Transaction Polymorphism in Java

Vincent Gramoli
EPFL, Switzerland
vincent.gramoli@epfl.ch

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

Abstract

Concurrent implementations of abstract types usually rely on lock-free primitives or locks and are highly tuned to support a finite set of efficient operations. However, it is very hard to extend such types for specific needs by adding new operations. The synchronization based on memory transactions, which is generally less efficient than based on lock-free primitives or locks, has been proposed in part to address this issue by ensuring that any set of transactional operations can run concurrently.

This paper introduces transaction polymorphism, a synchronization technique that consists of providing more control to the programmer than traditional (monomorphic) transactions to achieve comparable performance to generic lock-based and lock-free solutions.

We show in this paper that, maybe unsurprisingly, monomorphic transactions cannot reach the level of concurrency obtained with locks. As a drawback of sharing the same semantics that needs to be strong enough for various operations, monomorphic transactions seriously underutilize the concurrency potential of such operations. We describe a Java polymorphic transactional memory that includes three distinct transaction types, which we compare against four existing (monomorphic) STMs, as well as against existing lock-based and lock-free synchronization on a collection micro-benchmark and on the STMBench7 macro-benchmark. Our results show that polymorphism can be at least 2.4 times faster than any of these alternatives on 64 threads.

1. Introduction

Concurrent programming is an error-prone task even in Java because it is not easy to understand the semantics associated with some operation (i.e., method). For example, Iterators that are useful abstractions to parse data structures without requiring any knowledge of its implementation, are not atomic. More precisely in the JDK, the fail-fast Iterators raise an exception upon concurrent modification while the weakly consistent Iterators do not ensure that the first element iterated over is still present when iterating over another. Weak semantics are neither trivial to understand nor easy to detect especially when hidden in an existing Iterator-based operation, and implementing an operation the result of which depends on all elements, for example a sum or size operation, could give surprising results.

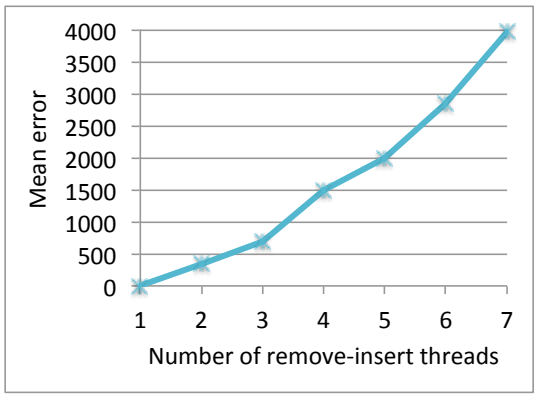


Figure 1. The mean error of the ConcurentLinkedQueue.size() operation as parallelism grows.

The java.util.concurrent package [27] that implements such weakly consistent Iterators provides invaluable low-level synchronization tools and lock-free data structures, one of which being the ConcurrentLinkedQueue data structure. We have recently reported an error regarding its size() operation that was described as atomic in the documentation while not being so in practice.[1] While the add operation of the concurrent skip list implementations is known to support a size operation that "is typically not very useful in concurrent applications"[2] because a concurrent insertion can occur anywhere in the skip list, the add operation of the ConcurrentLinkedQueue always

[1] We have reported this error to the concurrency group of the Java Specification Request (JSR) 166, that did not solve the issue but instead reported it into the documentation on May 20, 2010. We discuss the workaround in Section 5.

[2] Doug Lea. Documentation of JDK1.6.

appends an element to the tail of the queue. Yet, we have observed that the ConcurrentLinkedQueue.size() returns also inconsistent values in the access patterns described below.

If i threads remove and insert repeatedly, say j times, while another tries to compute the size of the data structure, then the size may return in the worst case a value that has an offset error of $i(j-1)$ even though the variation size is only of $\pm i$. This error grows as the number i of concurrent threads that remove/insert increases. Figure 1 indicates the mean error observed on a common dual core machine where $j = 1000$ and when from 1 to 7 concurrent threads remove/insert in a 2^{10} element ConcurrentLinkedQueue. Besides the need to understand the weak semantics associated with such an operation, a programmer needs to re-implement the operation with a stronger semantics to obtain accurate values. This result indicates the difficulty for novice programmers to reuse existing concurrent abstract data types.

The transaction paradigm is an appealing programming idiom for it guarantees to execute in isolation from the other existing transactions. Hence, provided that every operation of an abstract data type (i.e., object) is implemented within the open/close block of a transaction, any new operation (e.g., size) encapsulated within a transaction will also execute as atomically. Typically, a concurrency package implemented exclusively with transactions would provide a simple semantics to reason with. Hence, the novice programmer could reuse this concurrency package straightforwardly to write other transaction-based concurrent programs. Concurrent programming with transactions is simple in part for this reason and because it consists in delimiting regions of sequential code. As a drawback, transactions prevent the programmer from exploiting concurrency as they represent an inflexible synchronization mechanism.

A major limitation of transactions is, thus, that they do not provide control to the programmer. As the programmer cannot give hints on the semantics of each transaction, all transactions execute the same safest semantics—we refer to such transactions as *monomorphic*. Although this is sufficient for novice programmers that want an off-the-shelf solution for writing safely concurrent programs, it appears to be frustrating for expert programmers that cannot obtain the best performance out of it.

An example of such performance limitation, developed in Section 3.1 is that, despite its expertise a programmer will never achieve the same concurrency that can be obtained with locks. This concurrency limitation has a significant impact as it indicates that there exist some transactional programs that will never perform as well as their lock-based counterparts, whatever improvement could be made at the hardware level to diminish the overhead associated with transactional accesses.

We propose transaction polymorphism as an alternative to monomorphic transactions of most existing Software Transactional Memory (STM) implementations [8, 9, 11, 20, 35, 37]. These STMs are monomorphic in the sense that they execute the same semantics without differentiation for all transactions. In contrast, a polymorphic STM provides several transaction semantics among which the programmer can choose to implement a concurrent program. Typically, polymorphic transactions allow novice programmers to benefit from the simplicity of existing STMs by using exclusively the default transaction semantics so that the resulting program remains safe in any case, and it allows an expert programmer to decide to exploit concurrency by implementing each operation within a dedicated transaction semantics.

More specifically, our paper presents the following contributions: Our contributions are as follow:

1. We present two concurrent operations with different semantics such that the concurrency potential of only one of them is dramatically limited when they are implemented using traditional (monomorphic) transactions. In particular, we show that monomorphic transactions cannot achieve the same level of concurrency as lock-based algorithms. Specifically, we show that fine-grained locking techniques can be used to define non-transitive atomicity relations between low-level memory accesses, whereas a transaction, being an open/close block, requires the transitive closure of these atomicity relations, which diminishes its level of concurrency.
2. We implement a polymorphic software transactional memory (PSTM) that provides three types of transactions: *default*, *weak*, and *snapshot*. We illustrate the usefulness of these semantics on a collection micro-benchmark and on the STMBech7 macro-benchmark. Besides guaranteeing atomicity of accesses, this collection is implemented as a concurrent linked list that supports accesses of each type executing concurrently. Polymorphism does not violate extensibility of transactional abstract data types and ensures atomicity of all operations: a programmer can add a new atomic operation to the existing transactional data type without knowing how the data type and PSTM are implemented, and such that atomicity is preserved.
3. We illustrate the benefit of transaction polymorphism by comparing our PSTM against four monomorphic STMs (LSA [37], TL2 [9], Swis-

sTM [11] and NOrec [8]), lock-based and lock-free alternatives from the JDK. Although our solution suffers from the overhead associated with STMs to manage metadata and wrap shared accesses, it appears to be very efficient on workloads with various operation semantics: our results show that PSTM is 3.7 times faster than monomorphism, 4.7 times faster than lock-based solutions and 2.4 times faster than lock-free solutions on the highest level of parallelism we could test, i.e., 64 hardware threads.

Roadmap. Section 2 introduces concurrency relation between synchronization techniques. This relation motivates transactional polymorphism by comparing monomorphism to lock-based synchronization in Section 3. Our implementation of polymorphism, described in Section 4, provides three types of transactions within a single algorithm, PSTM. We discuss existing alternatives in Section 5 and show experimentally that PSTM outperforms monomorphic STMs, lock-based and lock-free alternatives in Section 6. Section 8 discusses nesting of polymorphic transactions and liveness guarantees of PSTM. Section 9 presents the related work and Section 10 concludes the paper.

2. Preliminary Definitions

A shared memory is partitioned into shared registers and metadata used for synchronization of register accesses. Each register represents an atomic unit as far as reads and writes are concerned. A *read* of shared register x that returns value v is denoted by $r(x):v$ or more simply $r(x)$; a write of v on x is denoted $w(x,v)$ or more simply $w(x)$. We refer to $\mathcal{A}$ as the set of read/write accesses. An *operation* π is a sequence of read and write accesses to shared registers and a *critical step* γ is a subsequence of an operation.

Semantics The *semantics* s of an operation π is an assignment of critical steps to a subsequence of its accesses. For example, consider that π is the following sorted linked list contains operation:

$$\pi_1 = r(x), r(y), r(z).$$

Its semantics assigns two critical steps, γ_1 and γ_2, to a sequence of two accesses each such that:

$$\begin{aligned} \gamma_1 &\mapsto r(x), r(y), \\ \gamma_2 &\mapsto r(y), r(z). \end{aligned}$$

This indicates that there should exist a point in the execution where the value returned by $r(x)$ and $r(y)$ where both present, and another point where the values returned by $r(y)$ and $r(z)$ were both present, but not necessarily a point at which both values from $r(x)$ and $r(z)$ were present. Intuitively, the semantics of an operation restricts the set of possible schedules comprising its inner accesses by defining its indivisible critical steps. For example, the semantics of operation π_1 allows its inner accesses to be scheduled with the accesses of an arbitrary operation π_2 in the following way:

$$r(x)_1, r(y)_1, w(x)_2, r(z)_1.$$

The reason is that the value returned by $r(x)_1$ and $r(y)_1$ may be present at a common point of the execution, right before $w(x)_2$ occurs. In contrast, the semantics of π_1 does not allow the following schedule with π_2':

$$r(x)_1, r(y)_1, w(y)_{2'}, w(z)_{2'}, r(z)_1.$$

Here, $r(y)_1$ and $r(z)_1$ cannot return values that were both present at any common point of the execution.

Synchronizations We consider three *synchronization techniques* (or *synchronizations* for short) to protect accesses to shared registers. (i) *lock-based synchronization* with $lock(x)$ and $unlock(x)$ functions taking a shared register as a parameter, (ii) *monomorphic synchronization* with $start(\bot)$ and *try-commit* events delimiting monomorphic transactions, and (iii) *polymorphic synchronization* with $start(p)$ and *try-commit* events, where p is the semantic hint. A *transactional operation* (resp. *lock-based operation*) is an operation whose sets of accesses $\mathcal{A}_{tx}$ (resp. $\mathcal{A}_\ell$) are extended with the events $start(*)$ and *try-commit*, $\mathcal{A}_{tx} = \mathcal{A} \cup \{start(*), \text{try-commit}\}$ (resp. $lock(x)$ and $unlock(x)$, $\mathcal{A}_\ell = \mathcal{A} \cup \{lock(x), unlock(x)\}$). We also refer to transactional operations as *transactions* for brevity.

- A lock-based operation π_i is *well-formed* if for each data item x every $lock(x)_i$ has a following $unlock(x)_i$ event in π_i (we say that x is *locked* by π_i between $lock(x)_i$ and $unlock(x)_i$);
- A transactional operation π is *well-formed* if it starts with a $start(*)$ event and ends by a matching *try-commit* event.

For example, the following lock-based operation is not well-formed as π_1 is still locked after the execution of π_1, i.e., an $unlock(x)$ is missing:

$$\pi_1 = lock(x), lock(y), r(x), r(y), unlock(y).$$

As another example, the following transactional operation π_2 is not well-formed as the transaction ends by a $r(y)$ event:

$$\pi_2 = start(\bot), lock(y), r(x), \text{try-commit}, r(y).$$

Schedules A lock-based (resp. transactional) *schedule* $\mathcal{I}$ is a sequence of events of well-formed lock-based (resp. transactional) operations. Below is a schedule

running example used in Figures 3 and 4 where processes p_1, p_2, p_3 execute operations π_1, π_2, π_3, respectively, is the following:

$$\mathcal{I}_1 = r_1(x), w_3(z), r_1(y), w_2(x), r_1(z).$$

A corresponding transactional schedule is the following:

$$\begin{aligned}\mathcal{I}_2 \quad = \quad & start_1(\bot), r_1(x), start_3(\bot), w_3(z), try\text{-}commit_3, r_1(y), \\ & start_2(\bot), w_2(x), try\text{-}commit_2, r_1(z), try\text{-}commit_1.\end{aligned}$$

Two critical steps γ_1 and γ_2 are concurrent in schedule $\mathcal{I}$ if they belong to distinct operation, and one of the events mapped by γ_1 is ordered in $\mathcal{I}$ after the first event of γ_2 but before the last event of γ_2. Take as an example the aforementioned semantics of π_1 and consider schedule $\mathcal{I}_1$. The first critical step γ_1 of π_1 and the critical step(s) of π_3 are concurrent. Similarly, the second critical step γ_2 of π_1 and the critical step(s) of π_2 are concurrent.

Histories Intuitively, a history H is the result of the execution of a schedule $\mathcal{I}$ by synchronization $\mathcal{S}$.

More formally, a *transactional history* H_{tx} is the result of the execution of the transactional schedule $\mathcal{I}_{tx}$ by a transactional memory where: (i) $start_i$ events in $\mathcal{I}_{tx}$ are $start(\mathsf{def})_i$ in H_{tx} if $\mathcal{S}$ is the monomorphic synchronization or are unchanged $\mathcal{S}$ is the polymorphic synchronization, (ii) one non-*start* event of π_i in $\mathcal{I}_{tx}$ may be replaced by $abort_i$ and in this case the schedule is considered *invalid*; and for the remaining events, (iii) $try\text{-}commit_i$ is replaced by $commit_i$, for any item x, $r(x)_i$ in $\mathcal{I}_{tx}$ is replaced by its corresponding execution $r(x) : v$ in H_{tx} that returns value v or $w(x)$ in $\mathcal{I}_{tx}$ is unchanged in H_{tx}.

For example, the history H_1 resulting from the execution of the transactional schedule $\mathcal{I}_1$ by the LSA [37] monomorphic STM is:

$$\begin{aligned}H_1 \quad = \quad & start_1(\bot), r_1(x), start_3(\bot), w_3(z), try\text{-}commit_3, \\ & r_1(y), start_2(\bot), w_2(x), try\text{-}commit_2, r_1(z), abort_1.\end{aligned}$$

A *lock-based history* H_ℓ is the result of the execution of the lock-based schedule $\mathcal{I}_\ell$ where for any item x, (i) $r(x)$ in $\mathcal{I}_\ell$ is replaced by its corresponding execution $r(x) : v$ that returns value v in H_ℓ, (ii) $w(x)$ and $unlock(x)$ in $\mathcal{I}_\ell$ are unchanged in H_ℓ. Note that the ordering of an input schedule $\mathcal{I}$ is preserved in the resulting history H.

A *sequential history* is a history where no two critical steps are concurrent. Two histories H_1 and H_2 are *equivalent* if they contain the same operations and all their operations have the same events in H_1 and H_2.

A transactional history is *valid* with respect to synchronization $\mathcal{S}$ if it is equivalent to a sequential history and if it does not result from the execution by $\mathcal{S}$ of an invalid schedule (with abort events). A lock-based history is *valid* with respect to synchronization $\mathcal{S}$ if it is equivalent to a sequential history and for each register x, every $lock(x)_i$ event occurs while x is not locked by any $\pi_j \neq \pi_i$.

Note that this notion of valid history is useful to identify synchronization that can execution operations in the order given by the underlying hardware without having to reschedule them. While rescheduling them would allow to improve permissiveness [19], it generally introduces some overhead that impacts performance. This notion is actually similar to the input acceptance [17] which measures the ability for an STM to commit all transactions given by a schedule without changing it. More precisely, this notion generalize input acceptance to a given synchronization $\mathcal{S}$.

Concurrency A schedule is *accepted* by synchronization $\mathcal{S}$ if its execution results in a valid history.

Definition 1 (Concurrency relation). *A synchronization $\mathcal{S}_1$* enables higher concurrency *than synchronization $\mathcal{S}_2$, denoted by $\mathcal{S}_1 \rightrightarrows \mathcal{S}_2$, if there exists a schedule accepted by $\mathcal{S}_1$ that is not accepted by $\mathcal{S}_2$.*

Using this definition, we can strictly compare the concurrency of two synchronizations: $\mathcal{S}_1$ *enables strictly higher concurrency* than another synchronization $\mathcal{S}_2$ if the following properties are satisfied: $\mathcal{S}_1 \rightrightarrows \mathcal{S}_2$ and $\mathcal{S}_2 \not\rightrightarrows \mathcal{S}_1$.

3. Polymorphism

Transaction polymorphism is a synchronization technique that allows multiple transactions, with distinct semantics, to run concurrently.[3] We describe transaction polymorphism by giving two simple operations that benefit from polymorphism to enable greater concurrency by differentiating their semantics.

1. The first operation is a *parse operation* $\pi = r(x_1), ..., r(x_k)$ whose semantics requires that for any $0 \leq i < k$, $r(x_i) \mapsto \gamma_i$ and $r(x_{i+1}) \mapsto \gamma_i$. Intuitively, it consists of parsing a data structure by reading all its elements in order but does not require that all accessed elements be present at a single point in time. The $\mathsf{contains(z)}$ of Figure 2(top) is a parse operation that looks for element z. It is consistent even though x is concurrently inserted after $r(w)$ occurs. Parse operations are also used to move in a search structure in order to insert or remove an element. We show,

[3] The term "polymorphism" is inspired by the polymorphism of object-oriented programming languages that associate a semantics to an operation depending on the object it acts upon, however, transaction polymorphism may apply to non-object-oriented programming languages by parameterizing explicit transaction start events as illustrated in Figure 4 and Algorithm 1 of Section 4.

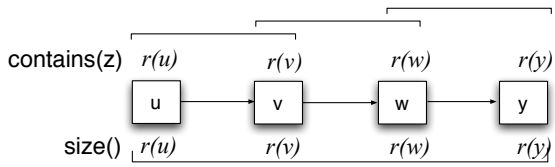


Figure 2. A sorted (lexicographically) linked list example in which two operations with same accesses have distinct semantics: contains(z) is a parse operation whereas size() is a snapshot operation.

in Section 3.1, that lock-based synchronization enables strictly more concurrency than monomorphic synchronization by presenting the hand-over-hand locking [2] implementation of such an operation as a counter-example.

2. The second operation is a *snapshot operation* $\pi = r(x_1), ..., r(x_k)$ whose semantics requires that for any $0 \leq i < k$, $r(x_i) \mapsto \gamma_1$. Intuitively, it also consists of parsing all the elements of a data structure but requires that all the values returned by the parse belong to an atomic snapshot: all values and no other were present in the data structure at the same point in time (which must belong to the operation duration interval). The size of Figure 2(bottom) is a snapshot operation. It requires, for example, that u and y were both present at the same time. Snapshot operations are necessary to iterate over a data structure to compute the sum of its elements or to print out its current status.

3.1 A Weak Transaction for Parses

We show that lock-based synchronization enables strictly higher concurrency than monomorphic synchronization. This motivates the need for polymorphism.

The first part of the proof of Theorem 3 ($\rightrightarrows$) relies on the fact that, unlike lock and unlock events, well-defined transactions are open-close blocks that cannot overlap as depicted by the schedule of Figure 3. Note that a similar argument can be used to show that in Java, reentrant locks enable higher concurrency than synchronized locks.

Lemma 1 ($\rightrightarrows$). *Lock-based synchronization enables higher concurrency than monomorphic synchronization.*

Proof. Let $\mathcal{S}_1$ be the lock-based synchronization and $\mathcal{S}_2$ be the monomorphic transaction synchronization. We show that $\mathcal{S}_1 \rightrightarrows \mathcal{S}_2$ using a schedule accepted when using locks but not accepted when using monomorphic transactions as a counter-example. Consider the schedule depicted in Figure 3. The transactions of Figure 3(b) cannot all commit as the writes from transactions of process p_2 and p_3 occur between the read of x and the read of z from the transaction of p_1, which would violate opacity of transactions. Note that we cannot split the operation in two transactions as the semantics of the operation could be violated during the execution of the schedule. Figure 3(a) depicts the same schedule that is made possible when using lock-based synchronization. □

Lemma 2 ($\not\Leftarrow$). *Monomorphic synchronization does not enable higher concurrency than lock-based synchronization.*

Proof. Let $\mathcal{S}_1$ be the lock-based synchronization and $\mathcal{S}_2$ be the monomorphic synchronization. We show that $\mathcal{S}_2 \not\rightrightarrows \mathcal{S}_1$ by explaining that lock-based can accept any schedule using monomorphic synchronization. For each transaction t_i, try to lock all memory locations that the transaction t_i attempts to access and release them all and restart if one of them is already locked. Then, executes the whole recorded accesses. Finally, unlock all locked memory locations. This reduction allows lock-based synchronization to accept any schedule the monomorphic synchronization can accept. □

By Lemma 1 and 2 we can derive the following Theorem 3.

Theorem 3. *Lock-based synchronization enables strictly higher concurrency than monomorphic synchronization.*

The problem of accepting such schedule is actually solved with a weaker form of transaction that may ignore conflicts induced by their low-level accesses. We refer to *weak transactions* as transactions that ignore at least one of the conflict induced by its accesses at some point during its execution and to *strong* transaction as non-weak transactions.

Weak transactions can be obtained using release actions [23] or more generally when implementing the elastic transaction model [13]. Release actions are used to discard explicitly from the read set the given read entry so that this entry will no longer be validated. 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. Getting back to the schedule presented in Figure 3 with $k = 1$, if p_1 executes an elastic transaction then $w(x)_2$ can occur between $r(y)_1$ and $r(z)_1$ as the entry with location x gets automatically discarded from the read set before $r(z)_1$ occurs. An elastic transaction executes the accesses following its read-only prefix as in a default transaction, keeping track of all read locations in its read set for later validation. We chose elastic transactions to implement the weak transactions of our polymorphic STM presented in Section 4.

p_1	p_2	p_3
lock(x)		
$r(x)$		
lock(y)		lock(z)
		$w(z)$
$r(y)$		unlock(z)
unlock(x)		
	lock(x)	
	$w(x)$	
lock(z)	unlock(x)	
$r(z)$		
unlock(y)		
unlock(z)		

(a) Lock-based execution (based on hand-over-hand locking).

p_1	p_2	p_3
start		
$r(x)$		
		start
		$w(z)$
$r(y)$		commit
	start	
	$w(x)$	
	commit	
$r(z)$		
abort		

(b) Default (monomorphic) transaction execution.

Figure 3. A schedule that is accepted by lock-based transactions but not by classical (monomorphic) transactions. One of the three monomorphic transactions cannot commit in (b) while the lock-based execution is valid in (a).

3.2 A Strong Transaction for Snapshots

We show that weak transactions are not sufficient to implement snapshot operations, we thus propose dedicated snapshot transactions, and show that polymorphism enables higher concurrency than monomorphism. The proofs are deferred to the Appendix.

Theorem 4. *Weak transactions cannot implement snapshot operation.*

Proof. Let π be a snapshot operation that reads memory locations x, y, and z and executes as in the schedule of Figure 4(b). This schedule executed by a weak transaction results in a history where $r(x)_1$ and $r(z)_1$ are not part of the same critical step because of $w(x)_2$ and $w(z)_3$ occurring concurrently. As the weak transaction executed by p_1 commits, the resulting history cannot be valid because there is no equivalent sequential history. Hence weak transactions cannot implement snapshot operations. □

By Theorem 4, we need another transaction semantics. Instead of replacing weak transactions by monomorphic transactions that would have a stronger semantics, we use polymorphic transactions and complement this weak transactions with stronger transactions.

A default single version transaction can be used to implement the snapshot operation, however, it would dramatically limit concurrency by disallowing concurrent modifications. To let snapshot operations a chance to commit while concurrent modifications commit, we introduce a novel transaction semantics based on multiversion concurrency control. Supporting snapshot transactions impose some requirements on the existing transaction semantics (weak and default) so that all updates create a backup copy of the old value before overriding it. As illustrated in Figure 4, maintaining only two versions (one old version and the current one) is sufficient for all three polymorphic transactions to commit in the schedule. We also implement such a transaction in Section 4.

Lemma 5. *Polymorphic synchronization enables higher concurrency than monomorphic synchronization.*

Proof. First, there exists some schedule that is accepted by a weak transaction that is not accepted by a default transaction; such a schedule is depicted in Figures 3 and 4. Second, there exist some operation semantics that weak transaction cannot implement as shown by Theorem 4. Hence monomorphic synchronization has to provide default transaction and by definition cannot also provide weak transactions. Therefore, monomorphic transactions cannot accept the schedule example as shown in Figure 3(b). □

Theorem 6. *Polymorphic synchronization enables strictly higher concurrency than monomorphic synchronization.*

Proof. Let $\mathcal{S}_1$ be the polymorphic synchronization and $\mathcal{S}_2$ be the monomorphic synchronization. The proof that $\mathcal{S}_2$ does not enable higher concurrency than $\mathcal{S}_1$ follows directly from the fact that polymorphic synchronization can provide default transactions. The conjunction of this and Lemma 5 leads to the result. □

4. Implementation of Polymorphism

Our Polymorphic STM, PSTM, differs from existing STMs by providing three types of transaction: *default*, *weak*, and *snapshot*.

Initially, $x = y = z = 0$.

p_1	p_2	p_3
start(snapshot)		
$r(x) : 0$		
		start(default)
		$w(z, 1)$
$r(y) : 0$		commit
	start(default)	
	$w(x, 2)$	
	commit	
$r(z) : 0$		
commit		

(a) Polymorphic transaction execution.

p_1	p_2	p_3
start(weak)		
$r(x)$		
		start(default)
		$w(z)$
$r(y)$		commit
	start(default)	
	$w(x)$	
	commit	
$r(z)$		
commit		

(b) Polymorphic transaction execution.

Figure 4. The same schedule as in Figure 3 that is accepted by polymorphic transactions by using either a weak transaction or a snapshot transaction to execute p_1 snapshot operation.

The pseudocode of PSTM is given in Algorithm 1. Conflicts are detected at the level of accesses to an object field, thus we say that PSTM is *field-based*. All transactions are *time-based* [14]. Each location is associated a versioned lock metadata and each transaction consults a global counter, *clock* (Line 13), and maintains version lower and upper bounds, resp. *lb* and *ub* (Lines 10 and 11), that help detecting whether an access is consistent. As other time-based STMs [9, 37], all transactions update the memory lazily by buffering writes into a write-set (Line 8) until it commits, and have *invisible reads*: none of the read accesses from any transaction is visible from other transactions.

4.1 Default transactions

The default transactions are similar to the transactions of LSA [37]. In fact, default transactions acquire locations eagerly, upon write at Line 17, this allows other transactions to detect a conflict at the time it accesses a location that will be updated by a concurrent transaction (Lines 18 and 37). When a default transaction reads a location with a higher version than *ub*, it attempts to extend its validity range (Line 55): this consists of validating that locations of the read set still have the same version, before upgrading *ub* to the higher version seen.

This extension cannot happen in other transactions because weak ones only maintain a truncated read set, and snapshot ones tend to serialize themselves early, returning old versions rather than new versions.

Upon commit, the read set is revalidated with no limit (∞) on the number of its entries (Line 60), the value-version pairs to override are backed up (Line 62), all write set entries are reported to memory (Line 63) with a strictly higher version obtained from the *clock* counter (Lines 58) and all locks are released (Line 65).

4.2 Weak transactions

Weak transactions are implemented using elastic transactions [13]. A weak transaction relaxes the atomicity of its read-only prefix by discarding automatically locations from its read set, *r-set*. PSTM truncates *r-set* to size 2 to keep track of at most two preceding read entries (Line 43) but validate only one entry upon read.

When a weak transaction has already written (i.e., it is no longer executing its read-only prefix) it behaves as a default transaction, recording all reads into its *r-set*. When a weak transaction still in its read-only prefix reads a location, it creates a new read entry in its *r-set* (Line 41) and discards the oldest (Line 43). The function getVerValVer at Line 40 is a spinning three-read process used to ensure that the value and version returned are consistent. If the read location has a higher version than *ub*, it tries to revalidate the last entry of its truncated read set (Line 47) to make sure the immediate preceding read is still consistent. This typically helps ensuring that a parse operation is still acting on the data structure. When a weak transaction writes for the first time, it has to revalidate the two entries of its *r-set*. A weak transaction commits as a default transaction except that its validation may occur on a truncated read set (Line 60).

4.3 Snapshot transactions

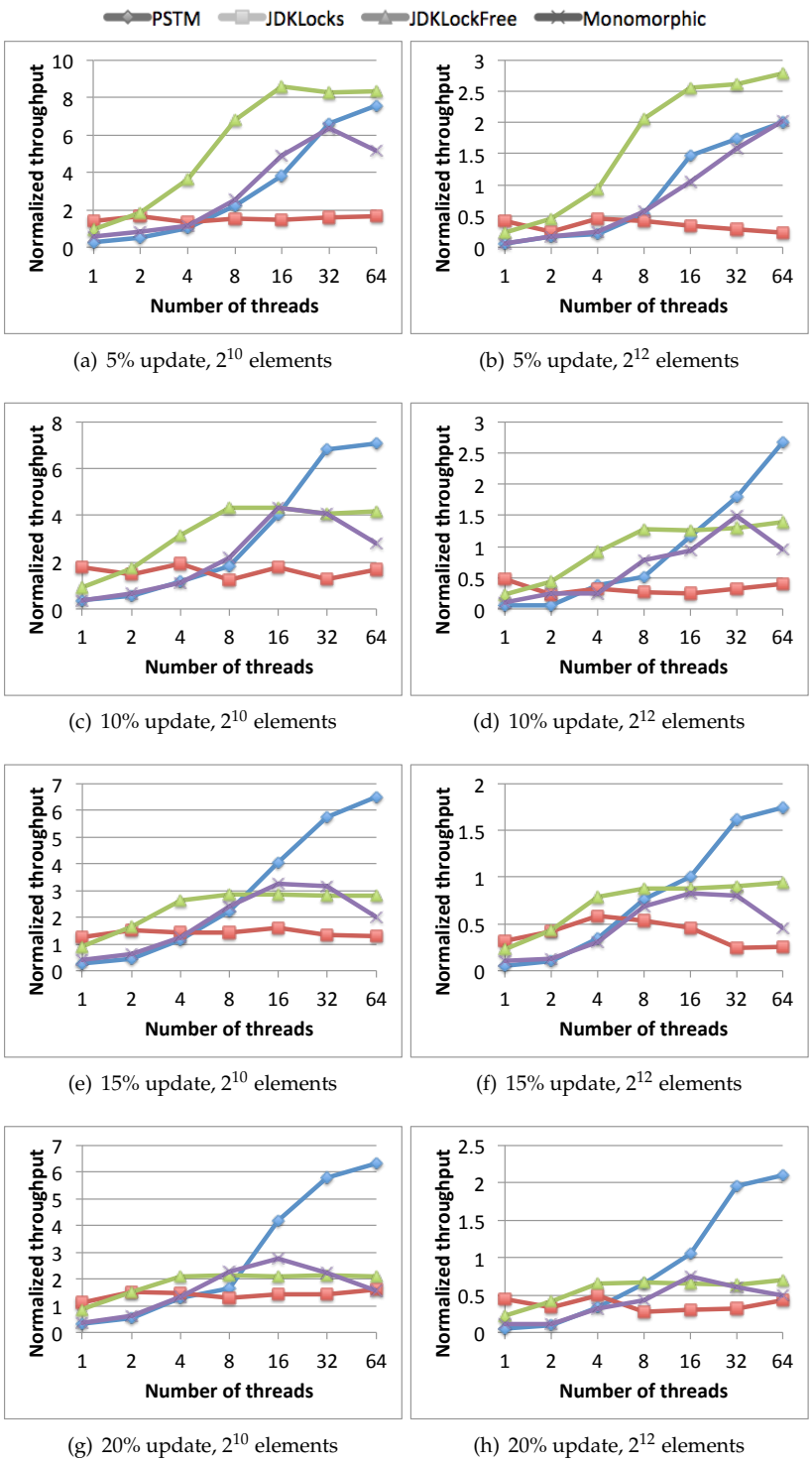
To exploit concurrency between update and snapshot operations, we propose a snapshot transaction that builds upon multiversion concurrency control. Multiversion concurrency control for transactions is not

Algorithm 1 PSTM, a polymorphic software transactional memory

```
1:  Domain:
2:      X the set of objects
3:      V the set of values
4:      T ⊆ ℕ, the set of versions

5:  State of transaction t:
6:      type ∈ {default, weak, snapshot}, initially ⊥
7:      bkp ⊂ X × T, backup of val-version pairs (initially
        ∅)
8:      w-set ⊂ X × V, the write set (initially ∅)
9:      r-set ⊂ X × T, the read set (initially ∅)
10:     lb ∈ ℕ, the transaction lower version (initially 0)
11:     ub ∈ ℕ, the transaction higher version (initially 0)

12: start(tx-type)_t: // Polymorphism requires a parameter
13:     ub ← clock
14:     lb ← clock
15:     type ← tx-type

16: write(ref, value)_t:
17:     lock = lock(ref)
18:     if lock.owner ∉ {t, ⊥} then // locked by other
19:         abort()
20:     if lock.owner = t then
21:         w-set ← w-set ∪ ⟨ref, value⟩
22:         return
23:     if lock.version > ub then // concurrently written value
24:         if ref ∈ r-set then abort()
25:     if type = weak and inROPrefix() then
26:         // revalidating for weak transaction
27:         try validate(2) catch-e abort()
28:     w-set ← w-set ∪ ⟨ref, value, lock.version⟩
29:     return

30: read(ref)_t:
31:     lock = getLock(ref)
32:     if lock.owner ≠ ⊥ then
33:         if lock.owner = t then // t has written ref
34:             for w ∈ w-set do
35:                 if w.ref = ref then
36:                     return w.val // return the written value
37:         else abort()
38:     else // ref is not locked
39:         if lock.version ≤ ub then
40:             ⟨val, ver⟩ = getVerValVer(ref)
41:             if type = weak and inROPrefix() then
42:                 r-set ← r-set ∪ {ref, val, lock.ver}
43:                 truncate(r-set, 2)
44:             else r-set ← r-set ∪ {ref, val, lock.ver}
45:             return val
46:         if type = weak and inROPrefix() then
47:             try validate(1) catch-e abort()
48:         else if type = snapshot then
49:             try oldVersion = bkp.getVersion(ref) catch-e
50:                 abort()
51:             if oldVersion ≤ ub then
52:                 lb = max(lb, oldVersion)
53:                 return bkp.getOldVersion(ref)
54:         else // type is default
55:             try extend() catch-e abort()

56: commit()_t: // try to commit
57:     if w-set ≠ ∅ then
58:         ts ← clock++ // fetch-and-increment
59:         if ts > ub + 1 then
60:             try validate(∞) catch-e abort
61:         for all w ∈ w-set do // necessary for multiversioning
62:             bkp ← bkp ∪ {⟨getLock(ref), getVal(ref)⟩}
63:             store(w.val, w.ref)
64:             set-ver(entry.field, ts)
65:             unlock(entry.field)
```

a novel idea [3]. The original lazy snapshot algorithm (LSA) [37] and the selective multiversion STM (SMV) [35] maintain multiple versions per accessed object to favor concurrency. The current field-based implementation of LSA favors concurrency by detecting conflicts at the granularity of fields but it favors lightweight metadata management by giving up multiversion concurrency control. In [36] the authors show that maintaining the minimum of versions per object that maximize the variety of output histories comes at a cost. The proposed useless-prefix multiversion (UP MV) STM guarantees this property but, as a drawback, does not support invisible reads.

Building on these recent results, we chose to maintain two versions at each location. All update transactions create a backup value-version pair before overriding them (Line 62). The snapshot transaction has simply to detect that the location it aims at accessing has a higher version than its upper bound *ub* (Line 39) to try getting an older version that could let it commit (line 49). Naturally, the snapshot transaction may have to abort if the older version is still too recent at Line 50 as we do not keep track of more than two versions here.



(a) 5% update, 2^{10} elements (b) 5% update, 2^{12} elements

(c) 10% update, 2^{10} elements (d) 10% update, 2^{12} elements

(e) 15% update, 2^{10} elements (f) 15% update, 2^{12} elements

(g) 20% update, 2^{10} elements (h) 20% update, 2^{12} elements

Figure 5. Throughput (normalized over sequential) obtained when using polymorphic transactions (PSTM), the lock-based synchronizedSet from the JDK, the lock-free copyOnWriteArraySet from the JDK and the highest throughput we obtained from our four monomorphic STMs (LSA, TL2, SwissTM, NOrec). Workloads include 10% of size, from 5% to 20% of add/remove and from 70% to 85% of contains.

5. Alternative Synchronization Techniques

We explore lock-based and lock-free alternatives to implement both snapshot and parse operations efficiently.

5.1 JDK Locks

In Section 3.1 we have presented hand-over-hand locking [2] as a solution to implement parse operations. Such a technique, as opposed to transactions, is limited as it does not naturally apply to snapshot operations. Actually, the programmer willing to implement a snapshot operation, say size, in addition to such lock-based parse operations, would need to lock all distinct elements separately which may add a significant overhead to the execution. Moreover, such an implementation may deadlock with another snapshot operation, say sum, if both access multiple data structures in different order.

As suggested by [34], the solution to atomically iterate over a set of locked elements is to use Collections.synchronized*(), which are wrappers converting a data structure implementing abstractions like Set, Map, Collection into their lock-based counterparts. We use this technique to implement our JDK lock-based version evaluated in Section 6.

5.2 JDK Lock-free

As already mentioned, lock-free data structures that are present in the java.util.concurrent package do not support snapshot operations. For instance, the size operation of the ConcurrentSkipListMap, ConcurrentSkipListSet and ConcurrentLinkedQueue may return values that do not represent the numbers of elements the corresponding data structures have ever had. This problem is common to all operations that use the existing Iterators like toString.

In [1] an interrupted snapshot for the Java size operation is provided, but it is very specific to snapshot objects that provide a scan and a localized update and does not apply to generic Iterator that can modify the structure in subtle ways. In [26] optimistic updates are described as getting a copy of the structure by holding locks, modifying the copy without holding any lock, then committing by switching the current state pointer from the current version to the new copy only if no concurrent changes have occurred in the meantime. This solution is similar to monomorphic transactions as it serializes all update operations even ones updating disjoint data.

An alternative present in the java.util.concurrent package of the JDK consists of using copyOnWriteArrayList and copyOnWriteArraySet that provide lock-freedom on accesses to set and list abstractions. More specifically, they initially store the structure elements into an array and they ensure that all elements get copied upon modification. If a concurrent modification occurs when the array is being parsed, then the whole array is copied and its reference is modified to point to the current version of the array. The modification is applied to this new version of the array but lets the parse operation execute on the older copy.

We have implemented this lock-free technique to evaluate our PSTM algorithm against, in Section 6, with reasonably low update ratios. There exist other lock-free solutions to this problem as one can make other operations lock-free using a reference that indicates the current version of the data structure [21] or when provided with a primitive modifying multiple data atomically [7].

6. Evaluating Synchronization Techniques

For the sake of comparison, we experimented our Polymorphic STM, PSTM, the JDK locks and the JDK lock-free techniques, as well as state-of-the-art STM libraries: LSA [37], TL2 [9], SwissTM [11] and NOrec [8] using the Deuce bytecode instrumentation framework [24].

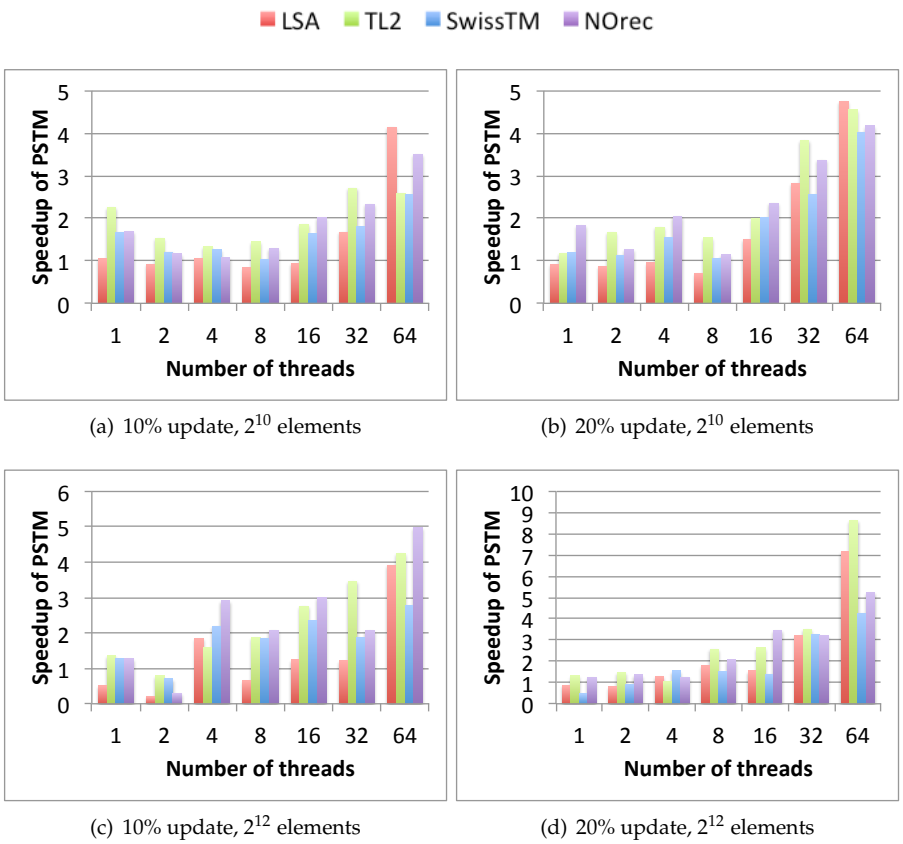
We have implemented an integer set with add, remove, contains and size operations on a sorted linked list data structure, using each of the techniques mentioned above. The workloads consist of 10% of size operations, from 5% to 20% of add/remove and from 70% to 85% of contains.

Our machine is an UltraSPARC T2 (Niagara 2) running up to 64 concurrent hardware threads on 8 cores. In the graph below, the throughput is averaged over 3 runs of 3 seconds each. Each throughput value results from the average throughput obtained from 3 runs of 3 seconds each (each run being preceded by an additional run of 10 seconds used to warmup the JVM).[4]

6.1 Polymorphic STM vs JDK

Figure 5 shows the throughput normalized over bare sequential code of PSTM, of existing monomorphic STMs, and of existing lock-free and lock-based solutions. By normalized throughput we mean for example that when the curve of PSTM is above 1, PSTM has a higher throughput than sequential running on a single thread. (The absolute throughput values observed range from 1 and 250 operations per milisecond.) About the monomorphic STMs curve, we have chosen, for each single point, the maximum throughput we obtained from LSA, TL2, SwissTM, and NOrec. The detailed speedup of PSTM over each of this STMs is presented in Section 6.3.

[4] For all our experiments, we used the Java HotSpot server VM v1.5.0_20-b02 in server mode and with 2G of initial and maximum Java heap size.



(a) 10% update, 2^{10} elements

(b) 20% update, 2^{10} elements

(c) 10% update, 2^{12} elements

(d) 20% update, 2^{12} elements

Figure 6. Speedup of PSTM over monomorphic STMs: LSA, TL2, SwissTM and NOrec, from 1 to 64 threads. (The throughput is identical when speedup has value 1.)

The overall performance of PSTM is better than synchronization alternatives. At very low levels of contention, when update ratio is 5% or at low number of threads, PSTM executes slower than lock-free and lock-based alternatives. The reason is that PSTM suffers from the overhead that is common to STM implementations including monomorphic ones. This overhead is however rapidly compensated as PSTM scales well with the contention whereas the lock-free solution scales badly and the lock-based solution does not even scale. More precisely, PSTM speeds up the existing lock-free solution by 2.4× on average, and the existing lock-based solution by 4.7× on average at the highest level of parallelism we have at our disposal (64 hardware threads).

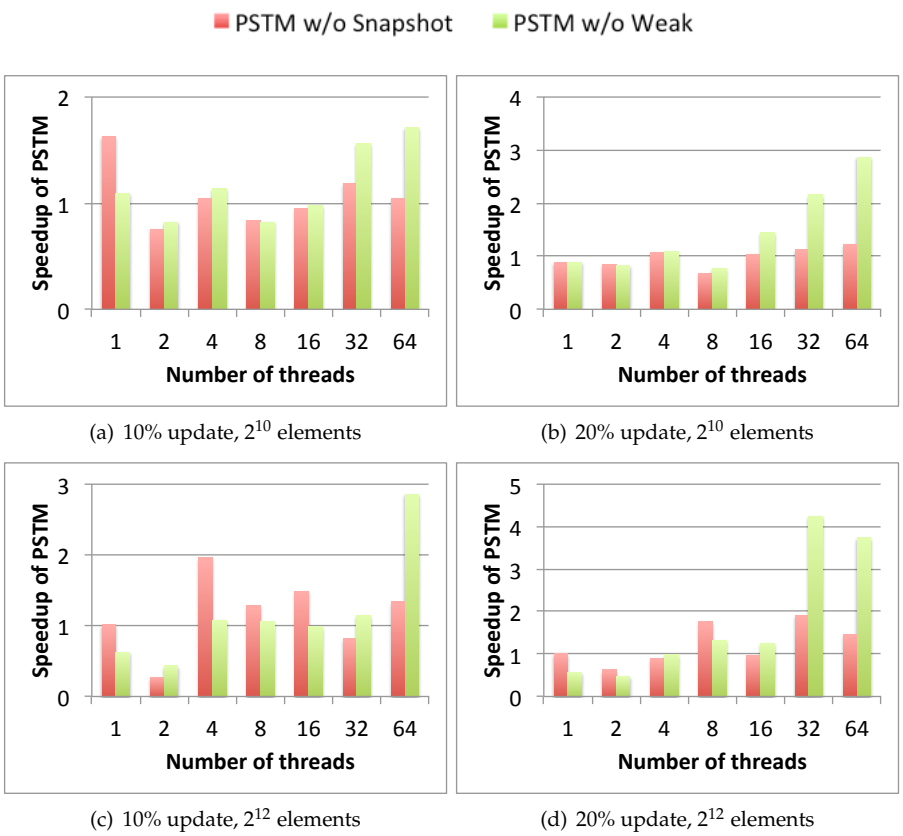
6.2 Polymorphism vs Monomorphism

Figure 6 gives the speedup of PSTM over monomorphic STMs, LSA, TL2, SwissTM, NOrec, as the throughput of PSTM divided by the throughput of the corresponding monomorphic STM.

These results show that PSTM scales better than other STMs. More precisely, PSTM presents a slight overhead at low levels of parallelism, typically when running a single thread but rapidly compensates this slight overhead in concurrent executions. This overhead is probably caused by the fact that polymorphism adds some necessary check at each access to determine the type of the current transaction and because it records one version at each write to provide multi-version concurrency control. At large levels of parallelism, PSTM is significantly more efficient as its polymorphism exploits adequately concurrency whereas monomorphic STM executes a single type of transaction, which has a fortiori the strongest semantics that also limits concurrency. More precisely, PSTM outperforms the tested monomorphic STMs by a factor of 3.7 on average on 64 threads. This improvement is specific to polymorphism as PSTM outperforms each monomorphic STM by at least a factor of 3.1 on 64 threads.

6.3 Comparing Transaction Semantics

We have also evaluated the advantage of combining three transaction semantics instead of only two. Figure 7 illustrates the speedup of using the three semantics (PSTM) over the use of only two of them. “PSTM



(a) 10% update, 2^{10} elements (b) 20% update, 2^{10} elements

(c) 10% update, 2^{12} elements (d) 20% update, 2^{12} elements

Figure 7. Speedup of PSTM over the variant that does not use snapshot transactions and the one that does not use weak transactions (The throughput is identical when speedup has value 1.)

without Snapshot" indicates the speedup of PSTM over the variant where all snapshot transactions have been replaced by default transactions. (All transactions of this variant are either default or weak.) "PSTM without Weak" indicates the speedup of PSTM over another variant where all weak transactions have been replaced by default transactions. (All transactions of this variant are either default or snapshot.)

The overall result is that PSTM speedup grows with the contention. The gain of using three distinct semantics over two is negligible or null at low levels of parallelism, however, it becomes significant at higher levels of parallelism. This result is not surprising as we expected the combination of the three semantics to be especially suited to limit the number of aborts, hence it is natural for its gain to be visible when the contention is high. Another interesting observation is that the speedup of PSTM over "PSTM without weak" is generally higher than its speedup over "PSTM without snapshot", seemingly indicating that weak transactions better boost performance than snapshot ones.

7. Polymorphism in STMBench7

To apply Transaction Polymorphism in more realistic settings, we evaluated it using the STMBench7 benchmark [18]. This section presents our results.

7.1 STMBench7 at a Glance

STMBench7 is an extended adaptation of the OO7 [5] database benchmark for software transactional memory. It has been implemented in both Java and C++ and we use the Java version here. Like OO7, STMBench7 aims at mimicking CAD/CAM/CASE applications without modeling any particular one. Its primary goal is to provide a set of workloads that correspond to realistic and complex object-oriented applications that benefit from multi-threading.

The data structure used by STMBench7 comprises a set of graphs and indexes. More precisely, it includes several modules, each containing a tree of assemblies. Each leave of these trees contains, in turn composite parts. The composite part has a document assigned to it and links to a graph of atomic parts which are connected via connection objects. The benchmark provides

Transaction	Original description	Semantics
Short Traversal 1	search & scan assembly	weak
Short Trav. 6–8	update docs & parts	default
Short Trav. 10	update a part	default
Queries 1–7	count parts/assemblies	snapshot
Query 6	retrieve assemblies	weak
Traversals 1, 6	count visited parts	snapshot
Trav. 2a–c, 3, 5	update visited parts	default
Trav. 4, 7–9	fetch characters	weak
Operations 9–15	update parts/assemblies	default
Operations 6–8	parse components	snapshot

Table 1. The semantics choices for STMBench7 transactions depending on the benchmark descriptions.

a set of operations, each being executed as a transaction. The various operations update the data structure, traverse it starting from the root or a random part in either top-to-bottom or bottom-to-top order, or search it.

7.2 Mapping Transactions to Semantics

Transaction polymorphism is tightly coupled to the semantics of a particular application. To associate each transaction provided by the benchmark its right semantics, we have studied the original meaning of each transaction and decided accordingly, as objectively as possible. Table 1 lists each benchmark transaction, its succinct description and the associated semantics we have chosen for it.

Some transactions that visit and count all visited elements should intuitively be snapshot transactions because (i) all their changes are applied to thread private data and they access shared fields exclusively in read mode, and (ii) they aim at acting on a consistent snapshot of the data structure to avoid the inaccuracy issue presented in the Introduction (Operations 6–8[5], Queries 1–7, Traversals 1–6). Other transactions are good candidates for weak transactions as they simply follow the linked elements of the data structure before updating any shared field, hence they do not require a large consistent view of the structure (Short Traversal 1, Query 6, Traversals 4, 7–9). Finally, the remaining transactions that update non localized shared fields are implemented as default ones (Operations 9–15, Short Traversals 6–8, Traversals 2a–c, 3, 5).

7.3 Settings

The experimental settings are the same as in the previous experiments (UltraSPARC T2 running Java HotSpot server VM v1.5.0_20-b02). The options of STMBench7 were `-w r -g stm -s stmbench7.impl.deucestm.DeuceSTMInitializer -l1 --no-sms`, meaning that the workload is dominated by read operations, STMBench7 uses the deuce runtime to launch the dedicated STMs, each run lasts 1 second (in addition to the benchmark initialization and finalization), structural modification are disabled and long traversals are enabled. As STMBench7 consumes a lot of memory, we had to shrink the default size of the graph structure of STMBench7 to better assess the STM performance. Recall that a single long traversal operation in STMBench7 can last up to half-an-hour [18]—this explains our choices to minimize the size of the benchmark and the length of each experiment.

We have compared the performance the PSTM against the performance of the monomorphic STMs used before (LSA, NOrec, TL2, SwissTM) in addition to the lock-based version, translating each transaction into a critical section. We are not aware of any lock-free implementation of macro-benchmark similar to STMBench7 or OO7 yet the only implementation we could envision would preclude concurrency when a single update operation runs. The throughputs of each STM and of locks (applied to each atomic block) averaged over three runs is depicted in Figure 8.

7.4 Results

Our conclusion is that our polymorphic STM gains become highly visible at high levels of parallelism. Up to 16 threads, polymorphism does not provide the best performance among all solutions. More precisely, its throughput is lower than LSA and NOrec, higher than SwissTM and locks and similar to TL2 up to 16 threads. We conjecture that the memory necessary by PSTM to track multiple versions of each accessed field incurs a significant overhead in a benchmark consuming a lot of memory, compared to lightweight solutions like the single-version LSA and NOrec. This overhead compensates the concurrency improvement of polymorphism when few transactions run concurrently.

It is noteworthy that up to the maximum level of parallelism (64 hardware threads), PSTM keeps scaling. As the lock-based solution does not scale and monomorphic STMs stop scaling at 16 threads, PSTM clearly outperforms other alternatives at 32 and 64 threads. On STMBench7, the highest throughput obtained from PSTM is 2.7 times higher than the best throughput obtained with any other solution, be it based on locks or monomorphic transactions. (In particular, as other alternatives do not scale, the speedup of PSTM becomes even at least 45 times on 64 threads.)

[5] Even though there are no operations named "Operation 1" to "Operation 5" in STMBench7, we kept the original operation naming for convenience.

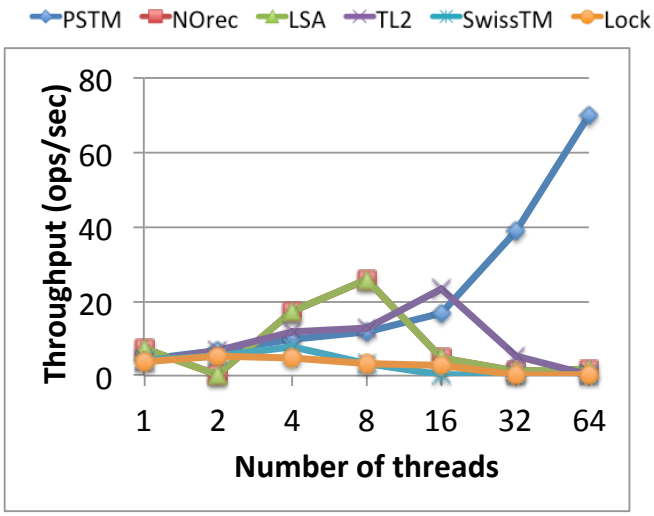


Figure 8. Throughput of our polymorphic STM (PSTM), of the monomorphic STMs (LSA, NOrec, TL2, SwissTM) and of locks obtained on STMBench7.

8. Discussion

In this section, we briefly discuss the liveness guarantees offered by PSTM and the various types of nesting induced when having transactions with different semantics.

8.1 Liveness

PSTM is obstruction-free [22] as it guarantees that some thread takes step eventually if executing for long enough in isolation (without, for example, being preempted by the scheduler). More particularly, PSTM is deadlock-free. Upon writing location x, a transaction t tries to acquire the lock of x until it commits or aborts, where it releases all locks. In case x is locked by a concurrent transaction, t aborts (Algorithm 1, Line 18). To remedy potential livelocks, one could simply use a separate contention manager to ensure that an aborted transaction backs off an increasing amount of time to let other transactions commit. We did not investigate contention management besides evaluating STMs with their default contention manager. One could benefit from transaction polymorphism to assign priorities depending on semantics.

8.2 Nesting

Transaction polymorphism raises crucial questions related to nesting. In short, what should be the semantics of a nested transaction? the semantics indicated by its parameter as if it was not nested, the parent transaction semantics, or the strongest of the two? Transaction nesting is appealing in Java to provide, for example, inheritance by encapsulation: one can extend an object by reusing an existing transactional operation but specializing its behavior.

On the one hand, if the inner transaction is part of a package developed by experts programmers to favor concurrency, it might be desirable to preserve the inner transaction semantics despite nesting (as in Algorithm 1). On the other hand, if a contains and a put transactional operations are encapsulated to obtain an addIfAbsent transactional operation then the semantics of the contains and the put should be changed upon encapsulation to avoid write-skew issues [16]. In this case, Line 15 of Algorithm 1 should be adapted so that the inner transaction would simply have to check the type of the outer transaction, and adopt the strongest of the two (or the default one if no such comparison is possible).

9. Related Work

Database transactions have been specialized for various kinds of application operations to favor their concurrency using different levels of isolation. For example, snapshot isolated transactions commit only if the values it has written have not been overwritten, thus relaxing the serializable transactions [12]. The Escrow transactional model [33] proposes transactions accessing aggregate fields. These transactions simply guarantee that the minimum and maximum values of the field are not exceeded during a speculative execution. Such transactions cohabit with transactions of other kinds in Fast Path [15], however, they always access different data: the former accessing main storage, the other accessing data entries. In contrast, transaction polymorphism targets shared memory applications and preserve the distinct semantics of two transactions even when they access common memory locations.

Various semantics for memory transactions have been suggested in the literature: strong atomicity [29], weak atomicities [30], irrevocability [4], best-effort hardware semantics [10] in independent TM algorithms. Polymorphism goes a step further to let a single TM algorithm run concurrently transactions of different semantics.

Coarse-grained transactions [25] aim at benefiting from high-level semantics to favor concurrency. The pessimistic form of coarse-grained transactions has already been evaluated on Java collections [6, 32] using open nesting. Both implementations suggest that a concurrent size should have a strong enough semantics to return accurate value, as opposed to what by provided in some `java.util.concurrent` data structures. Open nesting [31] enables concurrency by letting a child transaction commits before its parent transaction ends. Using open nesting requires to write a non-trivial abort handler for each transaction, making the

code of an open nested transaction twice longer than usual [32]. In addition, open nesting requires the programmer to define complex abstract locks for each operation to define precisely the set of existing operations it conflicts with, which leads to various problems including deadlocks [6, 32]. Our Java implementation of polymorphism, PSTM, is deadlock-free and requires instead a simple parameter indicating the semantics of a transaction, relieving the burden from the programmer of concurrent applications.

It only applies to invertible operations because an aborting transaction must compensate the effect of its high-level nested transactional operations. For example, this would prevent the programmer from reusing the removeAll collection operation as addAll does not compensate it. In contrast, PSTM is optimistic and can apply to any operation as it provides default transactions.

In addition, several researchers have investigated TM algorithms that adapt the type of transactions depending on workloads. AdaptiveSTM [28] adapts to the current workload by switching the algorithm among four design decisions to achieve the best performance among DSTM [23] and OSTM [20] in various workloads. Other algorithms, like LSA [37], set a transaction *update* field to true when the first write operation occurs to reduce the cost of validation when a read-only transaction tries to commit. These adaptations are automatic and does not give additional control to the programmer. The drawback is that the programmer cannot indicate to the STM the expected semantics of each transaction, hence the safest but most restrictive semantics is systematically chosen by the TM.

10. Conclusions and Open Problems

Paradoxically, the wide-spread adoption of the transaction paradigm is limited by its simplicity that prevents it from reaching the level of concurrency obtained with locks. We have presented transaction polymorphism to provide the programmer with more control on the transaction semantics. Our solution lets the advanced programmers exploit the concurrency of operations of various semantics while still being an off-the-shelf solution for beginners.

We suggested few transaction semantics that we implemented into a polymorphic STM in Java. Our experiments with operations of various semantics show that in addition to being faster than monomorphic STM, polymorphic STM outperforms existing lock-based and lock-free alternatives.

Our polymorphic STM implementation is dedicated to novice and advanced programmers, but further work by experts of the field is necessary to develop new transaction semantics. For example, there are other weak transaction semantics like snapshot isolation. The possibility but also the practical gain of combining them with the existing semantics remain an important open question.

Transaction polymorphism raises also fundamental composition questions. It is unclear whether it is more natural to compose two transactions into another that have a stronger semantics for the sake of safety or a less restrictive semantics for the sake of efficiency. The question becomes more intricate if the original transactions have different semantics or if the semantics are not comparable.

Acknowledgments

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

References

[1] Y. Afek, N. Shavit, and M. Tzafrir. Interrupting snapshots and the java size method. In *DISC*, 2009.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Readings in database systems*, 1988.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[4] A. Bieniusa, A. Middelkoop, and P. Thiemann. Brief announcement: actions in the twilight - concurrent irrevocable transactions and inconsistency repair. In *PODC*, pages 71–72, 2010.

[5] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, SIGMOD '93, pages 12–21, 1993.

[6] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *PPoPP*, 2007.

[7] D. Cederman and P. Tsigas. Supporting lock-free composition of concurrent data objects. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, pages 53–62, 2010.

[8] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPoPP*, 2010.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[10] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *SPAA*, 2010.

[11] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *CACM*, Apr 2011.

[12] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, June 2005.

[13] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, 2009.

[14] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *TPDS*, 21, 2010.

[15] D. Gawlick and D. Kinkade. Varieties of concurrency control in ims/vs fast path. *IEEE Database Eng. Bull.*, 8(2): 3–10, 1985.

[16] V. Gramoli, R. Guerraoui, and M. Letia. Composition vs concurrency. In *WTTM*, 2010.

[17] V. Gramoli, D. Harmanci, and P. Felber. On the input acceptance of transactional memory. *Parallel Processing Letters (PPL)*, 20(1), 2010.

[18] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *EuroSys*, pages 315–324, 2007.

[19] R. Guerraoui, T. A. Henzinger, and V. Singh. Permissiveness in transactional memories. In *Proceedings of the 22nd international symposium on Distributed Computing*, DISC '08, pages 305–319, 2008. ISBN 978-3-540-87778-3.

[20] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, 2003.

[21] M. Herlihy. A methodology for implementing highly concurrent data objects. *TOPLAS*, 15(5), 1993.

[22] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, 2003.

[23] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[24] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory. *Transactions on HiPEAC*, 5(2), 2010.

[25] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. *SIGPLAN Not.*, 45(1), 2010.

[26] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley, 2nd edition, 1999.

[27] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58:293–309, December 2005.

[28] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *DISC*, 2005.

[29] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5, 2006.

[30] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical weak-atomicity semantics for java STM. In *SPAA*, 2008.

[31] J. E. B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, February 2006.

[32] Y. Ni, V. Menon, A.-R. Abd-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP'07*, 2007.

[33] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11:405–430, December 1986. ISSN 0362-5915. doi: http://doi.acm.org/10.1145/7239.7265. URL `http://doi.acm.org/10.1145/7239.7265`.

[34] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley, 2005.

[35] D. Perelman and I. Keidar. SVM: Selective multi-versioning STM. In *Transact*, 2010.

[36] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, 2010.

[37] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, 2006.