Supporting Lock-Free Composition of Concurrent Data Objects

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

ABSTRACT

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.

1. INTRODUCTION

Lock-free data objects (i.e. lock-free concurrent data structures) 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 [10, 14, 15]. 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.

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 [4, 3], 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.

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.

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 [13, 9, 12, 2, 11, 16, 7, 6].

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

The results presented in this extended abstract appeared before in the proceedings of the 7th ACM international conference on Computing Frontiers [1].

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.

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 [8], 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. 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 a lock-free way 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 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 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.

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. 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 (where step four and five can be performed in any order):

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 detail in our earlier paper [1], 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 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 (or bookkeeping it for later use in subsequent invocations) 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.

The move operation is linearizable and lock-free if used together with two move-ready lock-free concurrent data objects [1].

4. 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. The experiment was performed using a move-ready version of the lock-free queue by Michael and Scott [9]. More experiments are available in our earlier paper [1].

The Intel Core i7 does not support a hardware DCAS, so we performed the experiments using a software DCAS based on the same idea as the one by Harris et al. [1, 5]. Lock-freedom is achieved by using a two-phase locking scheme with helping, so that a concurrent operation can help the DCAS operation finish. A full description is available in our earlier paper [1].

In the experiment two types of threads were used, one that performed only insert/remove operations, and one that only performed move operations. The number of threads, as well as the number of move-only threads, were varied between one and sixteen. We ran the 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.

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

5. DISCUSSION

In Figure 1, in the leftmost graph, we see that the performance increase sharply up to four threads, the number of cores on the processor, and 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 operation are performed, the performance does not scale as well, as can be seen in the two other graphs. The move operations are more expensive as they involve performing two operations and affects both data objects, which lowers the possible parallelism.

Regarding the backoff, we can see that with few move operations it hurts performance, whereas when the number of move operations increases it helps the performance. Unfortunately, it is typically hard to predict when this happens, making it difficult to design an optimal backoff function that works well in all scenarios.

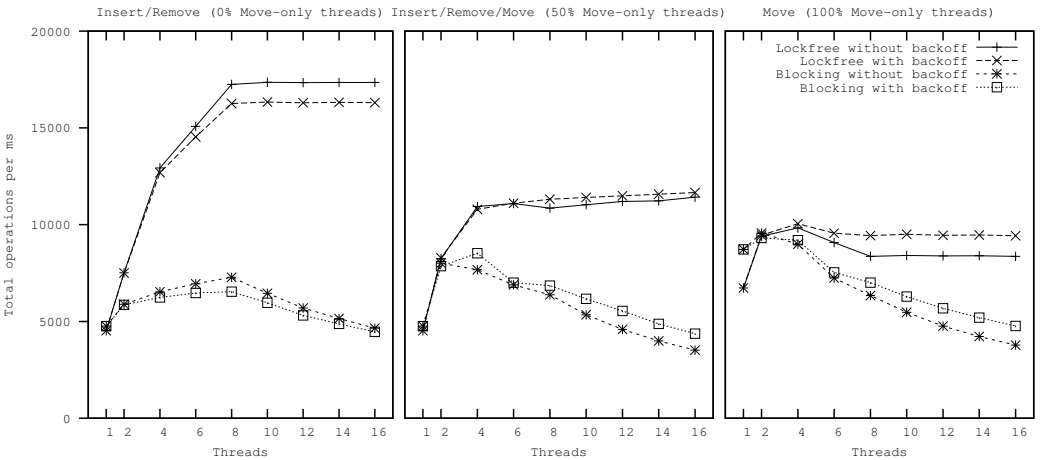

Figure 1: Results from the queue evaluation.

6. 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.

Acknowledgments

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

7. REFERENCES

[1] D. Cederman and P. Tsigas. Supporting Lock-Free Composition of Concurrent Data Objects. In *CF '10: Proceedings of the 7th ACM international conference on Computing Frontiers*, pages 53–62, 2010.

[2] 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.

[3] A. Gidenstam, M. Papatriantafilou, 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] M. M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[9] 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.

[10] H. Sundell and P. Tsigas. In *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.