Democratizing Transactional Programming

Vincent Gramoli and Rachid Guerraoui

EPFL
Switzerland

Abstract. The transaction abstraction is arguably one of the most appealing middleware paradigms. It lies typically between the programmer of a concurrent or distributed application on the one hand, and the operating system with the underlying network on the other hand. It encapsulates the complex internals of failure recovery and concurrency control, significantly simplifying thereby the life of a non-expert programmer.

Yet, some programmers are indeed experts and, for those, the transaction abstraction turns out to be inherently restrictive in its classic form. We argue for a genuine democratization of the paradigm, with different transactional semantics to be used by different programmers and composed within the same application.

1 A Brief History of Transaction

The transaction abstraction is in essence a middleware paradigm: it allows multiple processes running on one or more processors (machines) to interact. The transaction abstraction lies typically between the programmer of concurrent and distributed applications and the operating system. It encapsulates complex concurrency control and failure recovery mechanisms behind a simple user interface.

The transaction abstraction is very old. It dates back to the 70's when it was proposed as a means to ensure the *consistency* of shared data [1], determined with respect to a sequential behavior. To formalize this notion of consistency, the *serializability* definition recast the consistency of an execution of transactions in terms of its equivalence to a sequential execution of transactions [2]: concurrent accesses have to behave as if they were executing sequentially—in other words, they must be *atomic*. Since that definition, researchers have derived other variants, like opacity [3], applicable to different transactional contexts.

Formerly used in databases, the transaction abstraction was adapted for the first time as a language construct in the form of *guards* and *actions* [4] in particular to address issues like *robustness* to hardware failures. The programmability of transactions has subsequently been studied in distributed systems in various forms, e.g., Argus [5], Eden [6] and ACS [7]. During that period, the first hardware support for such a transactional construct was invented to introduce parallelism in functional languages by providing synchronization on multiple memory words [8].

Later, transactional memory was proposed for concurrent programming especially to remedy the existing difficulties of programming with locks, e.g., priority

F. Kon and A.-M. Kermarrec (Eds.): Middleware 2011, LNCS 7049, pp. 1–19, 2011.

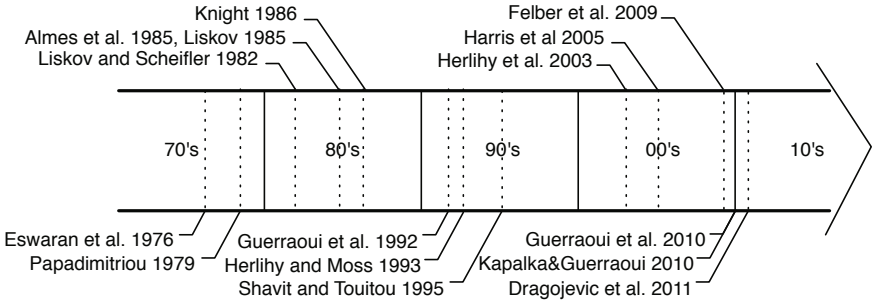

Fig. 1. A brief history of transactions

inversion, lock-convoying and deadlocks [9]. Since the advent of multicore architectures, the very notion of transaction memory has became an active topic of research[1]. Hardware implementations of such transactional systems [9] were generally limited by specific constraints and the programmer could only abstract away from these limitations using unbounded hardware transactions, a complex solution that most industrials are no longer exploring. Instead, a more hybrid tendency was adopted by implementing a best-effort hardware component that needs to be complemented by *software transactions* [10,11,12,13].

Software transactions were originally designed as a portable solution to execute a set of shared memory accesses fixed prior to execution [14]. Later software transactions were applied to a dynamic variant of this model, in which the control flow of the transaction was not predetermined [15]. Besides improvements stemming from the usage of timely information [16,17], new software transactions were derived to access more complex objects [18,19]. Nowadays, software transactions are even used in concurrent programs for the sake of coordinated failure recovery [20]. Despite these promising results, early investigations on the performance of software transactions have suggested their confinement to a research toy by questioning their ability to leverage multicore architectures [21].

Software transactions have finally won their spurs by outperforming sequential applications with only few cores. The result of [22] shows that an STM with manually instrumented benchmarks and explicit privatization outperforms 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. Even though the software overheads, induced by compiler instrumentation and transparent privatization, do not prevent transactions from outperforming sequential code, performance remains one of the major issue of transactions. Basically, an expert will never be able to extract as much concurrency from classic transactions than from synchronization primitives under the hood.

[1] A bibliography of the topic can be found at `http://www.cs.wisc.edu/trans-memory/biblio/list.html`.

Not surprisingly, researchers have kept exploring possible relaxations of the classic model since the early stage of transactions. Nesting models exploited *commutativity* of high level operations to favor concurrency [23,24]. In short, commutativity applies to operations whose order does not impact the transaction outcome. Such techniques would typically require the programmer to identify operations that can commute statically or to introduce code breakpoints. Others were dedicated to improve performance of typical contention hotspots: relaxed transactions were proposed for aggregate fields of database systems [25,26] on the one hand, and for search structures of multicore programs [15,27] on the other hand. A large majority of these relaxations rely however on complex code refactoring to improve performance and only a few, like [27], preserve both the sequential code and composition, most of them remaining non-exploitable by novice programmers.

To summarize, the transaction is an old appealing abstraction that has been the main topic of many practical and theoretical achievements in research, however, it has never been widely adopted in practice. Despite their genericness, transactions failed to be unified across distinct usages. Instead, transactions have always been tuned differently for different purposes, enforcing their incompatibility. An example is the recent adoption of transactions by IBM in their BlueGene/Q processor. This choice has been made in order to obtain the fastest supercomputer ever, yet only a very limited set of applications, which are supercomputing applications, will benefit from these highly tuned transactions. The incompatibility between distinct transactions breaks the appeal of the abstraction itself and prevents it from being used by the masses.

2 The Inherent Appeal of Transactions

The transaction paradigm is appealing for its simplicity as it preserves sequential code by hiding synchronization internals and its ability to promote 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 the sequential code, 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 or similar tx-begin/tx-commit delimiters, as depicted in Algorithm 1—the existing data organization appears unchanged (Algorithm 2 (left)).

Programming with transactions shifts the inherent complexity of concurrent programming to the implementation of the transaction abstraction which must be done once 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 transaction 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;
```

On the one hand, traditional synchronization techniques require generally 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, to prevent inconsistent memory deallocations, yet the memory management would not even be guaranteed to be simple, and may require additional re-engineering [28]. Using lock-based techniques, the programmer must explicitly declare and initialize all locks before protecting memory accesses as depicted in Algorithm 2; the programmer may even need to use a logical deletion technique as well as an additional validation phase to guarantee consistency [29].

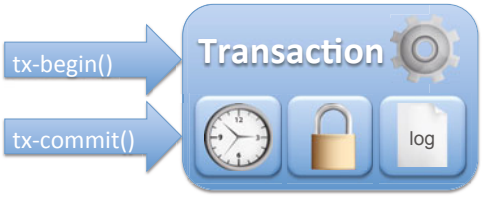

Fig. 2. The transaction abstraction hides complex synchronization mechanisms behind a simple interface

On the other hand, the transaction abstraction hides both synchronization internals and metadata management. If locks are internally used, they are declared and initialized transparently by the transaction system. Moreover, as the transaction system wraps memory accesses, a simple reference counting can keep track of the status of transactions accessing a particular location, before freeing the memory.

Despite its apparent simplicity and as depicted in Figure 2, the transaction system internally hides complex synchronization mechanisms. For example a single transactional system can exploit (i) time, to associate timestamps to values and guarantee that all values read belong to the same snapshot the transaction is acting upon; (ii) locks, for concurrent transactions to detect conflicts when accessing common data; and (iii) logs, to record operations that will be re-executed at commit time, or rolled back at abort time.

2.2 Enabling Composition

Transactions are also appealing for they allow concurrent programs to be reused in a modular fashion. More specifically, transactions allow Bob to compose existing transactional operations developed by Alice into a composite one that preserves the safety and liveness of its components [30] as depicted in Figure 3.

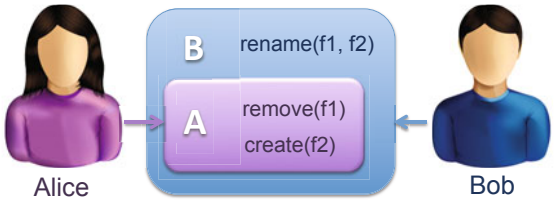

Fig. 3. Bob composes Alice's component operations remove and create into a new operation rename that preserves the safety and liveness of its components

By contrast, alternative synchronization techniques do not facilitate composition. For example, consider a simple directory abstraction mapping a name to a file. With 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 d_2 while another rename a file from d_2 to d_1, directories must be protected with care to avoid deadlocks. In the lock-based file system hierarchy of the Google File System [31], each directory at the same path depth has to be locked in a pre-determined ordering to prevent deadlock in such a scenario. In other words, Bob must first understand the locking strategy of Alice to ensure the liveness of his own operations. For the same 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 [32].

By contrast, a transaction system detects a conflict between the two renaming transactions and let only one of the two 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 and various strategies have been proposed [33].

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 makes 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:   unlock(&next → lock);
15:   result = (next → val == val);
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 during the point interval $(p_1; p_2)$ of α, in which it accesses x, is guaranteed that any of its other accesses during this interval will appear atomic with its access to x. For example, the process guarantees $atomicity(r(x), r(y))$ and $atomicity(r(y), r(z))$ but not $atomicity(r(x), r(z))$ in the following lock-based program:

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

Conversely, a process 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. Note that there is no way to ensure the two former atomicity relations with classic transactions without also ensuring the latter.

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

This lack of expressiveness when using transactions is directly implied by their syntax, which consists of an open/close block delimiting a compound statement [34]. In this sense, this expressiveness limitation is not related to the way transactions are used but to the transaction abstraction itself. This open/close block does neither accept a memory location nor a semantic hint as a parameter. Hence, it blindly guarantees that all its accesses appear as if there was an indivisible point in the execution where they *all* take effect.

Fig. 4. Among the correct linked list schedules, 20% of them are precluded when using transactions

3.2 Impact on Concurrency

The level of expressiveness is crucial especially when it restricts the set of acceptable schedules, and hence achievable concurrency, in a real workload. The low expressiveness of transactions translates actually into a concurrency loss on very common workloads. 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 $next \rightarrow val$; this is a false-conflict leading to unnecessary aborts. Such unnecessary aborts limit concurrency because they preclude schedules that would be correct, be all the transactions committed [35]. Conversely, the hand-over-hand locking program of Algorithm 3 allows such concurrent update (Line 7) when checking the value (Line 8), starting from the second iteration of the while-loop. Lock-free linked list algorithms [36,28] would not suffer such false-conflict either.

To quantify the impact of the limited expressiveness of transactions on the number of accepted schedules, consider that program P_t above executes concurrently with program $P_1 = \mathsf{transaction}\{w(x)\}$ and $P_2 = \mathsf{transaction}\{w(z)\}$. As there are four ways of placing the single access of one of these two programs 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 allowed in a linked list implementation; however, transactions that ensure opacity [3] (as it is the case for most classic ones) preclude four of these schedules: those in which P_t accesses x before P_1 ($P_t \prec P_1$), P_1 terminates before P_2 starts ($P_1 \prec P_2$) and in which P_2 accesses z before P_t ($P_2 \prec P_t$). The proportion of schedules precluded by transactions among all possible ones is depicted in Figure 4.

3.3 Impact on Performance

The metadata management overhead of transactions when starting, accessing shared memory and committing, is expected to be compensated by exploiting concurrency [22]. In scenarios like the previous linked list program where transactions fail to fully exploit all available concurrency, their performance cannot compete with lock-based or lock-free algorithms. 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. The conjunction of overhead and limited concurrency of transactions prevents them from outperforming well-engineered lock-based and lock-free alternatives.

To illustrate the impact on performance, we compared the existing Java concurrency package to a classic transaction library written in Java, TL2 [16], on a 64-way Niagara 2 machine. 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 as recommended for circumventing this limitation [37]. We compared it against the linked list implementation building upon TL2. The throughput speedups over sequential of classic transaction and the existing collection are depicted in Figure 5. The existing collection performs 2.2× faster than classic transactions on 64 threads.

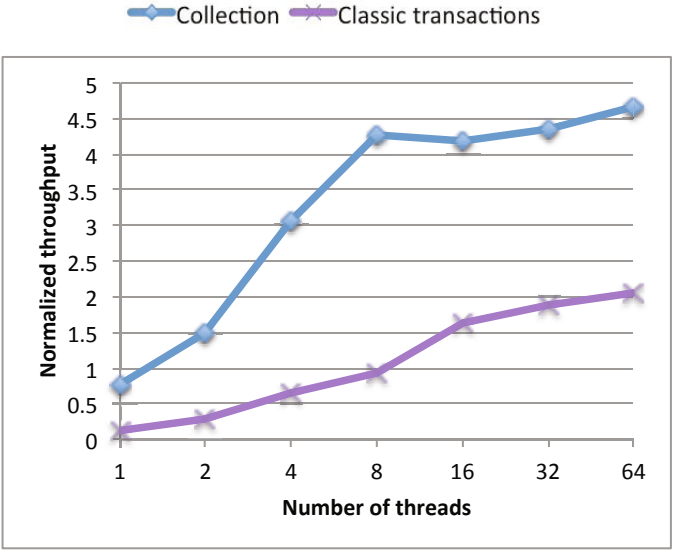

Fig. 5. Throughput (normalized over the sequential one) of classic transactions and the existing concurrent collection

4 Democratizing Transactions: The Challenge

Classic transactions share a single semantics for all types of applications. This simplifies the development of a transaction system by requiring the same guarantee for all its transactions, independently from their role in the concurrent applications. In some scenarios this semantics is, however, overly conservative and limits concurrency and performance (cf. Section 3). Without additional control, skilled programmers are frustrated by not being able to obtain highly efficient concurrent programs as depicted in Figure 6. In order to rather exploit adequately the concurrency allowed by the semantics of an application it is necessary to trade part of the simplicity of transactional memory for additional control.

We argue that for the transactional abstraction to really become a widely used programming paradigm it should be *democratized*. Not only is it important for transactions to be an off-the-shelf solution for novices, but also to give additional control to experts in concurrent programming.

Therefore, we believe that various transaction semantics should be able to run concurrently: a default semantics capturing the classic single-global-lock atomicity (i.e., opacity [3]), and more complex semantics capturing more subtle behaviors (e.g., elastic-opacity [27]). 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

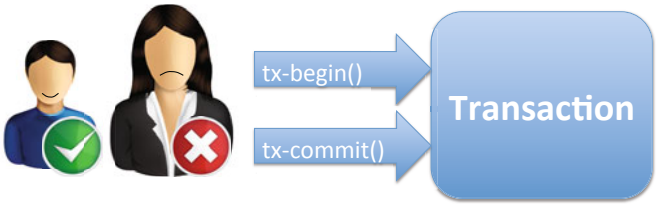

Fig. 6. The novice programmer benefits from the simplicity of transactions whereas the expert programmer is frustrated by its lack of flexibility

but simply delimiting its regions like for classic transactions. 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 Expressiveness and Simplicity

Several relaxed transaction models have been proposed as an alternative to the classic transaction model. Such relaxed models can generally achieve a greater level of flexibility than the classic model by avoiding unnecessary aborts thus tolerating additional schedules.

An explicit early `release` can be used to ask statically the transaction to ignore false conflicts [15] and hence avoid unnecessary aborts. For example, to achieve the same expressiveness as the lock-based linked list of Algorithm 3 one could use early release to force the transaction to unprotect some of its read locations while executing. More precisely, a release call of location x could indicate from which point of the transaction all conflicts involving its read of x can start being ignored. Despite increasing expressiveness, the use of early release may hamper transaction composition. Alice may implement an atomic linked list $\mathsf{add}(x)$ using early release, yet Bob cannot reuse Alice's code to develop an atomic $\mathsf{addIfAbsent}(x, y)$ that inserts x only if y is absent. Typically, the resulting operation would not be atomic: two instances $\mathsf{addIfAbsent}(x, y)$ and $\mathsf{addIfAbsent}(y, x)$ may insert concurrently x and y, leading to an inconsistency.

A first relaxing methodology consists in open nesting [24] that is considered effective to increase concurrency. The key underlying idea is that nested transactions typically commit before the outer transaction ends but pass a high level abstraction of their changes to the outer transaction. As a result, the changes committed by a nested transaction become immediately visible from concurrent transactions and abstract locks indicate which pairs of nested operations conflict. Open nesting may lead to deadlocks if the accesses to shared locations are not ordered with care [38]. Specifically, this problem is similar to the one raised with explicit locks as open nesting let the programmers acquire abstract locks even upon abort.

A second relaxing methodology is transactional boosting [39]. It benefits from commutativity by considering transactional operations at a high level of abstraction. If two high level operations commute, they can be executed in any order despite the conflicts between their low-level operations. To this end, high level operations are considered as a whole and the programmer must identify operations that commute and define inverse operations. Considering higher level operations diminishes the amount of information that needs to be logged and possibly rolled back. Each operation acquires an abstract lock similar to open nesting so that two operations conflict if and only if they do not commute. Upon abort, a transaction rolls back its changes by executing the appropriate inverse operations that compensate its logged operations. Typically such models require the programmer to identify commutative operations and to write an appropriate compensating block of action for each non-commutative operation, such a compensate block is typically as long as the corresponding transaction block itself.

Inherently more complex to use than the classic transaction model, these initial relaxed transaction models lost the appealing aspects of transactions: either by requiring significant code refactoring or by breaking composition. Therefore, it is crucial to guarantee sequential code preservation and transaction composition when deriving new relaxed models targeting high expressiveness.

4.2 Sequentiality and Composition

A relaxed transaction model preserving sequential code and guaranteeing composition was proposed as the elastic transaction model [27]. This model provides 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 are fully compatible with classic transactions thus inheriting the ability to compose of the classic model. Bob directly encapsulates Alice's elastic transactions, into another transaction, choosing between labeling it as elastic or classic, hence guaranteeing atomicity and deadlock-freedom of its own operation. Typically, Bob can easily compose Alice's elastic $\mathsf{add}(x)$ into a classic $\mathsf{addIfAbsent}(x, y)$.

In contrast with 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. For example, 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 clearly neither serializable [2] nor opaque [3] 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

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:  }
```

contradicting conflicts between transactions i and j, and the transactions could not both 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, a cut can split dynamically an elastic transaction depending on the interleaving of its accesses with other transaction accesses, yet it would be incorrect to replace statically the elastic transaction by multiple classic transactions, as the interleaving is not predictable. Second, identifying commutativity of accesses cannot enable the concurrency of elastic transactions because, depending on the current interleaving of accesses, two accesses that are (statically) non-commutative can be considered *dynamically-commuting* in

elastic transactions. 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.

4.3 Impact on Performance

To illustrate the benefit of combining relaxed and classic transactions, the collection benchmark was run in the exact same settings as the one used to obtain 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 classic transaction to ensure atomicity of all four operations. As an example, the Java pseudocode of the add operation based on an elastic transaction is depicted in Algorithm 4.

The performance of combining elastic transactions with classic transactions, is compared in Figure 7 against the performance obtained with the existing concurrent collection package and with the classic transactions alone. The best performance we obtained by combining elastic and classic transactions is higher than classic transactions alone by 3.5× and than the existing collection package by 1.6×. Unfortunately, the performance does not scale up to the maximum number of threads 64. We conjecture that the slow-down between 32 and 64

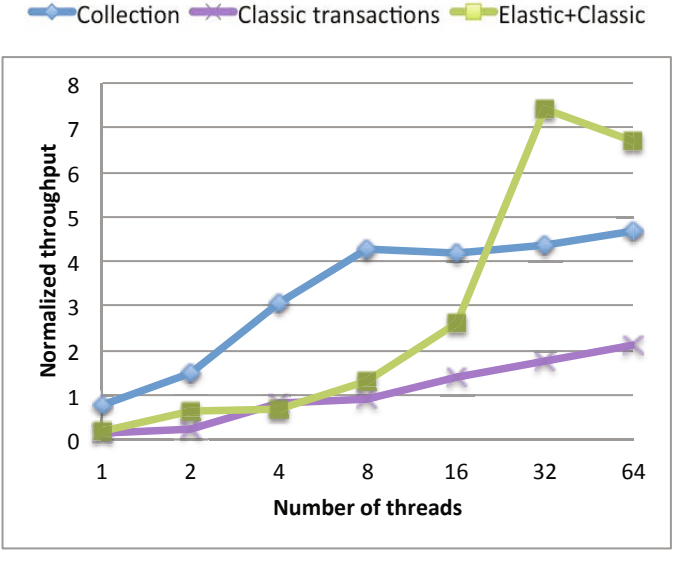

Fig. 7. Throughput (normalized over the sequential one) of elastic and classic transactions, the classic transactions alone and the existing concurrent collection

threads by repeatedly aborting the size operations, in the same vein as balance operations of the bank benchmark [40] or toxic transactions [41]. More precisely, the size executes within a classic transaction that has limited concurrency and which may thus produce an abort each time a concurrent update (add or remove) is modifying concurrently any location of the data structure.

5 Mixing Several Semantics

Mixing semantics means providing multiple transactions of different semantics to let the programmer choose the right semantics for each delimited region of the program. As these transactions can potentially access concurrently the same locations, it is crucial that one transaction does not alter the semantics of the others.

More precisely, we consider the semantic of classic transactions, opacity, to be the strongest one. Hence the novice can use exclusively the default semantics for all transactions, making sure that the resulting program is correct. Nevertheless, the expert can use a relaxed semantics that preserves sequential code (like elastic one) for some transactions and the classic one for others, to obtain higher expressiveness and better performance. The challenge is to preserve the semantics of all individual transactions. In the case of mixing elastic with classic transactions the resulting correct histories should thus be equivalent to a sequential legal history of elastic sub-transactions and classic transactions, as long as elastic sub-transactions result from consistent cuts (as required by the elastic transaction semantics). Consequently, mixing additional semantics may become rapidly challenging.

The key idea of mixing several semantics relies on providing various kinds of transactions among which the programmer can choose the adequate one that better matches its needs. More specifically, the programmer can start a transaction that executes the default intuitive semantics unless the tx-begin call is given some parameter, that serves as a hint to indicate the transaction semantics. With mixed semantics, not only does the transaction remain an off-the-shelf paradigm for novices, but it also gives control to the experts to boost the performance of some transactions (Figure 8).

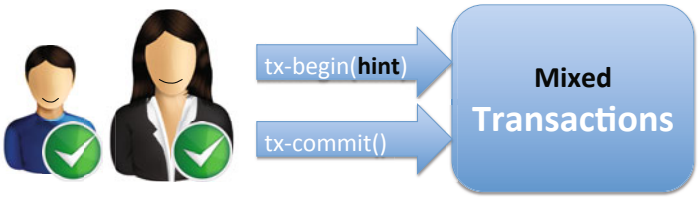

Fig. 8. Novice and expert programmers should both benefit from the simplicity and flexibility of a mixed transactional model

5.1 Combining Classic, Snapshot and Elastic Transactions

To go a step further in exploiting mixed transactions, here is an example of an additional transaction semantics, called *snapshot*, in addition to the two pre-existing ones, elastic and classic. This snapshot transaction semantics provides a way for the programmer to implement an atomic snapshot operation that can run concurrently with updating transactions (elastic or classic) modifying the data structure in a complex way (even at distinct locations). This is typically an appealing semantics to design an operation whose result depends on multiple elements of the data structure, like a Java Iterator. As an example, a size operation preserving sequential code and that is depicted in Algorithm 5 uses a snapshot transaction.

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:   }
```

The key idea is for the snapshot to detect the locations that have been concurrently modified and to exploit multiversion concurrency control to bypass these conflicts. Using a global counter and version numbers associated with location values, the snapshot can detect at read time whether a location has been concurrently updated by comparing its current version to the value of the global counter at the time the snapshot started. If such a concurrent modification is detected, the snapshot has to select an old value (with a lower version number) of the overwritten location that is consistent with the start time of the snapshot transaction (i.e., higher than the value of the global counter at the time it started).

More precisely, multiple versions must be maintained at each location by every update transaction, be they elastic or classic—in our case two versions were maintained, this was actually sufficient to speed up the performance significantly. All update transactions create a backup value-version pair before overwriting them. The snapshot transaction has simply to detect whether the location it aims at accessing has a higher version than its upper bound *ub* to try getting an older version that could let it commit. Naturally, the snapshot transaction may have to abort if the older version is still too recent as no transactions keep track of more than two versions here.

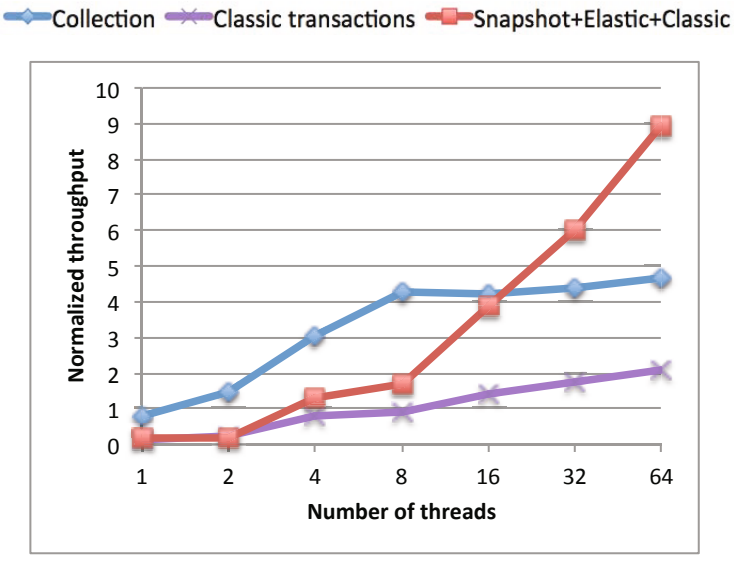

Fig. 9. Throughput (normalized over the sequential one) of the mixed transactions, the classic transaction and the collection package

5.2 Impact on Performance

The performance obtained when combining snapshot semantics in addition to the elastic one on the previous collection benchmark is depicted in Figure 9. The mixed transaction model performs 4.3× faster than the classic transaction model, TL2, and improves the concurrent collection package by 1.9× on 64 threads. Thanks to the snapshot semantics that remedy the scalability issue of the classic transactions, size operation in snapshot transactions commit more frequently than in default transactions. The reason is that a snapshot size returns potentially stale values that have been concurrently overwritten, while classic size would abort. Even though the overhead of the polymorphic transactions makes it slower than the concurrent collection package at low levels of parallelism, the performance scales well with the level of concurrency up to the maximum number of hardware threads we had at our disposal, and compensates the overhead effect at high level of parallelism.

6 Concluding Remarks

The transaction abstraction is in essence a middleware paradigm that allows multiple processes running on one or more processors (machines) to interact. The transaction abstraction was proposed long ago and has constituted an active area of research over the years.

Yet, transactions have not been widely adopted in practical concurrent and distributed programming and this is due, we believe, to their inherent cost and limited concurrency. In short, expert programmers need an alternative to bypass the simplicity of the concept and express their skills, potentially to obtain better performance.

We argue for democratizing the concept by enabling the co-existence of different semantics of it in the same application. Although a novice programmer will still be able to exploit the simplicity of the transaction abstraction in its default semantics, expert programmers would exploit, when possible, more expressive semantics of relaxed transaction models to gain in concurrency. This raises new challenges to guarantee that various semantics can cohabit smoothly in the same system but promises to further leverage the transaction abstraction.

References

1. Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicate locks in a database system. Commun. ACM 19, 624–633 (1976)
2. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26, 631–653 (1979)
3. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Morgan&Claypool (2010)
4. Liskov, B., Scheifler, R.: Guardians and actions: linguistic support for robust, distributed programs. In: POPL, pp. 7–19 (1982)
5. Liskov, B.: The Argus Language and System. In: Alford, M.W., Hommel, G., Schneider, F.B., Ansart, J.P., Lamport, L., Mullery, G.P., Zhou, T.H. (eds.) Distributed Systems. LNCS, vol. 190, pp. 343–430. Springer, Heidelberg (1985)
6. Almes, G.T., Black, A.P., Lazowska, E.D., Noe, J.D.: The eden system: A technical review. IEEE Trans. on Software Engineering SE-11(1), 43–59 (1985)
7. Guerraoui, R., Capobianchi, R., Lanusse, A., Roux, P.: Nesting Actions through Asynchronous Message Passing: the ACS Protocol. In: Madsen, O.L. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 170–184. Springer, Heidelberg (1992)
8. Knight, T.: An architecture for mostly functional languages. In: LFP, pp. 105–112 (1986)
9. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News 21, 289–300 (1993)
10. Dice, D., Lev, Y., Moir, M., Nussbaum, D.: Early experience with a commercial hardware transactional memory implementation. SIGPLAN Not. 44, 157–168 (2009)
11. Diestelhorst, S., Hohmuth, M., Pohlack, M.: Sane semantics of best-effort hardware transactional memory. In: WTTM (2010)
12. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M.L., Spear, M.F.: Hybrid NOrec: a case study in the effectiveness of best effort hardware transactional memory. In: ASPLOS, pp. 39–52 (2011)
13. Felber, P., Riviere, E., Moreira, W., Harmanci, D., Marlier, P., Diestelhorst, S., Hohmuth, M., Pohlack, M., Cristal, A., Hur, I., Unsal, O., Stenstrom, P., Dragojevic, A., Guerraoui, R., Kapalka, M., Gramoli, V., Drepper, U., Tomic, S., Afek, Y., Korland, G., Shavit, N., Fetzer, C., Nowack, M., Riegel, T.: The velox transactional memory stack. IEEE Micro 30(5), 76–87 (2010)

14. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
15. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
16. Dice, D., Shalev, O., Shavit, N.N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
17. Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
18. Dragojevic, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: PLDI, pp. 155–165 (2011)
19. Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. In: EuroSys, pp. 315–324 (2007)
20. Harmanci, D., Gramoli, V., Felber, P.: Atomic Boxes: Coordinated Exception Handling with Transactional Memory. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 634–657. Springer, Heidelberg (2011)
21. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., Chatterjee, S.: Software transactional memory: Why is it only a research toy? Queue 6, 46–58 (2008)
22. Dragojevic, A., Felber, P., Gramoli, V., Guerraoui, R.: Why STM can be more than a research toy. Commun. ACM 54(4), 70–77 (2011)
23. Lynch, N.A.: Multilevel atomicity a new correctness criterion for database concurrency control. ACM Trans. Database Syst. 8 (1983)
24. Moss, J.E.B.: Open nested transactions: Semantics and support. In: WMPI (2006)
25. Reuter, A.: Concurrency on high-traffic data elements. In: PODS, pp. 83–92 (1982)
26. O'Neil, P.E.: The escrow transactional method. ACM Trans. Database Syst. 11, 405–430 (1986)
27. Felber, P., Gramoli, V., Guerraoui, R.: Elastic Transactions. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 93–107. Springer, Heidelberg (2009)
28. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA, pp. 73–82 (2002)
29. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A Lazy Concurrent List-Based Set Algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
30. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP, pp. 48–60 (2005)
31. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: SOSP (2003)
32. Greenwald, M.: Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In: PODC, pp. 260–269 (2002)
33. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC, pp. 240–248 (2005)
34. Transactional Memory Specification Drafting Group: Draft specification of transactional language constructs for C++ (2009), `http://software.intel.com/file/21569`
35. Gramoli, V., Harmanci, D., Felber, P.: On the input acceptance of transactional memory. Parallel Processing Letters 20(1), 31–50 (2010)
36. Harris, T.: A Pragmatic Implementation of Non-Blocking Linked-Lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)

37. Peierls, T., Goetz, B., Bloch, J., Bowbeer, J., Lea, D., Holmes, D.: Java Concurrency in Practice. Addison-Wesley (2005)
38. Ni, Y., Menon, V., Abd-Tabatabai, A.-R., Hosking, A.L., Hudson, R.L., Moss, J.E.B., Saha, B., Shpeisman, T.: Open nesting in software transactional memory. In: PPoPP (2007)
39. Herlihy, M., Koskinen, E.: Transactional boosting: A methodology for highly-concurrent transactional objects. In: PPoPP (2008)
40. Harmanci, D., Gramoli, V., Felber, P., Fetzer, C.: Extensible transactional memory testbed. J. Parallel and Distrib. Comp. 70(10), 1053–1067 (2010)
41. Liu, Y., Spear, M.: Toxic transactions. In: Transact (2011)