Control transactions without compromising their simplicity for the sake of expressiveness, application concurrency, or performance.

BY VINCENT GRAMOLI AND RACHID GUERRAOUI

Democratizing Transactional Programming

THE TRANSACTION ABSTRACTION encapsulates the mechanisms used to synchronize accesses to data shared by concurrent processes, dating to the 1970s when proposed in the context of databases to ensure consistency of shared data.[7] This consistency was determined with respect to a sequential behavior through the concept of serializability;[25] concurrent accesses must behave as if executing sequentially or be atomic. More recently, researchers have derived other variants (such as opacity[13] and isolation[30]) applicable to different transactional contexts.

The transaction abstraction was first considered as a programming language construct in the form of guards and actions by Liskov and Scheifler more than 30 years ago,[22] then adapted to various programming models, including Eden,[1] ACS,[12] and Argus.[21] The first hardware support for a transactional construct was proposed in 1986 by Tom Knight,[19] basically introducing 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 trickiness and subtleties of using locks (such as priority inversion, lock-convoying, and deadlocks)[18] (see Figure 1).

Since the advent of multicore architectures approximately 10 years ago, the very notion of transactional memory has become an active topic of research (http://www.cs.wisc.edu/trans-memory/biblio/list.html). Hardware implementations of transactional systems[18] turned out to be limited by specific constraints programmers could "abstract away" only through unbounded hardware transactions. However, purely hardware implementations are complex solutions most industrial developers no longer explore. Rather, a hybrid approach was adopted through a best-effort hardware component that must be complemented by software transactions.[4]

Software transactions were originally designed in the mid-1990s as a reusable and composable solution to execute a set of shared memory accesses fixed prior to execution.[29] More recently, they were applied to handle when the control flow is not predetermined.[17] Early investigations of the performance of software transactions questioned their ability to leverage multicore architectures.[2] However, these results were revisited by Dragojevic et al.,[6] showing a highly optimized software-transactional

» **key insights**

■ **Though powerful, the transaction paradigm can sometimes restrict application concurrency and performance.**

■ **Democratizing transactional programming aims to make the paradigm useful for novice programmers who want concurrency to be transparent and for expert programmers who are able to address its underlying challenges while compromising neither composition nor correctness.**

■ **The key challenge is how to offer multiple synchronization semantics of sequences of shared data accesses without compromising safety and liveness.**
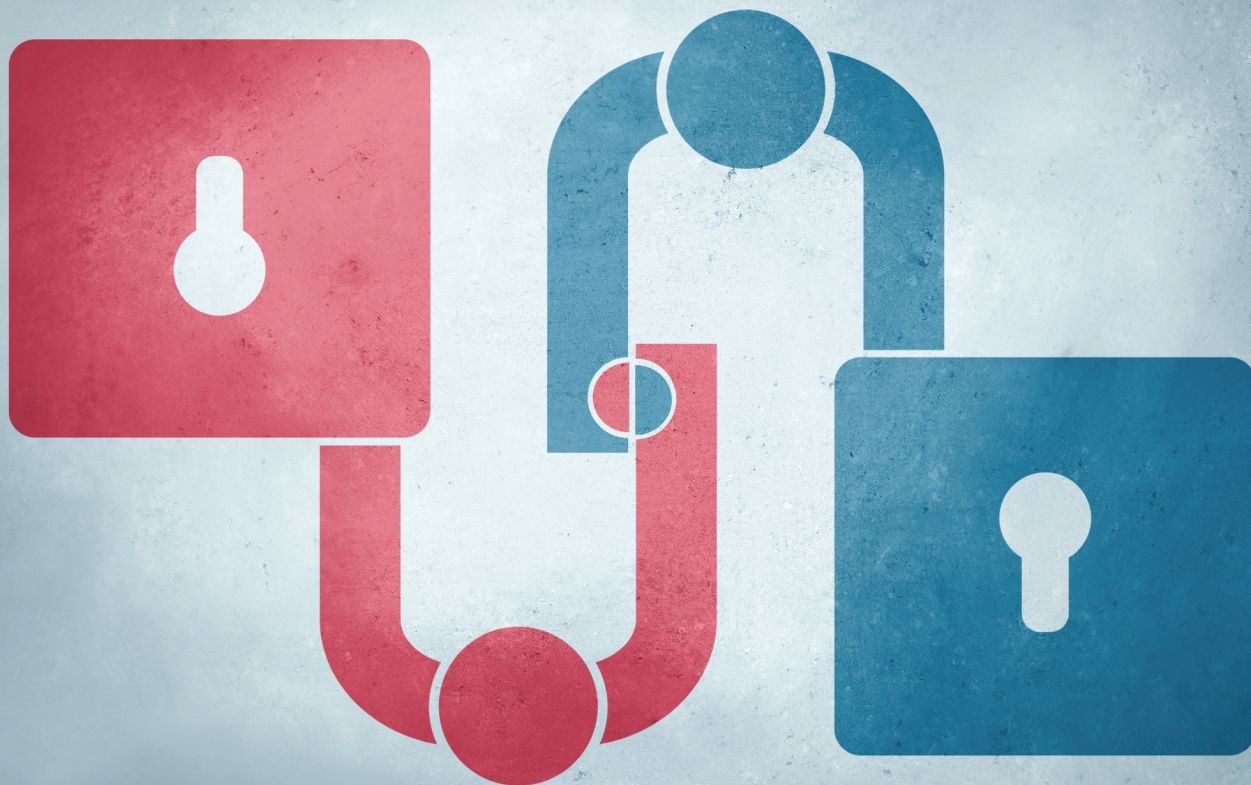
memory (STM) with manually instrumented benchmarks and explicit privatization whose throughput still outperforms sequential code by up to 29 times on SPARC processors with 64 concurrent threads and by up to nine times on x86 with 16 concurrent threads. However, performance remains the main obstacle preventing wide adoption of the transaction abstraction for general-purpose concurrent programming.

In classic form, transactions prevent expert programmers from extracting the same level of concurrency possible through more primitive synchronization techniques. This observation is folklore knowledge, yet we show for the first time, in this article through a simple example, that this limitation is inherent in the transaction concept in its classic form irrespective of how it is used. It can be viewed as the price of bringing concurrency to the masses and making it possible for average programmers to write parallel programs that use shared data. Nevertheless, some program-

mers are indeed concurrency experts and might find it frustrating if they are not able to use their skills to enhance concurrency and performance.

Not surprisingly, researchers have been exploring relaxation of the classic transaction model[23,24,27] that enables more concurrency. Doing so while keeping the simplicity of the original model has proved to be a challenge; the idea is to preserve the original sequential code while composing applications devised by different programmers, possibly with different skills.

Here, we endorse mixing different transaction semantics within the same application, with strong semantics to be used by novice programmers and weaker semantics by concurrency experts. The challenge is to ensure the polymorphic system mixing different semantics still enables code reuse, composing it in a smooth manner. Before describing how such mixing can be addressed, we take a closer look at the meaning of reuse and composition.

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 results;
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;
```

Preserving sequentiality. Transactions preserve the sequential code in that their use does not alter it beyond 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 remains unchanged.

Programming with transactions shifts the inherent complexity of concurrent programming to implementation of the transaction semantics that must be done once and for all. Due to transactions, writing a concurrent application follows a divide-and-conquer strategy where experts write a live, safe transactional system with an unsophisticated interface, and the novice writes a transaction-based application or delimits 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 programmers

first re-factorize the sequential code. Using lock-free techniques, they typically use subtle mechanisms (such as logical deletion[14]) to prevent inconsistent memory de-allocations. Using lock-based techniques, they usually explicitly declare and initialize all locks before using them to protect memory accesses, as in Algorithm 2 line 8.

The transaction abstraction hides both synchronization internals and metadata management. If locks or timestamps are used internally, 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. The wrappers can then exploit the metadata, locks, and timestamps to detect conflicting accesses and potentially abort a transaction.

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] (see Figure 2).

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

transactions, a programmer is able to compose the removal of a name and 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 and another user renames a file from d_2 to d_1, directories must be protected to avoid deadlocks; that is, 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` includes 50 lines of comments explaining the locking strategy. Lock-free techniques are even more complex, requiring a multi-word compare-and-swap operation 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 them resume and possibly commit; the other is restarted or resumed later. Deciding on a conflict-resolution strategy is the task of a dedicated service, or "contention manager," for which various strategies and implementations have been proposed.[28]

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

Lacking expressiveness. To reinforce our point that transactions are inherently limited in terms of expressiveness we define "atomicity" as a bi-

Figure 1. History of transactions.

Knight 1986
Almes et al. 1985, Liskov 1985
Liskov and Scheifler 1982
Felber et al. 2009
Harris et al. 2005
Herlihy et al. 2003

1970s   1980s   1990s   2000s   2010s

Eswaran et al. 1976
Papadimitriou 1979
Guerraoui et al. 1992
Herlihy and Moss 1993
Shavit and Touitou 1995
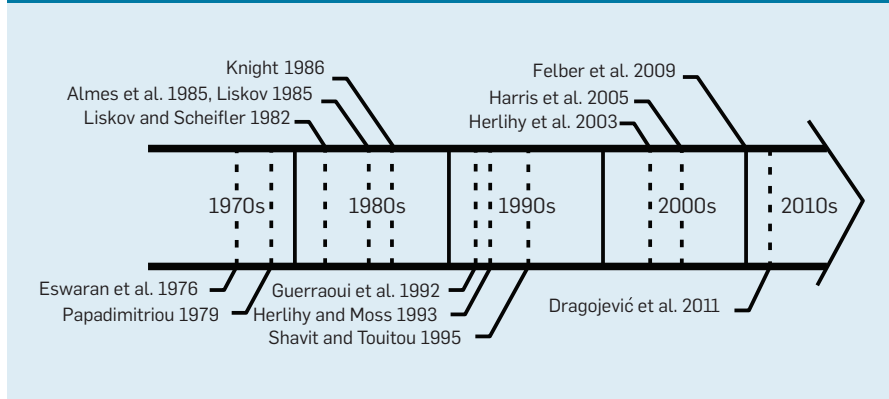Dragojević et al. 2011

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.
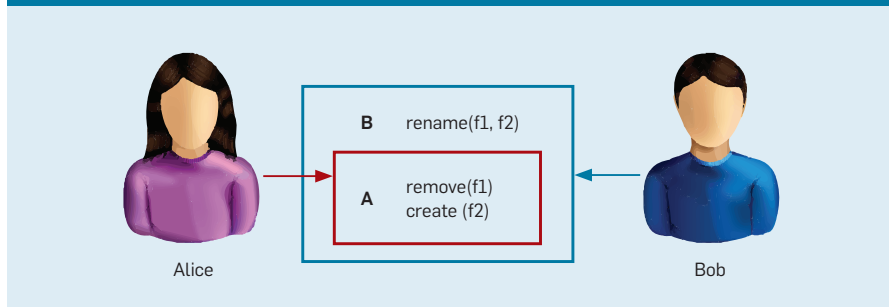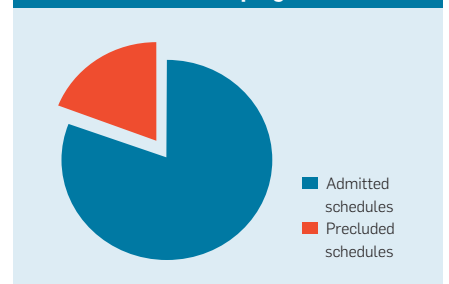
B   rename(f1, f2)
A   remove(f1)
    create (f2)

Alice   Bob

Figure 3. Transactions preclude 20% of the correct schedules of a simple concurrent linked list program.

■ Admitted schedules
■ Precluded schedules

nary 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 both occur at one common indivisible point of the execution. It is important to note this relation is not transitive; that is, $atomicity(\pi_1, \pi_2) \wedge atomicity(\pi_2, \pi_3) \not\Rightarrow atomicity(\pi_1, \pi_3)$.

Algorithm 3. An implementation of a linked list operation with locks

```
1:  lk-contains(val)_p:
2:    int results;
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;
```

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 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 $atomicity(r(x), r(z))$:

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

Conversely, a process P_t executing the following transaction block ensures $atomicity(r(x); r(y))$, $atomicity(r(y), r(z))$ but also $atomicity(r(x), r(z))$, the transitive closure of the atomicity relations guaranteed by P_ℓ. Using classic trans-

Performance remains the main obstacle preventing wide adoption of the transaction abstraction for general-purpose concurrent programming.

actions, there is no way to write a program with semantics similar to P_ℓ or ensure the two former atomicity relations without also ensuring the latter.

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

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 all take effect.

Effect on concurrency. Not surprisingly, the limited expressiveness of transactions translates into a concurrency loss; for example, consider the transactional linked list program 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 effect of the limited expressiveness of transactions on the number of accepted schedules, consider a concurrent program where the process P_t executes concurrently with processes $P_1 = \texttt{transaction}\{w(x)\}$ and $P_2 = \texttt{transaction}\{w(z)\}$. As there are four ways to place the single access of one of these two processes between accesses of P_t and five ways to place the remaining one in the resulting schedule, there are 20 possible schedules. Note that all are correct schedules of a sorted linked list implementation.

However, most transactional memory systems guarantee each of their executions 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 (strict-

ness). (This guarantee is often satisfied, as a large variety of transactional memory systems ensures opacity,[13] a consistency criterion even stronger than this strict serializability, as it additionally requires noncommitted transactions never observe an inconsistent state.) These transactional memory systems preclude four of these schedules (see Figure 3): those in which P_t accesses x before P_1 (P_t is serialized before P_1, or $P_t \prec P_1$), 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.

Worth noting is that a programmer could exploit weaker transactional memory systems to export these serializable histories.[10,26] Such systems would offer a transaction that might not be appropriate for all possible uses; for example, it might be possible that one transaction reads an inconsistent state before aborting. In fact, the concurrency limitation is due to transactional memory systems providing a unique but general-purpose transaction.

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

To depict the effect on performance, we compared the existing Java concurrency package to the classic transaction library TL2[5] on a 64-way Niagara 2 SPARC-based machine. Note this is the Java implementation of the TL2 algorithm that detects conflicts at the level of granularity of fields and is distributed within DeuceSTM,[20] a bytecode instrumentation framework offering a suite of TM libraries. We present the results obtained on a simple `Collection` benchmark of 2^{12} elements providing

> **To adequately exploit the concurrency allowed by the semantics of an application, programmers must be willing to trade simplicity for additional control.**

`contains`, `add`, `remove`, and `size` operations with an update ratio and a `size` ratio of 10%, respectively. As the existing lock-free data structures do not support atomic size we had to use the `copyOnWriteArraySet` workaround of this package, comparing it against the linked list implementation building on TL2.

Figure 4 uses the throughput (committing transactions per time unit) of the bare sequential implementation (without synchronization) as the baseline, illustrating the throughput speedup (over sequential) a programmer can achieve through 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 the existing collection performs 2.2x faster than classic transactions on 64 threads. The poor performance of classic transactions is due to their lack of concurrency, a problem addressed in the next section.

Democratizing Transactions

Traditionally, transactional systems ensure the same semantics for all their transactions, independent of their role in concurrent applications. However, as discussed, these semantics are overly conservative and, by limiting concurrency, could also limit performance. Without additional control, skilled programmers would be frustrated by not being able to obtain highly efficient concurrent programs. To adequately exploit the concurrency allowed by the semantics of an application, programmers must be willing to trade simplicity for additional control.

To be a widely used programming paradigm, the transactional abstraction must be democratized, or universally useful and available to all programmers. Not only should transactions be an off-the-shelf solution for novices, they should also permit additional control to experts in concurrent programming. Simple default semantics should be able to run concurrently with transactions of more complex semantics, capturing more subtle behaviors. The concurrency challenge is twofold: The

transaction abstraction must allow expert programmers to easily express hints about the targeted application semantics without modifying the sequential code, and the semantics of each transaction must be preserved, even though multiple transactions of different semantics can access common data concurrently. This second property, semantics, is crucial but makes development of a transactional system even more complex.

Relaxation and sequentiality. Several transaction models have been proposed as a relaxed alternative to the classic one. Examples are open nesting[24] and transactional boosting.[16] Both exploit commutativity by considering transactional operations at a high level of abstraction. Both also 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 acquisition of new locks at abort time, the programmer may follow lock-order rules or exploit timeouts. Alternatively, other approaches extend the interface of the transactional memory system with explicit mechanisms like functions `light-reads`, `unit-loads`, `snap`, and early `release`; for example, programmers can use early `release` explicitly to indicate from which point of a transaction all conflicts involving its read of a given location can be ignored.[17] The challenge is thus to achieve the same concurrency achievable through these models while preserving sequential code and composition of transactions.

The elastic transaction model[8] aims to preserve sequential code and guarantee composition, providing, together with the classic form of transaction model, a semantics of transactions that enables programmers to efficiently implement search structures. As in a classic transaction, the programmer must delimit the blocks of code that represent elastic transactions, 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.

Unlike classic transactions, during execution, an elastic transaction can be cut (by the elastic transactional system) into multiple classic transactions, depending on the conflicts it detects.

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)prev = head
 5:     E v
 6:
 7:     if next == null then // empty
 8:         head = newNode(E)(e,next)
 9:         return false
10:     while (v = next.getVal-
       ue()).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: }
```

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 $r(n)^i$ occurs before $w(n)^i$; the high-level insert operations of this history are atomic. A traditional transactional scheme would detect two conflicts between transactions i and j and prevent them both to commit. Nevertheless, 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 must label transaction i as being elastic to solve this issue. History $\mathcal{H}$ can then be viewed as the combination of several transactions:

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

In $f(\mathcal{H})$, elastic transaction i is 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 an expert programmer could accomplish with classic transactions for two main reasons: First, the cuts are tried dynamically at runtime depending on the interleaving of

Figure 4. Throughput (normalized over the sequential throughput) of classic transactions and existing concurrent collection.
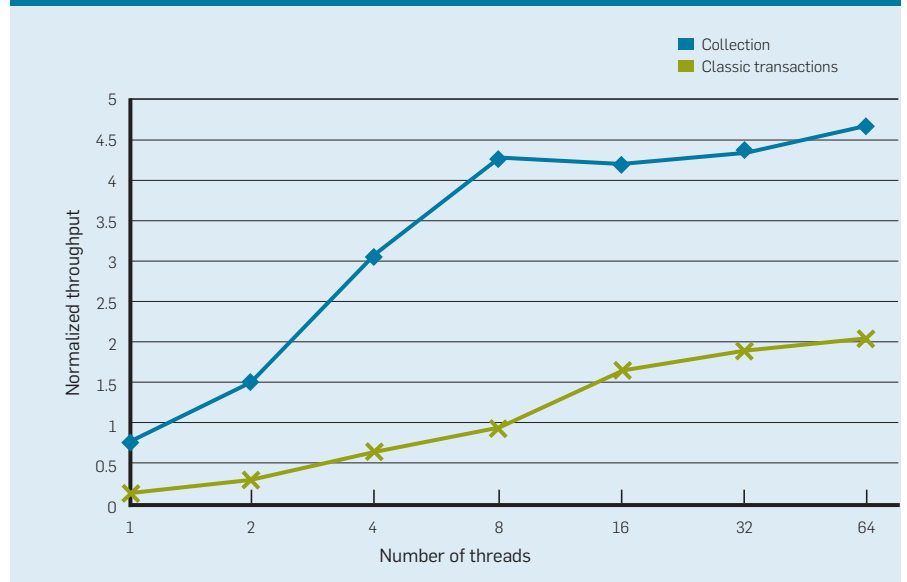
accesses; as this interleaving is generally nondeterministic, the programmer cannot just split transactions prior to execution and ensure correct executions. Second, as elastic transactions rely on dynamic information, they exploit more information than static commutativity of operations; for example, elastic transactions enable additional concurrency between two linked list adds by allowing the history involving transactions t_1 and t_2: $r(h)^{t1}$, $r(n)^{t2}$, $w(h)^{t2}$, $w(h)^{t1}$ in which neither $r(n)^{t2}$ and $w(n)^{t1}$ nor $r(h)^{t1}$ and $w(h)^{t2}$ commute.

Composition and mixture of semantics. The more semantics the transactional system provides, the more control it gives expert programmers, 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 bring added performance in search structures. A programmer can also consider the mix of the opaque classic and the relaxed elastic models with a new semantics we call "snapshot" semantics. This mix is particularly appealing for obtaining (efficiently) a result that depends on numerous elements of a data type (such as a Java Iterator); see, as an example, the snapshot transaction implementing a size method in Algorithm 5.

At first glance, providing as many forms as possible in a single toolbox system may seem to be the key solution for developing concurrent applications, but the challenge involves the mixture of these semantics. Mixing them 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; 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 the transaction can fetch the counter at start time and decide (while reading new locations) to return a value that has an appropri-

ate (not too recent) version consistent with this start time.

Algorithm 5. Java pseudocode of the size() operation with a snapshot transaction

```
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: }
```

However, the mixture of the snapshot with classic and elastic transactions requires the transaction system make sure all updates (elastic and classic) record the old value before overriding it.

The mixture problem might be more subtle if a relaxed transaction ignores a conflict involving a concurrent strong transaction that cannot ignore it. Elastic and opaque transactions typically handle this issue for read-write conflicts by requiring only the reading transaction decides on conflict resolution. Unlike writes, reads are idempotent so the semantics of the writing transaction is never altered by the outcome of the conflict resolution. Our solution relies on two features: having invisible reads, so the writing transaction does not observe the conflict, and

enforcing commit-time validation, so the reading transaction always detects the conflict.

A consequent algorithmic challenge relates to the composition of the semantics. Bob can directly nest Alice's elastic transactions into another transaction, choosing to label it as elastic, snapshot, or classic, guaranteeing atomicity and deadlock freedom of its own operation; for example, one can imagine Alice provides an elastic contains(x) Bob composes into a snapshot containsAll(C) method that returns successfully only if all elements of a collection C are present. For safety's sake, the strongest semantics of the related transactions (in this case the snapshot transaction) applies to all methods. Hence, a novice programmer, unaware of the various semantics, will always obtain a safe composite transactional method whose opacity would be conveyed to inner transactions. Which semantics to apply (when the semantics are incomparable) is an open question.

Effect on performance. To investigate the potential benefit of mixing transactions of different semantics, we ran the mixed transactions on the collection benchmarks in the exact same settings as before and reported both the new and the previously obtained results (see Figure 5). Each of the three parse operations—contains, add, and remove—is imple-

Figure 5. Throughput (normalized over the sequential throughput) of mixed transactions, classic transactions, and a collection package.
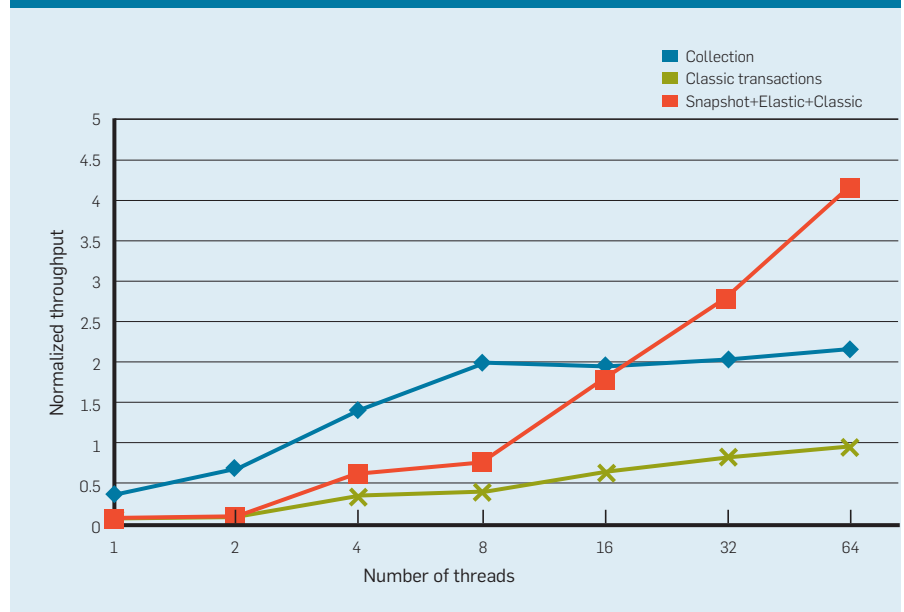
mented through an elastic transaction, and the `size` operation, which returns an atomic snapshot of the number of elements, is implemented through a snapshot transaction. The mixed transaction model performs 4.3x faster than the classic transaction model, TL2, improving on the concurrent collection package by 1.9x on 64 threads. Due to snapshot semantics, the size operation commits more frequently than with a classic transaction. The reason is a snapshot size could return values that were concurrently overridden, where classic size would 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, compensating for the overhead effect at high levels of parallelism.

The mixture of elastic and classic transactions has been shown to be effective in a non-managed language—C/C++—as well. It improved the performance of the tree library implemented in the transactional vacation-reservation benchmark by 15%;[3] it also improved the performance of a list-based set running on a many-core architecture by about 40x.[9]

Conclusion

The transaction is a proven, appealing abstraction that has been the main topic of many practical and theoretical achievements in research, despite never being widely adopted in practice. The reason the transaction abstraction is appealing as a programming construct is also the reason it might not be used in practice. That is, the appeal of transactions comes from their simplicity and bringing multicore programming to novice programmers. Average programmers can write concurrent code and, with little effort, use transactions to protect shared data against incorrectness. However, the simplicity of the concept is also its main source of rigidity, preventing expert programmers from exploiting their skills and enabling as much concurrency as they could, thereby limiting performance scalability. This limitation is inherent to the concept, not simply a matter of use.

Here, we have suggested a way out by truly democratizing the transaction concept and promoting the coexistence of different transactional semantics in the same application. Although novice programmers would still be able to exploit the simplicity of the transaction abstraction in its original (strong and hence simple) form, expert programmers would be able to exploit, whenever possible, more expressive semantics of relaxed transaction models to gain in concurrency.

As this polymorphism helps expert programmers take full advantage of transactions, they can likewise develop new efficient libraries that motivate other programmers to adopt this abstraction. It also raises new challenges for guaranteeing the various semantics can be used effectively in the same system. ⒸⒸ

References
1. Almes, G.T., Black, A.P., Lazowska, E.D., and Noe, J.D. The Eden system: A technical review. *IEEE Transactions on Software Engineering 11*, 1 (Jan. 1985), 43–59.
2. Cascaval, C., Blundell, C., Michael, M., Cain, H.W., Wu, P., Chiras, S., and Chatterjee, S. Software transactional memory: Why is it only a research toy? *Queue 6*, 5 (Sept. 2008), 46–58.
3. Crain, T., Gramoli, V., and Raynal, M. A speculation-friendly binary search tree. In *Proceedings of ACM SIGPLAN Conference on the Principles and Practice of Parallel Computing* (New Orleans, Feb. 23–27). ACM Press, New York, 2012, 161–170.
4. Dice, D., Lev, Y., Moir, M., and Nussbaum, D. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, D.C., Mar. 7–11). ACM Press, New York, 2009, 157–168.
5. Dice, D., Shalev, O., and Shavit, N. Transactional locking II. In *Proceedings of the International Symposium on DIStributed Computing, Vol. 4167 of LNCS* (Stockholm, Sept. 18–20). Springer, 2006, 194–208.
6. Dragojevic, A., Felber, P., Gramoli, V., and Guerraoui, R. Why STM can be more than a research toy. *Commun. ACM 54*, 4 (Apr. 2011), 70–77.
7. Eswaran, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. The notions of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.
8. Felber, P., Gramoli, V., and Guerraoui, R. Elastic transactions. In *Proceedings of the International Symposium on DIStributed Computing, Vol. 5805 of LNCS* (Elche, Spain, Sept. 23–25). Springer, 2009, 93–107.
9. Gramoli, V., Guerraoui, R., and Trigonakis, V. TM2C: A software transactional memory for many-cores. In *Proceedings of EuroSys* (Bern, Switzerland, Apr. 10–13). ACM Press, New York, 2012, 351–364.
10. Gramoli, V., Harmanci, D., and Felber, P. On the input acceptance of transactional memory. *Parallel Processing Letters 20*, 1 (Mar. 2010), 31–50.
11. Greenwald, M. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the Symposium on Principles of Distributed Computing* (Monterey, CA, July 21–24). ACM Press, New York, 2002, 260–269.
12. Guerraoui, R., Capobianchi, R., Lanusse, A., and Roux, P. Nesting actions through asynchronous message passing: The ACS protocol. In *Proceedings of the European Conference on Object-Oriented Programming, Vol. 615 of LNCS* (Utrecht, The Netherlands, June 29–July 3). Springer, 1992, 170–184.
13. Guerraoui, R. and Kapalka, M. *Principles of Transactional Memory*. Morgan & Claypool, San Rafael, CA, 2010.
14. Harris, T. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Symposium on DIStributed Computing, Vol. 2180 of LNCS* (Lisboa, Portugal, Oct 3–5). Springer, 2001, 300–314.
15. Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the ACM SIGPLAN Conference on the Principles and Practice of Parallel Computing* (Chicago, June 15–17). ACM Press, New York, 2005, 48–60.
16. Herlihy, M. and Koskinen, E. Transactional boosting: A methodology for highly concurrent transactional objects. In *Proceedings of ACM SIGPLAN Conference on the Principles and Practice of Parallel Computing* (Salt Lake City, Feb. 20–23). ACM Press, New York, 2008, 207–246.
17. Herlihy, M., Luchangco, V., Moir, M., and Scherer III, W.N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing* (Boston, July 13–16). ACM Press, New York, 2003, 92–101.
18. Herlihy, M. and Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Computer Architecture News 21*, 2 (May 1993), 289–300.
19. Knight, T. An architecture for mostly functional languages. In *Proceedings of the ACM Conference on LISP and Functional Programming* (Cambridge, MA, Aug. 4–6, 1986), 105–112.
20. Korland, G., Shavit, N., and Felber, P. Deuce: Noninvasive software transactional memory. *Transactions on High-Performance Embedded Architectures and Compilers 5*, 2 (Jan. 2010).
21. Liskov, B. The Argus language and system. In *Proceedings of Distributed Systems: Methods and Tools for Specification, An Advanced Course, Vol. 190 of LNCS*. Springer, 1985, 343–430.
22. Liskov, B. and Scheifler, R. Guardians and actions: Linguistic support for robust, distributed programs. In *Proceedings of the Symposium on the Principles of Programming Languages* (Albuquerque, NM). ACM Press, New York, 1982, 7–19.
23. Lynch, N.A. Multilevel atomicity a new correctness criterion for database concurrency control. *ACM Transactions on Database Systems 8*, 4 (Dec. 1983), 484–502.
24. Moss, J.E.B. Open nested transactions: Semantics and support. Poster presentation at the Workshop on Memory Performance (Austin, TX, 2006).
25. Papadimitriou, C.H. The serializability of concurrent database updates. *Journal of the ACM 26*, 4 (Oct. 1979), 631–653.
26. Ramadan, H.E., Roy, I., Herlihy, M., and Witchel, E. Committing conflicting transactions in an STM. In *Proceedings of the ACM SIGPLAN Conference on the Principles and Practice of Parallel Computing* (Raleigh, NC, Feb. 14–18). ACM Press, New York, 2009, 163–172.
27. Reuter, A. Concurrency on high-traffic data elements. In *Proceedings of the ACM Conference on the Principles of Database Systems* (Los Angeles, Mar. 29–31). ACM Press, New York, 1982, 83–92.
28. Scherer III, W.N. and Scott, M.L. Advanced contention management for dynamic software transactional memory. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing* (Las Vegas, July 17–25). ACM Press, New York, 2005, 240–248.
29. Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Ottawa, Aug. 20–23). ACM Press, New York, 1995, 204–213.
30. Shpeisman, T., Menon, V., Adl-Tabatabai, A.-L., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., and Saha, B. Enforcing isolation and ordering in STM. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (San Diego, CA). ACM Press, New York, 2007, 78–88.

Vincent Gramoli (vincent.gramoli@sydney.edu.au) is an assistant professor at the University of Sydney and a researcher at the National Information and Communication Technology Australia (NICTA).

Rachid Guerraoui (rachid.guerraoui@epfl.ch) is a professor at École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.