
50 upon becoming irrevocable, if `acquireReadset` failed; this happens iff an item read by T_k is updated by a transaction T_i,
9, 30, 70 whenever a transaction observes *ct.aborted*,
54, 81 whenever a transaction fails to lock *cleanRsetL* or *commitL*.

Apart from two last cases, all remaining are trivial: T_k and T_i access a common variable, at least one of the accesses is an update. From Lemma 4 we now, that if T_k reads aborted as true, then it is in conflict with some T_i. From Lemma 2 and Lemma 3 we know that if T_i acquires either *cleanRsetL* or *commitL*, then T_i and T_k both access a common variable and one of them updates it. So, we showed that in all lines where T_k forcibly aborts, there is a concurrent transaction T_i, and that T_k and T_i access a common variable, and one of the accesses is an update. Thus, T_k can be forced to abort iff it conflicts with T_i.

LEMMA 6. *If there is a conflict group Q of transactions $T_k, T_i, \ldots$ conflicting on a single variable, then once T_k acquires lock on x, no other revocable transaction may acquire cleanRsetL of T_k.*

Proof $cleanRsetL$ can be acquired by a T_i ($T_i \neq T_k$) in line 76 only. To reach line 76, T_i must have x in its write set. If T_i is revocable, to add x to its write set, it must acquire the lock on x. However, T_i cannot hold this lock, since it is already held by T_k.

THEOREM 7. *Our algorithm is strongly progressive.*

Proof We know from Lemma 4, that for a transaction to be forcibly aborted, it needs to be in a conflict group with another transaction. Thus, transactions with no conflicts (alone in a conflict group) can never be forcibly aborted. We also need to show that when a group of transactions $\{T_k, T_i, \ldots\} = Q$ conflicts on a single variable x, then at least one of the transactions is not forcibly aborted. Notice, that if a transaction $T_a \subset Q$ calls `abort` itself, then strong progressiveness trivially holds – T_a has not been forcibly aborted.
First, we consider a conflict on x_{irr}. In such case no transaction in Q can be forcibly aborted until it invokes `become irrevocable`, as no other conflicts occur (see Lemma 4). First step of this operation – acquiring $irrTransactionLock$ (line 47) – must succeed for one transaction (labeled T_k) due to the semantics of *try-lock()* operation. Next, T_k executes `acquireReadset` procedure. Since we know that T_k conflicts only on x_{irr}, it must succeed, as `acquireReadset` can fail iff T_k fails to win a lock on a variable from its read set, that is if there is a concurrent transaction updating that variable. Next, T_k needs to acquire locks in line 51. As the only conflict is on irrevocability (x_{irr}), from Lemma 2 and Lemma 3 we know that both locks are unlocked. This completes the *become irrevocable* operation, and since the irrevocable transaction simply has no forced abort routines, it trivially commits.
Next, we consider conflict on a transactional variable x. We start from analyzing a transaction T_k which issued an update on x and as the first (in real time order) acquired lock on x in line 22 or 37. If T_k is revocable, it may fail to complete the `write` in line 26; this happens iff then there is an irrevocable transaction T_i that accesses x. T_i cannot be forcibly aborted (is irrevocable), and is in Q (conflicts with T_k on x). Once T_k passes line 26 for x, it could be forced to abort in lines:

7 upon read of y, when T_k notices a transaction T_i updating y,
21, 26 upon updating y, when T_k notices an irrevocable transaction T_i accessing y,
22 upon updating y, when T_k notices a transaction T_i updating y,
47 upon becoming irrevocable, when an irrevocable transaction T_i is live; in this case, T_k and T_i conflict on x_{irr},
50 upon becoming irrevocable, if `acquireReadset` failed; this happens iff an item y read by T_k is updated by a transaction T_i (x is updated by T_k, so it cannot trigger that line),
9, 30, 70 whenever a transaction observes *ct.aborted*,
54, 81 whenever a transaction fails to lock *cleanRsetL* or *commitL*.

Lines 7, 21, 26, 22, 47 and 50 require a conflict on another variable (y or x_{irr}), so are not possible here. In any case $x \neq y$, as further reads/writes of x operate on a local buffer. From Lemma 4 we know that lines 9, 30, 70 ($aborted$ seen as true) mean the same as lines 54, 81 (failing to acquire *cleanRsetL* or *commitL*). From Lemma 6 we know that once T_k locked x, $cleanRsetL$ can no longer be taken by a peer revocable transaction. Notice, that $commitL$ can be locked by T_k or an irrevocable transaction only, and the latter locks it upon access to x. So, either T_k cannot be forcibly aborted, or a transaction in Q successfully became irrevocable, and, as irrevocable, cannot be forcibly aborted.
Summing up, we showed that in any case either a transaction T_a in Q aborts itself, or an irrevocable transaction T_i is in Q, or T_k cannot be forcibly aborted. So, in a conflict group Q on a single variable x there always exists a transaction (T_a, T_i or T_k) that is not forcibly aborted. Above we showed also that transactions with no conflict cannot be forcibly aborted, thus the proof is complete.

A.4 Trivia

Below, for sake of completeness, we present proofs for some obvious claims which we used to prove opacity.

A.4.1 Ad A.1.2 #1

(...) It is easy to show, that if T_k is revocable, then T_k aborts (...)
From the pseudocode we know that:
set *dirty(Irr)* $\prec_H$ kill *readers* $\prec_H$ op_c^p $\prec_H$ op_u^p $\prec_H$ clear *dirty(Irr)*
set *readers* $\prec_H$ check *dirty(Irr)* $\prec_H$ op_r
We know that $|op_c^p \prec_H op_r|$ and that for `read` to succeed:
kill *readers* $\prec_H$ set *readers*
and then since $|op_u^p \not\prec_H op_r|$ and $|op_u^p \prec_H$ clear *dirty(Irr)*$|$ we know that
set *dirty(Irr)* $\prec_H$ check *dirty(Irr)* $\prec_H$ clear *dirty(Irr)*
so obviously T_k reads dirty as true, what causes T_k to abort.

A.4.2 Ad A.1.2 #2

(...) op_c^p and op_c^b cannot happen in parallel if T_p and T_b both write to x (...)

Notation: we use *mrlo* as an abbreviation for *mostRecentLockOwner*.

We know that T_b and T_p update x, and that T_b held lock on x at op_c^b (as we know it is at least commit-pending and T_k read T_b as *mrlo*).

If T_p was revocable, T_p would need the lock on x at commit as well. So if T_p was revocable and op_c^p and op_c^b were parallel, then two transaction would hold the lock concurrently, which is not possible.

If T_p was irrevocable and failed to lock x, then from the pseudocode:

T_p : set^p *usedByIrr* $\prec_H$ read^p *mrlo* $\prec_H$ lock^p *mrlo.commitL* $\prec_H$ $\prec_H$ op_c^p $\prec_H$ clear^p *usedByIrr*

T_b : write^b *mrlo* $\prec_H$ check^b *usedByIrr* $\prec_H$ op_c^b

We defined op_c^b so that it stands for locking *commitL*, thus the line above is:

T_b : write^b *mrlo* $\prec_H$ check^b *usedByIrr* $\prec_H$ lock^b *commitL*

If op_c^p and op_c^b are parallel, then for T_b to commit, T_p must not read it as $mrlo$, so |read^p *mrlo* $\prec_H$ write^b *mrlo*| (notice that no new transaction can take the lock on x before the commits of T_b and T_p). Thus:

set^p *usedByIrr* $\prec_H$ read^p *mrlo* $\prec_H$ write^b *mrlo* $\prec_H$ $\prec_H$ check^b *usedByIrr* $\prec_H$ $\frac{op_c^p}{op_c^b}$ $\prec_H$ clear^p *usedByIrr*

and in such case T_b would observe *usedByIrr* as true, so it would have to abort. So, if T_p is irrevocable, then op_c^p and op_c^b must not be parallel either.

A.4.3 Ad A.1.2 #3

(...) $op_c^b \prec_H op_c^p \prec_H op_u^p \prec_H op_u^b$. If T_p is revocable, this is impossible (...)

Notation: we use *mrlo* as an abbreviation for *mostRecentLockOwner*.

If T_p is revocable, then it must hold lock between x at op_c^p and op_u^p. Obviously, if T_b were revocable, it would need to hold lock on x between op_c^b and op_u^b. So we must assume that T_b is irrevocable and failed to acquire lock on x (so that T_p can take it). Thus from the pseudocode:

T_b : set^b *usedByIrr* $\prec_H$ fail to lock^b x $\prec_H$ read^b *mrlo* $\prec_H$ $\prec_H$ lock^b `mrlo.commitL` $\prec_H$ op_c^b $\prec_H$ op_u^b $\prec_H$ clear^b *usedByIrr*

T_p : lock^p x $\prec_H$ write^p *mrlo* $\prec_H$ check^p *usedByIrr* $\prec_H$ op_c^p $\prec_H$ op_u^p

As we defined op_c^p as locking $commitL$, then we can write:

T_p : write^p *mrlo* $\prec_H$ check^p *usedByIrr* $\prec_H$ lock^p *commitL* $\prec_H$ op_u^p

For T_b not to force T_p to abort, |read^b *mrlo* $\prec_H$ write^p *mrlo*|, and we know that $|op_c^b \prec_H op_c^p \prec_H op_u^p \prec_H op_u^b|$. Thus:

set^b *usedByIrr* $\prec_H$ read^b *mrlo* $\prec_H$ write^p *mrlo* $\prec_H$ $\prec_H$ check^p *usedByIrr* $\prec_H$ op_u^p $\prec_H$ op_u^b $\prec_H$ clear^b *usedByIrr*

what is impossible, since we assume that T_p commits and T_p reads *usedByIrr* as true, so it has to abort.