Highly-scalable Concurrent Objects

Deepthi Devaki Akkoorath
University of Kaiserslautern
Germany
akkoorath@cs.uni-kl.de

Annette Bieniusa
University of Kaiserslautern
Germany
bieniusa@cs.uni-kl.de

ABSTRACT

Synchronisation is a bottleneck in shared memory concurrent programs. In this paper, we present *Mergeable Data Types* (MDTs) for relaxing synchronisation in shared memory system analogously to CRDTs in replicated systems. We present the key properties of MDTs and discuss the design of several MDTs.

Keywords

Concurrent Data Structures, Weak Consistency

1. INTRODUCTION

In concurrent programs with shared-memory synchronization, data structures are in general mutable and updates are executed in-place. The correctness condition that is traditionally applied to these objects is linearizability [8]. Linearizability induces restrictions on possible parallelism and imposes high cost due to coordination and synchronisation.

In contrast, replicated data types for distributed systems [12, 4] guarantee convergence while avoiding coordination and facilitating asynchronous replication.

An obvious question is how we can relax synchronization in concurrent shared-memory programs, while providing weaker, but well-defined semantics. Akin to CRDTs in distributed systems, we therefore propose *Mergeable Data Types (MDTs)*. The basic idea is as follows: A thread updates its private copy of an object and later merges its changes to the shared global object instance.

In prior work [2], we presented a programming model based on Software Transactional Memory employing mergeable objects. Transactions optimistically execute updates concurrently on shared objects and buffer the modifications. On commit, transaction merges its results to the global memory. Using mergeable objects guarantees that all concurrent updates can be merged, and thus transactions never abort due to update conflicts.

However, the CRDTs designs used for geo-replicated distributed systems tend to be too inefficient for our purpose because of their relatively expensive merge operation. In geo-replicated systems, the cost of merge operation is negligible compared to the high synchronisation cost. Hence CRDTs are not designed to keep the merge efficient. However, in shared memory concurrent programs, since the merge is executed synchronously it is important to reduce the cost of it. In this paper, we identify important properties of mergeable objects that can be employed for efficient implementations in shared memory systems. We discuss the implementation of several MDTs and provide guidelines how to design other mergeable objects.

PaPOC'16, April 18-21, 2016, London, United Kingdom
© 2016 ACM. ISBN 978-1-4503-4296-4/16/04...$15.00
DOI: http://dx.doi.org/10.1145/2911151.2911158

2. MERGEABLE OBJECTS

A mutable shared object O is said to be mergeable when concurrent updates can be merged meaningfully. It provides a global copy which is the last committed version accessible by all threads. A thread which needs to update O obtains a thread-local copy of O and modifies the thread-local copy. After having executed the updates, the thread can commit its changes by merging the local copy to the last committed version. Hence, we can identify two main properties for MDTs - *persistence* and *mergeability*.

2.1 Persistence

A data structure is said to be *persistent* if multiple versions of it are accessible. It is partially persistent if only some versions are accessible, and fully persistent if all versions are accessible [5]. Persistence is an important property for MDTs because it allows concurrent access to multiple versions. Multiple thread-local versions and global versions must co-exist.

There are different mechanisms to achieve persistence, such as copy-on-write or path copying [5]. Depending on the data type semantics, their implementations may use specific techniques to make it efficiently persistent. Hence, it is important to make persistence a property of the MDT rather than a universal mechanism implemented by a version manager.

Which versions must be persisted? Typically, not all versions have to be persisted. Old versions that are known to not be required and never be accessed again should be garbage-collected. The specific details depend on the actual system's transaction semantics, and are left for future work.

2.2 Mergeability

A shared object may get concurrently accessed and updated by multiple threads resulting in different versions of the object. These versions must be merged together to create a new version. Extending the Abstract Data Type definitions that specify their semantics, MDT s must be mergeable. This means that the MDT must define the semantics of concurrent updates on different versions and the semantics of the merge of two versions.

Which versions must be mergeable? In our case, every thread-local instance of an object must be mergeable with the last committed global version. On the other hand, local versions from two threads do not have to be mergeable. In our setting, all thread-local versions are branched of the global version and committed back to it. This is in contrast to state-based CRDTs in eventually consistent systems where all versions residing on different replicas must be mergeable.

There are two aspects of mergeability namely *semantic mergeability* and *structural mergeability.*

Semantic mergeability tackles the semantics of the data type operations with respect to the merge operation; i.e., whether the objects behaves "as expected" after merging. Similar to CRDTs, thread-local updates should be reflected semantically in the merged version. A deterministic merge guarantees that any concurrent updates can merge their results and obtain a consistent deterministic state.

Structural mergeability is related to how efficiently two versions can be merged. This refers to the low-level implementation details of the data structure. Copy-on-write for large data structures such as lists, trees, sets etc. is not efficient in terms of run-time and memory usage. Hence, it is important to share parts of data structure from multiple versions and only keep distinct the parts which are required to distinguish different versions. Different mechanism are employed in persistent data structures in functional programs [11]. Structural mergeability is inspired by these implementations, but having an efficient merge as additional requirement.

2.3 Examples

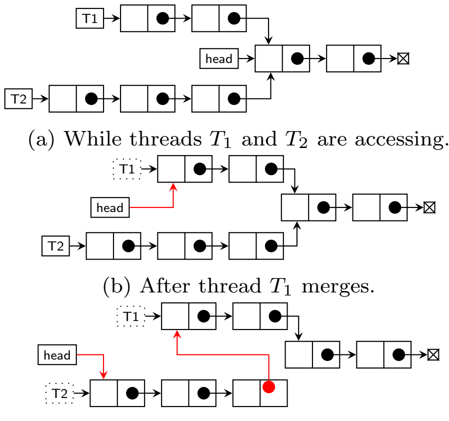
The implementation of mergeable data types must deal with the above two properties. In this section, we discuss the design for three MDTs.

Counter.

A shared counter can be incremented and decremented by concurrent threads. A mergeable counter is designed as follows: The global instance is represented by an integer; the thread-local instance has two integers, one representing the global value when starting the transaction and the other one counting the thread-local increments. The merge adds the local increments to the global value and resets the local increments to 0. Hereby, the merge relies on commutativity and arithmetic properties of increment. Due to its compact representation, persistence is achieved by copy-on-write.

Add-only Bag.

A bag is a set data structure allowing duplicate elements to be added. Here, threads can concurrently add elements without violating its semantical correctness. The Mergeable Add-only Bag can be implemented using a persistent linked list (cf. Fig. 1). Multiple threads can update the bag without copying the entire list to its thread-local storage. Multiple versions co-exits using a multi-headed list. The merge is done by adjusting the respective pointers.



(a) While threads T_1 and T_2 are accessing.
(b) After thread T_1 merges.
(c) After thread T_2 merges.

Figure 1: Mergeable Add-only Bag. T1 represents the local version of thread 1, T2 the local version of thread 2. head is the last committed version.

Observed-remove Set.

A set can be implemented using a form of binary search tree while providing the semantics of an observed-removed set [3]. The two update operations on a set are adding and removing elements. Concurrent adds of the same element make the element available in the set; on merge, the two concurrently added elements are considered as the same element. When two versions merge where one has an add operation and the other a concurrent remove of the same element, then the resulting version retains the element in the set.

In the implementation (Alg.1), persistence is achieved by assigning version numbers to each node. A node's VersionInfo consists of three elements; *first* denotes the version when the element was first added, *last* denotes the version when the element was last added, *removed* denotes when the element was removed. An element is valid in all versions between *first* and *removed.* When an element is re-added after a removal, a new set of VersionInfo is added to the list.

When a *val* is being added, the potential parent node of the *val* is identified and this information is stored locally. While merging, the val is inserted as a child of the parent and the vinfo.first is set to the new version number. If a node corresponding to *val* already exists, then vinfo.last is set to the new version number.

When a *val* is being removed, the node and its vinfo.last is identified. While merging, if there was any concurrent add of the same *val* (i.e if vinfo.last is different from what it observed), then this remove is ignored. Otherwise, the node is marked as removed by setting vinfo.removed to the new version number.

3. RELATED WORKS

Concurrent Data Structures.

Scalable designs of concurrent linearizable data structures has been an active area of research [10]. Generic mechanisms to implement concurrent data structures such as using Transactional Memory [7], Read-copy-update [9] are also widely studied. However, the correctness criteria applied in all recent literature is linearizability. In this paper we

Algorithm 1 OR-Set using a binary search tree.

```
TREE = {NODE root, INT vid, LIST localadded, LIST localremoved }
NODE = { val, NODE left, NODE right, VERSIONINFO vinfo}
VERSIONINFO = LIST (first, last, removed)

function ADD(val, TREE t)
    NODE p = findParent(val, t.root)
    localadded.add(⟨val, p⟩)
end function

function REMOVE(val, TREE t)
    NODE p = findNode(val, t.root)
    if p $\neq$ null then
        vid = p.vinfo.last
    end if
    localremoved.add(⟨p,vid⟩ )
    if val in localadded then
        localadded.remove(val)
    end if
end function

function LOOKUP(val, TREE t)
    NODE p = findNode(val, t.root)
    if p == Null then return false
    end if
    if p.vinfo.removed == 0 and p.vinfo.first $\leq$ t.vid then
        return true
    else if p.vinfo.first $\leq$ t.vid $<$ p.vinfo.removed then
        return true
    end if
    return false
end function

function MERGE(TREE t)
    newvid = t.vid+1
    for ⟨val,p⟩ in localadded do
        NODE n = findOrCreateNode(val, p),
        if n.vinfo.head.removed $>$ 0 then
            n.vinfo.add.newVinfo(newvid, newvid, 0),
        else
            n.vinfo.head.last = newvid
        end if
    end for
    for ⟨p,obvid⟩ in localremoved do
        if p.vinfo.removed = 0 then do nothing
        else if p.vinfo.head.last $>$ obvid then do nothing
        else if p.vinfo.headlast == obvid then
            p.vinfo.head.removed = newvid
        end if
    end for
end function
```

explore an alternative: by allowing parallel versions to co-exists, such that threads can update/read in parallel without having the order of operations compatible to a sequential specification of the data structure.

Transactional Data Structures.

In most cases Transactional Memory(TM) cannot exploit the design of concurrent linearizable data structures. Research in Transactional Data Structures aim for scalable data structure designs to use with in a TM.

Transactional Boosting [6] is a method to convert linearizable objects into a transactional object that can be used with in a composable transaction. Concurrent commutative method invocations on a transactionally boosted object can execute without aborting the transaction. MDT aims at object specific conflict resolution which may allow non-commutative operations to occur in parallel.

Consistency Oblivious Programming [1] is an approach to design concurrent algorithms to execute in an optimistic way. Using COP, operations on a concurrent data structure can be designed to work efficiently with STM.

4. CONCLUSION

Mergeable Data Types in shared memory concurrent programs can be considered analogously to CRDTs in replicated distributed systems. However, it is important to notice that the implementations of these two data types must be adapted to the specific system at hand. We identified two important properties of MDTs that determine the actual efficiency of an implementation: persistence and mergeability. We also present the novel design of an OR-Set MDT that implements persistence within the data structure.

5. ACKNOWLEDGMENTS

This research is supported in part by the European FP7 project 609 551 SyncFree.

6. REFERENCES

[1] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *Proceedings of the 15th International Conference on Principles of Distributed Systems*, OPODIS'11, pages 65–79, 2011.

[2] D. D. Akkoorath and A. Bieniusa. Transactions on mergeable objects. In *Programming Languages and Systems*, pages 427–444. Springer, 2015.

[3] A. Bieniusa, M. Zawirski, N. M. Preguiça, M. Shapiro, C. Baquero, V. Balegas, and S. Duarte. An optimized conflict-free replicated set. *CoRR*, abs/1210.3368, 2012.

[4] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, 2012.

[5] A. Fiat and H. Kaplan. Making data structures confluently persistent. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 537–546. Society for Industrial and Applied Mathematics, 2001.

[6] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 207–216, 2008.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300. ACM, 1993.

[8] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[9] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[10] M. Moir and N. Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[11] C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[12] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011.