Democratizing Transactional Programming

Vincent Gramoli
NICTA and University of Sydney

Rachid Guerraoui
EPFL

Errata: Figure 5 of the printed copy of the CACM paper has a wrong scale on the y-axis, the online version as well as the current version correct this. The number of precluded schedules in Figure 3 is actually 3 out of 20, hence 15% (instead of 20%).

1 A Brief History of Transactions

The transaction abstraction encapsulates the mechanisms used to synchronize the accesses to data shared by concurrent processes. The abstraction dates back to the 70's when it was proposed in the context of databases to ensure the *consistency* of shared data [7]. This consistency was determined with respect to a sequential behavior through the concept of *serializability* [25]: concurrent accesses need to behave as if they were executing sequentially—in other words, they must be *atomic*. Since then, researchers have derived other variants (like isolation [30] and opacity [13]) applicable to different transactional contexts.

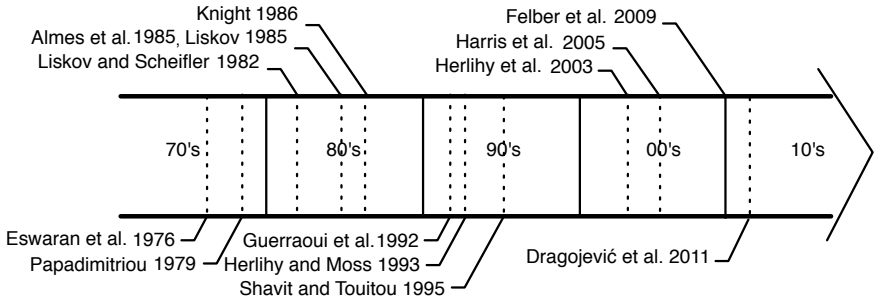

Figure 1: A brief history of transactions

The transaction abstraction was considered for the first time as a programming language construct in the form of *guards* and *actions* in [22]. Then it was adapted to various programming models, e.g., Argus [21], Eden [1] and ACS [12]. The first hardware support for a transactional construct was proposed in [19]. It basically introduced parallelism in functional languages by providing synchronization for multiple memory words. Later, the notion of transactional memory was proposed in the form of hardware support for concurrent programming to remedy the difficulties of using locks, e.g., priority inversion, lock-convoying and deadlocks [18] (cf. Figure 1).

Since the advent of multicore architectures, the very notion of transactional memory has become an active topic of research[1]. Hardware implementations of transactional systems [18] turned out to be limited by specific constraints the programmer could only "abstract away" from using unbounded hardware transactions. Purely hardware implementations are however complex solutions that most industrials are no longer exploring. Instead, a hybrid approach was adopted by implementing a best-effort hardware component that needs to be complemented by *software transactions* [4].

Software transactions were originally designed as a reusable and composable solution to execute a set of shared memory accesses fixed prior to the execution [29]. Later, they were applied to handle the case where the control flow was not predetermined [17]. Early investigations on the performance of software transactions questioned their ability to leverage multicore architectures [2]. Those results were however revisited in [6], where it was shown that a highly optimized STM, with manually instrumented benchmarks and explicit privatization, whose throughput can still outperform sequential code by up to 29 times on SPARC with 64 concurrent threads and by up to 9 times on x86 with 16 concurrent threads. Nonetheless, performance remains the main obstacle preventing the wide adoption of the transaction abstraction for general purpose concurrent programming.

In short, and as we show in this paper, in their classic form, transactions prevent us from extracting the same level of concurrency possible with more primitive synchronization techniques. This is folklore knowledge, yet we show for the first time and through a simple example, that this is inherent to the transaction concept in its classic form and is irrespective of how it is used. One can view this limitation as the price of bringing concurrency to the masses and making it possible for average programmers to write parallel programs that use shared data. However, some programmers are

[1] `http://www.cs.wisc.edu/trans-memory/biblio/list.html`.

concurrency experts and these might find it frustrating not to be able to use their skills in enhancing concurrency and performance.

Not surprisingly, researchers have been exploring relaxations of the classic transaction model [23, 24, 27] that enable more concurrency. Nevertheless, it proved challenging to do so while keeping the simplicity of the original model, namely, the ability to (1) preserve the original sequential code and (2) compose applications devised by different programmers, possibly with different skills.

We argue in this paper for mixing different transaction semantics within the same application; strong semantics to be used by novice programmers and weaker semantics to be used by concurrency experts. The challenge is to make sure the polymorphic system mixing different semantics can still enable to reuse code and compose it in a smooth manner. Before describing how this can be addressed, it is important to have a closer look into the very meaning of reuse and composition.

2 The Inherent Appeal of Transactions

The transaction paradigm is appealing for its simplicity as it preserves sequential code and promotes concurrent code composition.

Algorithm 1 An implementation of a linked list operation with transactions

```
1:  tx-contains(val)_p:
2:    int result;
3:    node *prev, *next;
4:    transaction {
5:      curr = set → head;
6:      next = curr → next;
7:      while next → val < val do
8:        curr = next;
9:        next = curr → next;
10:     result = (next → val == val);
11:   }
12:   return result;
```

2.1 Preserving sequentiality

Transactions preserve the sequential code in that their usage does not alter it, besides segmenting it into several transactions. More precisely, the

regions of sequential code that must remain atomic in a concurrent context are simply delimited, typically by a transaction{...} block as depicted in Algorithm 1—the original structure depicted in Algorithm 2 (left) remains unchanged.

Programming with transactions shifts the inherent complexity of concurrent programming to the implementation of the transaction semantics which must be done once and for all. Thanks to transactions, writing a concurrent application follows a divide-and-conquer strategy where experts have the complex task of writing a live and safe transactional system with an unsophisticated interface so that the novice has simply to write a transaction-based application, namely, delimit regions of sequential code.

Algorithm 2 The linked list node

```
1: Transactional structure node:
2:     intptr_t val;
3:     struct node * next;
4:     // Metadata management is implicit
5: Lock-based structure node_lk:
6:     intptr_t val;
7:     struct node_lk * next;
8:     volatile pthread_spinlock_t lock;
```

Traditional synchronization techniques generally require the programmer to first re-factorize the sequential code. Using lock-free techniques, the programmer would typically need to use subtle mechanisms, like logical deletion [14], to prevent inconsistent memory deallocations. Using lock-based techniques, the programmer must usually declare and initialize explicitly all locks before using them to protect memory accesses, as depicted in Algorithm 2, Line 8.

The transaction abstraction hides both synchronization internals and metadata management. If locks or timestamps are internally used, they are declared and initialized transparently by the transactional system. All memory accesses within a transaction block are transparently instrumented by the transactional system as if they were *wrapped*. These wrappers can exploit the metadata, locks and timestamps to detect conflicting accesses and to potentially abort a transaction.

2.2 Enabling composition

Transactions allow Bob to compose existing transactional operations developed by Alice into a composite operation that preserves the safety and liveness of its components [15] as depicted in Figure 2.

Alternative synchronization techniques do not facilitate composition. Consider a simple directory abstraction mapping a name to a file. With

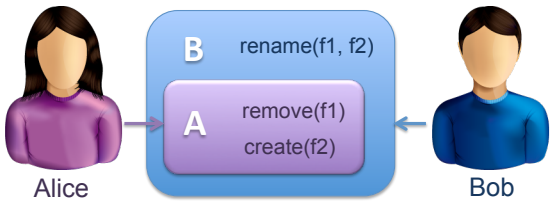

Figure 2: Bob composes Alice's component operations remove and create into a new operation rename that preserves the safety and liveness of its components

transactions, one can compose the removal of a name and the creation of a new name into a rename action. If a user renames a file from one directory d_1 to another directory d_2, while another user renames a file from d_2 to d_1, directories must be protected to avoid deadlocks. In other words, Bob must first understand the locking strategy of Alice to ensure the liveness of his own operations. For this reason, the header of the Linux kernel file `mm/filemap.c` comprises 50 lines of comments explaining the locking strategy. Existing lock-free techniques are even more complex as they require a multi-word compare-and-swap to make the two renaming actions atomic while retaining concurrency [11].

In contrast, a transactional system detects a conflict between the two renaming transactions and lets only one of the two resume and possibly commit; the other one is restarted or resumed later. Deciding upon the conflict resolution strategy is the task of a dedicated service, called a contention manager, for which various strategies and implementations have been proposed [28].

3 The Inherent Limitations of Transactions

A transaction delimits a region of accesses to shared locations and protects the set of locations that is accessed in this region. By contrast, a (fine-grained) lock generally protects a single location even though it is held during a series of accesses as depicted in Algorithm 3. This is a crucial difference between transactions and locks in terms of expressiveness, concurrency and performance.

Algorithm 3 An implementation of a linked list operation with locks

```
 1: lk-contains(val)_p:
 2:   int result;
 3:   node_lk *prev, *next;
 4:   lock(&set → head → lock);
 5:   curr = set → head;
 6:   lock(&curr → next → lock);
 7:   next = curr → next;
 8:   while next → val < val do
 9:     unlock(&curr → lock);
10:     curr = next;
11:     lock(&next → next → lock);
12:     next = curr → next;
13:   unlock(&curr → lock);
14:   result = (next → val == val);
15:   unlock(&next → lock);
16:   return result;
```

3.1 Lacking expressiveness

To make our point that transactions are inherently limited in terms of expressiveness we define *atomicity* as a binary relation over shared memory accesses π and π' of a single transaction within an execution α: $atomicity(\pi, \pi')$ is true if π and π' appear in α as if they were both occurring at one common indivisible point of the execution. It is important to notice that this relation is not transitive, i.e., $atomicity(\pi_1, \pi_2) \wedge atomicity(\pi_2, \pi_3) \not\Rightarrow atomicity(\pi_1, \pi_3)$. In fact, as π_2 may appear to have executed at several consecutive points of the execution, the points at which π_1 and π_2 appear to have occurred may be disjoint from the points at which π_2 and π_3 appear to have occurred.

A process locking x (with mutual exclusion) during the point interval $(p_1; p_2)$ of α, in which it accesses x, guarantees that any of its other accesses during this interval will appear atomic with its access to x. For example, in the following lock-based program where $r(x)$ and $w(x)$ denote respectively read and write accesses to shared variable x, process (or more precisely thread) P_ℓ guarantees $atomicity(r(x), r(y))$ and $atomicity(r(y), r(z))$ but not $atomicity(r(x), r(z))$:

$$P_\ell = \mathsf{lock}(x)\ r(x)\ \mathsf{lock}(y)\ r(y)\ \mathsf{unlock}(x)\ \mathsf{lock}(z)\ r(z)\ \mathsf{unlock}(y)\ \mathsf{unlock}(z).$$

Conversely, a process P_t executing the following transaction

block ensures $atomicity(r(x), r(y))$, $atomicity(r(y), r(z))$ but also $atomicity(r(x), r(z))$, which is the transitive closure of the atomicity relations guaranteed by P_ℓ. Using classic transactions, there is no way to write a program with a semantics similar to P_ℓ, namely to ensure the two former atomicity relations without also ensuring the latter.

$$P_t = \mathsf{transaction}\{\ r(x)\ r(y)\ r(z)\ \}.$$

This lack of expressiveness is not related to the way transactions are used but to the transaction abstraction itself. The open/close block somehow blindly guarantees that all the accesses it encapsulates appear as if there was an indivisible point in the execution where they *all* take effect.

3.2 Impact on concurrency

Not surprisingly, the low expressiveness of transactions translates into a concurrency loss. For example, consider the transactional linked list program depicted in Algorithm 1. Clearly, the value of the $head \rightarrow next$ pointer observed by the transaction (Line 6) is no longer important when the transaction is checking whether the value *val* corresponds to a value of a node further in the list (Line 7), yet a concurrent modification of $head \rightarrow next$ can invalidate the transaction when reading $next \rightarrow val$ as transactions enforce atomicity of all pairs of accesses; this is a false-conflict leading to unnecessary aborts. Conversely, the hand-over-hand locking program of Algorithm 3 allows such a concurrent update (Line 7) when checking the value (Line 8), starting from the second iteration of the while-loop.

To quantify the impact of the limited expressiveness of transactions on the number of accepted schedules, consider a concurrent program where the process P_t above executes concurrently with processes $P_1 = \mathsf{transaction}\{w(x)\}$ and $P_2 = \mathsf{transaction}\{w(z)\}$. As there are four ways of placing the single access of one of these two processes between accesses of P_t and five ways of placing the remaining one in the resulting schedule, there are twenty possible schedules. Note that all are correct schedules of a sorted linked list implementation.

However, most transactional memory systems guarantee that each of their execution is equivalent to an execution where sequences of reads and writes representing transactions are executed one after another (serializability) in an order where no transaction terminating before another start is ordered after (strictness). (This is actually often the case, as a large variety of transactional memory systems ensure *opacity* [13] a consistency criterion that is even stronger than this strict-serializability as it additionally requires

Figure 3: Transactions preclude 15% of the correct schedules of a simple concurrent linked list program.

that non-committed transactions never observe an inconsistent state.) These transactional memory systems actually preclude three of these schedules as depicted in Figure 3: those in which P_t accesses x before P_1 (P_t is serialized before P_1, i.e., $P_t \prec P_1$), P_1 terminates before P_2 starts ($P_1 \prec P_2$) and in which P_2 accesses z before P_t ($P_2 \prec P_t$). This limitation translates here into concurrency loss.

It is worth noting that one could exploit weaker transactional memory systems to export these serializable histories [26, 10]. These would offer a transaction that may not be appropriate for all possible usages. For example, it would be possible that one transaction reads an inconsistent state before aborting. In fact, the concurrency limitation stems from trying to provide a unique but general-purpose transaction.

3.3 Impact on performance

The metadata management overhead of software transactions when starting, accessing shared memory, and committing, is typically expected to be compensated by exploiting concurrency [6]. In scenarios like the previous linked list program where transactions fail to fully exploit all available concurrency, their performance cannot compete with other synchronisation methodologies. Recall that this is due to the expressiveness limitation inherent to transactions—it is thus not tied to the way transactions are used but to the abstraction itself.

To illustrate the impact on performance, we compared the existing Java concurrency package to the classic transaction library, TL2 [5], on a 64-way Niagara 2 machine. Note that this is the Java implementation of the

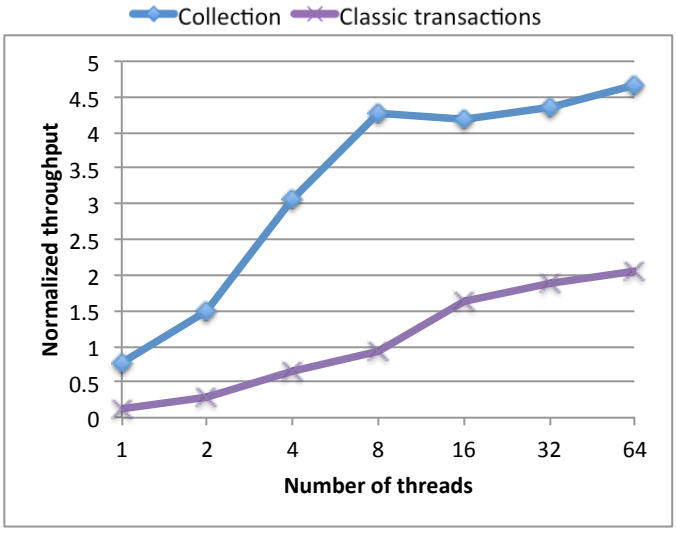

Figure 4: Throughput (normalized over the sequential one) of classic transactions and the existing concurrent collection.

TL2 algorithm, it detects conflicts at the granularity of fields and is distributed within DeuceSTM [20]. We present the results obtained on a simple Collection benchmark of 2^{12} elements providing contains, add, remove and size operations with an update and a size ratios of 10% each. As the existing lock-free data structures do not support atomic size we had to use the copyOnWriteArraySet workaround of this package. We compared it against the linked list implementation building upon TL2.

Figure 4 uses the throughput (amount of committing transactions per time unit) of the bare sequential implementation (without synchronization) as the baseline. It illustrates the throughput speedup (over sequential) one can achieve using either the classic transactions or the existing java.util.concurrent package. When its normalized throughput is 1, the throughput of the corresponding concurrent implementation equals the throughput of the sequential implementation. In particular, the graph indicates that the existing collection performs 2.2× faster than classic transactions on 64 threads. The poor performance of classic transactions is due to their lack of concurrency, problem that will be addressed in the next section.

4 Democratizing Transactions: the Challenge

Traditionally, transactional systems ensure the same semantics for all its transactions, independently of their role in the concurrent applications. As we discussed, these semantics are, however, overly conservative and, by limiting concurrency, may also limit performance (cf. Section 3). Without additional control, skilled programmers will be frustrated by not being able to obtain highly efficient concurrent programs. In order to adequately exploit the concurrency allowed by the semantics of an application it is necessary to trade simplicity off for additional control.

We argue that for the transactional abstraction to really become a widely used programming paradigm it should be *democratized*. Not only should transactions be an off-the-shelf solution for novices, but they should also permit additional control to concurrent programming experts. Therefore, we believe that a simple default semantics should be able to run concurrently with transactions of more complex semantics capturing more subtle behaviors. The challenge is twofold. First, the transaction abstraction should allow the expert programmers to easily express hints about the targeted application semantics without modifying the sequential code. Second, the semantics of each transaction must be preserved even though multiple transactions of different semantics can access common data concurrently.

This second property is crucial and makes the development of a transactional system even more complex.

4.1 Relaxation and sequentiality

Several transaction models have been proposed as a relaxed alternative to the classic one. Two examples are open nesting [24] and transactional boosting [16]. They both exploit commutativity by considering transactional operations at a high level of abstraction. They acquire abstract locks to apply nested operations and require the programmer to specify compensating actions or inverse operations to roll-back these high-level changes. To avoid deadlocks due to the acquisition of new locks at abort time, the programmer may follow lock-order rules or exploit timeouts. Alternatively, other approaches consists in extending the interface of the transactional memory system with explicit mechanisms like various functions, *light reads*, `unit-loads`, SNAP and early `release`. For example, early `release` can be used explicitly by the programmer to indicate from which point of the transaction all conflicts involving its read of a given location can be ignored [17]. The challenge is thus to achieve the same concurrency achievable with these

models while preserving sequential code and composition of transactions.

The elastic transaction model [8] aims precisely at preserving sequential code and guaranteeing composition. This model provides, together with the classic form, a semantics of transactions that enables to efficiently implement search structures. Just like for a classic transaction, the programmer must simply delimit the blocks of code that represent elastic transactions, thus preserving sequential code as depicted in Algorithm 4. Elastic transactions bypass deadlocks by updating memory only at commit time, avoiding the need to acquire additional locks upon abort.

Algorithm 4 Java pseudocode of the add() operation with elastic transactions

```
1:  public boolean add(E e):
2:    transaction(elastic) {
3:      Node⟨E⟩ prev = null
4:      Node⟨E⟩ next = head
5:      E v
6:
7:      if next == null then // empty
8:        head = newNode⟨E⟩(e, next)
9:        return false
10:     while (v = next.getValue()).compareTo(e) < 0 do // non-empty
11:       prev = next
12:       next = next.getNext()
13:       if next == null then break
14:     if v.compareTo(e) == 0 then
15:       return false
16:     if prev == null then
17:       Node⟨E⟩ n = new Node⟨E⟩(e, next)
18:       head = n
19:     else prev.setNext(new Node⟨E⟩(e, next))
20:     return true
21:   }
```

In contrast to classic transactions, during its execution, an elastic transaction can be cut (by the elastic transactional system) into multiple classic transactions, depending on the conflicts it detects. Consider the following history of shared accesses in which transaction j adds 1 while transaction i is parsing the data structure to add 3 at its end.

$$\mathcal{H} = r(h)^i, r(n)^i, r(h)^j, r(n)^j, w(h)^j, r(t)^i, w(n)^i.$$

This history is neither serializable [25] nor opaque [13] since there is no history in which transactions i and j execute sequentially and where $r(h)^i$

occurs before $w(h)^j$ and $r(n)^j$ occurs before $w(n)^i$ (yet the high level insert operations of this history are atomic). A traditional transactional scheme would detect two conflicts between transactions i and j, and would not allow them both to commit. Nonetheless, history $\mathcal{H}$ does not violate the correctness of the integer set: 1 appears to be added before 3 in the linked list and both are present at the end of the execution.

The programmer has simply to label transaction i as being elastic to solve this issue. Then, history $\mathcal{H}$ can be viewed as the combination of several pieces:

$$f(\mathcal{H}) = \boxed{r(h)^i, r(n)^i}^{s_1}, r(h)^j, r(n)^j, w(h)^j, \boxed{r(t)^i, w(n)^i}^{s_2}.$$

In $f(\mathcal{H})$, elastic transaction i has been cut into two transactions s_1 and s_2. Crucial to the correctness of this cut, no two modifications on n and t have occurred between $r(n)^{s_1}$ and $r(t)^{s_2}$. Otherwise the transaction would have to abort.

These cuts enable more concurrency than what the expert programmer could do with classic transactions. First, the cuts are dynamically tried at runtime depending on the interleaving of accesses. As this interleaving is generally non-deterministic, a programmer cannot just split transactions prior to execution and ensure correct executions. Second, as elastic transactions rely on dynamic information they exploit more information than static commutativity of operations. For example, elastic transactions enable additional concurrency between two linked list adds by allowing the history involving transactions t_1 and t_2: $r(h)^{t_1}, r(n)^{t_2}, w(h)^{t_2}, w(n)^{t_1}$ in which neither $r(n)^{t_2}$ and $w(n)^{t_1}$ nor $r(h)^{t_1}$ and $w(h)^{t_2}$ commute.

4.2 Composition and mixture of semantics

The more semantics the transactional system provides, the more control it gives to experts, allowing them to boost performance. The opacity semantics of classic transactions benefit the novice programmer as they are always safe to use. The elastic transactions can also bring added performance in search structures. Interestingly, one could also consider the mix of the opaque classic and the relaxed elastic models with a new semantics, the *snapshot* semantics. This is particularly appealing for obtaining efficiently a result that depends on numerous elements of a data type, like a Java Iterator. As an example, a snapshot transaction implementing a size method is depicted in Algorithm 5.

At first glance, providing as many forms as possible in a single toolbox system may seem the key solution to help develop concurrent applications

once and for all, the challenge lies however in the mixture of these semantics. Mixing these semantics requires letting them access the same shared data concurrently. It is crucial that the semantics of each individual transaction is not violated by the execution of concurrent transactions of potentially different semantics.

Algorithm 5 Java pseudocode of the size() operation with snapshot transactions

```
1:  public int size():
2:    transaction(snapshot) {
3:      int n = 0
4:      Node⟨E⟩ curr = head
5:
6:      while curr ≠ null do
7:        curr = curr.getNext()
8:        n++
9:      return n
10:   }
```

For example, the key idea for highly concurrent snapshot semantics is to exploit multi-version concurrency control to let snapshots commit while concurrent (elastic or classic) updates commit. A typical implementation of a snapshot is to exploit a global counter and a version number per written value, so that the transaction can fetch the counter at start time and decide while reading new locations to return a value that has an appropriate (not too recent) version consistent with this start time. The mixture of the snapshot with classic and elastic transactions requires, however, to make sure all updates (elastic and classic) must record the old value before overriding it.

The mixture problem might even be subtler if a relaxed transaction ignores a conflict that involves a concurrent strong transaction that cannot ignore it. Typically, elastic and opaque transactions handle this issue for read-write conflicts by requiring that only the reading transaction decides upon the conflict resolution. Unlike writes, reads are idempotent so that the semantics of the writing transaction are never altered by the outcome of the conflict resolution. Our solution relies on (1) having invisible reads so that the writing transaction does not observe the conflict and (2) enforcing commit-time validation so that the reading transaction always detect the conflict.

A consequent challenge relates to the composition of the semantics. Bob can directly nest Alice's elastic transactions, into another transaction, choosing between labelling it as elastic, snapshot, or classic, thus guaranteeing

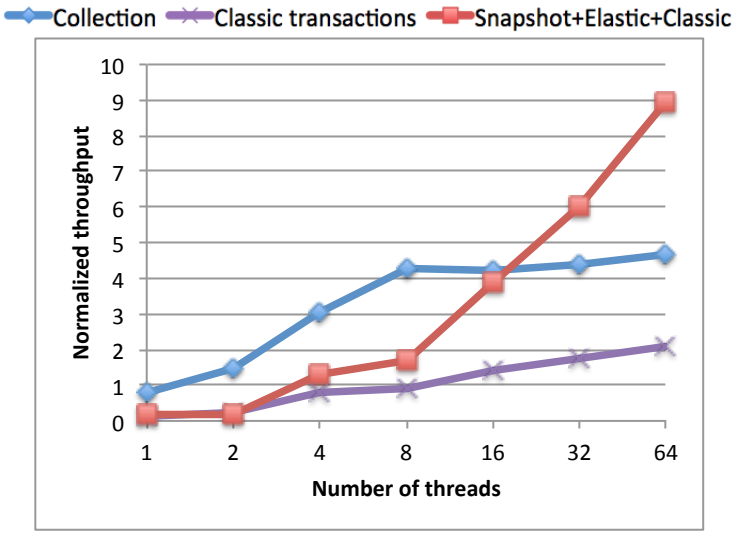

Figure 5: Throughput (normalized over the sequential one) of mixed transactions, classic transactions and the collection package.

atomicity and deadlock-freedom of its own operation. Typically, Bob may use various semantic transactions while encapsulating because Alice. For example, one can imagine that Alice provides an elastic contains(x) that Bob composes into a snapshot containsAll(C) method that returns successfully only if all elements of a collection C are present. For the sake of safety, the strongest semantics of the involved transactions, in this case the snapshot one, applies to all. Hence, a novice, unaware of the various semantics, will always obtain a safe composite transactional method whose opacity would be conveyed to inner transactions. Deciding upon which semantics to apply, when the semantics are incomparable, is an open question.

4.3 Impact on performance

To illustrate the benefit of mixing transactions of different semantics we run the mixed transactions on the collection benchmarks in the exact same settings as before and reported both the new and the previously obtained results in Figure 5. Each of the three parse operations contains, add, and remove is implemented with an elastic transaction, and the size operation, which returns an atomic snapshot of the number of elements, is implemented with a snapshot transaction. The mixed transaction model performs 4.3×

faster than the classic transaction model, TL2, and improves on the concurrent collection package by 1.9× on 64 threads. Thanks to the snapshot semantics, the size operation commits more frequently than with classic transactions. The reason is that a snapshot size returns potentially values that have been concurrently overridden, where classic size would simply be aborted. Even though the overhead of polymorphic transactions makes them slower than the concurrent collection package at low levels of parallelism, the performance scales well, which compensates for the overhead effect at high levels of parallelism.

Since then, the mixture of elastic and classic transactions has also been shown effective in a non-managed language, C/C++, as well. First, it improved the performance of the tree library implemented in the transactional vacation reservation benchmark by 15% [3]. Second, it improved the performance of a list-based set running on a many-core architecture by about 40 times [9].

5 Concluding Remarks

The transaction is an old appealing abstraction that has been the main topic of many practical and theoretical achievements in research. It has however never been widely adopted in practice.

We argue that the reason why the transaction abstraction is appealing as a programming construct is also the reason why it might not be used in practice. In short, the appeal comes from the simplicity and the very fact that transactions will enable bringing multicore programming to the masses. Average programmers can write concurrent code and, with little effort, use transactions to protect shared data against incorrectness. Yet, the simplicity of the concept is also its main source of rigidity. It prevents expert programmers from exploiting their skills and from enabling as much concurrency as they could, thereby limiting performance scalability. We showed that this limitation is inherent to the concept and is not simply a matter of usage.

We suggest a way out by truly democratizing the transaction concept and promoting the co-existence of different transactional semantics in the same application. Although a novice programmer will still be able to exploit the simplicity of the transaction abstraction in its original, strong and hence simple, form, expert programmers would exploit, whenever possible, more expressive semantics of relaxed transaction models to gain in concurrency. As this polymorphism helps experts take full advantage of transactions, we

expect to see new efficient libraries that will motivate other programmers to adopt this abstraction. This polymorphism also raises new challenges to guarantee that the various semantics can be used effectively in the same system.

References

[1] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The eden system: A technical review. *IEEE Trans. on Software Engineering*, SE-11(1):43–59, 1985.

[2] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, 2008.

[3] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *PPoPP*, pages 161–170, 2012.

[4] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, pages 157–168, 2009.

[5] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, volume 4167 of *LNCS*, 2006.

[6] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[7] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19:624–633, 1976.

[8] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *DISC*, volume 5805 of *LNCS*, pages 93–107, 2009.

[9] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. TM2C, a software transactional memory for many-cores. In *EuroSys*, pages 351–364, 2012.

[10] Vincent Gramoli, Derin Harmanci, and Pascal Felber. On the input acceptance of transactional memory. *Parallel Processing Letters*, 20(1):31–50, 2010.

[11] Michael Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *PODC*, pages 260–269, 2002.

[12] Rachid Guerraoui, Riccardo Capobianchi, Agnes Lanusse, and Pierre Roux. Nesting actions through asynchronous message passing: the ACS protocol. In *ECOOP*, volume 615 of *LNCS*, pages 170–184, 1992.

[13] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Morgan&Claypool, 2010.

[14] Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *LNCS*, pages 300–314, 2001.

[15] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[16] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[17] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[18] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, 1993.

[19] Tom Knight. An architecture for mostly functional languages. In *LFP*, pages 105–112, 1986.

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

[21] Barbara Liskov. The argus language and system. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, volume 190 of *LNCS*, pages 343–430, 1985.

[22] Barbara Liskov and Robert Scheifler. Guardians and actions: linguistic support for robust, distributed programs. In *POPL*, pages 7–19, 1982.

[23] Nancy A. Lynch. Multilevel atomicity a new correctness criterion for database concurrency control. *ACM Trans. Database Syst.*, 8, 1983.

[24] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *WMPI*, 2006.

[25] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, 1979.

[26] Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. Committing conflicting transactions in an STM. In *PPoPP*, 2009.

[27] Andreas Reuter. Concurrency on high-traffic data elements. In *PODS*, pages 83–92, 1982.

[28] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC*, pages 240–248, 2005.

[29] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[30] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI*, pages 78–88, 2007.