

Supporting Lock-Free Composition of Concurrent Data Objects*

Daniel Cederman and Philippas Tsigas
Department of Computer Science and Engineering
Chalmers University of Technology, Sweden
{cederman,tsigas}@chalmers.se

ABSTRACT

Lock-free data objects offer several advantages over their blocking counterparts, such as being immune to deadlocks and convoying and, more importantly, being highly concurrent. However, composing the operations they provide into larger atomic operations, while still guaranteeing efficiency and lock-freedom, is a challenging algorithmic task.

We present a lock-free methodology for composing highly concurrent linearizable objects together by unifying their linearization points. This makes it possible to relatively easily introduce atomic lock-free move operations to a wide range of concurrent objects. Experimental evaluation has shown that the operations originally supported by the data objects keep their performance behavior under our methodology.

Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming

General Terms

Algorithms, Experimentation

Keywords

composition, data structures, lock-free

1. INTRODUCTION

Lock-free data objects offer several advantages over their blocking counterparts, such as being immune to deadlocks, priority inversion, and convoying, and have been shown to work well in practice [27, 31, 32]. They have been included in Intel's Threading Building Blocks Framework [17], the

*This work was partially supported by the EU as part of FP7 Project PEPPER (www.pepper.eu) under grant 248481 and the Swedish Research Council under grant number 37252706. Daniel Cederman was supported by Microsoft Research through its European PhD Scholarship Programme.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CF'10, May 17–19, 2010, Bertinoro, Italy.

Copyright 2010 ACM 978-1-4503-0044-5/10/05 ...\$10.00.

NOBLE library [27] and the Java concurrency package [20], and will be included in the forthcoming parallel extensions to the Microsoft .NET Framework [24]. However, the lack of a general, efficient, lock-free method for composing them makes it difficult for the programmer to perform multiple operations together atomically. To efficiently glue together multiple objects, and their respective operations, one needs to perform an often challenging task that requires an efficient algorithmic design for every particular composition. The task is made difficult by the fact that lock-free data objects are often too complicated to be trivially altered.

Composing blocking data objects also puts the programmer in a difficult situation, as it requires knowledge of the way locks are handled internally (in the implementation of the objects themselves), in order to avoid deadlocks. It is not possible to build on lock-based components without examining their implementations and even then the drawbacks of locking will not go away.

Software Transactional Memories (STMs) provide good composability [13], but have problems with high overhead and have poor support for dealing with non-transactional code [3, 19]. They require, with few exceptions, that the data objects are rewritten to be handled completely inside the STM, which lowers performance compared to pure non-blocking data objects.

1.1 Composing

With the term composing we refer to the task of binding together multiple operations in such a way that they can be performed as one, without any intermediate state being visible to other processes. In the literature the term is also used for nesting, making one data object part of another, which is an interesting problem, but outside the scope of this paper.

Composing lock-free concurrent data objects, in the context that we consider in this paper, has been an open problem in the area of lock-free data objects. There exists customized compositions of specific concurrent data objects, including the composition of lock-free flat-sets by Gidenstam et al. that constitute the foundation of a lock-free memory allocator [9, 8], but no generic solution.

Using blocking locks to compose lock-free operations is not a viable solution, as it would reduce the concurrency and remove the lock-freedom guarantees of the operations. The reason for this is that the lock-free operations would have to acquire a lock before executing, in order to ensure that they are not executed concurrently with any composed operations. This would cause the operations to be executed sequentially and lose their lock-free behavior. Simply put, a

generic way to compose concurrent objects, without foiling the possible lock-freedom guarantees of the objects, has to be lock-free itself.

1.2 Contributions

The main contribution of this paper is to provide a methodology to introduce atomic move operations, that can move elements between objects of different types, to a large class of already existing concurrent objects without having to make significant changes to them. It manages this while preserving the lock-free guarantees of the object and without introducing significant performance penalties to the previously supported operations. Move operations are an important part of the core functionality needed when composing any kind of containers, as they provide the possibility to shift items between objects.

In our methodology we present a set of properties that can be used to identify suitable concurrent objects and we describe the mostly mechanical changes needed for our move operation to function together with the objects. The properties required by our methodology are fulfilled by a wide variety of lock-free data objects, among them lock-free stacks, queues, lists, skip-lists, priority queues, hash-tables and dictionaries [30, 23, 29, 7, 28, 34, 21, 14].

Our methodology is based on the idea of decomposing and then arranging lock-free operations appropriately so that their linearization points can be combined to form new composed lock-free operations. The linearization point of a concurrent operation is the point in time where the operation can be said to have taken effect. Most concurrent data objects that are not read- or write-only support an insert and a remove operation, or a set of equivalent operations that can be used to modify its content. These two types of operations can be composed together using the method presented in this paper to make them appear to take effect simultaneously. By doing this we provide a lock-free atomic operation that can move elements between objects of different types. To the best of our knowledge this is the first time that such a general scheme has been proposed.

As a proof of concept we show how to apply our method on two commonly used concurrent data objects, the lock-free queue by Michael and Scott [23] and the lock-free stack by Treiber [30]. Experimental results on an Intel multiprocessor system show that the methodology presented in the paper, applied to the previously mentioned lock-free implementations, offers significantly better performance and scalability than a composition method based on locking. The proposed method does this in addition to its qualitative advantages regarding progress guarantees that lock-freedom offers. Moreover, the experimental evaluation has shown that the operations originally supported by the data objects keep their performance behavior while used as part of our methodology.

2. THE MODEL

The model considered is the standard shared memory model, where a set of memory locations can be read from and written to, by a number of processes that progress asynchronously. Concurrent data objects are composed of a subset of these memory locations together with a set of operations that can use read and write instructions, as well as other atomic instructions, such as compare-and-swap (CAS).

We require all concurrent data objects to be linearizable to assure correctness.

Linearizability is a commonly used correctness criterion introduced by Herlihy and Wing [16]. Each operation on a concurrent object consists of an invocation and a response. A sequence of such operations makes up a history. Operations in a concurrent history can be placed in any order if they occur concurrently, but an operation that finishes before another one is invoked must appear before the latter. If the operations in any actual concurrent history can be re-ordered in such a way, so that the history is equivalent to a correct sequential history, then the concurrent object is linearizable. One way of looking at linearizability is to think that an operation takes effect at a specific point in time, the linearization point. All operations can then be ordered according to the linearization point to form a sequential history.

3. THE METHODOLOGY

The methodology that we present can be used to unify the linearization points of a remove and an insert operation for any two concurrent objects, given that they fulfill certain requirements. We call a concurrent object that fulfills these requirements a *move-candidate* object.

3.1 Characterization

DEFINITION 1. *A concurrent object is a move-candidate if it fulfills the following requirements:*

1. *It implements linearizable operations for insertion and removal of a single element.*
2. *Insert and remove operations invoked on different instances of the object can succeed simultaneously.*
3. *The linearization points of the successful insert and remove operations can be associated with successful CAS operations, (on a pointer), by the process that invoked it. Such an associated successful CAS can never lead to an unsuccessful insert or remove operation.*
4. *The element to be removed is accessible before the linearization point.*

To implement a move operation, the equivalent of a remove and insert operation needs to be available or be implemented. A generic insert or remove operation would be very difficult to write, as it must be tailored specifically to the concurrent object, which motivates the first requirement.

Requirement 2 is needed since a move operation tries to perform the removal and insertion of an element at the same time. If a successful removal invalidates an insertion, or the other way around, then the move operation can never succeed. This could happen when the insert and remove operations share locks between them or when they are using memory management schemes such as hazard pointers [22], if not dealt with explicitly. With shared locks there is the risk of deadlocks, when the process could be waiting for itself to release the lock in the remove operation, before it can acquire the same lock in the insert operation. Hazard pointers, which are used to mark memory that cannot yet be reused, could be overwritten if the same pointers are used in both the insert and remove operations.

Requirement 3 requires that the linearization points can be associated with successful CAS operations. The linearization points are usually provided together with the algorithmic description of each object. Implementations that use the LL/SC¹ pair for synchronization can be translated to ones that use CAS by using the construction by Doherty et al. that implements the LL/SC functionality from CAS [6]. The requirement also states that the CAS operation should be on a variable holding a pointer. This is not a strict requirement; the reason for it is that the DCAS operation used in our methodology often needs to be implemented in software due to lack of hardware support for such an operation. By only working with pointers it makes it easier to identify words that are taking part in a DCAS operation. The last part, which requires the linearization point of an operation to be part of the process that invoked it, prevents concurrent data objects from using some of the possible helping schemes, but not the majority of them. For example, it does not prevent using the commonly used helping schemes where the process that helps another process is not the one that defines the linearization point of the process helped. As described in Section 1.2, there is a large class of well-known basic and advanced data objects that fulfill this requirement.

Requirement 4 is necessary as the insert operation needs to be invoked with the removed element as an argument. The element is usually available before the linearization point, but there are data objects where the element is never returned by the remove operation, or is accessed after the linearization point for efficiency reasons.

3.2 The Algorithm

The main part of the algorithm is the actual move operation, which is described in the following section. Our move operation makes heavy use of a DCAS operation that is described in detail in Section 3.2.2.

3.2.1 The Move Operation

The main idea behind the move operation is based on the observation that the linearization points of many concurrent objects' operations is a CAS and that by combining these CASs and performing them simultaneously, it would be possible to compose operations. A move operation does not need an expensive general multi-word CAS, so an efficient two word CAS customized for this particular operation is good enough. We would like to simplify the utilization of this idea as much as possible, and for this reason we worked towards three goals when we designed the move operation:

- The changes required to adapt the concurrent data object should be minimal and be possible to perform mechanically.
- The performance impact on the normal operations of the concurrent data objects should be minimized.
- The move operation should be lock-free if the insert and remove operations are lock-free.

With these goals in mind we decided that the easiest and most generic way would be to reuse the remove and insert

¹LL (Load-Link) and SC (Store-Conditional) are used together. LL reads a value from a memory location and SC can then only write a new value at the same location if the memory location has not been written to since the last LL.

operations that are already supported by the object. By definition a move-candidate operation has a linearization point that consists of a successful CAS. We call the part of the operation prior to this linearization point the init-phase and the part after it the cleanup-phase. The move can then be seen as taking place in five steps:

1st step. The init-phase of the remove operation is performed. If the removal fails, due for example to the element not existing, the move is aborted. Otherwise the arguments to the CAS at the potential linearization point are stored. By requirement 4 of the definition of a move-candidate, the element to be moved can now be accessed.

2nd step. The init-phase of the insert operation is performed using the element received in the previous step. If the insertion fails, due for example to the object being full, the move is aborted. Otherwise the arguments to the CAS at the potential linearization point are stored.

3rd step. The CASs that define the linearization points, one for each of the two operations, are performed together atomically using a DCAS operation with the stored CAS arguments. Step two is redone if the DCAS failed due to a conflict in the insert operation. Steps one and two are redone if the conflict was in the remove operation.

4th step. The cleanup-phase for the insert operation is performed.

5th step. The cleanup-phase for the remove operation is performed.

To be able to divide the insert and remove operations into the init- and cleanup-phases without resorting to code duplication, it is required to replace all possible linearization point CASs with a call to the `scas` operation. The task of the `scas` operation is to restore control to the move operation and store the arguments intended for the CAS that was replaced. The `scas` operation is described in Algorithm 3 and comes in two forms, one to be called by the insert operations and one to be called by the remove operations. They can be distinguished by the fact that the `scas` for removal requires the element to be moved as an argument. If the `scas` operation is invoked as part of a normal insert or remove, it reverts back to the functionality of a normal CAS. This should minimize the impact on the normal operations.

If the DCAS operation used is a software implementation that uses helping, it might be required to use hazard pointers to disallow reclaiming of the memory used by it. In those cases the hazard pointers can be given as an argument to the `scas` operation and they will be brought to the DCAS operation. The DCAS operation provided in this paper uses helping and takes advantage of the support for hazard pointers.

If the DCAS in step 3 should fail, this could be for one of two reasons. First, it could fail because the CAS for the insert failed. In this case the init-phase for the insert needs to be redone before the DCAS can be invoked again. Second, it could fail because the CAS for the remove failed. Now we need to redo the init-phase for the remove, which means

that the insert operation needs to be aborted. For concurrent objects such as linked lists and stacks there might not be a preexisting way for the insert to abort, so code to handle this scenario must be inserted. The code necessary usually amounts to freeing allocated memory and then return. The reason for this simplicity is that the abort always occurs before the operation has reached its linearization point. If the insertion operation can fail for reasons other than conflicts with another operation, there is also a need for the remove operation to be able to handle the possibility of aborting.

Depending on whether one uses a hardware implementation of a DCAS or a software implementation, it might also be required to alter all accesses to memory words that could take part in DCAS, so that they access the word via a special read-operation designed for the DCAS.

A concurrent object that is a move-candidate (Definition 1) and has implemented all the above changes is called a *move-ready* concurrent object. This is described formally in the following definition.

DEFINITION 2. *A concurrent object is move-ready if it is a move-candidate and has implemented the following changes:*

1. *The CAS at each linearization point in the insert and remove operations have been changed to scas.*
2. *The insert (and remove) operation(s) can abort if the scas returns ABORT.*
3. *(All memory locations that could be part of a scas are accessed via the read operation.)*

The changes required are mostly mechanical once the object has been found to adhere to the move-ready definition. This object can then be used by our move operation to move items between different instances of any concurrent move-ready objects. Requirement 3 is not required for systems with a hardware based DCAS.

Theorem 2 in Section 4 states that the move operation is linearizable and lock-free if used together with two move-ready lock-free concurrent data objects.

3.2.2 DCAS

The DCAS operation performs a CAS on two distinct words atomically (See Algorithm 1 for its semantics). It is unfortunately not commonly available in hardware, some say for good reasons [5], so for our experiments it had to be implemented in software. There are several different multi-word compare-and-swap methods available in the literature [18, 2, 15, 10, 1, 25, 26, 12] and ours uses the same basic idea as in the solution by Harris et al.

Lock-freedom is achieved by using a two-phase locking scheme with helping². First an attempt is made to change both the words involved, using a normal CAS, to point to a descriptor that holds all information required for another process to help the DCAS complete. See lines D10 and D14 in Algorithm 4. If any of the CASs fail, the DCAS is unsuccessful as both words need to match their old value. In this case, if one of the CASs succeeded, its corresponding word must be reverted back to its old value. When a word holds the descriptor it cannot be changed by any other non-helping process, so if both CASs are successful, the DCAS

²Lock-freedom does not exclude the use of locks, in contrast to its definition-name, if the locks can be revoked.

Algorithm 1 Semantics of the DCAS operation.

```

struct DCASDesc
word old1, old2, new1, new2
word *ptr1, *ptr2
[word *hp1, *hp2]
word res

dres DCAS(desc)
if (*desc.ptr1 ≠ desc.old1)
    return FIRSTFAILED
if (*desc.ptr2 ≠ desc.old2)
    return SECONDFAILED
*desc.ptr1 ← desc.new1
*desc.ptr2 ← desc.new2
return SUCCESS

```

as a whole is successful. The two words can now be changed one at a time to hold their respective new values. See lines D28 and D29.

If another process wants to access a word that is involved in a DCAS, it first needs to help the DCAS operation finish. The process knows that a word is used in a DCAS if it is pointing to a descriptor. This is checked at line D34 in the read operation. In our experiments we have marked the descriptor pointer by setting its least significant bit to one. This is a method introduced by Harris et al. [11] and it is possible to use since we assume that the word will contain a pointer and that pointers will be aligned to the word size of the system. Using the information in the descriptor it tries to perform the same steps as the initiator, but marks the pointer to the descriptor it tries to swap in with its thread id. This is done to avoid the ABA-problem, which can occur since CAS cannot distinguish a word that has been changed from A to B and then back to A again, from a word whose value has remained A. Unless taken care of in this manner, the ABA-problem could cause the DCAS to succeed multiple times, one for each helping process.

Our DCAS differs from the one by Harris et al. in that i) it has support for reporting which, if any, of the operations has failed, ii) it does not need to allocate an `RDCSSDescriptor` as it only changes two words, iii) it has support for hazard pointers, and iv) it requires two fewer CASs in the uncontended case. These are, however, minor differences and for our methodology to function it is not required to use our specific implementation. Performance gains and practicality reasons account for the introduction of the new DCAS. The DCAS is linearizable and lock-free according to Theorem 1.

4. PROOF

THEOREM 1. *The DCAS is lock-free and linearizable with possible linearization points at D10, D17, and D24, and follows the semantics as specified in Algorithm 1.*

Due to space constraints, we direct the reader to the technical report for the proof [4].

THEOREM 2. *The move operation is linearizable and lock-free if applied to two lock-free move-ready concurrent objects.*

PROOF. We consider DCAS an atomic operation as shown by Theorem 1. All writes, except the ones done by the DCAS operation, are process local and can as such be ignored.

Algorithm 2 Basic operations.

```
bool remove([key],*item)
...
while(unsuccesful)
...
result ← scas(ptr, old, new, element, [hp])
// Only needed when insert can fail
[if(result=ABORT)]
[abort]
[return false]
...
...

bool insert([key],item)
...
while(unsuccesful)
...
result ← scas(ptr, old, new,[hp])
if(result=ABORT)
abort
return false
...
...
```

The move operation starts with an invocation of the remove operation. If it fails, it means that there were no elements to remove from the object and that the linearization point must lie somewhere in the remove operation, since requirement 1 of the definition of a move-candidate states that the operations should be linearizable. If the process reaches the first `scas` call, the insert operation is invoked with the element to be removed as an argument. If the insert fails before it reached the second `scas` call, it was not possible to insert the element. In this case the `insfailed` variable is not set at line `M32` and `scas` will abort the remove operation. The linearization point in this case is somewhere in the insert operation. In both these scenarios, whether it is the remove or the insert operation that fails, the move operation as a whole is aborted.

If the process reached the second `scas` call, the one in the insert operation, the DCAS operation is invoked. If it is successful, then both the insert and remove operation must have succeeded according to requirement 3 of the definition of a move-candidate. By requirement 1, they can only succeed once, which makes the DCAS the linearization point. If the DCAS fails nothing is written to the shared memory and either the insert or both the remove and the insert operations are restarted.

Since the insert and remove operations are lock-free, the only reason for the DCAS to fail is that another process has made progress in their insertion or removal of an element. This makes the move operation as a whole lock-free. \square

5. CASE STUDY

To get a better understanding of how our methodology can be used in practice, we apply it to two commonly used concurrent objects, the lock-free queue by Michael and Scott [23] and the lock-free stack by Treiber [30]. The objects use hazard pointers for memory management and the selection of them is motivated in the paper by Michael [22].

5.1 Queue

The first task is to see if the queue is a move-candidate as defined by Definition 1:

1. The queue fulfills the first requirement by providing de-

Algorithm 3 Move operation.

```
thread local variables
desc, ltarget, lskey, ltkey, insfailed

M1 bool move(source, target, [skey, tkey])
M2 desc ← new DCASDesc
M3 desc.res ← UNDECIDED
M4 [lskey ← skey, ltkey ← tkey]
M5 ltarget ← target
M6 result ← source.remove([lskey], tmp)
M7 desc ← 0
M8 return result

M9 fbool scas(ptr, old, new, element, [hp])
M10 if(desc ≠ 0)
M11 desc.ptr1 ← ptr
M12 desc.old1 ← old
M13 desc.new1 ← new
M14 [desc.hp1 ← hp]
M15 insfailed ← true
M16 result ← ltarget.insert([ltkey], element)
M17 if(insfailed)
M18 return ABORT
M19 return result
M20 else
M21 return cas(ptr, old, new)

M22 fbool scas(ptr, old, new, [hp])
M23 if(desc ≠ 0)
M24 desc.ptr2 ← ptr
M25 desc.old2 ← old
M26 desc.new2 ← new
M27 [desc.hp2 ← hp]
M28 result ← DCAS(desc, true)
M29 if(result != SUCCESS)
M30 desc ← new DCASDesc(desc)
M31 desc.res ← UNDECIDED
M32 insfailed ← false
M33 if(result = FIRSTFAILED)
M34 return ABORT
M35 if(result = SECONDFAILED)
M36 return false
M37 return true
M38 else
M39 return cas(ptr, old, new)
```

queue and enqueue operations, which have been shown to be linearizable [23].

2. The insert and remove operations share hazard pointers in the original implementation. By using a separate set of hazard pointers for the dequeue operation we fulfill requirement number 2, as no other information is shared between two instances of the object.
3. The linearization points can be found on lines `Q34`, and `Q14` and both consist of a successful CAS, which is what requirement number 3 asks for. There is also a linearization point at line `Q29`, but it is not taken in the case of a successful dequeue. These linearization points were provided together with the algorithmic description of the object, which is usually the case for the concurrent linearizable objects that exist in the literature.
4. The linearization point for the dequeue is on line `Q34` and the value that is read in case of a successful CAS is available on line `Q33`, which must be executed before line `Q34`.

The above simple observations give us the following lemma in a straightforward way.

Algorithm 4 Double word compare-and-swap.

```
D1 desc DCAS(desc, initiator)
D2 [if(¬initiator)]
D3   [hp1 ← desc.hp1, hp2 ← desc.hp2]
D4 if(desc.res = SUCCESS ∨ SECONDFAILED)
D5   if(desc is marked)
D6     cas(desc.ptr2, desc, desc.old2)
D7   else
D8     cas(desc.ptr1, desc, desc.old1)
D9   return desc.res
D10 if(initiator ∧ ¬cas(desc.ptr1, desc.old1, desc))
D11   return FIRSTFAILED
D12
D13 mdesc ← mark(unmark(desc), threadID)
D14 p2set ← cas(desc.ptr2, desc.old2, mdesc)
D15 if(¬p2set)
D16   if(*desc.ptr2.ptr ≠ desc)
D17     cas(desc.res, UNDECIDED, SECONDFAILED)
D18     if(desc.res = SUCCESS)
D19       return desc.res
D20     if(desc.res = SECONDFAILED)
D21       cas(desc.ptr1, desc, desc.old1)
D22       return desc.res
D23
D24 cas(desc.res, UNDECIDED, mdesc)
D25 if(desc.res = SECONDFAILED)
D26   if(p2set) cas(desc.ptr2, mdesc, desc.old2)
D27   return desc.res
D28 cas(desc.ptr1, desc, desc.new1)
D29 cas(desc.ptr2, desc.res, desc.new2)
D30 desc.res ← SUCCESS
D31 return desc.res

D32 word read(*ptr)
D33 result ← *ptr
D34 while(result is DCASDesc)
D35   hpd ← result
D36   if(hpd = *ptr)
D37     DCAS(result, false)
D38   result ← *ptr
D39 return result
```

LEMMA 1. *The queue by Michael and Scott is a move-candidate.*

After making sure that the queue is a move-candidate we need to replace the CAS operations at the linearization points on lines Q₃₄ and Q₁₄ with calls to the `scas` operation. If we are using a software implementation of DCAS we also need to alter all lines where words are read that could be part of a DCAS, so that they access them via the read operation. For the queue these changes need to be done on lines Q₆, Q₇, Q₈, Q₁₀, Q₂₃, Q₂₄, Q₂₅, Q₂₆, and Q₂₈.

One must also handle the case of `scas` returning `ABORT` in the enqueue. Since there has been no change to the queue, the only thing to do before returning from the operation is to free up the allocated memory on line Q₁₆. The enqueue cannot fail so there is no need to handle the `ABORT` result value in the dequeue operation.

The move operation can now be used with the queue. In Section 6 we evaluate the performance of the move-ready queue when combined with another queue, and when combined with the Treiber stack.

5.2 Stack

Once again we first check to see if the stack fulfils the requirements of the move-candidate definition:

1. The push and pop operations are used to insert and remove elements and it has been shown that they are

Algorithm 5 Lock-free queue by Michael and Scott [23].

```
Q1 bool enqueue(val)
Q2 node ← new Node
Q3 node.next ← 0
Q4 node.val ← val
Q5 while(true)
Q6   ltail ← read(tail)
Q7   hp1 ← ltail; if(hp1 != read(tail)) continue
Q8   lnext ← read(ltail.next)
Q9   hp2 ← lnext
Q10  if(ltail != read(tail)) continue
Q11  if(lnext != 0)
Q12    cas(tail, ltail, lnext)
Q13    continue
Q14  res ← scas(ltail.next, 0, node, hp1)
Q15  if(res = abort)
Q16    free node
Q17  return false
Q18  if(res = true)
Q19    cas(tail, ltail, node)
Q20  return true

Q21 bool dequeue(*val)
Q22 while(true)
Q23   lhead ← read(head)
Q24   hp3 ← lhead; if(hp3 != read(head)) continue
Q25   ltail ← read(tail)
Q26   lnext ← read(lhead.next)
Q27   hp4 ← lnext
Q28   if(lhead != read(head)) continue
Q29   if(lnext == 0) return false
Q30   if(lhead == ltail)
Q31     cas(tail, ltail, lnext)
Q32     continue
Q33   *val ← lnext.val
Q34   if(scas(head, lhead, lnext, val, hp3))
Q35     free lhead
Q36   return true
```

linearizable. Vafeiadis has, for example, given a formal proof of this [33].

2. There is nothing shared between instances of the object, so the push and pop operations can succeed simultaneously.
3. The linearization points on lines s_7 and s_{22} are both CAS operations. The linearization point on line s_{17} is not a CAS, but it is only taken when the source stack is empty and when the move can not succeed. The conditions in the definition only require successful operations to be associated to a successful CAS.
4. The element to be removed is available on line s_{21} , which is before the linearization point on line s_{22} .

The above simple observations give us the following lemma in a straightforward way.

LEMMA 2. *The stack by Treiber is a move-candidate.*

To make the stack object move-ready we change the CAS operations on lines s_7 and s_{22} to point to `scas` instead. We also need to change the read of `top` on lines s_5 , s_{15} , and s_{19} , if we are using a software implementation of DCAS, so that it goes via the read operation. Since push can be aborted we also need to add a check after line s_7 that looks for this condition and frees allocated memory.

The stack is now move-ready and can be used to atomically move elements between instances of the stack and other move-ready objects, such as the previously described queue.

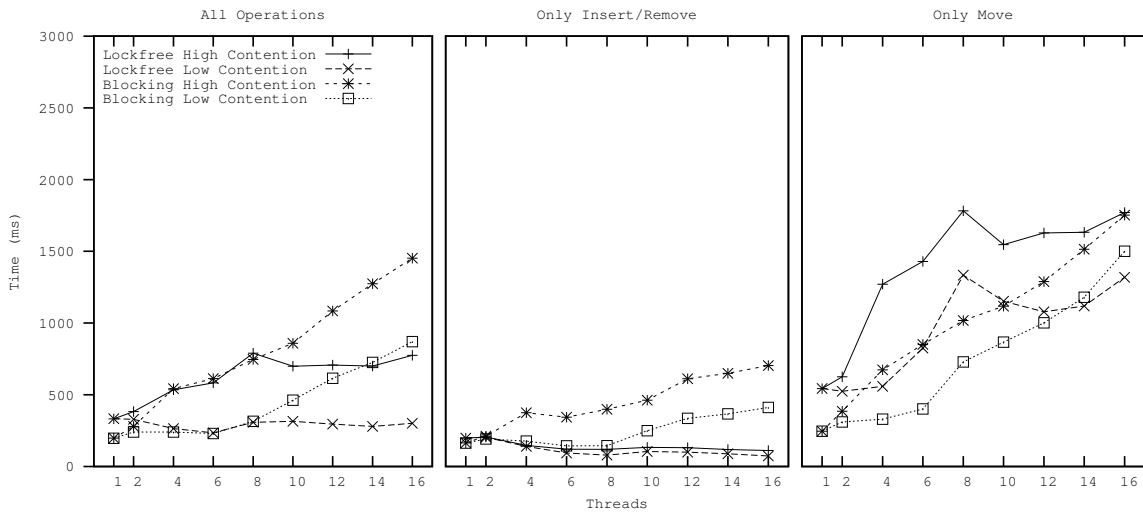


Figure 1: Results from the queue/stack evaluation without back-off.

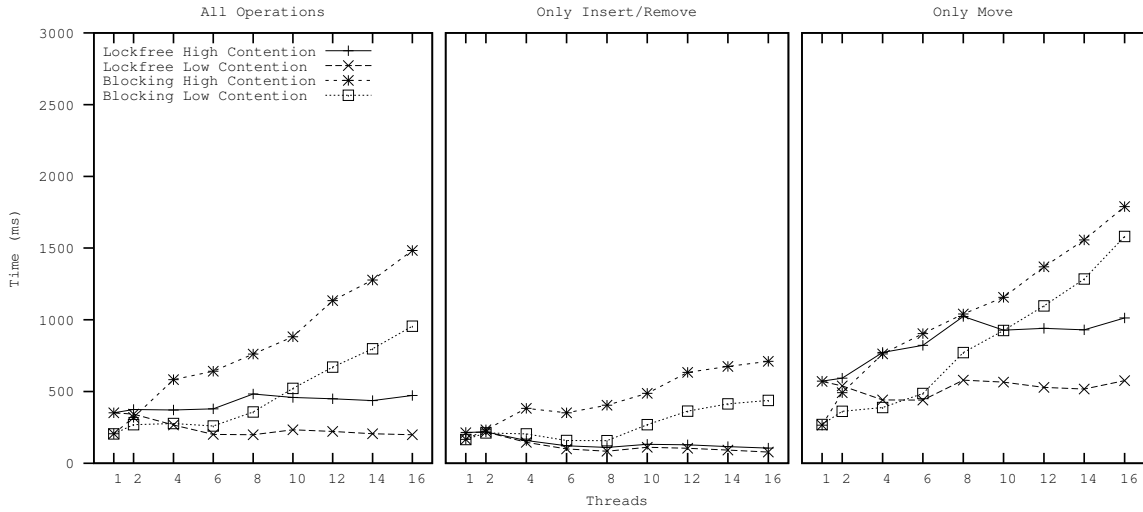


Figure 2: Results from the queue evaluation without back-off.

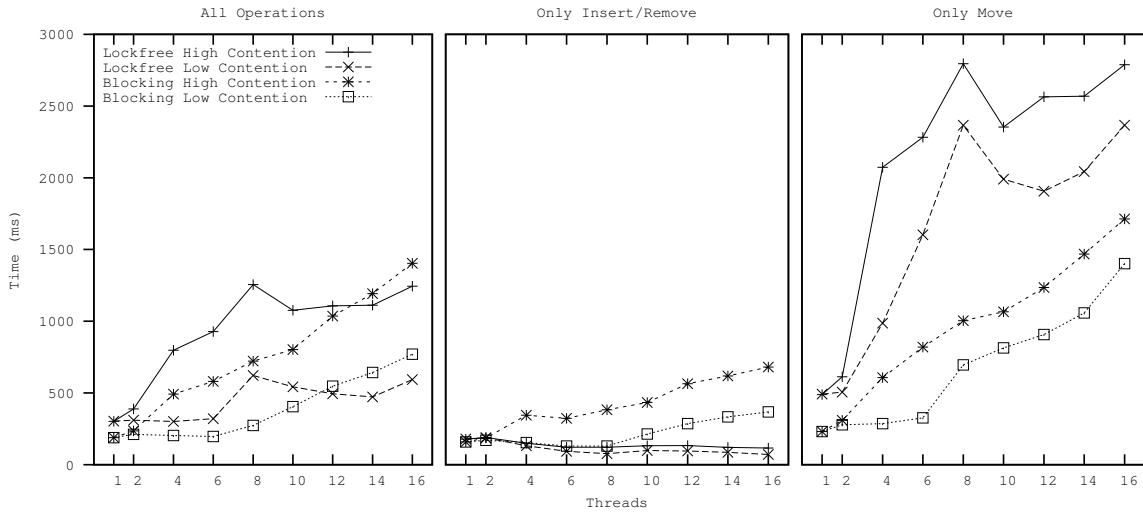


Figure 3: Results from the stack evaluation without back-off.

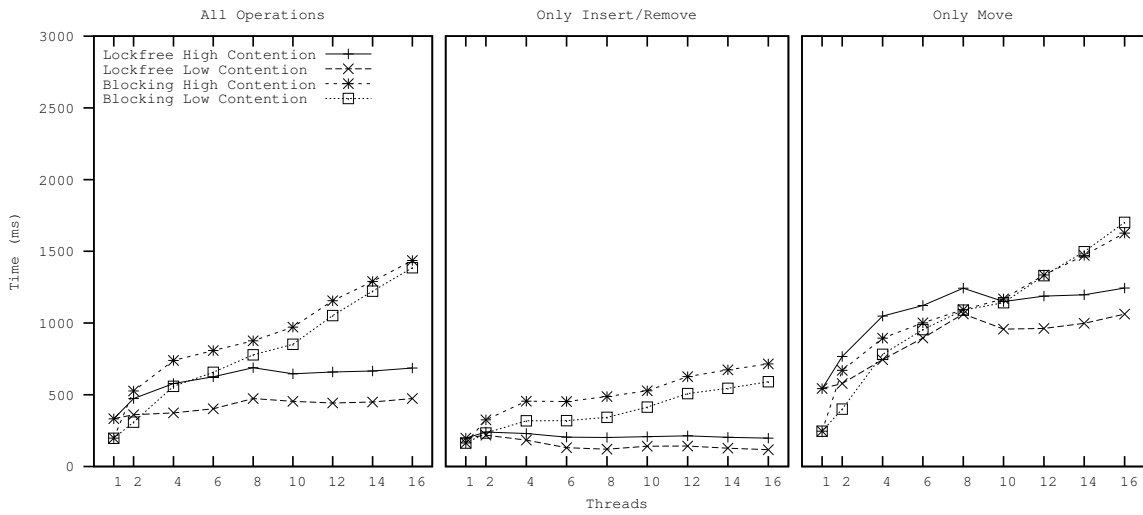


Figure 4: Results from the queue/stack evaluation with back-off.

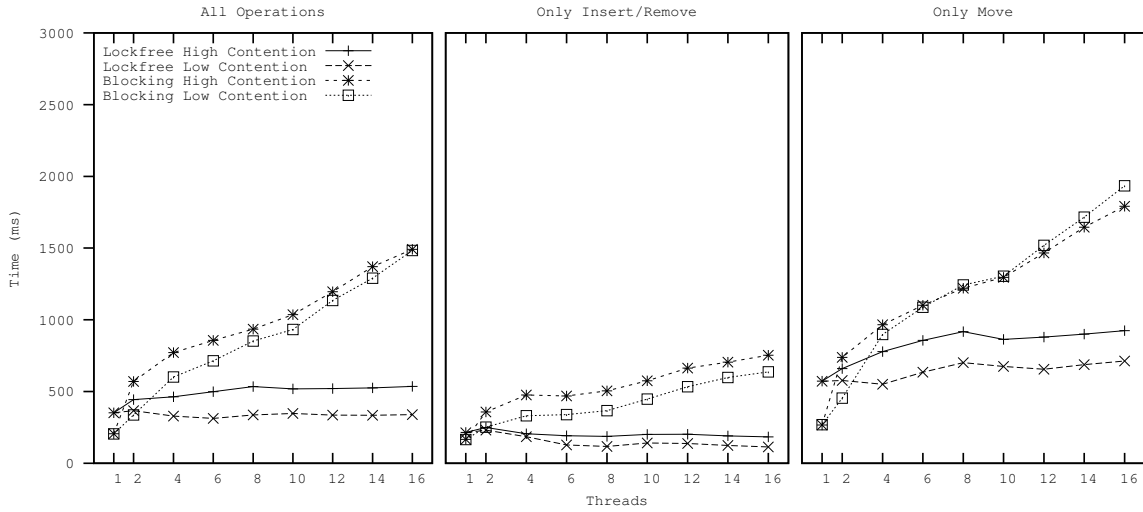


Figure 5: Results from the queue evaluation with back-off.

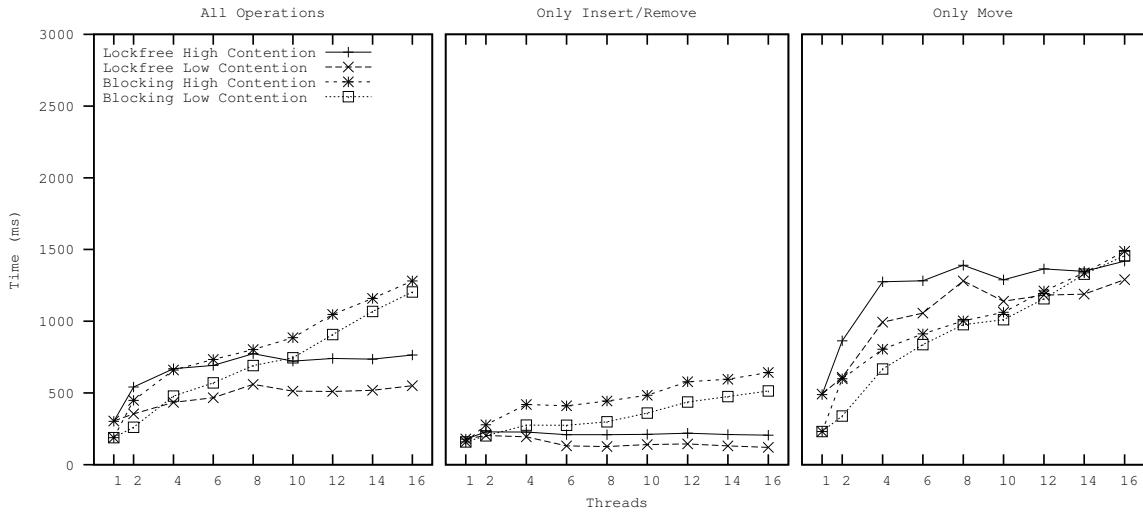


Figure 6: Results from the stack evaluation with back-off.

Algorithm 6 Lock-free stack by Treiber [30].

```
S1 bool push(val)
S2 node ← new Node
S3 node.val ← val
S4 while(true)
S5   ltop ← read(top)
S6   node.next ← ltop
S7   res ← scas(top, ltop, node)
S8   if(res = abort)
S9     free node
S10    return false
S11   if(res = true)
S12     return true

S13 bool pop(val)
S14 while(true)
S15   ltop ← read(top)
S16   if(ltop = 0)
S17     return false
S18   hp ← ltop
S19   if(read(top) != ltop)
S20     continue
S21   val ← ltop.val
S22   if(scas(top, ltop, ltop.next, val))
S23     free ltop
S24   return true
```

In the next section we evaluate the performance of the move-ready stack when combined with another stack as well as when combined with the Michael and Scott queue.

6. EXPERIMENTS

The evaluation was performed on a machine with an Intel Core i7 950 3 GHz processor and 6 GB DDR3-1333 memory. All experiments were based on either two queues, two stacks, or one queue and one stack. The stack used was the lock-free stack by Treiber and the queue was the lock-free queue by Michael and Scott [30, 23].

Each thread randomly performs operations from a set of either just move operations, or just insert/remove operations, or both move and insert/remove operations. A total of five million operations were distributed evenly to between one and sixteen threads and each trial was run fifty times.

For reference we compared the lock-free concurrent objects with blocking implementations of the same objects, using test-test-and-set to implement the locks. We did the experiments both with and without a backoff function. The backoff function was used to lower the contention so that every time a thread failed to acquire the lock or, in case of the lock-free objects, failed to insert or remove an element due to a conflict, the time it waited before trying again was doubled.

All implementations used the same lock-free memory manager. Freed nodes are placed on a local list with a capacity of 200 nodes. When the list is full it is placed on a global lock-free stack. A process that requires more nodes accesses the global stack to get a new list of free nodes. Hazard pointers were used to prevent nodes in use from being reclaimed.

Two load distributions were tested, one with high contention and one with low contention, where each thread did some local work for a variable amount of time after they had performed an operation on the object. The work time is picked from a normal distribution and the work takes around $0.1\mu s$ per operation on average for the high contention distribution and $0.5\mu s$ per operation on the low contention distribution.

The total time taken for all threads to finish their allotted operations with no backoff function, excluding the time it took to perform the local work, is shown in Figures 1, 2, 3, 4, 5 and 6. The local work time was subtracted from the result to emphasize the synchronization overhead.

7. DISCUSSION

The results for only the remove/insert operations show that the lock-free versions scale with the number of threads, while the blocking drops in performance when the contention rises. With only move operations, the result is similar for the queue, whereas for the lock-free stack, the result is worse.

The main difference introduced by adding a backoff is that the gap between the high contention and low contention scenario gets smaller, for both the blocking and the lock-free version. One can also see, that for the stack, the result is much better with the backoff than without, for both versions. However, it is typically hard to predict the contention level, which often varies during runtime, making it difficult to design an optimal backoff function that works well during both high and low contention.

The graphs with the result of all operations can be seen as an average between the two other graphs and shows that the lock-free data objects scale quite well. It should also be noted that it is not possible to combine a blocking move operation with non-blocking insert/remove operations.

8. CONCLUSION

We present a lock-free methodology for composing highly concurrent linearizable objects by unifying their linearization points. Our methodology introduces atomic move operations that can move elements between objects of different types, to a large class of already existing concurrent objects without having to make significant changes to them.

Our experimental results demonstrate that the methodology presented in the paper, applied to the classical lock-free implementations, offers better performance and scalability than a composition method based on locking. These results also demonstrate that it does not introduce noticeable performance penalties to the previously supported operations of the concurrent objects.

Our methodology can also be easily extended to support n operations on n distinct objects, for example to create functions that remove an item from one object and insert it into n others atomically.

9. REFERENCES

- [1] J. H. Anderson and M. Moir. Universal Constructions for Multi-Object Operations. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 184–193, 1995.
- [2] J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing Wait-Free Objects on Priority-Based Systems. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 229–238, 1997.
- [3] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, 2008.

- [4] D. Cederman and P. Tsigas. Supporting Lock-Free Composition of Concurrent Data Objects. *Computing Research Repository*, abs/0910.0366, 2009.
- [5] S. Doherty, D. L. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit, and G. L. Steele, Jr. DCAS is not a Silver Bullet for Nonblocking Algorithm Design. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 216–224, 2004.
- [6] S. Doherty, M. Herlihy, V. Luchangco, and M. Moir. Bringing Practical Lock-Free Synchronization to 64-bit Applications. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 31–39, 2004.
- [7] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, 2004.
- [8] A. Gidenstam, M. Papatrantaifilou, and P. Tsigas. Allocating Memory in a Lock-Free Manner. In *ESA '05: Proceedings of the 13th Annual European Symposium on Algorithms*, pages 329–342, 2005.
- [9] A. Gidenstam, M. Papatrantaifilou, and P. Tsigas. NBmalloc: Allocating Memory in a Lock-Free Manner. *Algorithmica*, 2009.
- [10] P. H. Ha and P. Tsigas. Reactive Multi-Word Synchronization for Multiprocessors. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 184–193, 2003.
- [11] T. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.
- [12] T. Harris, K. Fraser, and I. A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 265–279, 2002.
- [13] T. Harris, S. Marlow, S. Peyton-Jones, and M. P. Herlihy. Composable Memory Transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005.
- [14] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 300–314, 2001.
- [15] M. P. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.
- [16] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
- [17] Intel. Threading Building Blocks, 2009.
- [18] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160, 1994.
- [19] J. Larus and C. Kozyrakis. Transactional Memory. *Communications of the ACM*, 51(7):80–88, 2008.
- [20] D. Lea. The Java Concurrency Package (JSR-166), 2009.
- [21] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, 2002.
- [22] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.
- [23] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.
- [24] Microsoft. Parallel Computing Developer Center, 2009.
- [25] M. Moir. Transparent Support for Wait-Free Transactions. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.
- [26] N. Shavit and D. Touitou. Software Transactional Memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, 1995.
- [27] H. Sundell and P. Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Lecture Notes in Computer Science, 2002.
- [28] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1438–1445, 2004.
- [29] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.
- [30] R. K. Treiber. Systems programming: Coping with parallelism. In *Technical Report RJ 5118*, April 1986.
- [31] P. Tsigas and Y. Zhang. Evaluating the Performance of Non-Blocking Synchronization on Shared-Memory Multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 29(1):320–321, 2001.
- [32] P. Tsigas and Y. Zhang. Integrating Non-Blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies. In *WOSP '02: Proceedings of the 3rd international workshop on Software and performance*, pages 55–67, 2002.
- [33] V. Vafeiadis. Shape-Value Abstraction for Verifying Linearizability. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 335–348, 2009.
- [34] J. D. Valois. Lock-free linked lists using compare-and-swap. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222, 1995.