PROGRAMMING MULTI-CORE AND MANY-CORE COMPUTING SYSTEMS

CONTRIBUTORS

V. GRAMOLI EPFL, Switzerland.
R. GUERRAOUI EPFL, Switzerland.

CONTENTS IN BRIEF

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

PROGRAMMING WITH TRANSACTIONAL MEMORY

Vincent Gramoli[1], Rachid Guerraoui[1]

[1]EPFL, Switzerland

1.1 INTRODUCTION

The transactional memory (TM) paradigm simplifies concurrent programming by hiding all synchronizations from the standpoint of the application programmer. Basically, it relieves the programmer of the lock management burden, therefore the programmer does no longer have to explicitly acquire and release locks but simply has to delimit regions of sequential code that should appear as atomic, the *transactions*.

Transactions cannot block each other, hence guaranteeing deadlock-free executions. In contrast, purely lock-based systems are known to be complex and for example in [5] the authors identified that 34% of Linux bugs [5] were due to synchronization while in [11] the authors thought having found 8 potential deadlock bugs in the Linux kernel v2.5. Transactional memory provides a transaction abstraction appealing for it remedies numerous concurrency problems.

It is the role of the TM to execute transactions concurrently yet guaranteeing that their execution is equivalent to an execution in which they would be serialized. The synchronization mechanics are thus still present but transferred to the TM program underlying the applications. TM greatly simplifies concurrent programming by hiding these complex mechanics from the application programmer.

More specifically, the TM program provides a basic high-level interface for the programmer to convert a sequential program into a concurrent one: `begin`, `read`, `write`, `commit`. The `begin` and `commit` delimit the *transaction*, i.e., the region of code that should appear as being executed atomically in isolation from the rest of the system. The `read` and `write` calls are wrappers to usual memory accesses that are used to redirect memory loads and stores that belong to a transaction. Besides making sure that the transaction executes the corresponding `load` (resp. `store`), the `read` (resp. `write`) maintains the metadata defining the transaction state.

Section 1.2 illustrates the simplicity of transactions to solve a common concurrency problem. Section 1.3 summarizes TM support in existing programming languages and presents the transactional constructs for C, C++ and Java. Section 1.4 introduces TM implementations by discussing hardware support and presenting a running example for software support. Section 1.5 outlines causes for performance limitations TMs may suffer from and Section 1.6 introduces recent solutions to cope with these limitations.

1.2 CONCURRENCY MADE SIMPLE

The transactional memory paradigm initiated a complexity shift between the development of the concurrent applications to the development of the transactional memory programs. This section illustrates the simplicity of writing a TM-based concurrent program with the dining philosopher problem.

The dining philosopher problem is a common concurrency control problem proposed by Dijkstra and reformulated by Hoare [24], in which five philosophers sitting around a table with a rice bowl in front of each of them and a single chopstick between each pair of consecutive bowls (see Figure 1.1), alternate between thinking and eating. A philosopher can eat only if he acquires the two chopsticks near his bowl, hence not all philosophers can eat at the same time. Solving the dining philosopher problem relies in designing an algorithm that avoids *starvation* so that each philosopher is guaranteed to eat eventually.

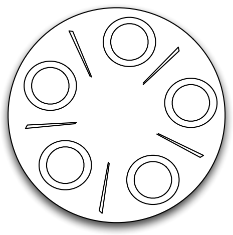
Figure 1.1 The five chopsticks shared by the five dining philosophers to eat rice.

A naive spinning lock-based solution that consists of acquiring the left chopstick before trying to acquire the right one suffers from deadlock, as all philosophers could potentially acquire the left one and wait forever for the right one to be released by his right neighbor. Djikstra solution consists of acquiring and releasing all chopsticks in a predetermined order, however, if the placement of the chopstick is not adequate, then the solution may be very costly forcing many philosophers to release their chopstick to let a single one eat. Other deadlock-free solutions are not starvation-free as they may suffer from livelock, where philosophers keep acquiring and releasing chopstick without being able to eat.

Algorithm The try-to-eat procedure of the dining philosopher i.

```
1: try-to-eat()_i:
2:   begin()                          ▷ transaction starts
3:     pickup(chpks[i])
4:     pickup(chpks[i + 1 mod 5])
5:     eat()
6:     putdown(chpks[i + 1 mod 5])
7:     putdown(chpks[i])
8:   commit()                         ▷ transaction ends

9: pickup(chpk)_i:
10:    chpk.owner = i                 ▷ write memory

11: putdown(chpk)_i:
12:    chpk.owner = ⊥                 ▷ write memory
```

This algorithm depicts a transaction-based try-to-eat procedure that shows how TM can simply solve the dining philosopher problem. The all-or-nothing transaction semantics guarantees that the transaction either acquires both chopsticks of none of them. If the transaction commits the philosopher can eat. Conversely, if the transaction aborts which means that concurrent transactions conflict while trying to access a common chopstick, then the philosopher does not prevent other philosophers from eating, granting additional time to his neighbors to finish eating. This example is quite simple but a general guideline for programmers is to write transactions that are as short as possible. If the transactions are too long, then it is likely that one of them may have to abort and restart. A separate contention manager plays the role of avoiding livelocks in which a transaction gets repeatedly aborted. For instance, contention managers either map a high priority to aborting transactions [32] or make aborting transactions back off some time before restarting [22].

The simplicity comes from the fact that synchronization complexities are transferred to the TM itself and are hidden from the programmer. In fact, the exposed synchronization is reduced to begin and commit delimiters, the rest being bare sequential code. No locking primitives are exposed to the programmer and deadlock-freedom is inherently guaranteed by the hidden TM. TM implementations are detailed in Section 1.4.

1.3 TM LANGUAGE CONSTRUCTS

Several compilers support transaction language constructs. For instance, there exist at least three compilers that compiles transactions in C at the time of writing: the prototype DTMC[1], the TM branch of gcc[2], and the Intel® C++ STM compiler (icc)[3]. The Glasgow Haskell Compiler provides support for transactions in Haskell [19]. Multi-

[1]http://www.velox-project.eu/software/dtmc
[2]http://www.velox-project.eu/software/gcc-tm
[3]http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/

verse offers support for TM in Java and has been used in Scala [3]. Fortress supports transactional language constructs and Pugs compiles and interpret transactions in Perl 6. .NET Framework 4 supports software transactional memory. Multiple third-party programs allow also to support transactions in other languages, like STMlib for OCaml and Durus for Python.

1.3.1 C/C++ Programming Languages

The first attempts to support TM in production compilers led to the definition of dedicated transactional language constructs in a C++ specification draft [16], resulting originally from the collaboration of Intel®, IBM® and Sun® Microsystems (Oracle®). This draft describes the constructs to delimit a compound statement that is identified by the compiler as a transaction, in addition to several subtleties we list here. For several years now, this specification has been reviewed and discussed by academicians and other industrials on the tm-language mailing list[4], for the sake of language expressiveness and compliance with existing TM systems and other languages.

In C, a transaction is simply delimited using the block

```
__tm_atomic{ ... }
```

while in C++ a transaction is delimited by a `__transaction{ ... }` block where `__transaction{` (or equivalently `__transaction[[atomic]]{`) indicates the point in the code where the corresponding transaction `begin` should be called. The closing bracket } indicates the point in the code where the corresponding transaction `commit` should be called. Within this block, memory accesses are instrumented by the compiler to call the transactional `read` and `write` wrappers. More precisely, the binary files produced by the compiler call a dedicated TM runtime library through an appropriate application binary interface (ABI) specified in [6]. This ABI is used for both C and C++ and has been optimized for the Linux OS and x86 architectures to reduce the overhead of the TM calls and to allow fast accesses to thread-specific metadata shared by existing TMs.

Transaction nesting, which consists in encapsulating a transaction block into another, is allowed and the `[[outer]]` keyword is explicitly used to indicate that a transaction cannot be nested inside another. Generally, it is not allowed to redirect the control flow to some point in the context of a transaction, but exceptions can be raised within the context of transaction to redirect the control-flow outside the transaction context by propagating the exception.

Irrevocable transactions do not execute speculatively and are used to execute actions that cannot be rolled back once executed, this is typically necessary in cases where an action has some external side-effects like I/O have. Attribute in C++1x-style indicate whether the transaction executes speculatively as by default `__transaction[[atomic]]{}` or has to execute without being aborted

[4]`http://groups.google.com/group/tm-languages`

`__transaction[[relaxed]]{}`, say in *irrevocable* mode. Only irrevocable transactions can execute calls with irrevocable side-effects, and for example

```
[[transaction_unsafe]] void fire_missile{};
```

declares a `fire_missile` function that can only be called in an irrevocable transaction. The attribute `[[transaction_safe]] void do_work{};` is especially used to indicate the opposite, that function `do_work` does not have to be called in an irrevocable transaction and can execute speculatively as part of a transaction prone to abort.

1.3.2 Java^{TM} Programming Language

In Java, TM supports has been initially proposed using the combination of Java annotations for identifying transactional accesses and a corresponding bytecode instrumentation framework that instruments transactional accesses either at load-time or statically, prior to execution. Multiverse[5] and Deuce [26] are two such JVM agents that instrument transactional accesses of the annotated bytecode resulting from a concurrent program.

Multiverse distinguishes `@TransactionalObject` and `@TransactionalMethod` annotations that apply respectively to Java objects and methods. All instance methods of an annotated object are thus instrumented transactions and annotated methods allow to specify which methods of a non-annotated object are transactions. Additionally, annotating a method of an already annotated objects allows to differentiate explicitly read-only methods from update methods using `@Transactional(readonly = true)`. Such differentiation is useful for the underlying TM to optimize the validation of a read-only transaction that commits.

Deuce instruments methods annotated with the `@Atomic` keyword and uses a clear interface a TM should provide: `begin` (namely `init`), `read` (namely `onReadAccess`), `write` (namely `onWriteAccess`), `commit` methods and additional `beforeReadAccess` and `abort` (namely `rollback`) methods. The current distribution features classical transactions of state-of-the-art software TMs developed collaboratively like TL2 [9], LSA [30], NOrec [7] but also $\mathcal{E}$-STM [13] that combines classical and elastic transactions, described in Section 1.6.2.2.

Unlike C/C++ compilers, the aforementioned bytecode instrumentation frameworks cannot consider an arbitrary compound statement as a transaction because they rely on annotation mechanisms. The Java precompiler TMJava[6] remedies this limitation by extending Java with transactional blocks. More specifically, TMJava supports the `__transaction{...}` language construct in Java and outputs a purely Java annotated program whose bytecode can be instrumented with Deuce.

It is noteworthy that TMs have recently found other applications in Java, including coordinated exception handling [1]. In this case, the failure atomicity guaranteed by

[5] `http://multiverse.codehaus.org`
[6] `http://tinystm.org/tmjava`

transactions is useful for recovering from an inconsistent state even in a concurrent environment.

1.4 IMPLEMENTING A TM

Programming with transactions shifts the synchronization complexity from the application to the TM. This section discusses the TM implementations in hardware and then in software, illustrating different design choices.

1.4.1 Hardware Support

Hardware transactional memory (HTM) has already shown promising results for leveraging parallelism in the Linux OS [31] where transactions are combined with spinlocks. HTMs scale better than locks in some scenarios. For example, if a single lock is protecting multiple elements of a data structure, then concurrent transactions accessing these elements may not abort each other. Finally, transactions are inherently compositional and deadlock-free, an additional reason for using HTMs over locks.

The scheme of the MetaTM [31] is similar to Log-TM [27]. When a thread executing a transaction executes a modification, the modification becomes immediately visible and the old value is recorded into an undo-log. If the transaction aborts, it then rolls back by reverting the value to the old value. A thread can stop a transaction and restart it afterwards which facilitates interrupt handling. HTMs have been applied to bus-based systems in which the network-on-ship communications have to be diminished. In [25], the authors propose an HTM that targets object-oriented programming. This object-aware HTM attempts to avoid the high abort ratio induced by considering conflicts at the level of objects that can be arbitrarily large. The key idea is to provide an object cache with object address and field offset that are cached.

HTMs suffer, however, from some limitations. For example, some HTMs require transactions to be of limited size [20] because of the limited hardware resource, like bounded cache size. Some HTMs require specific system events or instructions to be executed outside transactions [8, 27]. Despite being dedicated to OSes, some others cannot support transaction suspension, migration or context switches [18, 25]. Finally, transactions must be small enough to provide responsive irrevocable I/O despite their speculative behaviors. The development of the Rock processor [8], which provided a best-effort hardware transactional memory, has been canceled. This processor used aggressive speculation to provide high single-thread performance in addition to high system throughput. For these reasons, it is unlikely that future TM implementations will be purely hardware and upcoming TMs are expected to contain a software component.

1.4.2 Software Support

We present a running example of software transactional memory (STM) that does not need special hardware support—it is implemented in software. First, we present a 2-phase-locking STM by giving the pseudocode of its `begin`, `read`, `write` and `commit` functions, then we derive four variants that enable greater concurrency among transactions.

1.4.2.1 *Two phase locking* We first present a naive STM algorithm whose transactions use two-phase-locking. Each read and write access of a transaction t tries to acquire a lock on the accessed memory location. If t succeeds in acquiring all the locks, t is granted an exclusive access to these locations, and t commits. If t cannot acquire a lock, it detects a conflict and may abort. Upon commit or abort, t releases all the locks it acquired. The TM presented in the algorithm below serializes transactions that access common locations as its transaction semantics is two-phase-locking with no distinction on the type of accesses: acquiring locks on all accessed locations (1^{st} phase) and releasing them all in a row (2^{nd} phase).

Algorithm Naive TM for transaction t.

```
 1: begin()_t:
 2:    r-set ← ∅
 3:    w-set ← ∅
 4:    w-log ← ∅

 5: write(x, v)_t:
 6:    if ⟨x, v'⟩ ∉ w-set then
 7:       while !cas(lock(x), unlocked, locked) do
 8:          abort-and-restart()        ▷ contention mgmt
 9:       w-log = w-log ∪ {⟨x, store(x, v)⟩}
10:       w-set = w-set ∪ {⟨x, v⟩}
11:    return(ok)

12: read(x)_t:
13:    if ⟨x, ∗⟩ ∉ w-set then
14:       while !cas(lock(x), unlocked, locked) do
15:          abort-and-restart()        ▷ contention mgmt
16:       v = load(x)
17:       r-set = r-set ∪ {⟨x, v⟩}
18:    else let v be such that ⟨x, v⟩ ∈ w-set
19:    return(v)

20: commit()_t:
21:    for ⟨x, ∗⟩ ∈ w-set do unlock(x)
```

This algorithm depicts the pseudocode of the 2-phase-locking TM algorithm that lets a transaction commits only if it obtains exclusive accesses. Such a transaction is likely to detect a conflict preventing it from committing. As the contention management policy simply aborts and restarts the transaction upon conflict detection (Lines 8 and 15), the same scenario will likely occur later. For brevity, we omitted the description of the abort-and-restart() procedure that consists of resetting the shared locations and thread-private metadata to their default value (abort) and redirecting the control flow to the begin() of the transaction (restart). Below, we explore several modifications of such naive TM algorithm that leads to better performance.

1.4.2.2 *Read sharing* For the previous TM to enable greater concurrency, one can allow *read sharing* to let concurrent transactions read the same memory location.

Algorithm Read sharing TM for transaction t.

```
1: write(x, v)_t:
2:    ...
3:    while !cas(lock(x), unlocked, w-locked) do
4:    ...

5: read(x)_t:
6:    ...
7:    while !cas(lock(x), w-unlocked, r-locked) do
8:    ...
```

This algorithm presents the modifications to obtain a TM with read sharing. The idea is simply to change the mutex locks into read/write locks. The read operation acquires a read lock on location x only if the write lock on x is not acquired, i.e., $lock(x)$ is w-unlocked (Line 7), the write operation acquires a write lock on x only if neither the read nor the write lock on x is acquired, i.e., $lock(x)$ is unlocked (Line 3).

1.4.2.3 Time of update The previous TM algorithms execute update *in-place*, meaning that each modification of a write access is immediately reported in memory. Another approach, called *deferred* update, aims at recording each write access into a redo-log and to report their logged modifications to the memory at commit time. The algorithm below presents the changes to make to the previous one to obtain a TM with deferred-update transactions. Note that by postponing the time of memory update, there is no need to maintain an undo-log with the old version, $w\text{-}log$ (Line 5), as no value has to be reverted in case of abort.

Algorithm Deferred update TM for transaction t.

```
1:  write$(x, v)_t$:
2:     if $\langle x, v' \rangle \notin w\text{-}set$ then
3:        while !cas(lock$(x)$, unlocked, w-locked) do
4:           abort-and-restart()
5:        ~~$w\text{-}log = w\text{-}log \cup \{\langle x, \text{store}(x, v)\rangle\}$~~
6:        $w\text{-}set = w\text{-}set \cup \{\langle x, v \rangle\}$
7:     return(ok)

8:  commit$()_t$:
9:     for $\underline{\langle x, v \rangle \in w\text{-}set}$ do $\underline{\text{store}(x, v)}$
10:    for $\langle x, * \rangle \in w\text{-}set$ do unlock$(x)$
```

In-place update transactions have a lightweight commit phase but a costly abort phase. They require to record all their write accesses into an undo-log to revert the memory appropriately upon abort. Moreover, they have to protect their modifications until commit time to preserve their isolation. Otherwise the TM could suffer from *cascading aborts*: a transaction t that aborts forces the transactions that have read t's modifications to abort as well, provoking, in turn, additional aborts.

Problems due to the lack of isolation can be even more dramatic: division-by-zero, infinite loops... A TM copes with these issues if it ensures opacity [17], i.e. any execution it produces is equivalent to an execution where transactions execute sequentially, respecting the order of non-concurrent transactions in the original execution, and where all transactions, including aborting ones, do not observe a non-committed state.

Deferred update transactions have to replay the redo-log at commit-time but do have a lightweight abort phase as no modifications have to be reported in memory in case the transaction aborts. Additionally, a deferred update transaction may make its modifications invisible from concurrent transactions until it commits, which may represent a waste of effort as running transactions that will eventually abort keep cores busy. Invisible write transaction t has, however, the advantage of letting other transactions access same locations and commit before t reaches its commit phase. We discuss in the next section how to make write invisible.

1.4.2.4 *Write invisibility*

A TM with in-place updates cannot have invisible writes, as by modifying the shared location before it commits, other concurrent transactions accessing the same location are guaranteed to see the modification. Conversely in a deferred update transaction, the writes can be invisible until commit time. Deferred update transactions can also have visible writes, by changing the metadata associated with some locations. For example, the TM resulting from modifications of the previous algorithm have visible writes as it locks a location x at the time the write on x is executed and before commit time. While accessing the same location x, concurrent transactions detect that the lock on x has been acquired.

Algorithm Invisible write TM for transaction t.

```
write($x, v$)$_t$:
  **if** $\langle x, v' \rangle \notin w\text{-}set$ **then**
    **while** ~~!cas(lock($x$), unlocked, w-locked)~~ **do**
      ~~abort-and-restart()~~
    $w\text{-}set = w\text{-}set \cup \{\langle x, v \rangle\}$
  return(ok)

commit()$_t$:
  **for** $\langle x, v \rangle \in w\text{-}set$ **do**
    **while** !cas(lock($x$), unlocked, w-locked) **do**
      abort-and-restart()
    store($x, v$)
  **for** $\langle x, * \rangle \in w\text{-}set$ **do** unlock($x$)
```

As illustrated in the algorithm above, the TM has simply to postpone the locking of x from the write time to the commit time to make writes invisible from concurrent transactions.

1.4.2.5 *Time-based Implementations*

In 2006, time-based TM algorithms [9, 30] were suggested as an alternative to purely lock-based TMs.

The aforementioned lock-based TM implementations rely on locks that simply indicate whether a memory location is protected. Typically, a transaction t_1 locks a memory location between the time it reads it and the time it commits preventing a concurrent transaction t_2 from overwriting the read value during this time interval. This prevention is however restrictive as t_1 could still be serialized before t_2 even if t_2 is allowed to write. An alternative is to use a global counter so that each transaction and each memory location get assigned an associated version, whose comparison indicates whether memory locations can be read by the transaction.

Algorithm Time-based TM for transaction t.

```
begin()$_t$:
  ...
  $ts \leftarrow counter$                       ▷ read global counter

commit()$_t$:
  $t \leftarrow counter$++                      ▷ set a new version
  **for** $\langle x, v, t \rangle \in w\text{-}set$ **do**
    **while** !cas(lock($x$), unlocked, w-locked) **do**
      abort-and-restart()
    store($x, v, t$)                            ▷ record new version
  **for** $\langle x, *, * \rangle \in w\text{-}set$ **do** unlock($x$)

read($x$)$_t$:
  **if** $\langle x, *, * \rangle \notin w\text{-}set$ **then**
    **while** !cas(lock($x$), w-unlocked, r-locked) **do**
      abort-and-restart()
    $\langle v, t \rangle$ = load($x$)                    ▷ get version
    **if** ($t > ts$) **then** try-update()       ▷ validate
    $r\text{-}set = r\text{-}set \cup \{\langle x, v, t \rangle\}$
  **else** let $v$ be such that $\langle x, v, * \rangle \in w\text{-}set$
  return($v$)
```

Transaction t calls try-update when it reads a memory location whose version t is more recent than the time ts the transaction started, indicating that the location has been concurrently updated. The try-update tries to update ts by checking if all previously read locations have not been overwritten since t has started. Upon failure, the transaction aborts.

1.4.2.6 Read invisibility A TM that executes *invisible read* transactions ensure that no transactions can detect read accesses of pending transactions. If transaction t executes invisible reads then no concurrent transactions can detect related conflicts, thus it is the role of t to validate its set of read accesses, $r\text{-}set$, at commit time to make sure that all these remain consistent. Although a monomorphic TM algorithm with exclusive read accesses cannot have invisible reads, allowing shared read accesses from two concurrent transactions, as we did in modifications of Section 1.4.2.2, permits having invisible reads. (Transaction monomorphism and polymorphism are discussed in Section 1.6.2.) We now present the additional modifications necessary to obtain a TM with invisible reads.

Algorithm Invisible read TM for transaction t.

```
1: read($x$)$_t$:
2:    if $\langle x, *, * \rangle \notin w\text{-}set$ then
3:       while ~~!cas(lock(x), w-unlocked, r-locked)~~ do
4:          ~~abort-and-restart()~~
5:       ...
6: commit()$_t$:
7:    for $\langle x, v, t \rangle \in r\text{-}set$ do
8:       if $v \neq$ load($x$) then abort-and-restart()
9:    ...
```

The read no longer acquires locks, so that concurrent transactions cannot observe that a location has been read-locked. The reading transaction could check that the current location is locked by an updating transactions and abort if so, but it does not have to check whether its read is consistent at the time it reads, if all transactions have invisible writes. The validation of an invisible read transaction is done at commit time by checking that all its read accesses are still valid at the commit time, if not a conflict is detected (Line 8).

1.5 PERFORMANCE LIMITATIONS

TM has become very popular these ten past years. Few years ago however, several researchers expressed their skepticism about this programming paradigm. This tendency follows the well known Gartner hype-cycle and seems to indicate that TM, after having been on the peak of inflated expectations is reaching the trough of disillusionment. This reveals a critical period after which TM becomes a mature idea.

1.5.1 Trough of Disillusionment

This disillusionment was expressed in an experimental paper, invited for publications in the October 2008 issue of CACM, questioning the capability of STM-based

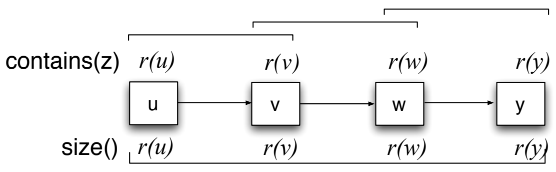

Figure 1.2 Sorted (lexicographically) linked list example in which two operations with same accesses have distinct semantics: `contains(z)` is a parse operation whereas `size()` is a snapshot operation.

concurrent applications to even speedup the performance one could obtain from a single-threaded sequential application [4]. Differing results [10], which appeared recently also in CACM, state that STM is finally a mechanism that scales significantly the performance of various applications as the level of parallelism increases, provided that the underlying multicore machine features enough hardware parallelism.

A more theoretical paper shed some light on some formal appealing properties, like read invisibility and strict disjoint-access-parallelism, transactional memory implementations fail in guaranteeing [2]. Disjoint access parallelism properties may present a lighter impact when placed in the shared memory context where accessing shared data is fast and where the TM bottleneck is more susceptible to come from the lack of concurrency or the ignorant contention management policy than the sharing of centralized counter metadata. Recently, a workshop outlined a potential gap between recent theoretical concerns about TM and its practical concerns [29].

To conclude, as TM has become more and more popular for the challenging goal of simplifying concurrent programming, several researchers expressed their skepticism about the adoption of such paradigm mainly due to the overhead of their software implementations. After this trough of disillusionment, TM have finally won its spurs, perhaps already reaching the slope of enlightenment that is common to mature ideas.

1.5.2 Classical Transactions limit Concurrency

In a concurrent environment, two operations may look very similar even though they do not share the same semantics as explained in [15]. For example, this is the case for a $\texttt{contains}(z)$ operation parsing a linked list data structure and failing in finding element z and another operation $\texttt{size}()$ capturing an atomic snapshot of the number of elements of this data structure. Both operations have the same sequence of read/write accesses, yet they have distinct semantics. Figure 1.2 depicts the reads $r(*)$ and write $w(*)$ of these operations.

The $\texttt{contains}(z)$ is consistent even though y is concurrently inserted after $r(x)$ occurs. Identifying the modification of the next pointer of x to insert y as a conflict with $r(x)$ would unnecessarily limit concurrency: we thus refer to such a conflict as a *false-conflict*. Conversely, the $\texttt{size}()$ requires for example that x and y, which are both counted, were both present in the linked list at the same time. Hence, `contains`(z) enables theoretically more concurrency than `size`$()$ as it tolerates concurrent updates and a fine-grained locking technique (e.g., hand-over-hand locking) will naturally benefit from this additional concurrency. To implement these two different operations with transactions, the programmer encapsulates all accesses within transaction delimiters. The semantics of the transactions has of course to be strong enough to support both semantics. The drawback is that transaction-based `contains` does not enable greater concurrency than `size`, thus classical transactions abort in an over-conservative manner.

Despite having the appealing property of composition, transaction-based algorithms are known to execute generally slower than lock-based and lock-free alternatives, in part for this aforementioned reason. In the next section we present how an average programmer can use a TM library to implement a concurrent application that overcomes these issues.

1.6 RECENT SOLUTIONS

We describe two different techniques: exposing commutativity and using transaction polymorphism. The former technique exploits concurrency between operations by letting the expert programmer explicitly ignore ordering constraints between commutative operations. The latter technique exploits operation concurrency by using polymorphic transactions for various operation semantics.

1.6.1 Exposing Commutativity

Novel TM models [28, 21, 13] have been proposed to overcome the lack of concurrency. The idea common to these models is to let the TM ignore the false read-write conflicts when executing high-level operations that are commutative. As a drawback of exploiting commutative operations, delimiting transactions within a sequential program is no longer sufficient to obtain a concurrent program.

1.6.1.1 Open nesting The open nesting model [28] allows a transaction to commit and report its changes to the memory while being nested inside a transaction that has not committed yet. The key idea is to enable higher concurrency by splitting high level operations into transactions and to define appropriate abort handlers that would compensate the effect of committed inner transactions if their parent transaction aborts. To enable higher concurrency, each transaction keeps track of the high level operations that they have executed. In some cases though, they also have to keep track of lower level operations to be able to compose.

1.6.1.2 Transactional boosting Transactional boosting [21] aims at transforming a linearizable object into a transactional object by implementing transactions that call the high-level operations from an external thread-safe library. These operations do not have to execute speculatively, and can for example be implemented using lock-free primitives. As the library operations apply directly their low-level changes in memory without keeping track of them, the library can only provide invertible operations and must accordingly provide their respective inverses.

. The commutativity-based transaction models are more complex to use than other transaction models as a sequential program cannot be converted into a concurrent program by simply using transaction delimiters. The application programmer has to identify the pairs of commutative operations and must implement specific compensating actions or complex abort handlers for each transaction.

1.6.2 Transaction Polymorphism

A recently suggested synchronization technique, called *transactional polymorphism* [15], provides transaction control to the average programmers to improve performance of their concurrent applications. The key idea is to provide transactions of distinct semantics that can execute concurrently, while preserving their respective semantics.

The average programmer can exploit these various semantics to implement high-level linearizable operations without annihilating their concurrency, whereas the novice can ignore these semantics and use the default transaction semantics for all operations. Although the programmer needs to have some understanding of the various transaction semantics to enhance concurrency of all operations, the novice programmer can straightforwardly implement a potentially slower but safe concurrent program using only transactions with default semantics.

We present how the average programmer can enable greater concurrency by implementing `contains`(z) and `size`$()$ operation presented in Figure 1.2 using transaction polymorphism. First we present two potential relaxed transaction semantics, early release and elastic transaction, and then present complementary form of transactions.

1.6.2.1 Early release The early release mechanism extends the traditional TM interface with a release action. The release [23] is the action of forgetting past reads during the execution of their transaction. This mechanism enhances concurrency by decreasing the number of low-level conflicts for some pointer structures: some of the unnecessary low-level conflicts involving `contains`(z) can thus be ignored. As opposed to the explicit `Select For Update` that strengthens snapshot isolation in database systems, early release provides an explicit `release` that weakens serializability. It requires the programmer to carefully determine when and which objects in every transaction can be safely released [33]: if an object is released too early then inconsistencies may happen.

Such a technique exposes additional calls to the programmer, hence to benefit from early release the programmer cannot simply delimit regions of a sequential program to obtain a concurrent one. The simple TM interface as well as the ABI mentioned in the Introduction do not support such explicit calls.

1.6.2.2 Elastic transactions Elastic transactions enable transaction polymorphism without the need to change the TM interface. The programmer simply has to delimit regions of sequential code to obtain a concurrent program.

Syntactic sugar. Elastic transaction [13] is a transactional model that respects the classical ABI but enables greater concurrency than classical transactions. Two versions (in C and Java) of $\mathcal{E}$-STM, combining elastic and classical transactions, have been released[7], and the Java version is currently part of the distribution of Deuce [26], the bytecode instrumentation framework described in Section 1.3.2.

The key idea is that an elastic transaction is a particular form of transaction that executes `begin`/`read`/`write`/`commit` events differently from classical transactions. The programmer has simply to add a parameter to the `begin` delimiter to differentiate an elastic transaction from a classical transaction. As the `begin` delimiters already support parameters in icc and gcc, there is no need to change the ABI to support elastic transactions. Similarly, the Deuce bytecode instrumentation framework also supports meta-information given as parameters to `begin` delimiters. As an example Deuce annotates methods that represent transactions with parameterized annotations and supports elastic transactions. An an example the following `addAll` method adds multiple elements to a `Collection`, atomically:

```
Deuce support for elastic transactions
@Atomic(metainf=“elastic”)
public void addAll(Collection c) {
        for (x : c) this.add(x);
}
```

Semantics. We now present the semantics of elastic transactions. An elastic transaction executes its read-only prefix in a hand-over-hand style by recording the $i^{th}, i+1^{st}, ..., i+k^{th}$ read locations before discarding the i^{th} one. In the linked list example of Figure 1.2, if p_1 executes the `contains`(z) operation in an elastic transaction with $k = 1$, then a concurrent transaction can execute $w(x)$ between $r(y)$ and $r(z)$ as the potential conflicts involving the $r(x)$ start being ignored. An elastic transaction executes the accesses following its read-only prefix as in a default transaction, keeping track of all conflicts with its read accesses for later validation. In Figure 1.6.2.2, all modifications performed by the series of `add` are hence part of the same default transaction.

Elastic transactions are compatible with classical transactions, in the sense that the TM providing elastic transactions also provides classical transactions that can all run concurrently while preserving their respective semantics. Due to this polymorphism, the elastic transaction model allows composition as opposed to usual relaxed transaction models.

For example, one could try to reuse `contains` and `add` operations by composing them to implement an `addIfAbsent`(x, y) operation, which inserts x given that y is absent. However, because the relaxed `contains` ignores some low-level conflicts to enhance concurrency, the concurrent execution of `addIfAbsent`(x, y) and `addIfAbsent`(y, x) leads to an inconsistent state where both x and y are present. The elastic transaction model solves this issue by letting an elastic transaction check

[7] `http://lpd.epfl.ch/gramoli/php/estm.php`

whether it is nested inside another transaction and if so determines the type of its enclosing transaction to determine its own type. Hence, if the enclosing transaction is regular it can switch its type to regular to prevent inconsistencies. If the enclosing transaction is elastic, then the nested one does not have to change its type.

1.6.2.3 Other Aspects

Transaction polymorphism, has been shown to enable more concurrency than monomorphic STMs in [15], hence conveying the intuition that transactions may achieve comparable performance to other synchronization techniques. Experimentations have confirmed these thoughts by showing that transaction polymorphism is more efficient than lock-based, and lock-free synchronization techniques to implement atomic `Collections` in Java [14], where snapshot operations and parse operations use distinct forms of transactions. Finally, other appealing features of transaction polymorphism lies in the way the contention manager could associate priority to each transaction depending on their form.

1.7 CONCLUSION

After years of investigation, TM has become a mature technique that allows to greatly simplify concurrent programming. A complete transactional memory stack has been released [12], it features TM-based applications, TM language extensions for Java, C and C++, x86 instruction set extension for TM support in hardware as well as hybrid mechanisms that combine both software and hardware mechanisms. The major limitations having caused skepticism of users have been addressed: I/O are now supported by compilers and with polymorphism a TM compensates its performance limitations at high level of parallelism.

Building on top of the current advances on the topic of TM, the notions we have presented here aim at giving any programmer of concurrent applications the possibility to exploit the power of transactional memory.

Acknowledgments

The research leading to some results presented in this paper has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement numbers 216852 and 248465.

GLOSSARY

TM Transactional Memory. A concurrent programming paradigm consisting of delimiting regions of sequential code whose execution should look as atomic.

REFERENCES

1. *Atomic Boxes: Coordinated Exception Handling with Transactional Memory*, 2011.
2. Hagit Attiya. The inherent complexity of transactional memory and what to do about it. In *Proceedings of the 29th annual ACM symposium on Principles of distributed computing (PODC)*, 2010.
3. Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: High performance concurrent sets and maps for STM. In *Proceedings of the 29th annual ACM symposium on Principles of distributed computing (PODC)*, 2010.
4. Calin Cascaval, Colin Blundell, Maged M. Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.
5. A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An emprical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating systems principles (SOSP)*, 2001.
6. Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009.
7. Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 67–78, New York, NY, USA, 2010. ACM.
8. D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2009.

9. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *20th International Symposium on Distributed Computing (DISC)*, 2006.
10. Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why stm can be more than a research toy. *Commun. ACM*, 54:70–77, April 2011.
11. Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating systems principles (SOSP)*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.
12. P. Felber, E. Riviere, W.M. Moreira, D. Harmanci, P. Marlier, S. Diestelhorst, M. Hohmuth, M. Pohlack, A. Cristal, I. Hur, O.S. Unsal, P. StenstroÌš andm, A. Dragojevic, R. Guerraoui, M. Kapalka, V. Gramoli, U. Drepper, S. TomicÌ~ and, Y. Afek, G. Korland, N. Shavit, C. Fetzer, M. Nowack, and T. Riegel. The velox transactional memory stack. *Micro, IEEE*, 30(5):76 –87, sept.-oct. 2010.
13. Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *Proceedings of the 23rd International Symposum on Distributed Computing (DISC)*, volume 5805 of *LNCS*, pages 93–107. Springer-Verlag, sep 2009.
14. Vincent Gramoli and Rachid Guerraoui. Technical Report EPFL-REPORT-163379, EPFL, 2011.
15. Vincent Gramoli and Rachid Guerraoui. Brief announcement: Transaction polymorphism. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.
16. Transactional Memory Specification Drafting Group. Draft specification of transactional language constructs for c++, 2009. http://software.intel.com/file/21569.
17. Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP)*, pages 175–184, New York, NY, USA, 2008. ACM.
18. Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, , and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA)*, 2004.
19. Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, pages 48–60, New York, NY, USA, 2005. ACM.
20. M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.
21. Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, 2008.
22. Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

23. Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Proceedings of the 22nd annual ACM symposium on Principles of distributed computing (PODC)*, 2003.

24. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

25. B. Khan, M. Horsnell, I. Rogers, M. Lujan, A. Dinn, and I. Watson. A first insight into object-aware hardware transactional memory. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2008.

26. Guy Korland, Nir Shavit, and Pascal Felber. Deuce: Noninvasive software transactional memory. *Transactions on HiPEAC*, 5(2), 2010.

27. K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

28. J. Eliot B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues (WMPI 2006)*, February 2006.

29. Srivatsan Ravi, Vincent Gramoli, and Victor Luchangco. Transactional memory, linking theory and practice. *SIGACT News*, 41:109–115, December 2010.

30. Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing (DISC)*, 2006.

31. C. J. Rossbach, O. S. Hofmann, D. E. Porter, H. E. Ramadan, B. Aditya, and E. Witchel. TxLinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

32. William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM symposium on Principles of distributed computing (PODC)*, pages 240–248, New York, NY, USA, 2005. ACM.

33. Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*, Jun 2006.