Composition vs Concurrency

Vincent Gramoli Rachid Guerraoui Mihai Letia

Abstract

Composing software is notoriously difficult, especially when it is concurrent. Two fine-grained locked operations may for instance easily deadlock upon composition. One of the most appealing features of transactions is, on the other hand, that they compose. Indeed they do so, but only in their original, orthodox, form. As we discuss in the paper, as soon as we slightly relax the model to enable more concurrency, we incur the risk of breaking encapsulation.

1 Composition

In the sequential world, composition is well understood, at least to some extend. Given an abstract data type with two operations, π_1 and π_2, say written in some imperative extensible language, one can add to the type a third operation, π_3, that encapsulates the two previous operations and executes them sequentially; namely: $\pi_3 = \pi_1 \circ \pi_2$. Assuming the sequential specifications of π_1 and π_2, one can derive that of π_3. For instance, given a set abstraction, $\pi_1 = \mathsf{remove}(x)$ and $\pi_2 = \mathsf{insertIfNotPresent}(y, x)$, which aims at inserting y if x is not in the set, it is clear that y gets inserted when π_3 executes.

Consider now the fact that multiple instances of operation π_3, can run concurrently. It is natural to ask what composition semantics is obtained. Clearly, this depends on (a) the properties that the individual operations π_1 and π_2 satisfy, and (b) the actual semantics of the composition.

In our context, we assume that the operations ensure *atomicity* and *deadlock-freedom* and we ask which form of composition preserves these properties.[1]

We say that two operations π_1 and π_2, ensuring each atomicity and deadlock-freedom *compose*, if atomicity and deadlock-freedom are preserved by the resulting composition $\pi_3 = \pi_1 \circ \pi_2$.

To illustrate this, recall that a fine-grained locking transformation does not compose: $\pi_1 = \mathsf{remove}$ and $\pi_2 = \mathsf{insert}$ cannot be encapsulated into $\pi_3 = \mathsf{move}$ as explained in [4]. The reason is that a concurrent execution with π_3' moving x to y and π_3'' moving y to x would be deadlock-prone. Note also that any execution where π_3 runs concurrently with π_1 or π_2 must also be linearizable for π_1 and π_2 to compose.

[1] An operation is *atomic* if all histories produced by a concurrent execution involving this operation and any other existing operation results in a linearizable history. An operation is *deadlock free* if it progresses despite the concurrent execution of any exiting operation.

2 Transactions

A memory transaction is an appealing concurrent programming abstraction as it makes programs easily extensible. The transactional transformation consists of encapsulating all operations in a transaction. For instance, a programmer simply has to encapsulate two transactional operations π_1 and π_2 into another transaction to obtain the composition π_3 that is atomic and deadlock-free if the invidiual operations are. With the same technique another programmer can compose the compositions themselves, and so on.

Transactions, in their original form, limit however concurrency as they detect conflicts at the read/write level and may over-conservatively abort even though the execution would be correct at the application level [3]. Several alternative transactional models have been proposed to execute transactions depending on their application-level semantics [1, 2, 5–7] and improve concurrency. As we discuss below however, these models usually annihilate one of the most appealing aspects of the transactional model, i.e., its composition. We discuss below the non-composability of various transactional models.

Inversion-based transactions. According to this model, transactions execute a series of operations, each being inverted upon abort. This model encompasses Transactional Boosting [6] and Open-Nesting [7] models. The goal is to reduce false-conflicts by considering conflicts only between high level sub-operations rather than between reads and writes.

To illustrate the idea, consider that such a transaction executes up to a certain point where a conflict between its operations and a concurrent transaction forces it to abort. This transaction has then to undo all the operations that it has executed by running inverse operations. For example, some set operations remove and insert are naturally the inverse of each other and can be appended in a common transaction to obtain a collection operation, move. If a move successfully executes a $\mathsf{remove}(x)$ and then aborts,

it will simply roll-back its changes by executing insert(x), the appropriate inverse operation.

The drawback is that operations cannot always be inverted without breaking original abstraction. A simple example is the impossibility to compose a π_1 = removeAll and a π_2 = insertAll into a π_3 = moveAll as the inverse operation of π_1 cannot be defined with the existing operations. Hence, the only way for a programmer to obtain a moveAll by composing existing code would be to break the existing abstraction and to reimplement insertAll and removeAll, as well as implement their respective inverse.

Escaped transactions. According to this model, transactions use explicit *escape* mechanisms instead of inverse operations to enhance concurrency. In addition to identifying transactional reads/writes by delimiting transactions, the programmer must place these escape mechanisms within transactions to indicate when and which accessed locations can be safely unprotected. Early Release [5] and View Transaction [1] belong to this model as they expose release and light-read primitives, respectively, to the programmer. The release consists of discarding, *a posteriori* from the transaction read-set, some logged read while the light-read primitive executes a read operation that is not logged. As non-logged reads are not visible from the transaction, the unnecessary read-write conflicts involving them are simply ignored.

As we discuss below, this model does not allow for composition either. To illustrate this, assume that a transaction t protects a memory location x in the time interval $[\tau, \tau']$, even though t is still pending at τ'. Although this suffices to implement a π_1 = insert operation in a sorted linked list (because parsed elements do not need to remain protected), this is inadequate to implement π_3 = insertIfNotPresent(x, y) by composing some π_2 = contains and this π_1. Typically, a transaction executing such insertIfNotPresent(x, y) would unprotect element x before completing, and a concurrent insertIfNotPresent(y, x) transaction would lead to an inconsistent state where both operations succeed in inserting x and y. This execution is clearly not atomic.

Mixed-granularity transactions. Another model of interest is the one where we use differing granularities to protect sets of locations that are accessed transactionally. This mixed-granularity class of transactions include Transactional Predication [2]. The advantage is to let the user genuinely tune the number of locations protected by a single metadata to reach the ideal tradeoff between metadata accesses and false-sharing: The more locations for a given metada, the less accesses to metadata. Conversely, the less locations per metadata the less false-sharing we encounter.

This approach fails in composing operations as it might use differing granularities for distinct operations. Consider the presence of two Map operations—π_1 = get operation returning the value associated with a given key and π_2 = put operation inserting a new key-value pair. Composing multiple of these put operations into a π_3 = putAll is not possible because the get and put consult boolean metadata to detect conflicts whereas operation putAll has to update the counter metadata of the element stripe it updates to indicate that new pairs have been added. As they access disjoint metadata to insert key-value pairs, the get cannot detect that a new putAll have inserted.

3 Concurrency and Composition

As opposed to other models, the elastic transactional model [3] combines elastic transactions for concurrency and normal transactions for composition. Unlike escaped transactions, which use explicit primitives that cannot be disabled upon composition without breaking abstractions, elastic transactions use the same begin-read-write-commit interface as normal transactions, yet their read/write behave differently depending on the type of the transaction that is either given by its parent transaction (if some exists), or given as an argument to its begin (otherwise): when encapsulated in a normal transactional operation, the elastic operation behave as normal. The drawback is when all elastic transactions are encapsulated in normal transactions, performance becomes as bad as in the normal model. We are currently introducing multiversioning in this model to obtain high concurrency and composition even in this particular scenario.

References

[1] Y. Afek, A. Morrison, and M. Tzafrir. View transactions: Transactional model with relaxed consistency checks. In *PODC*, page 65, 2010.

[2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High performance concurrent sets and maps for STM. In *PODC*, pages 6–15, 2010.

[3] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, pages 93–107, 2009.

[4] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[5] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[6] E. Koskinen and M. Herlihy. Concurrent non-commutative boosted transactions. In *PODC*, pages 272–273, 2009.

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