J. Parallel Distrib. Comput. 100 (2017) 103–127

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Elastic transactions[☆]

Pascal Felber[a], Vincent Gramoli[b,*], Rachid Guerraoui[c]

[a] University of Neuchâtel, Rue Emile-Argand 11, B-114, CH-2000 Neuchâtel, Switzerland
[b] School of Information Technologies, Bldg J12, 1, Cleveland St, University of Sydney, NSW 2006, Sydney, Australia
[c] EPFL Station 14, CH-1015 Lausanne, Switzerland

HIGHLIGHTS

- Elastic transaction is a new relaxed transactional model.
- Elastic transaction offers better performance than classic transaction when used instead.
- Elastic transaction is especially suited for search structures.

ARTICLE INFO

Article history:
Received 23 December 2014
Received in revised form
27 August 2016
Accepted 7 October 2016
Available online 27 October 2016

Keywords:
Transactional memory
Concurrent data structures

ABSTRACT

This paper presents *elastic transactions*, an appealing alternative to traditional transactions, in particular to implement search structures in shared memory multicore architectures. Upon conflict detection, an elastic transaction might drop what it did so far within a separate transaction that immediately commits, and resume its computation within a new transaction which might itself be elastic.

We present the elastic transaction model and an implementation of it, then we illustrate its simplicity and performance on various concurrent data structures, namely *double-ended queue*, *hash table*, *linked list*, and *skip list*. Elastic transactions outperform classical ones on various workloads, with an improvement of 35% on average. They also exhibit competitive performance compared to lock-based techniques and are much simpler to program with than lock-free alternatives.

1. Introduction

Transactions are an appealing synchronization paradigm for they enable average programmers to leverage modern multicore architectures. The power of the paradigm lies in its abstract nature: there is no need to know the internals of shared object implementations, it suffices to delimit every critical sequence of shared object accesses using transactional boundaries. The inherent difficulty of synchronization is hidden from the programmer and encapsulated inside the transactional memory, implemented once and for all by experts in concurrent programming.

Not surprisingly, and precisely because it hides synchronization issues, the transaction abstraction may severely hamper parallelism. This is particularly true for search data structures where transactions do not know a priori where to insert an element unless they possibly explore a big part of the data structure. Search structures implement key abstractions like queues, heaps, key–value stores, or collections but turn out to be the contention hot spots of applications aiming at leveraging modern multicore machines [59]. In an attempt to minimize this contention, transactions are typically chosen to synchronize search structures by redirecting shared accesses at run-time, instead of conservatively protecting extra memory locations ahead of time.

To illustrate the limitation of transactions, consider the bucket hash table depicted in Fig. 1 implementing an integer set and exporting operations search, insert, and remove. A bucket, itself implemented with a sorted linked list as in [44], indicates where an integer should be stored. Consider furthermore a situation involving two concurrent transactions: the first seeks to insert an integer m at some position whereas the second searches for an integer n and reads m. In a strict sense, there is a read–write conflict that may cause to block or abort one of the transactions; yet this

[☆] A preliminary extended abstract of this work has been published in the proceedings of DISC 2009 (Felber et al., 2009) [15], the current version extends it by generalizing the model, applying it to additional data structures, and comparing it against existing synchronization alternatives.
[*] Corresponding author.
E-mail addresses: pascal.felber@unine.ch (P. Felber), vincent.gramoli@sydney.edu.au (V. Gramoli), rachid.guerraoui@epfl.ch (R. Guerraoui).

http://dx.doi.org/10.1016/j.jpdc.2016.10.010
0743-7315/© 2016 Elsevier Inc. All rights reserved.

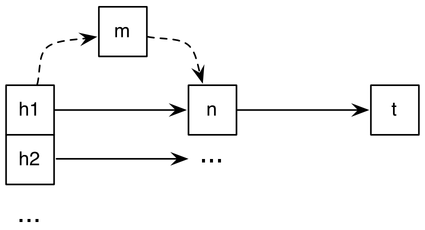

Fig. 1. A bucket hash table where a transaction (insert(m)) invalidates an almost complete transaction (search(n)) that accesses the same bucket.

is a false (search–insert) conflict. Because they are sensitive to these kinds of conflicts, regular transactions hamper concurrency, and this might have a significant impact on performance, should the data structures be large and shared by many concurrent transactions. It is important to notice here that the issue is not related to the way transactions are used, but to the paradigm itself. More specifically, assuming transactions in their traditional sense, i.e., accessing shared objects through read and write primitives, even an expert programmer *has to choose between violating consistency and hampering concurrency*.

Addressing the issue above with locks is simpler. A well-known lock-based technique to access the aforementioned sorted linked list is to parse it starting from its first head element, by acquiring multiple consecutive elements, before releasing the first of these. The technique is called *hand-over-hand locking* [4]: it looks like a *right hand* acquires the $i+1$st elements, then the *left hand* releases the ith before it acquires the $i+2$nd, and so on. This enables a level of concurrency that is hard to get with regular transactions for these are open-closed blocks that cannot overlap with one another. Instead, a transaction keeps track of all its accesses during its entire lifespan, hence a concurrent update on the ith element triggers a conflict even at the point where the transaction accesses the $i+2$nd element. This lack of concurrency is problematic in numerous data structures in which a big part of the data structure must be parsed in order to find the targeted location.

Several transactional models were proposed to cope with similar problems. The theory of commutativity [38] helps identifying particular transactional operations that can be reordered without affecting the semantics of the execution. This commutativity was key to multiple transactional models. The consistency criterion of multi-level serializability [69] exploits this commutativity to reorder low level operations within operations at a higher level of abstractions. Similar to these models, elastic transactions do not relieve the programmer from the burden of understanding the semantics of the operations, but as far as we know, no existing transactional models can exploit the runtime information in search data structure executions to decide dynamically whether operations commute.

We propose *elastic* transactions, an efficient alternative to traditional transactions for such search data structures. Just like for a regular transaction yet differently from most transaction models as we discuss in Section 2, the programmer must simply delimit the blocks of code that represent elastic transactions. Nevertheless, during its execution, an elastic transaction can be *cut* (by the elastic transactional memory) into multiple regular transactions, depending on the conflict it encountered at runtime. Intuitively, the cut allows to automatically decide at runtime whether operations commute.

More specifically, upon conflict detection an elastic transaction decides whether it can cut itself; if so it commits the past accesses as if they were part of a regular transaction before resuming into a continuation transaction until it encounters a new conflict or commits. A cut is prohibited if, between the times of two of its consecutive accesses on two locations, these two locations get updated by other transactions. Only in this rare case does the elastic transaction abort. In other words, it is not possible for the elastic transaction to abort if, during the interval where the elastic transaction executes a pair of consecutive accesses on two locations, at most one of these locations gets updated. In case the elastic transaction does not abort, a cut could cause the elastic transaction to execute a constant number of additional accesses before committing the past ones. In a sense, these few extra accesses can be viewed as a partial roll-back that is the price to pay to avoid aborting the elastic transaction. We will see later that, as a result, there is no need to undo any write.

At this point, one might ask why we propose a new transactional model instead of using locks. The reasons are twofold: unlike locks, elastic transactions (i) can be combined with other transactions to permit extensibility through code composition and (ii) enable the direct reuse of sequential code. To illustrate code composition, consider again a hash table implementation. Consider however that this implementation now extends the integer set abstraction into a dictionary abstraction aimed at exporting a move operation, which modifies the key of a value. Given a transactional integer set, one has simply to encapsulate a transactional remove and a transactional insert into a single transaction to obtain an atomic move. By contrast, using locks explicitly is known to be a difficult task [50,55] prone to deadlocks when one process moves from bucket ℓ_1 to bucket ℓ_2 while another moves from ℓ_2 to ℓ_1. Given a lock-based integer set, the programmer must know the granularity of internal locks, like the size of lock stripes, to make sure that the new move and existing updates on common parts are mutually exclusive. Even so, the original implementation tuned to provide an efficient integer set interface may provide an inefficient extension.

Finally, lock-free implementations can neither ensure both extensibility and concurrency. To ensure the atomicity of the move resulting from the composition of lock-free remove and insert, one could modify a copy of the data structure before switching a pointer from one copy to another [27]. This, however, prevents two concurrent updates, which modify disjoint locations, from succeeding. One could also use a multi-word compare-and-swap instruction [21] but this is often considered inefficient and upcoming architectures rather favor general-purpose transactions. A remarkable example of the lack of extensibility of efficient lock-free algorithm is the complete redesign of a hash table structure into a split-ordered linked list to support a lock-free resize operation [58]. Although the resize allows hash table buckets to move among consecutive list nodes, it does not allow nodes to move among hash table buckets.

Elastic transactions: a primer

To give an indication of the main idea underlying elastic transactions, consider the integer set abstraction implemented using the linked list data structure. Each of the insert, remove, and search operations consists of lower-level operations: some reads and possibly some writes. Consider an execution in which two transactions, i and j, try to insert keys 3 and 1. Each insert transaction parses the nodes in ascending order up to the node before which it should insert its key. Let $\{2\}$ be the initial state of the integer set and let h, n, t denote respectively the memory locations where the head pointer, the single node (its key and next pointer) and the tail key are stored. Let $\mathcal{H}$ be the following history of operations where transaction j inserts 1 while transaction i is parsing the data structure to insert 3 at its end. (We indicate only operations of non-aborting transactions and omit commit events for simplicity.)

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

This history is clearly not serializable [51] since there is no sequential history where $r(h)^i$ occurs before $w(h)^j$ and $r(n)^j$ occurs before