Progress Guarantees When Composing Lock-Free Objects[*]

Nhan Nguyen Dang and Philippas Tsigas

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
{nhann,tsigas}@chalmers.se

Abstract. Highly concurrent and reliable data objects are vital for parallel programming. Lock-free shared data objects are highly concurrent and guarantee that at least one operation, from a set of concurrently executed operations, finishes after a finite number of steps regardless of the state of the other operations. Lock-free data objects provide progress guarantees on the object level. In this paper, we first examine the progress guarantees provided by lock-free shared data objects that have been constructed by composing other lock-free data objects. We observe that although lock-free data objects are composable when it comes to linearizability, when it comes to progress guarantees they are not. More specifically we show that when a lock-free data object is used as a component (is shared) by two or more lock-free data objects concurrently, these objects can no longer guarantee lock-free progress. This makes it impossible for programmers to directly compose lock-free data objects and guarantee lock-freedom. To help programmability in concurrent settings, this paper presents a new synchronization mechanism for composing lock-free data objects. The proposed synchronization mechanism provides an interface to be used when calling a lock-free object from other lock-free objects, and guarantees lock-free progress for every object constructed. An experimental evaluation of the performance cost that the new mechanism introduces, as expected, for providing progress guarantees is also presented.

1 Introduction

A concurrent data object is lock-free if it guarantees that at least one, among all concurrent operations, finishes after a finite number of steps. Lock-free data objects are immune to deadlocks and livelocks, and typically provide high scalability and performance [12] [11] [20] [22], especially in shared memory multiprocessor architectures. Several lock-free implementations of fundamental data structures have been introduced in the literature, such as queues [15] [21] [9], priority queues [18], linked-lists [23] [19] [18] [10], and hashtables [7] [17] [4]. Moreover, the problem of composing lock-free data objects has been considered recently in an effort to support the use of lock-free objects in the context of complex software development. Composite data

[*] 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 37252706.

structures, which are built by nesting multiple basic data structures, were first studied by Cohen and Campell [5]. Recently, Gidenstam et al. [8] and Cederman and Tsigas [3] studied the problem of composing two operations from two different lock-free objects into one compound atomic operation. These results made it possible to perform complex atomic operations such as *moves* that could move an item from one lock-free data object to another lock-free data object in a lock-free way.

Petrank and Steensgaard [16] also studied the problem of composing lock-free programs and services. They provided new formal definitions of lock-freedom, the bounded and unbounded lock-freedom and they extended them to programs and services. These new definitions allowed the authors to formally state and prove the composition theorem. The theorem guarantees lock-free progress for a lock-free program when composing with a service supporting lock-freedom, using the new definitions. This contribution is a step towards formally studying lock-freedom. However, the paper did not consider the case when multiple programs share a service and compete with each other to use it. This way of composing programs and services can affect their progress guarantees.

In this work, we address the lock-free composition problem but from the perspective of object-oriented programming and we do not consider changing the definition of lock-freedom in order to guarantee composition. In object-oriented programs, one lock-free object can be concurrently shared by other lock-free objects. In this setting, composition of several lock-free objects in one object is possible. When examining progress guarantees provided by these objects, we found that they can not provide the lock-free progress guarantee offered by the shared objects that compose them. To help solve this problem, a synchronization mechanism is proposed for a lock-freedom progress guarantee. By applying this mechanism when composing lock-free objects, we can compose as many objects as possible without fear of losing lock-freedom of the individual participants.

The rest of this paper is organized as follows. Section 2 examines the progress guarantees for lock-free objects in a composition. Then, the new synchronization mechanism for composing lock-free objects is proposed in section 3. Section 4 presents a set of experiments to evaluate our synchronization mechanism in practice. A conclusion of our work and discussions about future improvements come last in the section 5.

2 Progress Guarantee When Composing Lock-Free Data Objects

This section examines progress guarantees by lock-free objects used in an object-oriented program. The program can also contain blocking objects. However, since we are considering composing lock-free objects, blocking objects can be taken away without degradation of generality. In the remainder of this paper, all objects mentioned are lock-free.

2.1 Lock-Free Data Objects

Lock-free objects are objects that provide lock-free progress guarantee for their operation executions. The guarantee ensures that some among its concurrent operations succeed after a finite number of steps of their own execution. To provide such a guarantee, lock-free objects usually use non-blocking synchronization primitives to synchronize concurrent accesses to shared memory among the concurrent operations. Two

Algorithm 1. A template of a lock-free object

```
1 class LF
2   word *ptr
3   public op(args)
4    while (1)
5      oldVal ← *ptr
6      newVal ← calculate(args)
7      if (CAS(ptr, oldVal, newVal))
8        return
```

Algorithm 2. Operation Descriptor

```
9  struct OpDesc
10   void *oper(void *args)
11   void *args
12   bool done
13   Object src


16
```

synchronization primitives that are commonly used are Compare-And-Swap (*CAS*), Load-Link/Store-Conditional (*LL/SC*). *CAS* [12] takes three arguments: an address, an expected value, and an update value. If the value at the address is equal to the expected value, it is replaced by the update value; otherwise the value is left unchanged. *LL/SC* is a pair of instructions. The *LL* instruction reads from an address. A later *SC* instruction attempts to store a new value at the address. The instruction succeeds if content of the address are unchanged since that thread issued the earlier *LL* instruction to it. The instruction fails if the content has changed in the interval. These instructions are equally powerful since they both have an infinitive consensus number [12].

By observing several lock-free implementation of fundamental data structures such as queues [15] [21], linked-lists [23], and memory allocators [14], we found a common template that most of these implementations followed presented in Algorithm 1. The template object *LF* offers one operation *op*, which takes generalized arguments *args*. This operation computes a *newVal* (line 6) and updates it to *ptr* variable. In a multi-threaded environment, several threads can try to update *ptr* concurrently. Therefore, the *CAS* primitive is used to keep each update atomic. Examples of an *LF* object and an operation *op* that it supports are a lock-free $Queue$ [15] and its *enqueue* operation, respectively. The *enqueue* operation creates a new node containing the new value and inserts it to the $head$ of the queue (by a *CAS*) to become the new $head$ node.

2.2 Examining Lock-Free Progress Guarantee in Object-Oriented Program

An object-oriented program comprised by three lock-free objects is examined as an example. Among the objects, one, O_{21}, is concurrently shared by the other objects: O_{11} and O_{12}. All are assumed to be implemented by using the above template.

During the executions of O_{11} and O_{12}'s operations, they invoke operations in O_{21} and wait for the returned results. Object O_{21} is lock-free and therefore, always has some executed operations, invoked by O_{11} or O_{12}, finish and return after a finite number of executed steps. But, O_{21} provides no mechanism to ensure fairness among the executions invoked by different objects. As a result, that only executed operations called by one object (e.g O_{11}) succeed while those called by the other object fail to succeed is possible. Consequently, the former object progresses while the latter does not and fails to provide lock-freedom. So, composition causes a lock-free conflict point at O_{21} for O_{11} and O_{12}. When it is the case, lock-freedom of objects that conflict can be violated.

This lock-free conflict concept can be generalized. There can be several objects sharing another object. An object sharing another object can also be shared by other objects and become itself a conflict point. This sharing scenario creates a hierarchy of sharing lock-free objects together with the respective hierarchy of lock-free conflicts.

Our objective is to introduce a new synchronization mechanism enhancing the shared object so that it supports the lock-free property of the sharing objects.

3 A Synchronization Mechanism for Composing Lock-Free Objects

3.1 Our Approach

A new synchronization mechanism for sharing lock-free objects is proposed. Application of this mechanism enhances objects with the capability to maintain fairness among all the objects that invoke its operations. This fairness ensures that any invoking object has at least one operation returned after a finite number of steps. In other words, no object starves because of performing operations at the shared object.

In detail, the proposed synchronization mechanism keeps track of all invocations by sharing objects to the shared object's operations. When those by an object are unsuccessful to execute the instruction(s) at the linearization point many times, the mechanism will announce one of the operations. When such an announcement is made, later invocations help finish the announced operation before performing their expected operations. Completion of the announced operation allows the sharing object to progress.

The description of the proposed synchronization mechanism are introduced in the two next subsections. A correctness proof for the mechanism is also presented.

3.2 The Operation Descriptor

The new synchronization mechanism is introduced so that an unfinished operation can be helped to finish. The operation can be executed by more than one thread but the mechanism guarantees that only at most one execution can successfully complete. To make this helping scheme possible, a description of the operation and its execution status is needed. Any thread can read the description and execute the operation it describes.

The data structure *OpDesc* illustrated in Algorithm 2 is such an operation descriptor. *OpDesc* contains a function pointer **oper* to the operation, along with arguments for the operation; a boolean variable *done* records the status of the operation (finished or unfinished); *src* is a unique identity of the object that invokes this operation.

An $OpDesc$ object encapsulates an operation (e.g $enqueue$ operation) provided by shared lock-free object. The mechanism introduces a special kind of operation which can help executing other operations. In other words, operations that can read $OpDesc$ and execute the operation it described. We call them "super-operations". The term "operation", from this point, refer to an operation representing functionality that other objects want to perform at the shared object, which is described as an $OpDesc$ object.

3.3 The Synchronization Mechanism

The implementation of our synchronization mechanism for the lock-free object *LF* is presented in Algorithm 3. The new object *CLF* provides the same interface as that *LF* does to other objects. However each method in the interface is associated with a super-operation instead of an operation.

Any operation *op* in LF is re-written into a pair of one public method *op* (a super-operation) and one private one *op_m* (an operation). The operation *CLF.op_m* executes steps to make changes to the *CLF* object similar to that *LF.op* does to the *LF* object. The difference between *CLF.op_m* and *LF.op* is additional steps required by the

Algorithm 3. A lock-free object employing the proposed synchronization mechanism

```
class CLF
 word *ptr
 OpDesc hlps[M], EMPTY;      //EMPTY.done=true

 public op(src, args)
  OpDesc me(src, &op_m, (void*)args), hlp

  for(int i ← 0; i < M; i++) {
    hlp ← hlps[i];
    hp_x ← hlp;              //protect hlp with hazard pointer
    if (hlp != hlps[i]) continue;
    if (!hlp.done) *hlp.oper(me, hlp)

  if (¬me.done) op_m(me, me)

 private op_m(OpDesc me, OpDesc hlp)
  while (¬hlp.done)
    for (tries=0; tries<$T_{MAX}$ ∧ ¬hlp.done; tries++)
      oldVal ← *ptr
      newVal ← calculate(hlp.args)
      tmp ← hlps[hlp.src]
      if (DCAS(ptr, oldVal, newVal, &hlp.done, false, true))
        counter[hlp.src] ← 0;
        CAS(hlps[hlp.src], tmp, EMPTY);
        break;

    if (¬hlp.done)
      if (++counter[me.src] ≥ $O_{MAX}$)
        announce(me)

 void announce(OpDesc me)
   curr ← hlps[me.src]
   if(curr.done)
     CAS(hlps[me.src], curr, me)
```

synchronization mechanism that will be discussed later. *CLF.op*, is to provide the same interface as that *LF* but the content is totally new. When *CLF.op* is invoked, it is expected to perform modifications on *CLF* similar to functionality of operation $LF.op$. The functionality is now implemented in $CLF.op\_m$. In addition, *CLF.op* can help finish other *CLF.op_m* operations that other objects want to perform.

When *CLF.op* is invoked (assuming by object O_i) to perform the operation $CLF.op\_m$, it does not perform the operation immediately. Instead, it first creates an $OpDesc$ describing the operation (line 22) which it can perform by itself (line 30) or any thread can help finishing the operation. Then it checks if there are operations of any object needing help to finish (line 24). If there are such operations, the super-operation will execute these operations (line 28). The checking for any object that needs help is performed through a newly introduced array *hlps[]*. When one among the objects needs help, one of the concurrent operations the object performs will be placed in *hlps[]* at a dedicated position for the object. Other concurrent super-operation executions then can help to finish that one. We assume that there are M objects sharing *CLF* object. Therefore, $hlps[]$ can have M elements that one is assigned to an object.

The operation *CLF.op_m* introduces two main changes compared to *LF.op*. The first change is that a Double-Compare-And-Swap (*DCAS*) is used instead of a *CAS* in *LF.op* (line 7). *DCAS* atomically compares and exchanges values at two separate memory locations. Lock-free implementations of *DCAS* have been introduced in [6] and [3]. In $CLF.op\_m$, the *DCAS* performs modification of **ptr* and a status variable atomically. The former is similar to *CAS* in *LF.op*. The latter is to set the execution status variable of $OpDesc$. This status variable, which is allowed to be changed only once, makes sure that an $OpDesc$ only succeeds once even when multiple threads are executing it.

The second change in *CLF.op_m* is the introduction of a counter array $counter[]$ to record the numbers of times invocations by sharing objects try (but fail) to commit the changes to the shared object *CLF*. The counter at position i is increased after a failed *DCAS* execution (line 38) in an operation invoked by object O_i. When this number reaches a threshold, an executed operation invoked by O_i will be announced in *hlps[]* to be helped.

Due to this change, the loop inside this operation is also modified. Our algorithm could have followed the idea of increasing the counter after every failed *DCAS*. In this case, the counter at any position would be shared among several threads and need synchronization for every update which decreases the performance. To avoid this high overhead, in our design, this counter was split into two counters. One local counter *tries* for each operation execution and a shared one ($counter[]$) to record number of tries the executions invoked by the object have made. When $tries$ reach a threshold T_{MAX}, an update to *counter[me.src]* is made. And if this counter reaches its threshold O_{MAX}, one of the operation executions whose *src* is the same as *me.src* is announced.

In addition to those changes, a *CAS* is added to remove the reference from the announcement array $hlps[]$ to a successful operation hlp. This avoids any unsafe reference to hlp in the future when its hazard-pointer protection (line 26) is removed. The memory used by hlp can safely be reclaimed later by a memory reclamation scheme.

In short, the synchronization mechanism guarantees that new invocations of *CLF*'s operations helps finish on-going executed operations that need help. Then they executes the operation they are supposed to perform. With this mechanism, objects invoking operations of *CLF* always has one of the invocations finish after a finite number of steps. Therefore, these objects make progress.

3.4 ABA Problem

Similar to other lock-free objects, our mechanism also encounters the ABA problem. The ABA problem happens when the content at an address changes from A to B, and then changes back to A. *CAS* cannot distinguish this case and the case where the content is unchanged. A number of methods have been introduced to tackle with ABA problem such as tagging [1], hazard pointers [13]. In addition, memory words used by lock-free objects must be protected from deletion by concurrent threads when they are in use and reclaimed when they are no more used. Safe Memory Reclamation with hazard pointers introduced in [13] is used for these purposes.

3.5 Linearizability

This section states the lemmas for the linearizability and lock-freedom property of *CLF*. Due to the space limitation, the proofs for these lemmas are not included in this version of the paper.

Lemma 1. *Regardless of the number of threads executing an operation op_m with the same value of hlp argument, only one can succeed.*

Lemma 2. *CLF is linearizable with the linearization point at line 38.*

Lemma 3. *The presented object CLF is lock-free.*

3.6 How Does the Proposed Synchronization Mechanism Resolve Lock-Free Conflicts?

When a lock-free object is concurrently used by other lock-free objects $O_1 \dots O_M$, it can become a lock-free conflict and block the progress of those objects. This section will prove that when there is such a conflict point at *CLF*, our mechanism can resolve the conflict. Therefore, *CLF* does not block lock-free progress of the objects using it.

A scenario of using *CLF* is a program containing M lock-free objects $O_1 \dots O_M$ and one *CLF* object. An object O_i can have at most n concurrent invocations (executed by n threads) to *CLF.op* to perform an intended *CLF.op_m* (referred to as *me*). Each invocation creates an execution of operation *CLF.op*. We seek a bound of the maximum number of steps (a step is one execution of *DCAS*) performed by these executions between any two successful operations. If this bound is finite, it guarantees that any object that uses *CLF* progresses. The lemmas and theorem below figure out this bound.

Lemma 4. *An object O_i can make at most n concurrent invocations to super-operation CLF.op. Starting from when the last invocation returns (or when the program starts, if there is no such invocation), if any of these invocations has executed:*

$$U\_BOUND = T_{MAX}.O_{MAX} \tag{1}$$

steps, one of the following condition must hold:

- *at least one invocation finished. Or*
- *one of these concurrent CLF.op_m operations has been announced.*

Lemma 5. *When an operation me is announced in $hlps$, either me or another operation that has the same src as $me.src$ finishes after it has executed at most*

$$HELP\_BOUND = n(M-1)+1$$

steps since when the announcement is made.

Theorem 1. *When CLF is shared by several objects by invoking to CLF's super-operation op, there is always one, among all invocations by one object, finishing after executing a finite number of steps.*

Proof. From lemma 4, there must be one among the invocations from *O* which finishes before any of them has executed U_BOUND steps. Otherwise, one of the invocations has its operation me announced.

If me is announced, lemma 5 stated that one of the operations whose src is the same as $me.src$ (including me) finishes after it has executed at most $HELP\_BOUND$ steps since the announcement is made. Therefore, one of the invocations from one object returns after executing at most:

$$U\_BOUND + HELP\_BOUND = n(M-1) + T_{MAX}.O_{MAX} + 1$$

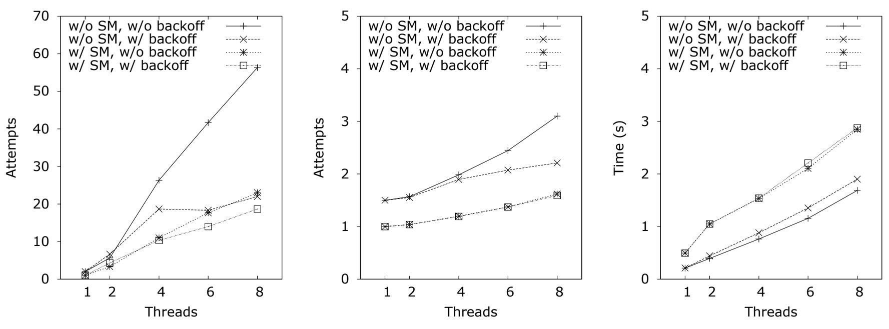
steps; where:

- T_{MAX} is the number of steps executed by an operation before it checks if it should announce itself.
- O_{MAX} is the number of times T_{MAX} was reached by all invocations from one object.
- n is the maximum number of concurrent operations of *CLF* that can be executed.
- M is the number of objects that are sharing *CLF*.

4 Experimental Evaluation

For our experimental evaluation we considered the composition scenario where a program containing a number of pseudo objects sharing one queue. The queue is an implementation of the Michael-Scott Queue [15] enhanced with the proposed synchronization mechanism. A set of experiments to evaluate the effectiveness and performance cost of our synchronization mechanism was performed and the results are presented.

In our experiments, the program was executed to perform queue's operations at three contention levels. In high contention, each thread performed one operation right after another. In medium contention, "other work" with a ratio following the normal distribution between 0 and 1 was performed between two consecutive operations. The "other work" was a fixed-times spin loop of a simple calculation. In low contention, "other work" was always performed between two consecutive operations. An exponential back-off was also used after any failed *DCAS*. The program can be run by one to 8



(a) attempts(max)/op **(b)** attempts(avg)/op **(c)** Execution time

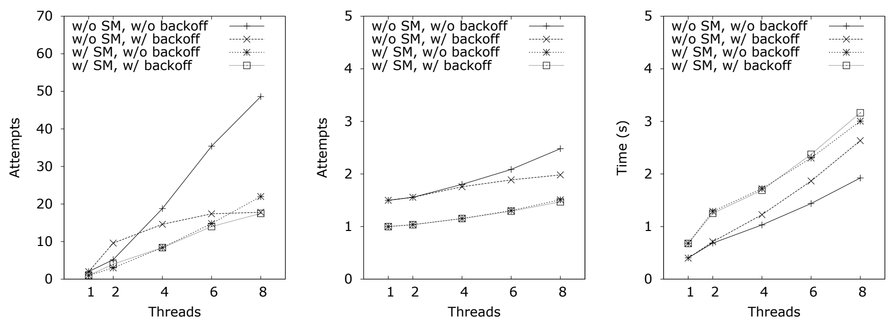
Fig. 1. Measurement results in high contention level

threads and each thread performs 1 000 000 queue operations. Each experiment is the program configured to one contention level and with or without back-off, and set up with a specific number of threads. Each experiment ran five times on a platform with two Intel Core i7 quad-core processors and the average result of the runs was reported. When running the experiments, no other users were using the system.

Three measurements were recorded. The first two were the maximum and average number of attempts between two consecutive successful operations invoked by one object. The maximum number of attempts is an indicator to know whether the proposed synchronization mechanism helped the sharing objects before they starved. The lower this number, the more likely an object is to be helped. On the other hand, the average number of attempts, helps answer a question: does the synchronization mechanism cause the total number of attempts to perform the set of operations increasing? The third measurement was the time it took to finish a run.

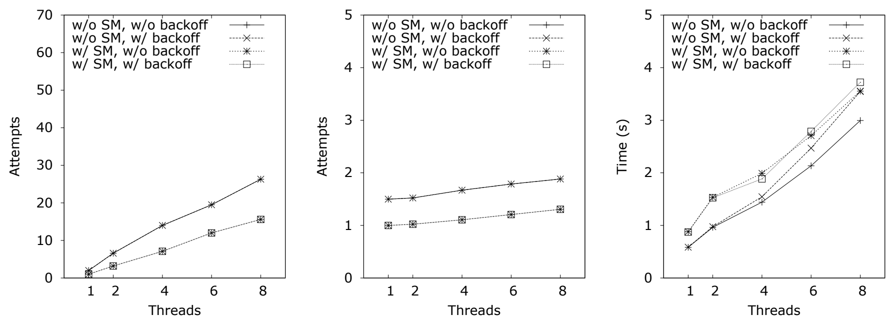
Fig. 1 presents the experimental results for the case of high contention. Fig. 1a shows that our synchronization mechanism (w/ SM) significantly reduced the maximum number of attempts to finish one operation when there was no back-off. In the case where no synchronization mechanism was used (w/o SM), the maximum number of attempts when back-off is used (w/ backoff) is much lower than when it is not (w/o backoff). The reason is that back-off reduces the contention among threads and, therefore, lowers the number of attempts. Even though, in this case, there is no lock-free progress guarantee for the sharing objects. The average number of attempts in Fig. 1b shows that when our synchronization mechanism is used, one queue operation needs, on average, about only two thirds of the number of attempts compared to when it is not used. Similar improvements when the synchronization mechanism was used are also observed in medium and low contention levels as shown in Figs. 2a, 2b, 3a, and 3b.

Fig. 1c shows the time to finish all operations at high contention level. Either with or without back-off, the execution time of the runs where our synchronization mechanism was used took about 1.7 of those where the original queue is used. This degradation in performance is because of the overhead cost when applying our synchronization mechanism to achieve the lock-freedom property. In medium and low contention levels, our synchronization performed better which reduced the ratios to 1.5 (Fig. 2c) and 1.2



(a) attempts(max)/op **(b)** attempts(avg)/op **(c)** Execution time

Fig. 2. Measurement results in medium contention level

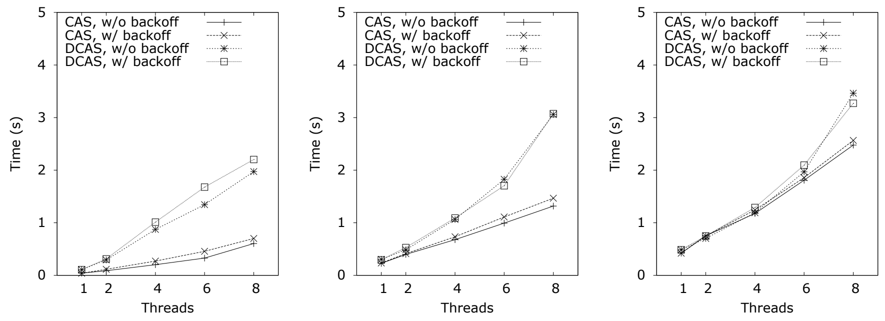


(a) attempts(max)/op **(b)** attempts(avg)/op **(c)** Execution time

Fig. 3. Measurement results in low contention level

(Fig. 3c) respectively. Especially, in low contention level with back-off, the performance of the queue where our synchronization was used is closer to that when it was not used. Our synchronization mechanism performed better in these contention levels than in high contention levels. This is consistent with the previous result that fewer attempts were performed to finish one queue operation in lower contention level. In addition, when the number of attempts were fewer, the number of cases that the synchronization mechanism was activated to help "unlucky object" were fewer too.

We performed additional experiments to analyze the overhead cost by measuring the performance of *DCAS* comparing to that of *CAS*. The experimental setup was similar to the one described in previous experiments. The only difference was that the queue operations were replaced by an operation containing a simple mathematical calculation and a *DCAS* (or *CAS*). The performance result in Fig. 4 shows that *DCAS* is much more expensive than *CAS* especially in high and medium contention levels. In low contention level, execution time of a *DCAS* operations is quite comparable to that of a *CAS*. These results support a claim that *DCAS* contributes a big portion to the overhead cost of our synchronization mechanism.



(a) High contention **(b)** Medium contention **(c)** Low contention

Fig. 4. Performance of DCAS and CAS

In brief, the experimental results demonstrate that our synchronization mechanism reduces the maximum number of attempts in all the contention level cases. The presented experimental results support the theoretical proofs. The results also show, as expected, that there is a performance overhead cost in order to achieve lock-freedom when composing. The software-implemented *DCAS* mainly contributes to this cost. We expect that with the use of a hardware-supported *DCAS* such as the Advanced Synchronization Facility by Advanced Micro Devices [2], this cost will be reduced significantly.

5 Conclusion

This paper presents our observation on progress guarantees provided by lock-free objects that concurrently share other lock-free objects. We found that these sharing objects can not provide lock-free progress guarantee as expected. A new synchronization mechanism for composing lock-free objects is proposed in order to provide lock-free progress guarantees for each individual. The experimental results show the effectiveness of the new mechanism. A preliminary study for the performance cost introduced by the new mechanism is also presented.

The assumption of the fixed number M of sharing objects should be studied further and if possible removed. Additional experiments can be performed to investigate the influence of choosing T_{MAX} and O_{MAX} on the performance of the mechanism. In addition, an implementation of the mechanism that uses a hardware-supported *DCAS* such as Advanced Synchronization Facility by Advanced Micro Devices is expected to reduce the performance cost.

References

1. IBM System/370 Extended Architecture, Principles of Operations. No. SA22-7085. IBM Publication (1983)
2. AMD: Advanced Synchronization Facility - Proposed Architectural Specification. No. 45432/rev 2.1, AMD (2009)

3. Cederman, D., Tsigas, P.: Supporting lock-free composition of concurrent data objects. In: Conf. Computing Frontiers, pp. 53–62. ACM, New York (2010)
4. Click, C.: A lock-free wait-free hash table, lecture notes in Course EE380 (2006-2007), Stanford University (2007), `http://www.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`
5. Cohen, D., Campbell, N.: Automatic composition of data structures to represent relations. In: Proceedings of KBSE 1992, pp. 182–191 (September 1992)
6. Fraser, K., Harris, T.: Concurrent programming without locks. ACM Trans. Comput. Syst. 25(2) (2007)
7. Gao, H., Groote, J., Hesselink, W.: Almost wait-free resizable hashtables. In: Proceedings of IPDPS 2004, p. 50a (2004)
8. Gidenstam, A., Papatriantafilou, M., Tsigas, P.: Allocating memory in a lock-free manner. Algorithmica 58, 304–338 (2005)
9. Gidenstam, A., Sundell, H., Tsigas, P.: Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 302–317. Springer, Heidelberg (2010)
10. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Lecture Notes in Computer Science, pp. 300–314. Springer, Heidelberg (2001)
11. Herlihy, M.: A methodology for implementing highly concurrent objects. ACM Trans. Program. Lang. Syst. 15(5), 745–770 (1993)
12. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, San Francisco (2008)
13. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. 15(6), 491–504 (2004)
14. Michael, M.M.: Scalable lock-free dynamic memory allocation. SIGPLAN Not. 39(6), 35–46 (2004)
15. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of PODC 1996, pp. 267–275 (1996)
16. Petrank, E., Musuvathi, M., Steesngaard, B.: Progress guarantee for parallel programs via bounded lock-freedom. In: Proceedings of PLDI 2009, pp. 144–154 (2009)
17. Purcell, C., Harris, T.: Non-blocking hashtables with open addressing. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 108–121. Springer, Heidelberg (2005)
18. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. 65(5), 609–627 (2005)
19. Sundell, H., Tsigas, P.: Lock-free and practical doubly linked list-based deques using single-word compare-and-swap 3544, 240–255 (2005)
20. Tsigas, P., Zhang, Y.: Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. SIGMETRICS Perform. Eval. Rev. 29, 320–321 (June 2001)
21. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proceedings of SPAA 2001, pp. 134–143 (2001)
22. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In: Proceedings of the 3rd International Workshop on Software and Performance WOSP 2002, pp. 55–67 (2002)
23. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proceedings of PODC 1995, pp. 214–222. ACM, New York (1995)