Supporting Lock-Free Composition of Concurrent Data Objects: Moving Data Between Containers

Daniel Cederman and Philippas Tsigas
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, priority inversion and convoying. They have also been shown to work well in practice. 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 a wide variety of 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 lock-free containers. This move operation allows data to be transferred from one container to another, in a lock-free way, without blocking any of the operations supported by the original container.

For a data object to be suitable for composition using our methodology it needs to fulfil a set of requirements. These requirement are however generic enough to be fulfilled by a large set of objects. To show this we have performed case studies on six commonly used lock-free objects (a stack, a queue, a skip-list, a deque, a doubly linked-list and a hash-table) to demonstrate the general applicability of the methodology. We also show that the operations originally supported by the data objects keep their performance behavior under our methodology.

Index Terms—lock-free, data structures, containers, composition

1 Introduction

Data objects, such as concurrent containers, can be implemented with different types of progress guarantees. *Lock-free* data objects offer several advantages over their blocking counterparts, such as being immune to deadlocks, priority inversion, and convoying. They have also been shown to work well in practice [1], [2], [3]. They have been included in Intel's Threading Building Blocks Framework [4], the NOBLE library [1], the PEPPHER framework [5], the Java concurrency package [6], and the parallel extensions to the Microsoft .NET Framework [7]. 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. Defining efficient atomic operations spanning multiple objects requires careful algorithmic design for every particular composition. Such an approach is challenging and time-intensive. 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. In order to avoid deadlocks, the programmer must know the way locks are handled internally in the implementation of the objects themselves. 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 [8], but have poor support for dealing with non-transactional code with side-effects [9], [10]. This requires the data objects to be rewritten to be handled completely inside the STM, which lowers performance compared to a pure non-blocking implementation.

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. No intermediate state should be visible to any other process. In the literature the term is also used for nesting, making one data object part of another. This 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, is an open problem in the area of lock-free data objects. There do however exist customized compositions of specific concurrent data objects. This includes the composition of lock-free flat-sets by Gidenstam et al. that constitute the foundation of a lock-free memory allocator [11], [12]. But thus far there are no generic solutions.

Using blocking locks to compose lock-free operations is not a viable solution. It would reduce the concurrency and remove the lock-freedom guarantees of the operations.

1.2 Contributions

The main contribution of our paper is a systematic methodology for defining *atomic move* operations that span lock-free containers of potentially distinct types. An atomic move operation allows elements to be moved from one container to another, atomically. We demonstrate the methodology on a large class of already existing concurrent objects without having to make significant changes to them and without impacting the efficiency of their existing primitives.

In our methodology we present a set of properties that can be used to identify suitable concurrent objects. We also 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 [13], [14], [15], [16], [17], [18], [19], [20].

Non-collection based data objects, such as for example counters, do not map readily to the required properties. While the same method for composing can be used in many instances, these data objects are not suitable for use with the main contribution of this paper, the generic move operation.

Our methodology is based on the idea of decomposing and then rearranging 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. Many commonly used concurrent data objects 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 six commonly used concurrent data objects, the lock-free queue by Michael and Scott [14], the lock-free stack by Treiber [13], the lock-free skip-list, doubly linked-list and deque by Sundell and Tsigas [17], [21], and the lock-free hash-table by Michael [19]. Experimental results on an Intel multiprocessor system show that the methodology presented in the paper offers better performance and scalability, in most cases, than a composition method based on locking. This is especially the case for data objects with disjoint memory accesses, such as the skip-list and hash-table. 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.

The methodology requires the use of a dual-word CAS (DCAS) or a multi-word CAS (MWCAS) operation, depending on the number of objects to compose. In this paper we provide a software implementation of a DCAS operation that works for pointers. It has been designed to support hazard pointers and to provide information on which of the two words that caused it to fail, in case of failure. We provide a correctness proof for it in Section 4. This DCAS operation is used in all our experiments.

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 [22].

Linearizability is a commonly used correctness criterion introduced by Herlihy and Wing [22]. 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 reordered 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:

Requirement 1) It implements linearizable operations for insertion and removal of a single element.

Requirement 2) Insert and remove operations invoked on different instances of the object can succeed simultaneously.

Requirement 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.

Requirement 4) The element to be removed is readable before the CAS operation associated with 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 [23], if not dealt with explicitly. With shared locks there is the risk of deadlocks. 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 are used to prevent reclamation of memory that is currently referenced by the process. They could be accidentally 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. However, the linearization point does not need to be at the actual CAS operation. See Section 5.4 for an example. The linearization points are usually provided together with the algorithmic description of each object. Implementations that use the LL/SC pair for synchronization can be translated to ones that use CAS by using the construction by Doherty et al. [24]. 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. Working only with pointers makes it easier to identify words that are taking part in a DCAS operation. The last part requires the linearization point of an operation to be associated with an execution step by the process that invoked the operation. This 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 fulfills 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 CAS operation, but there are data objects where the element is never returned by the remove operation. This makes it unsuitable for a move operation. The element could also be accessed after the CAS operation for efficiency reasons, in which case the algorithm can often be rewritten.

3.2 The Algorithm

The main part of the algorithm is the actual move operation, which is described in the following section.

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 are each associated with a CAS. By combining these CASs and performing them simultaneously, it would be possible to compose operations.

By definition a move-candidate operation has a linearization point that is associated with 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:

Step 1) 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 associated with 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.

Step 2) 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 associated with the potential linearization point are stored.

Step 3) The CASs that are associated with 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.

Step 4) The cleanup-phase for the insert operation is performed.

Step 5) 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 CASs associated with a possible linearization point with a call to the `scas` operation described in Algorithm 3. 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 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 is implemented in software with helping, hazard pointers might be needed to disallow reclaiming of the memory referred throughout the operation. 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 used 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. In some cases the insertion operation can fail for reasons other than conflicts with another operation. In those cases there is also a need for the remove operation to be able to handle the possibility of aborting.

If one uses a software DCAS, it might be required to alter all accesses to memory words that could take part in the DCAS. Accesses to these words must then go 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 associated with 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) If a software DCAS is used, 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, 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 [25], [26], [27], [28], [29], [30], [31] and ours uses the same basic idea as in the solution by Harris et al [31]. Our definition is not concerned with DCAS for arbitrary word contents, but assumes the words actually contain aligned pointers. This allow us to use the lower bits of the pointers for management purposes.

Lock-freedom is achieved by using a two-phase locking scheme with helping.[1] First an attempt is made to change both the words involved, using a normal CAS, to point to a descriptor. The descriptor 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. This means that if both CASs are successful, the DCAS 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, a method introduced by Harris [20]. Using the information in the descriptor it tries to perform the same steps as

1. Lock-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)
  atomic
    if(*desc.ptr1 ≠ desc.old1)
      return *desc.res ← FIRSTFAILED
    if(*desc.ptr2 ≠ desc.old2)
      return *desc.res ← SECONDFAILED
    *desc.ptr1 ← desc.new1
    *desc.ptr2 ← desc.new2
    return *desc.res ← SUCCESS
```

Algorithm 2 Basic operations.

```
bool remove([key],*item)
  ...
  while(unsuccessful)
    ...
    // element is the element to be (re)moved.
    // See Requirement 4.
    result ← scas(ptr, old, new, element, [hp])
    // Only needed when insert can fail
    [if(result=ABORT)]
      [abort]
      [return false]
    ...
  ...
```

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

the initiator. The only difference is that it marks the pointer to the descriptor it tries to swap in with its thread id. This is done to avoid the ABA-problem. A CAS operation 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. A latecoming process could still cause a pointer to temporarily point to the descriptor, even after the DCAS is completed. If another process reads the pointer at that time, the process needs to change the pointer back to its old value. This is done in lines D4 to D9. Line D3 is used to protect the words with hazard pointers. See Lemma 6 in Section 4.

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 Correctness Proofs

In this section we show that the *DCAS* operation is correct, linearizable and lock-free (Theorem 1). We do this by first showing that the initiating process, and any potential helping processes, all receive the same result value (Lemma 1 and 2). If the result value is `SUCCESS`, then there was a time when the two pointers both pointed to a DCAS descriptor (Lemma 3). They then both changed their value to the *new* values given as input to the DCAS (Lemma 4). If the result value is `FIRSTFAILED` or `SECONDFAILED`, then neither of the two pointers changed their value to the *new* values (Lemma 5). Lemma 6 shows that the DCAS operation is compatible with hazard pointers.

In Theorem 2 we show that the *move* operation is linearizable and lock-free if applied to two lock-free move-ready concurrent objects.

Lemma 1: The DCAS descriptor's `res` variable can only change from `UNDECIDED` to `SECONDFAILED` or from `UNDECIDED` to a marked descriptor and consequently to `SUCCESS`.

Proof: The `res` variable is set at lines D17, D24, and D30. On lines D17 and D24 the change is made using CAS, which assures that the variable can only change from `UNDECIDED` to `SECONDFAILED` or to a marked descriptor. Line D30 writes `SUCCESS` directly to `res`, but it can only be reached if `res` differs from `SECONDFAILED` at line D25, which means that it must hold a marked descriptor as set on line D24 or already hold `SUCCESS`. □

Lemma 2: The initiating and all helping processes will receive the same result value.

Proof: DCAS returns the result value at lines D9, D11, D19, D22, D27, and D31. Lines D22 and D27 are only executed if `res` is equal to `SECONDFAILED` and we know by Lemma 1 that the result value cannot change after that. Lines D31 and D19 can only be executed when `res` is `SUCCESS` and by the same Lemma the value can not change. Line D9 only returns when the result value is either `SUCCESS` or `SECONDFAILED` and as stated before these value cannot change. Line D11 returns `FIRSTFAILED` when the initiator process fails to announce the DCAS, which means that no other process will help the operation to finish. □

Lemma 3: If and only if the result value of the DCAS is `SUCCESS`, then $*\text{ptr}_1$ held a descriptor at the same time as $*\text{ptr}_2$ held a marked descriptor.

Proof: Line D28 can only be reached if the CASs

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)
```

Algorithm 4 Double word compare-and-swap.

```
D1 dres 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
```

at lines D10 and D14 were successful. The values of $*ptr_1$ and $*ptr_2$ are not changed back until lines D28 and D29, so just before the first process reaches line D28 $*ptr_1$ holds a descriptor and $*ptr_2$ holds a marked descriptor. □

Lemma 4: If and only if the result value of the DCAS is SUCCESS, then $*ptr_1$ has changed value from old_1 to the descriptor to new_1 and $*ptr_2$ has changed value from old_2 to a marked descriptor to new_2 once.

Proof: On line D10 $*ptr_1$ is set to the descriptor by the initiating process as otherwise the result value would be FIRSTFAILED. On line D14, $*ptr_2$ is set to a marked descriptor by any of the processes. By contradiction, if all processes failed to change the value of $*ptr_2$ on line D14, the result value would be set to SECONDFAILED on line D17. On line D24 the res variable is set to point to a marked descriptor. This change is a step on the path to the SUCCESS result value and thus must be taken. On line D28 $*ptr_1$ is changed to new_1 by one process. It can only succeed once as the descriptor is only written once by the initiating process. This is in contrast to $*ptr_2$ which can hold a marked descriptor multiple times due to the ABA-problem at line D14. When $*ptr_2$ is changed to new_2 it could be changed back to old_2 by a process outside of the DCAS. The CAS at line D14 has no way of detecting this. This is the reason why we are using a marked descriptor that is stored in the res variable using CAS, as this will allow only one process to change the value of $*ptr_2$ to new_2 on line D29. A process that manages to store its marked descriptor to $*ptr_2$, but was not the first to set the res variable, will have to change it back to its old value. □

Lemma 5: If and only if the result value of the DCAS is FIRSTFAILED or SECONDFAILED, then $*ptr_1$ was not changed to new_1 in the DCAS and $*ptr_2$ was not changed to new_2 in the DCAS due to either $*ptr_1 \neq old_1$ or $*ptr_2 \neq old_2$.

Proof: If the CAS at line D10 fails, nothing is written to $*ptr_1$ by any processes since the operation is not announced. The CAS at line D24 must

fail, since otherwise the result value would not be SECONDFAILED. This means the test at line D25 will succeed and the operation will return before line D29, which is the only place that $*ptr_2$ can be changed to new_2. □

Lemma 6: If the initiating process protects $*ptr_1$ and $*ptr_2$ with hazard pointers, they will not be written to by any helping process unless that process also protects them with hazard pointers.

Proof: If the initiating process protects the words, they will not be unprotected until that process returns. At this point the final result value must have been set. This means that if the test at D4 fails for a helping process, the words were protected when the process local hazard pointers were set at line D3. If the test did not fail, then the words are not guaranteed to be protected. But in that case the word that is written to at line D6 or D8 is the same word that was read in the read operation. That word must have been protected earlier by the process calling it, as otherwise it could potentially read from invalid space. Thus the words are either protected by the hazard pointers set at line D3 or by hazard pointers set before calling the read operation. □

Lemma 7: DCAS is lock-free.

Proof: The only loop in DCAS is part of the read operation that is repeated until the word read is no longer a DCAS descriptor. The word can be assigned the same descriptor, with different process id, for a maximum number of $p - 1$ times, where p is the number of processes in the system. This can happen when each helping process manages to write to $*ptr_2$ due to the ABA-problem mentioned earlier. This can only happen once for each process per descriptor as it will not get past the test on line D4 a second time.

So, a descriptor appearing on a word means that either a process has started a new DCAS operation or that a process has made an erroneous helping attempt. Either way, one process must have made progress for this to happen, which makes the DCAS lock-free. □

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.

Proof: Lemma 2 gives that all processes return the same result value. According to Lemmata 4 and 5, the result value can be seen as deciding the outcome of the DCAS. The result value is set at D17 and D24, which become possible linearization points. It is also set at D30, but that comes as a consequence of the CAS at line D24. The final candidate for linearization point happens when the CAS at line D10 fails. This happens before the operation is announced so we do not need to set the res variable.

Lemma 4 proves that when the DCAS is successful it has changed both $*ptr_1$ and $*ptr_2$ to an intermediate state from a state where they were equal to old_1 and old_2, respectively. Lemma 3 proves that they were in this intermediate state at the same time before they got their new values, according to Lemma 4 again. If the DCAS was unsuccessful then nothing is changed due to either $*ptr_1 \neq old_1$ or $*ptr_2 \neq old_2$. This is in accordance with the semantics specified in Algorithm 1.

Lemma 7 gives that DCAS is lock-free. □

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. All writes, except the ones done by the DCAS operation, are process local and can as such be ignored.

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. This is so 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. □

5 Case Study

To get a better understanding of how our methodology can be used in practice, we apply it to six commonly used concurrent objects. These are the lock-free queue by Michael and Scott [14], the lock-free stack by Treiber [13], the lock-free skip-list, doubly linked-list and deque by Sundell and Tsigas [17], [21], and the lock-free hash-table by Michael [19]. A technical report is available with more detailed code listings showing some of the algorithms before and after adaptation [32].

Algorithm 5 Lock-free queue by Michael and Scott [14].

```
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
```

5.1 Queue

The first case is the lock-free queue by Michael and Scott [14] that uses hazard pointers for memory management. We start by making sure that it is a move-candidate as defined by Definition 1. All line references are to Algorithm 5.

Requirement 1) The queue fulfills the first requirement by providing dequeue and enqueue operations, which have been shown to be linearizable [14].

Requirement 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.

Requirement 3) The linearization points can be found on lines Q34, and Q14 and both consist of a successful CAS, which fulfils requirement number 3. 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.

Requirement 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.

Lemma 8: 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. They should access them via the `read` operation instead. 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

The second case is the lock-free stack by Treiber [13]. We look at the version by Michael that uses hazard pointers for memory management [23]. We first check to see if the stack fulfils the requirements of the move-candidate definition. All line references are to Figure 8 in the paper by Michael [23].

Requirement 1) The push and pop operations are used to insert and remove elements and it has been shown that they are linearizable. Vafeiadis has, for example, given a formal proof of this [33].

Requirement 2) There is nothing shared between instances of the object, so the push and pop operations can succeed simultaneously.

Requirement 3) The linearization points on lines 5 and 11 are both CAS operations. The linearization point on line 7 is not a CAS, but it is only taken when the source stack is empty and when the move cannot succeed. The conditions in the definition only require successful operations to be associated to a successful CAS.

Requirement 4) The node holding the element to be removed is available on line 10, which is before the linearization point on line 11.

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

Lemma 9: The stack by Treiber is a move-candidate.

To make the stack object move-ready we change the CAS operations on lines 5 and 11 to point to `scas` instead. We also need to change the read of `top` on lines 3, 6, and 9, 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 5 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. In Section 6 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.

5.3 Skip-List

The third case is the lock-free skip-list by Sundell and Tsigas [17] that uses reference counting for memory management. We first check to see if the skip-list fulfils the requirements of the move-candidate definition. All line references are to the skip-list paper [17].

Requirement 1) The insert and remove operations have been proved to be linearizable [34].

Requirement 2) There is nothing shared between instances of the object, so the insert and remove operations can succeed simultaneously.

Requirement 3) The linearization point of a successful delete operations is the CAS at line D8. The linearization point for a successful insert operation is either the CAS at line I26, when inserting a new node, or the CAS at line I13, when updating the value of a node. The linearization points for unsuccessful operations can be ignored.

Requirement 4) The element to be removed is read at line D6, which is before the linearization point on line D8.

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

Lemma 10: The skip-list by Sundell and Tsigas is a move-candidate.

To make the skip-list object move-ready we change the CAS operations on lines I13, I26 and D8 to `scas` operations. If we are using a software DCAS we need to change all reads of a nodes value and all reads of the `next` pointer on the lowest level so that they are done via the `read` operation. If the insert operation should be aborted we need to free the allocated node and also to lower all reference counters to not interfere with the memory management.

The skip-list is now move-ready. In Section 6 we evaluate the performance of the move-ready skip-list.

5.4 Deque

The fourth case is the lock-free double ended queue by Sundell and Tsigas [21] that uses Beware&Cleanup (B&C), a combination of hazard pointers and reference counting, for memory management [35]. We first check to see if the deque fulfils the requirements of the move-candidate definition. All line references are to the deque paper [21].

Requirement 1) The pushLeft, pushRight, popLeft and popRight operations are linearizable according to Theorem 1 in the deque paper [21].

Requirement 2) The deque uses hazard pointers which normally needs to be handled. However, the B&C memory reclamation scheme dynamically allocates more hazard pointers if needed, which prevents them from being overwritten before being released. This allows an insert and remove operation on two different instances of the object to succeed concurrently.

Requirement 3) The successful insert and remove operations have their linearization points on line L7, R7, PR11 and PL3. The three first of these are CAS operations, but the fourth is a read operation. Fortunately, the read operation is only the linearization point when it is associated with a successful CAS operation on line PL13, which means that requirement 3 is fulfilled. The linearization points of unsuccessful operations can be ignored.

Requirement 4) The reference to the node that will be popped if the popLeft operation is successful, is taken at line PL3, which is before the CAS at PL13. For a successful popRight operation the reference to the node that will be popped is taken at either line PR2 or PR5, both which are before the CAS at PR11.

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

Lemma 11: The double ended queue by Sundell and Tsigas is a move-candidate.

To make the deque object move-ready we need to extend the B&C scheme so that it provides a compareAndSwapRef operation that uses `scas` instead CAS. All reads of the `next` pointer in each node also needs to be done via the `read` operation, if we are using a software DCAS. Since the two push operations can be aborted we also need to add a check after line L7 and R7 to check for this condition. If aborted, we need to free the newly allocated node and release all hazard pointers before returning.

The deque is now move-ready.

5.5 Doubly linked-list

The fifth case is the doubly linked-list with support for concurrent cursors that the previously mentioned deque builds upon [21]. The linked list is traversed and modified using cursors that support standard operations such as next, prev, delete and insertBefore and insertAfter. We first check to see if the doubly linked-list fulfils the requirements of the move-candidate definition. All line references are to the doubly linked-list paper [21].

Requirement 1) The delete, insertBefore and insertAfter operations are linearizable according to Theorem 1 in the double linked-list paper [21].

Requirement 2) It uses hazard pointers, but these are handled automatically by the B&C memory reclamation scheme. This allows insertion and removal operations on two different instances of the linked-list to succeed concurrently.

Requirement 3) The successful insert and remove operations have their linearization points on line D8, IB11 and IA8. These are all CAS operations. The linearization points of unsuccessful operations can be ignored.

Requirement 4) The cursor is pointing to the element to delete, which makes it trivially accessible before the CAS operation at D8.

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

Lemma 12: The doubly linked-list by Sundell and Tsigas is a move-candidate.

To make the doubly linked-list move-ready we need to, as with the deque, to extend the B&C scheme so that it provides a compareAndSwapRef operation that uses `scas` instead CAS. All reads of the `next` pointer in each node also needs to be done via the `read` operation, if we are using a software DCAS. Since the two insert operations can be aborted we also need to add a check after line IB11 and IA8 to check for this condition. If aborted, we need to free the newly allocated node and release all hazard pointers before returning.

The doubly linked-list is now move-ready.

5.6 Hash-table

The last case is the lock-free hash-table by Michael [19] that uses hazard pointers for memory management. We first check to see if the hash-table fulfils the requirements of the move-candidate definition. All line references are to Figure 7 in the hash-table paper [19].

Requirement 1) The insert and remove operations are both linearizable, as stated in Theorem 7 in the hash-table paper [19].

Requirement 2) The insert and remove operations both utilizes the same find operation that sets three hazard pointers. Due to this, the algorithm needs to be changed so that the find operation uses a disjoint set of hazard pointers depending on if it is called from the insert or remove operation.

Requirement 3) The possible linearization point for a successful operation is the CAS at line A3 for the insert operation and the CAS at line B2 for the delete operation. The linearization points for unsuccessful operations at line D1 and D3 can be ignored.

Requirement 4) The element to be removed is available after the call to the find operation at line B1, which is before the linearization point at line B2.

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

Lemma 13: The hash-table by Michael is a move-candidate.

The hash-table is made move-ready by changing the CAS operations on lines A3 and B2 to `scas` instead. If we use a software DCAS we need to change all reads of each nodes `next` pointer to go via the `read` operation instead. Since inserts can be aborted, we also need to add a check after line A3 that checks for this condition and returns false if it is the case that the insert has been aborted. The algorithm will then free the allocated memory.

The hash-table is now move-ready.

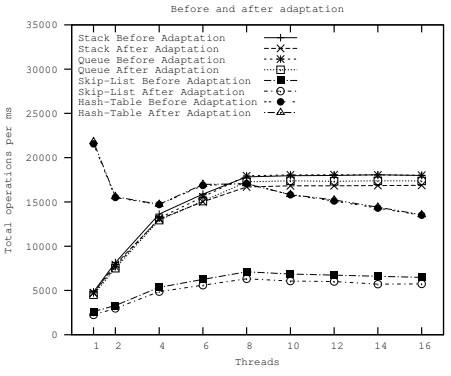

Fig. 1. Total number of operations per millisecond *before* and *after* adaptation of the data objects.

6 Experiments

The evaluation was performed on a machine with an Intel Core i7 950 3 GHz processor and 6 GB DDR3-1333 memory. The processor has four cores with hyper-threading, providing us with eight virtual processors in total. All experiments were based on either two queues, two stacks, two skip-lists, two hash-tables, or one queue and one stack. The stack used was the lock-free stack by Treiber [13], the queue was the lock-free queue by Michael and Scott [14], the skip-list used was the lock-free skip-list by Sundell and Tsigas [17], and the hash-table used was the lock-free hash-table by Michael [19].

In the experiments two types of threads were used, one that performed only insert/remove operations, and one that only performed move operations. For the hash-table and skip-list, the keys used in the operations were picked in such a way as to guarantee that the operation could succeed. The skip-list was 8 levels high and the hash-table had 1024 buckets. The number of threads, as well as the number of move-only threads, were varied between one and sixteen. We ran each experiment for five seconds and measured the number of operations performed in total per millisecond. Move operations were counted as two operations to normalize the result. Each experiment was run fifty times, taking the average as the results.

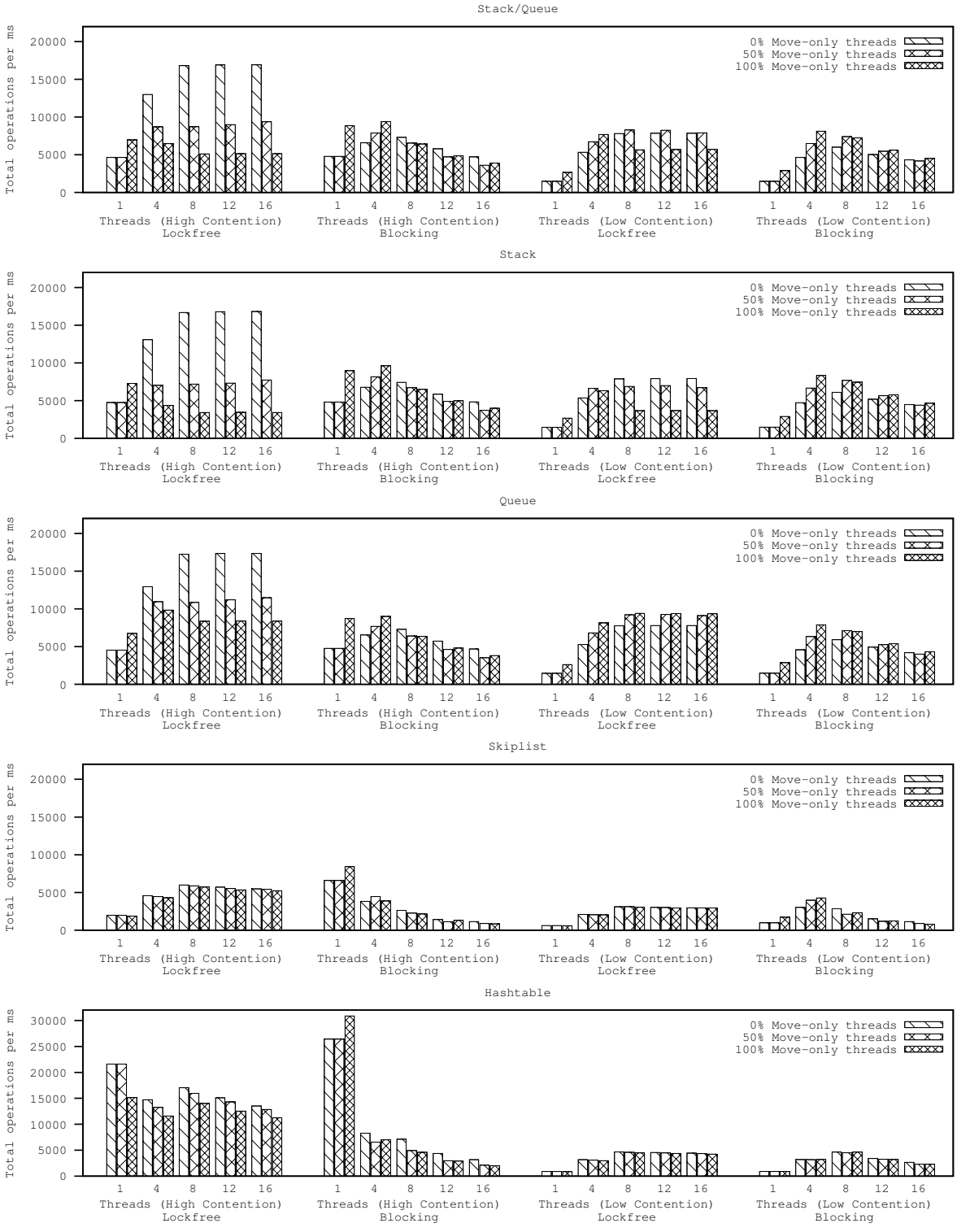

Fig. 2. Results from the stack/queue, stack/stack, queue/queue, skip-list/skip-list and hash-table/hash-table evaluation.

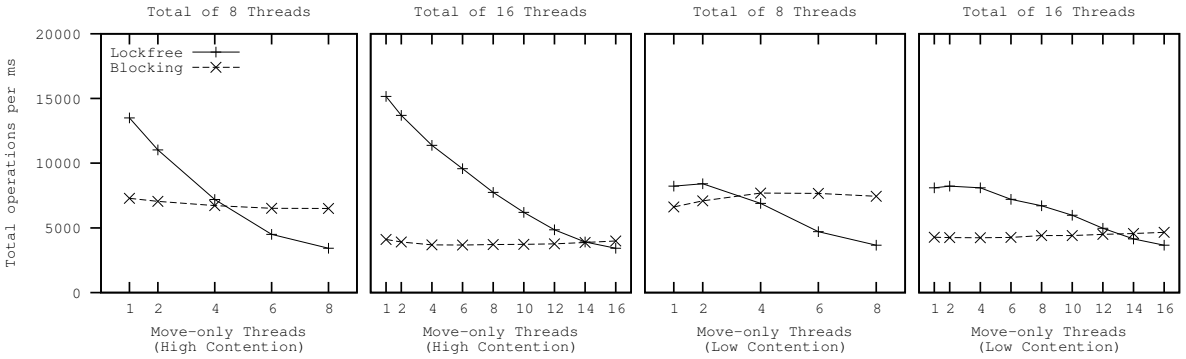

Fig. 3. Stack/Stack - Varying number of move-only threads.

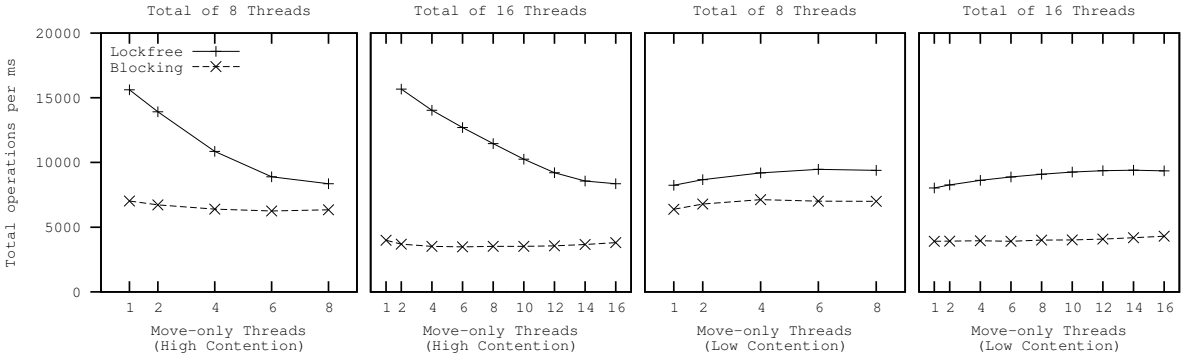

Fig. 4. Queue/Queue - Varying number of move-only threads.

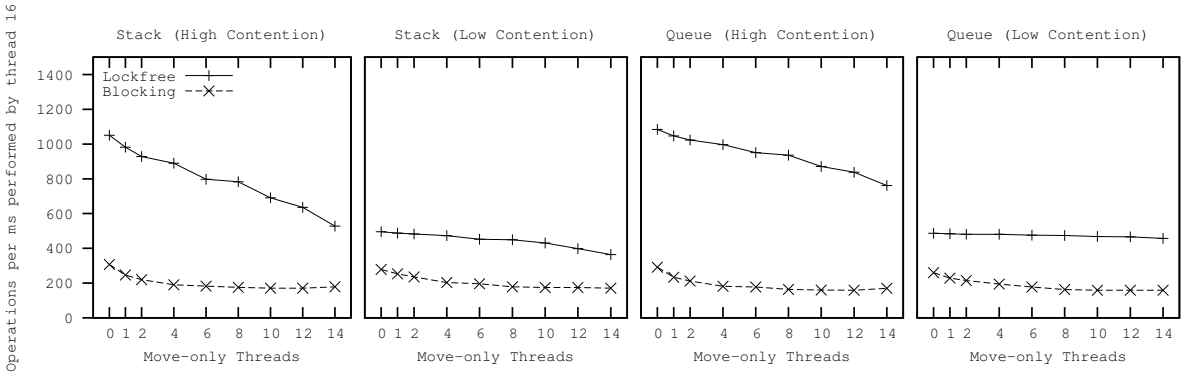

Fig. 5. Effect on thread performing insert/remove operations when other threads gradually turn into move-only threads.

For reference we compared the data objects with blocking implementations of the same objects, using test-test-and-set to implement the locks.

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 by the queue, stack and hash-table to prevent nodes in use from being reclaimed. The skip-list used reference counting.

Two load distributions were tested, one with high contention and one with low contention, where each process 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.

Figure 1 shows how the performance of the insert/remove operations was affected by the adaptation to the move operation. Figure 2 shows the number of operations in total per millisecond that was achieved performing either just insert/remove operations, or just move operations, or an even mix of the two. The experiments were performed using both high and low contention. In Figures 3 and 4 we take a closer look at what happens when the ratio of

processes performing only move operations is varied. We used a total of either eight or sixteen processes and ran the experiment under both low and heavy contention.

In Figure 5 we examine how the performance of a thread, that is only performing insert/remove operations, is affected when gradually more threads go from performing just insert/remove operations to performing just move operations.

7 Discussion

The stack and queue have very few access points, which limits the offered parallelism. The experiments on these two data objects are thus to be seen as showing some of the worst case scenarios. The skip-list and the hash-table on the other hand allows multiple operations on disjoint memory locations to succeed concurrently, and should thus scale better.

Figure 1 shows that the modifications done to the data objects do not degrade the performance of the normal operations. This is important, as it means that the adapted data object can also be efficiently used in stages where the move operation is not needed.

In Figure 2, when having no move-only threads, we see that the performance increase sharply up to four threads. This is the number of cores on the processor. It then increases more slowly up to eight threads, the number of cores times two for hyper-threading. After eight threads there is no increase in performance as there are no more processing units. After this point the blocking version drops in performance when more threads are added.

When more move operations are performed, the performance does not scale as well. The move operations are more expensive as they involve performing two operations and affects both data objects. This lowers the possible parallelism. This effect is hardly noticeable for the lock-free skip-list and the hash-table. Threads performing operations on these data objects have a high likelihood of accessing disjoint memory. This lowers the number of conflicts. For the queue the performance is still better than for the blocking version, however on the stack it is actually worse. All operations on this data object need to alter the same memory word. This leads to a high number of conflicts and retries. Looking at Figure 3 we see that there is a threshold were the ratio of move-only threads makes the lock-free version worse than the blocking. This threshold does not exist for the queue. Regardless of the ratio it is faster than the blocking version, though more moves lowers the performance for the high contention case. For the low contention case the performance remains the same. In this scenario there are fewer conflicts and less operations needs to be helped. In general the difference in performance between the blocking and lock-free methods becomes lower when the contention decreases. It should however be noted that it is not possible to combine a blocking move operation with non-blocking insert/remove operations.

In Figure 5 we see how the performance of a thread doing insert/remove operations is affected by the kind of operations that the other threads are performing. We can see that for the blocking implementations the performance stays roughly the same no matter the number of move operations. Once the lock has been acquired there is not much difference in the amount of work thats needs to be done. On the other hand the lock-free stack loses in performance as more move operations are performed. This is due to the higher probability of a thread needing help another thread finish its operation. The amount of helping required is lowered when the contention drops. For the queue, which can support two operations succeeding at the same time, the drop in performance due to helping is half that of the stack. In all cases the lock-free implementations are faster than their blocking counterparts.

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, often offers better performance and scalability than a composition method based on locking. This is especially the case for data objects with disjoint memory accesses, such as the skip-list and hash-table. These results also demonstrate that it does not introduce noticeable performance penalties to the previously supported operations of the concurrent objects.

By using a MWCAS instead of a DCAS, our methodology can 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.

8.1 Acknowledgments

This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481 and the Swedish Research Council under grant number 37252706.

References

[1] 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*, ser. Lecture Notes in Computer Science. Springer Verlag, 2002.

[2] P. Tsigas and Y. Zhang, "Evaluating the Performance of Non-Blocking Synchronization on Shared-Memory Multiprocessors," *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, no. 1, pp. 320–321, 2001.

[3] ——, "Integrating Non-Blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies," in *Proceedings of the 3rd international workshop on Software and performance*. New York, NY, USA: ACM, 2002, pp. 55–67.

[4] Intel, "Threading Building Blocks," 2009.

[5] S. Benkner, S. Pllana, J. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "Peppher: Efficient and productive usage of hybrid computing systems," *Micro, IEEE*, vol. 31, no. 5, pp. 28 –41, sept.-oct. 2011.

[6] D. Lea, "The Java Concurrency Package (JSR-166)," 2009.

[7] Microsoft, "Parallel Computing Developer Center," 2009.

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. P. Herlihy, "Composable Memory Transactions," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2005, pp. 48–60.

[9] 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*, vol. 6, no. 5, pp. 46–58, 2008.

[10] J. Larus and C. Kozyrakis, "Transactional Memory," *Communications of the ACM*, vol. 51, no. 7, pp. 80–88, 2008.

[11] A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "NBmalloc: Allocating Memory in a Lock-Free Manner," *Algorithmica*, 2009.

[12] ——, "Allocating Memory in a Lock-Free Manner," in *ESA '05: Proceedings of the 13th Annual European Symposium on Algorithms*, 2005, pp. 329–342.

[13] R. K. Treiber, "Systems programming: Coping with parallelism," in *Technical Report RJ 5118*, April 1986.

[14] 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*. ACM Press, 1996, pp. 267–275.

[15] H. Sundell and P. Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *Journal of Parallel and Distributed Computing*, vol. 65, no. 5, pp. 609–627, 2005.

[16] 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*. New York, NY, USA: ACM, 2004, pp. 50–59.

[17] H. Sundell and P. Tsigas, "Scalable and lock-free concurrent dictionaries," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2004, pp. 1438–1445.

[18] 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*. New York, NY, USA: ACM, 1995, pp. 214–222.

[19] 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*. New York, NY, USA: ACM, 2002, pp. 73–82.

[20] T. Harris, "A Pragmatic Implementation of Non-blocking Linked-Lists," in *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*. London, UK: Springer-Verlag, 2001, pp. 300–314.

[21] H. Sundell and P. Tsigas, "Lock-free deques and doubly linked lists," *J. Parallel Distrib. Comput.*, vol. 68, pp. 1008–1020, July 2008.

[22] M. P. Herlihy and J. M. Wing, "Linearizability: a Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[23] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[24] 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*. New York, NY, USA: ACM, 2004, pp. 31–39.

[25] 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*. New York, NY, USA: ACM, 1994, pp. 151–160.

[26] M. P. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 5, pp. 745–770, 1993.

[27] 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*. Los Alamitos, CA, USA: IEEE Computer Society, 2003, pp. 184–193.

[28] 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*. New York, NY, USA: ACM, 1995, pp. 184–193.

[29] M. Moir, "Transparent Support for Wait-Free Transactions," in *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1997, pp. 305–319.

[30] N. Shavit and D. Touitou, "Software Transactional Memory," in *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM, 1995, pp. 204–213.

[31] 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*. London, UK: Springer-Verlag, 2002, pp. 265–279.

[32] D. Cederman and P. Tsigas, "Adapting Lock-Free Concurrent Data Objects to Support a Generic Move Operation," Computer Science and Engineering, Chalmers University of Technology, Technical report 2012-11, 2012.

[33] V. Vafeiadis, "Shape-Value Abstraction for Verifying Linearizability," in *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 335–348.

[34] H. Sundell and P. Tsigas, "Scalable and lock-free concurrent dictionaries, extended version," Computing Science, Chalmers University of Technology, Technical report, December 2003.

[35] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas, "Efficient and reliable lock-free memory reclamation based on reference counting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, pp. 1173–1187, August 2009.

Daniel Cederman received his MSc and PhD degree in computer science at Chalmers University of Technology. Currently he is working as a PostDoc at the Distributed Computing and Systems group at Chalmers. He is involved in the European FP7 project PEPPHER on performance portability and programmability for heterogeneous many-core architectures. Research interests include non-blocking synchronization, load balancing, and algorithms and data structures for many-core processors.

Philippas Tsigas received the BSc degree in mathematics from the University of Patras, Greece, and the PhD degree in computer engineering and informatics from the same University, in 1994. Currently, he is working as a professor in the Department of Computer Science and Engineering at Chalmers University of Technology. From 1993 to 1994, he was with the National Research Institute for Mathematics and Computer Science in the Netherlands (CWI), Amsterdam. From 1995 to 1997, he was with the Max-Planck Institute for Computer Science, Saarbrucken, Germany. He joined Chalmers University of Technology in 1997. He is the cofounder and the head of the Distributed Computing and Systems research group at Chalmers. He is the initiator and one of the designers of NOBLE, a library of nonblocking data structures. His research interests include parallel and distributed computing, parallel and distributed systems, and information visualization.