A Dynamic Elimination-Combining Stack Algorithm

Gal Bar-Nissan, Danny Hendler and Adi Suissa
Department of Computer Science
Ben-Gurion University

June 9, 2021

Abstract

Two key synchronization paradigms for the construction of scalable concurrent data-structures are *software combining* and *elimination*. Elimination-based concurrent data-structures allow operations with *reverse semantics* (such as *push* and *pop* stack operations) to "collide" and exchange values without having to access a central location. Software combining, on the other hand, is effective when colliding operations have *identical semantics*: when a pair of threads performing operations with identical semantics collide, the task of performing the combined set of operations is delegated to one of the threads and the other thread waits for its operation(s) to be performed. Applying this mechanism iteratively can reduce memory contention and increase throughput.

The most highly scalable prior concurrent stack algorithm is the *elimination-backoff stack* [5]. The elimination-backoff stack provides high parallelism for symmetric workloads in which the numbers of *push* and *pop* operations are roughly equal, but its performance deteriorates when workloads are asymmetric.

We present DECS, a novel Dynamic Elimination-Combining Stack algorithm, that scales well for all workload types. While maintaining the simplicity and low-overhead of the elimination-bakcoff stack, DECS manages to benefit from collisions of both identical- and reverse-semantics operations. Our empirical evaluation shows that DECS scales significantly better than both blocking and non-blocking best prior stack algorithms.

1 Introduction

Concurrent stacks are widely used in parallel applications and operating systems. As shown in [11], LIFO-based scheduling reduces excessive task creation and prevents threads from attempting to dequeue and execute a task which depends on the results of other tasks. A concurrent stack supports the *push* and *pop* operations with linearizable LIFO semantics. *Linearizability* [7], which is the most widely used correctness condition for concurrent objects, guarantees that each operation appears to have an atomic effect at some point between its invocation and response and that operations can be combined in a modular way.

Two key synchronization paradigms for the construction of scalable concurrent data-structures in general, and concurrent stacks in particular, are *software combining* [13, 3, 4] and *elimination* [1, 10]. Elimination-based concurrent data-structures allow operations with reverse semantics (such as *push* and *pop* stack operations) to "collide" and exchange values without having to access a central location. Software combining, on the other hand, is effective when colliding operations have identical semantics: when a pair of threads performing operations with identical semantics collide, the task of performing the combined set of operations is delegated to one of the threads and the other thread waits for its operation(s) to be performed. Applying this mechanism iteratively can reduce memory contention and increase throughput.

The design of efficient stack algorithms poses several challenges. Threads sharing the stack implementation must synchronize to ensure correct linearizable executions. To provide scalability, a stack algorithm must be highly parallel; this means that, under high load, threads must be able to synchronize their operations without accessing a central location in order to avoid sequential bottlenecks. Scalability at high loads should not, however, come at the price of good performance in the more common low contention cases. Hence, another challenge faced by stack algorithms is to ensure low latency of stack operations when only a few threads access the stack simultaneously.

The most highly scalable concurrent stack algorithm known to date is the lock-free *elimination-backoff stack* of Hendler, Shavit and Yerushalmi [5] (henceforth referred to as the HSY stack). It uses a single elimination array as a backoff scheme on a simple lock-free central stack (such as Treiber's stack algorithm [12][1]). If the threads fail on the central stack, they attempt to eliminate on the array, and if they fail in eliminating, they attempt to access the central stack once again and so on. As shown by Michael and Scott [9], the central stack of [12] is highly efficient under low contention. Since threads use the elimination array only when they fail on the central stack, the elimination-backoff stack algorithm enjoys similar low contention efficiency.

The HSY stack scales well under high contention if the workload is symmetric (that is, the numbers of *push* and *pop* operations are roughly equal), since multiple pairs of operations with reverse semantics succeed in exchanging values without having to access the central stack. Unfortunately, when workloads are asymmetric, most collisions on the elimination array are between operations with identical semantics. For such workloads, the performance of the HSY stack deteriorates and falls back to the sequential performance of a central stack.

Recent work by Hendler et al. introduced *flat-combining* [2], a synchronization mechanism based on coarse-grained locking in which a single thread holding a lock performs the combined work of other threads. They presented flat-combining based implementations of several concurrent objects, including a flat-combining stack (FC stack). Due to the very low synchronization overhead of flat-combining, the FC

[1] Treiber's algorithm is a variant of an algorithm previously introduced by IBM [8].

stack significantly outperforms other stack implementations (including the elimination-backoff stack) in low and medium concurrency levels. However, since the FC stack is essentially sequential, its performance does not scale and even deteriorates when concurrency levels are high.

Our Contributions:

This paper presents DECS, a novel Dynamic Elimination-Combining Stack algorithm, that scales well for all workload types. While maintaining the simplicity and low-overhead of the HSY stack, DECS manages to benefit from collisions of both identical- and reverse-semantics operations.

The idea underlying DECS is simple. Similarly to the HSY stack, DECS uses a contention-reduction layer as a backoff scheme for a central stack. However, whereas the HSY algorithm uses an elimination layer, DECS uses an *elimination-combining layer* on which concurrent operations can dynamically either eliminate or combine, depending on whether their operations have reverse or identical semantics, respectively. As illustrated by Figure 1-(a), when two identical-semantics operations executing the HSY algorithm collide, both have to retry their operations on the central stack. With DECS (Figure 1-(b)), every collision, regardless of the types of the colliding operations, reduces contention on the central stack and increases parallelism by using either elimination or combining. Since combining is applied iteratively, each colliding operation may attempt to apply the combined operations (multiple *push* or multiple *pop* operations) of multiple threads - its own and (possibly) the operations delegated to it by threads with which it previously collided, threads that are awaiting their response.

We compared DECS with a few prior stack algorithm, including the HSY and the FC stacks. DECS outperforms the HSY stack on all workload types and all concurrency levels; specifically, for asymmetric workloads, DECS provides up to 3 times the throughput of the HSY stack.

The FC stack outperforms DECS in low and medium levels of concurrency. The performance of the FC stack deteriorates quickly, however, as the level of concurrency increases. DECS, on the other hand, continues to scale on all workload types and outperforms the FC stack in high concurrency levels by a wide margin, providing up to 10 times its throughput.

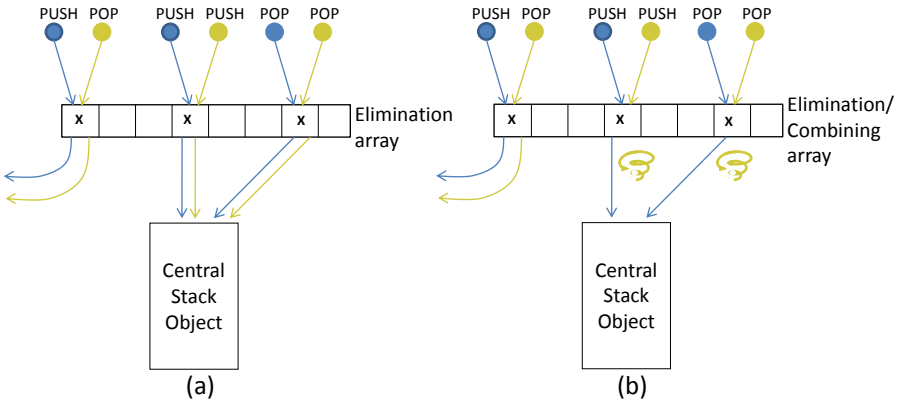


Figure 1: Collision-attempt scenarios: (a) Collision scenarios in the elimination-backoff stack; (b) Collision scenarios in DECS.

For some applications, a nonblocking [6] stack may be preferable to a blocking one because lock-freedom is more robust in the face of thread failures. Whereas the elimination-backoff stack is lock-free, both the FC and the DECS stacks are blocking. We present NB-DECS, a lock-free [6] variant of DECS that allows threads that delegated their operations to a combining thread and have waited for too long to cancel their "combining contracts" and retry their operations. The performance of NB-DECS is slightly better than that of the HSY stack when workloads are symmetric and for *pop*-dominated workloads, but it provides significantly higher throughput for *push-dominated* asymmetric workloads.

The remainder of this paper is organized as follows. We describe the DECS algorithm in Section 2 and report on its performance evaluation in Section 3. A high-level description of the NB-DECS algorithm and its performance evaluation are provided in Section 4. We conclude the paper in Section 5 with a short discussion of our results. Detailed correctness proof sketches and pseudo-codes of some functions that were omitted from the body of the paper are presented in the appendix.

2 The Dynamic Elimination-Combining Algorithm

In this section we describe the DECS algorithm. Figure 2-(a) presents the data-structures and shared variables used by DECS. Similarly to the HSY stack, DECS uses two global arrays - *location* and *collision* - which comprise its elimination-combining layer. Each entry of the *location* array corresponds to a thread $t \in \{1..N\}$ and is either *NULL* or stores a pointer to a *multiOp* structure described shortly. Each non-empty entry of the *collision* array stores the ID of a thread waiting for another thread to collide with it. DECS also uses a *CentralStack* structure, which is a singly-linked-list of *Cell* structures - each comprising an opaque *data* field and a *next* pointer. Threads iteratively do the following: first, they try to apply their operation to the *CentralStack* structure and, if they fail, they access the elimination-combining layer (see Figure 6 in the appendix[2]).

Push or *pop* operations that access the elimination-combining layer may be combined. Thus, in general, operations that are applied to the central stack or to the elimination-combining layer are *multi-ops*; that is, they are either *multi-pop* or *multi-push* operations which represent the combination of multiple *pop* or *push* operations, respectively. A multi-op is performed by a *delegate thread*, attempting to perform its own operation and (possibly) also those of one or more *waiting threads*. The *length* of a multi-op is the number of operations it consists of (which is the number of corresponding waiting threads plus 1). Each multi-op is represented by a *multiOp* structure (see Figure 2-(a)), consisting of a thread identifier *id*, the operations type (*PUSH* or *POP*) and a *Cell* structure (containing the thread data in case of a multi-push or empty in case of a multi-pop). The *next* field points to the structure of the next operation of the multiOp (if any). Thus, each multiOp is represented by a *multiOp list* of structures, the first of which represents the operation of the delegate thread. The *last* field points to the last structure in the multiOp list and the *length* field stores the multi-op's length. The *cStatus* field is used for synchronization between a delegate thread and the threads awaiting it and is described later in this section.

[2]For lack of space, some of the more straightforward pseudo-code is presented in the appendix.

Central Stack Functions

Figures 2-(b) and 2-(c) respectively present the pseudo-code of the `cMultiPop` and `cMultiPush` functions applied to the central stack.

The `cMultiPop` function receives as its input a pointer to the first *multiOp* record in a multi-op list of *pop* operations to be applied to the central stack. It first reads the central stack pointer (line **6**). If the stack is empty, then all the *pop* operations in the list are linearized in line **6** and will return an *empty* indication. In lines **9–14**, an EMPTY_CELL is assigned as the response of all these operations and the *cStatus* fields of all the multiOp structures is set to FINISHED in order to signal all waiting threads that their response is ready. The `cMultiPop` function then returns *true* indicating to the delegate thread that its operation was applied.

If the stack is not empty, the number m of items that should be popped from the central stack is computed (lines **15–19**); this is the minimum between the length of the multi-pop operation and the central stack's size. The *nTop* pointer is set accordingly and a *CAS* is applied to the central stack attempting to atomically pop m items (line **20**). If the CAS fails, *false* is returned (line **34**) indicating that `cMultiPop` failed and that the multi-pop should be next applied to the elimination-combining layer.

If the CAS succeeds, then all the multi-pop operations are linearized when it occurs. The `cMultiPop` function proceeds by iterating over the multi-op list (lines **23–32**). It assigns the m cells that were popped from the central stack to the first m pop operations (line **27**) and assigns an EMPTY_CELL to the rest of the pop operations, if any (line **25**). It then sets the *cStatus* of all these operations to FINISHED (line **30**), signalling all waiting threads that their response is ready. The `cMultiPop` function then returns *true*, indicating that it was successful (line **33**).

The `cMultiPush` function receives as its input a pointer to the first *multiOp* record in a multi-op list of *push* operations to be applied to the central stack. It sets the *next* pointer of the last cell to point to the top of the central stack (line **36**) and applies a *CAS* operation in an attempt to atomically chain the list to the central stack (line **37**). If the CAS succeeds, then all the *push* operations in the list are linearized when it occurs. In this case, the `cMultiPush` function proceeds by iterating over the multi-op list and setting the *cStatus* of the *push* operations to FINISHED (lines **38–41**). It then returns *true* in line **42**, indicating its success. If the CAS fails, `cMultiPush` returns *false* (line **44**) indicating that the multi-push should now be applied to the elimination-combining layer.

Elimination-Combining Layer Functions

The `collide` function, presented in Figure 2-(d), implements the elimination-combining backoff algorithm performed after a multi-op fails on the central stack.[3] It receives as its input a pointer to the first *multiOp* record in a multi-op list. A delegate thread executing the function first *registers* by writing to its entry in the *location* array (line **46**) a pointer to its *multiOp* structure, thus advertising itself to other threads that may access the elimination-combining layer . It then chooses randomly and uniformly an index into the collision array (line **47**) and repeatedly attempts to swap the value in the corresponding entry with its own ID by using *CAS* (lines **48–50**).

[3]This function is similar to the `LesOP` function of the HSY stack and is described for the sake of presentation completeness.

Figure 2: (a): Data structures, (b), (c): central stack operations, (d): the `collide` function.

(a) Data Structures and Shared Variables

```
1  define Cell: struct {data: Data, next: Cell };
2  define multiOp: struct
     {id,op,length,cStatus:int, cell: Cell, next,last:
     multiOp };
3  global CentralStack: Cell;
4  global collision: array of [1,...,N] of int init
     EMPTY;
5  global location: array of [1,...,N] of multiOp
     init null;
```

(b) **boolean** `cMultiPop`(*multiOp*: mOp)

```
6  top = CentralStack;
7  if top = null then
8      repeat
9          mOp.cell = EMPTY_CELL;
10         mOp.cStatus = FINISHED;
11         mOp=mOp.next;
12     until mOp = null ;
13     return true;
14 end
15 nTop = top.next;
16 m = 1;
17 while nTop ≠ null ∧ m < mOp.length do
18     nTop = nTop.next, m++;
19 end
20 if CAS(&CentralStack, top, nTop) then
21     mOp.cell = top;
22     top = top.next;
23     while mOp.next ≠ null do
24         if top = null then
25             mOp.next.cell = EMPTY_CELL;
26         else
27             mOp.next.cell = top;
28             top = top.next;
29         end
30         mOp.next.cStatus = FINISHED;
31         mOp.next = mOp.next.next;
32     end
33     return true;
34 else return false;
```

(c) **boolean** `cMultiPush`(*multiOp*: mOp)

```
35 top = CentralStack;
36 mOp.last.cell.next=top;
37 if CAS(&CentralStack, top, mOp.cell) then
38     while mOp.next ≠ null do
39         mOp.next.cStatus = FINISHED;
40         mOp.next = mOp.next.next;
41     end
42     return true;
43 else
44     return false;
45 end
```

(d) **boolean** `collide`(multiOp: mOp)

```
46 location[id] = mOp;
47 index = randomIndex();
48 him = collision[index];
49 while CAS(&collision[index], him, id)=false
   do
50     him = collision[index];
51 end
52 if him ≠ EMPTY then
53     oInfo = location[him];
54     if oInfo ≠ NULL ∧ oInfo.id ≠ id ∧
       oInfo.id=him then
55         if CAS(&location[id], mOp,
           NULL)=true then
56             return activeCollide(mOp,
               oInfo);
57         else
58             return
               passiveCollide(mOp);
59         end
60     end
61 end
62 wait();
63 if CAS(&location[id], mOp, NULL)=false
   then
64     return passiveCollide(mOp);
65 end
66 return false;
```

A thread that initiates a collision is called an *active collider* and a thread that discovers it was collided with is called a *passive collider*. If the value read from the collision array entry is not null (line **52**), then it is a value written there by another registered thread that may await a collision. The delegate thread (now acting as an active collider) proceeds by reading a pointer to the other thread's multiOp structure *oInfo* (line **53**) and then verifies that the other thread may still be collided with (line **54**).[4]

If the tests of line **54** succeed, the delegate thread attempts to *deregister* by CAS-ing its *location* entry back to *NULL* (line **55**). If the CAS is successful, the thread calls the `activeCollide` function (line **56**) in an attempt to either combine or eliminate its operations with those of the other thread. If the CAS fails, however, this indicates that some other thread was quicker and already collided with the current thread; in this case, the current thread becomes a passive thread and executes the `passiveCollide` function (line **58**).

If the tests of line **54** fail, the thread attempts to become a passive collider and waits for a short period of time in line **62** to allow other threads to collide with it. It then tries to deregister by CAS-ing its entry in the *location* array to *NULL*. If the CAS fails - implying that an active collider succeeded in initiating a collision with the delegate thread - the delegate thread, now a passive collider, calls the `passiveCollide` (line **64**) function in an attempt to finalize the collision. If the CAS succeeds, the thread returns *false* indicating that the operation failed on the elimination-combining layer and should be retried on the central stack.

The `activeCollide` function (figure 3-(a)) is called by an active collider in order to attempt to combine or eliminate its operations with those of a passive collider. It receives as its input pointers to the *multiOp* structures of both threads. The active collider first attempts to swap the passive collider's *multiOp* pointer with a pointer to its own *multiOp* structure by performing a *CAS* on the *location* array in line **67**. If the CAS fails then the passive collider is no longer eligible for collision and the function returns *false* (line **76**), indicating that the executing thread must retry its multi-op on the central stack. If the *CAS* succeeds, then the collision took place. The active collider now compares the type of its multi-op with that of the passive collider (line **68**) and calls either the `combine` or the `multiEliminate` function, depending on whether the multi-ops have identical or reverse semantics, respectively (lines **68–73**). Observe that `activeCollide` returns *true* in case of elimination and *false* in case of combining. The reason is the following: in the first case it is guaranteed that the executing thread's operation was matched with a reverse-semantics operation and so was completed, whereas in the latter case the operations of the passive collider are delegated to the active collider which must now access the central stack again.

The `passiveCollide` function (figure 3-(b)) is called by a passive collider after it identifies that it was collided with. The passive collider first reads the multi-op pointer written to its entry in the *location* array by the active collider and initializes its entry in preparation for future operations (lines **84–85**). If the multi-ops of the colliding threads-pair are of reverse semantics (line **86**) then the function returns *true* in line **90** because, in this case, it is guaranteed that the colliding delegate threads exchange values. Specifically, if the passive thread's multi-op type is *pop*, the thread copies the cell communicated to it by the active collider (line **88**).

If both multi-ops are of identical semantics, then the passive collider's operations were delegated to the active thread and the executing thread ceases to be a delegate thread. In this case, the thread waits until it is signalled (by writing to the *cStatus* field of its *multiOp* structure) how to proceed. There are two possibilities:

[4]Some of the tests of line **54** are required because *location* array entries are not re-initialized when operations terminate (for optimization reasons) and thus may contain outdated values.

Figure 3: (a) The `activeCollide`, (b) `passiveCollide` and, (c) `combine` functions.

```
(a) boolean activeCollide(multiOp:aInf,pInf)
67 if CAS(&location[pInf.id], pInf, aInf) then
68 |   if aInf.op = pInf.op then
69 |   |   combine(aInf,pInf);
70 |   |   return false;
71 |   else
72 |   |   multiEliminate(aInf,pInf);
73 |   |   return true;
74 |   end
75 else
76 |   return false;
77 end
```

```
(b) boolean passiveCollide(multiOp:pInf)
84 aInf = location[pInf.id];
85 location[pInf.id] = null;
86 if pInf.op ≠ aInf.op then
87 |   if pInf.op = POP then
88 |   |   pInf.cell = aInf.cell;
89 |   end
90 |   return true;
91 else
92 |   await(pInf.cStatus ≠ INIT);
93 |   if pInf.cStatus = FINISHED then
94 |   |   return true;
95 |   else
96 |   |   pInf.cStatus = INIT;
97 |   |   return false;
98 |   end
99 end
```

```
(c) combine(multiOp: aInf, pInf)
78 if aInf.op = PUSH then
79 |   aInf.last.cell.next = pInf.cell;
80 end
81 aInf.last.next = pInf;
82 aInf.last = pInf.last;
83 aInf.length = aInf.length + pInf.length;
```

(1) *cStatus = FINISHED* holds in line **93**. In this case, the thread's operation response is ready and it returns *true* in line **94**. (2) *cStatus = RETRY* holds (line **95**) indicating that the executing thread became a delegate thread once again. This occurs if a thread to which the current thread's operation was delegated eliminated with a multi-op that had a shorter list than its own and the first operation in the "residue" is the current thread's operation. In this case, the thread changes the value of its $cStatus$ back to *INIT* (line **96**) and returns *false*, indicating that the operation should be retried on the central stack.

The *combine* function (figure 3-(c)) is called by an active collider when the operations of both colliders have identical semantics. It receives as its input pointers to the *multiOp* structures of the two colliders. It delegates the operations of the passive collider to the active one by concatenating the multiOp list of the passive collider to that of the active collider, and by updating the *last* and *length* fields of its *multiOp* record accordingly (lines **81**–**83**). In addition, if the type of both multi-ops is *push*, then their cell-lists are also concatenated (line **79**); this allows the delegate thread to push all its operations to the central stack by using a single *CAS* operation.

The *multiEliminate* function is called by an active collider when the operations of both colliders have reverse semantics. It matches as many pairs of reverse-semantics operations as possible. If there is a residue of *push* or *pop* operations, it signals the first waiting thread in the residue list by writing the value *RETRY* to the *cStatus* field of its *multiOp* structure. The signaled thread becomes a delegate thread again and retries

its multi-op on the central stack. The full pseudo-code of the *multiEliminate* function and its description are deferred to the appendix.

3 DECS Performance Evaluation

We conducted our performance evaluation on a Sun SPARC T5240 machine, comprising two UltraSPARC T2 plus (Niagara II) chips, running the Solaris 10 operating system. Each chip contains 8 cores and each core multiplexes 8 hardware threads, for a total of 64 hardware threads per chip. We ran our experiments on a single chip to avoid communication via the L2 cache. The algorithms we evaluated are implemented in C++ and the code was compiled using GCC with the -O3 flag for all algorithms.

We compare DECS with the Treiber stack[5] and with the most effective known stack implementations: the HSY elimination-backoff stack, and a flat-combining based stack.[6] [7]

In our experiments, threads repeatedly apply operations to the stack for a fixed duration of one second and we measure the resulting *throughput* - the total number of operations applied to the stack - varying the level of concurrency from 1 to 128. Each data point is the average of three runs. We measure throughput on both symmetric (push and pop operations are equally likely) and asymmetric workloads. Stacks are pre-populated with enough cells so that pop operations do not operate on an empty stack also in asymmetric workloads.

Figures 4-(a) through (c) compare the throughput of the algorithms we evaluate in symmetric (50% push, 50% pop), moderately-asymmetric (25%push, 75% pop) and fully-asymmetric (0% push, 100% pop) workloads, respectively. It can be seen that the DECS stack outperforms both the Treiber stack and the HSY stack for all workload types and all concurrency levels.

Symmetric workloads

We first analyze performance on a symmetric workload, which is the optimal workload for the HSY stack. As shown in Figure 4-(a), even here the HSY stack is outperformed by DECS by a margin of up to 31% (when the number of HW threads is 64). This is because, even in symmetric workloads, there is a non-negligible fraction of collisions between operations of identical semantics from which DECS benefits but the HSY stack does not. Both DECS and the HSY stack scale up until concurrency level 64 - the number of hardware threads. When the number of software threads exceeds the number of hardware threads, the HSY stack more-or-less maintains its throughput whereas DECS slightly declines but remains significantly superior to the HSY stack.

The FC stack incurs the highest overhead in the lack of contention (concurrency level 1) because the single running thread still needs to capture the FC lock. Due to its low synchronization overhead it then exhibits a steep increase in its throughput and reaches its peak throughput at 24 threads, where it outperforms DECS by approximately 33%. The FC stack does not continue to scale beyond this point, however, and its

[5] We evaluated two variants of the Treiber algorithm - with and without exponential backoff. The variant using exponential backoff performed consistently better and is the version we compare with.

[6] We downloaded the most updated flat-combining code from https://github.com/mit-carbon/Flat-Combining.

[7] The Treiber, HSY and DECS algorithms need to cope with the "ABA problem" [8], since they use dynamic-memory structures that may need to be recycled and perform CAS operations on pointers to these structures. We implemented the simplest and most common ABA-prevention technique that includes a tag with the target memory locations so that both the memory location and the tag are manipulated together atomically, and the tag is incremented with each update of the target memory location [8].

throughput rapidly deteriorates as the level of concurrency rises. For concurrency levels higher than 40, its performance falls below that of DECS and it is increasingly outperformed by DECS as the level of concurrency is increased: for 64 threads, DECS provides roughly 33% higher throughput, and for 128 threads DECS outperforms FC by a factor of 4. For concurrency levels higher than 96, the throughput of the FC stack is even lower than that of the Treiber algorithm. The reason for this performance deterioration is clear: the FC algorithm is essentially sequential, since a single thread performs the combined work of other threads. The Treiber algorithm exhibits the worst performance since it is sequential and incurs significant synchronization overhead. It scales moderately until concurrency level 16 and then more-or-less maintains its throughput.

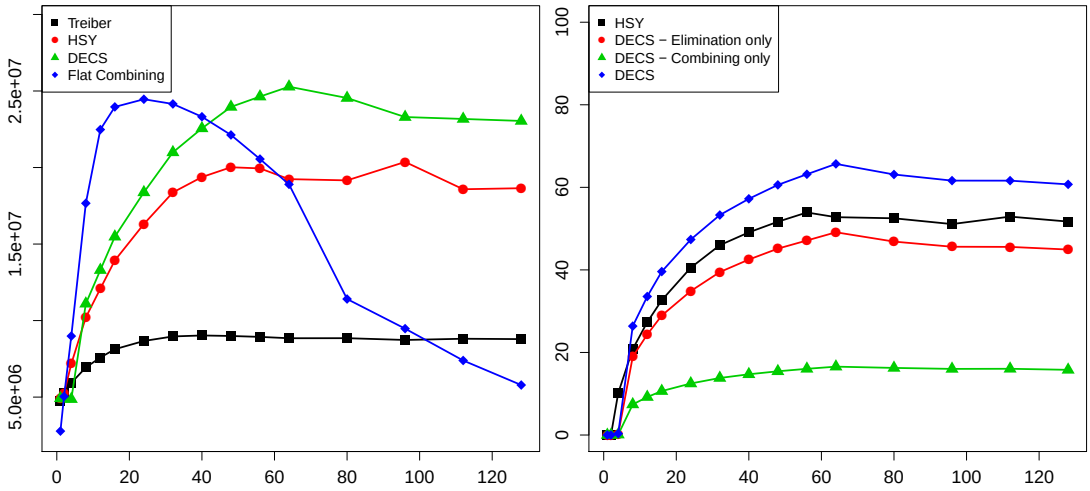
Figure 4-(d) provides more insights into the behavior of the DECS and HSY stacks in symmetric workloads. The HSY curve shows the percentage of operations completed by elimination. The DECS curve shows the percentage of operations not applied directly to the central stack. These are the operations completed by either elimination or combining.[8] The curves titled "Elimination only" and "Combining only" show a finer partition of the DECS operations according to whether they completed through elimination or combining. It can be seen that the overall percentage of operations not completed on the central stack is higher for DECS than for the HSY stack by up to 30% (for 64 threads), thus reducing the load on the central stack and allowing DECS to perform better than the HSY stack.

Asymmetric workloads

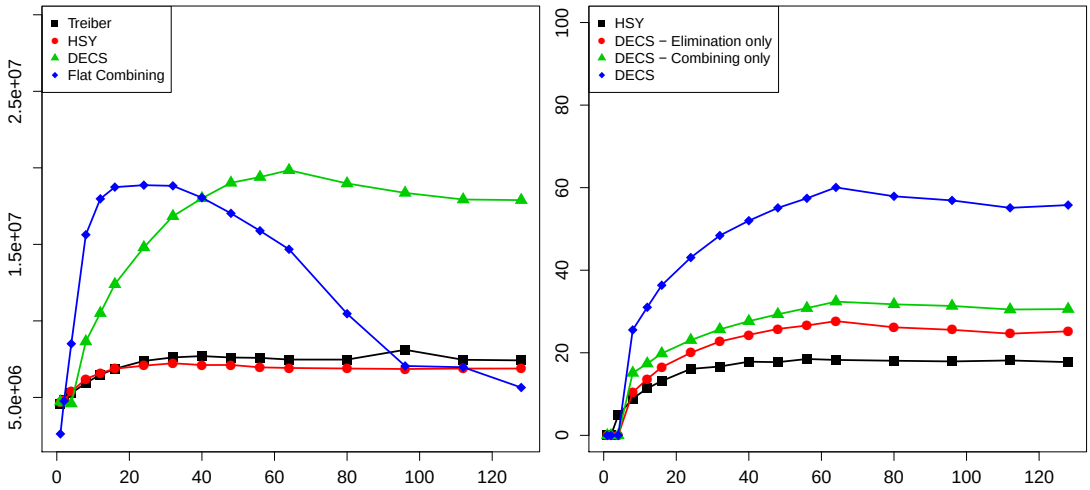
Figures 4-(b) and 4-(c) compare throughput on moderately- and fully-asymmetric workloads, respectively. The relative performance of DECS, the FC and the Treiber stacks is roughly the same as for the symmetric workload; nevertheless, DECS performance decreases because, as can be seen in Figures 4-(e) and 4-(f), the ratio of DECS operations that complete via elimination is significantly reduced for the 25% push workload and is 0 for the 0% push workload. This reduction in elimination is mostly compensated by a corresponding increase in the ratio of DECS operations that complete by combining.

The performance of the HSY stack, however, deteriorates for asymmetric workloads because, unlike DECS, it cannot benefit from collisions between operations with identical semantics. When the workload is moderately asymmetric (Figure 4-(b)), the HSY stack scales up to 32 threads but then its performance deteriorates and falls even below that of the Treiber algorithm for 48 threads or more. In these levels of concurrency, the low percentage of successful collisions makes the elimination layer counter-effective. The throughput of the DECS algorithm exceeds that of the HSY stack by a factor of up to 3. The picture is even worse for the HSY algorithm for fully asymmetric workloads (Figure 4-(c)), where it performs almost consistently worse than the Treiber algorithm. In these workloads, DECS' throughput exceeds that of the HSY algorithm significantly in all concurrency levels 8 or higher; the performance gap increases with concurrency up until 64 threads and DECS provides about 3 times the throughput for all concurrency levels 64 or higher.

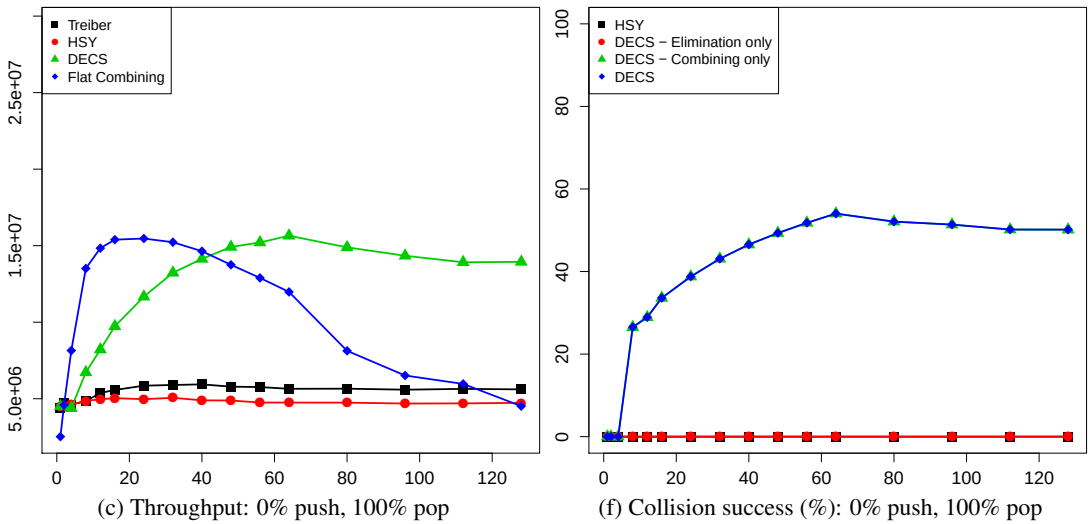
[8] Whenever a muli-op is applied to the central stack, the operation of the delegate thread is regarded as applied directly to the central stack and those of the waiting threads are counted as completed by combining. Similarly, when two multi-ops of reverse semantics collide, the operations of the delegate threads are counted as completed by elimination and those of the waiting threads as completed by combining.



(a) Throughput: 50% push, 50% pop



(b) Throughput: 25% push, 75% pop



(c) Throughput: 0% push, 100% pop

(d) Collision success (%): 50% push, 50% pop

(e) Collision success (%): 25% push, 75% pop

(f) Collision success (%): 0% push, 100% pop

Figure 4: Throughput and collision success rates. X-axis: threads #; Y-axis in (a)-(c): throughput.

4 The Nonblocking DECS Algorithm

For some applications, a nonblocking stack may be preferable to a blocking one because it is more robust in the face of thread failures. The HSY stack is nonblocking - specifically lock-free [6] - and hence guarantees global progress as long as some threads do not fail-stop. In contrast, both the FC and the DECS stacks are blocking. In this section, we provide a high-level description of NB-DECS, a lock-free variant of our DECS algorithm that allows threads that delegated their operations to another thread and have waited for too long to cancel their "combining contracts" and retry their operations. A full description of the algorithm is deferred to the appendix. We also present a comparative evaluation of the new algorithm.

Recall that waiting threads await a signal from their delegate thread in the `passiveCollide` function (line **92** in Figure 3). In the DECS algorithm, a thread awaits until the delegate thread writes to the *cStatus* field of its *multiOp* structure but may wait indefinitely. In NB-DECS, when a thread concludes that it waited "long enough" it attempts to *invalidate* its *multiOp* structure. To prevent race conditions, invalidation is done by applying *test-and-set* to a new *invalid* field added to the *multiOp* structure. A delegate thread, on the other hand, must take care not to assign a cell of a valid *push* operation to an invalid multi-op structure of a *pop* operation. This raises the following complications which NB-DECS must handle.

1. A delegate thread may pop invalid cells from the central stack. Therefore, in order not to assign an invalid cell to a *pop* operation, the delegate thread must apply *test-and-set* to each popped cell to verify that it is still valid (and if so to ensure it remains valid), which hurts performance.

2. A delegate thread performing a pop multi-op must deal with situations in which some of its waiting threads invalidated their multi-op structures. If the delegate thread were to pop from the central stack more cells than can be assigned to valid multi-op structures in its list, linearizability would be violated. Consequently, unlike in DECS, the delegate thread must pop items from the central stack *one by one*, which also hurts the performance of NB-DECS as compared with DECS.

3. The `multiEliminate` function, called by an active delegate thread when it collides with a thread with reverse semantics, must also verify that valid cells are only assigned to valid pop multi-ops. Once again, *test-and-set* is used to prevent race conditions.

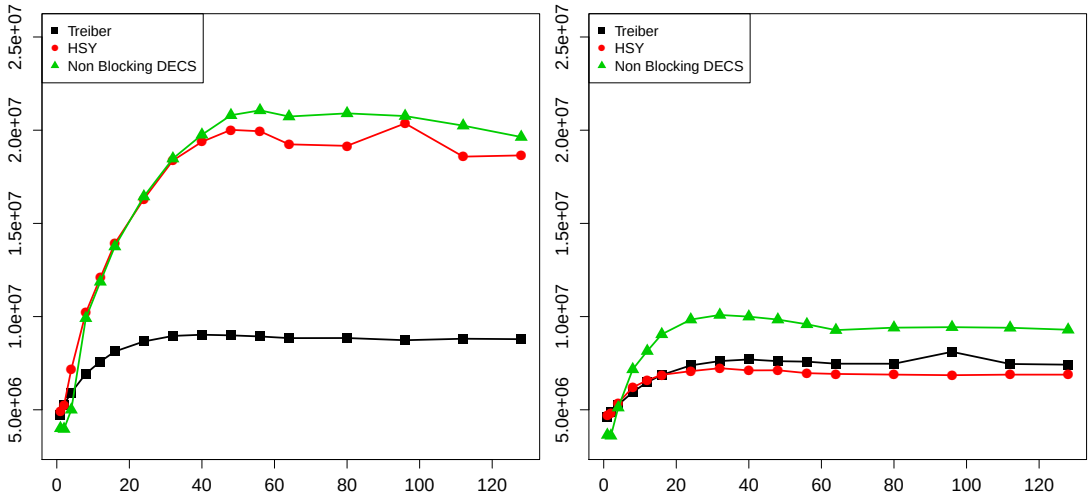
NB-DECS performance evaluation.

Due to the extra synchronization introduced in NB-DECS for allowing threads to invalidate operations that are pending for too long, the throughput of NB-DECS is, in general, significantly lower than that of the (blocking) DECS stack. We compare NB-DECS with two other lock-free algorithms: Treiber and the HSY stack. As shown in Figure 5, the performance of the NB-DECS and HSY stacks on symmetric workloads is almost identical (Figure 5-(a)), with a slight advantage to NB-DECS for concurrency levels of 36 or more, and they both scale significantly better than the Treiber stack.

For moderately-asymmetric workloads, NB-DECS performs much better than the HSY stack but its advantage is much greater when there is a majority of *push* operations. The reason for this difference is that the extra synchronization added to NB-DECS (as compared with DECS) hurts *pop* operations more than it does *push* operations. Specifically, complication 2. above hurts the performance of multi-pop operations applied to the central stack (since they need to pop cells one by one) but multi-push operations to the central

stack may still push their entire list atomically. For workloads with 75% *push* operations (Figure 5-(b)), NB-DECS outperforms the HSY stack significantly and the margin increases with concurrency. Specifically, for 56 threads or more, NB-DECS outperforms HSY by more than 70%. For workloads with 25% *push* (Figure 5-(b)), the difference is smaller but still significant and NB-DECS outperforms HSY by about 35% for concurrency levels 24 or higher.

The state of affairs is similar for fully-asymmetric workloads. When the workload consists of *push* operations only (Figure 5-(d)), NB-DECS scales nicely up to 56 threads and then more-or-less retains its throughput, delivering performance of up to 2 times that of the HSY stack. On the other hand, when the workload consists of only *pop* operations the extra synchronization hurts NB-DECS and the difference in performance is much smaller.



(a) Throughput: 50% push, 50% pop

(c) Throughput: 25% push, 75% pop

(b) Throughput: 75% push, 25% pop

(d) Throughput: 100% push, 0% pop

Figure 5: Throughput of lock-free algorithms. X-axis: threads #; Y-axis: throughput.

5 Discussion

We present DECS, a novel Dynamic Elimination-Combining Stack algorithm. Our empirical evaluation shows that DECS scales significantly better than (both blocking and nonblocking) best known stack algorithms for all workload types, providing throughput that is significantly superior to that of both the elimination-backoff stack and the flat-combining stack for high concurrency levels. We also present NB-DECS - a lock-free variant of DECS. NB-DECS provides lower throughput than (the blocking) DECS due to the extra synchronization required for satisfying lock-freedom but may be preferable for some applications since it is more robust to thread failures. NB-DECS significantly outperforms the elimination-backoff stack, the most scalable prior lock-free stack on almost all workload types. The key feature that makes DECS highly effective is the use of a dynamic elimination-combining layer as a backoff scheme for a central data-structure. We believe that this idea may be useful for obtaining high-performance implementations of additional concurrent data-structures.

Acknowledgements: We thank Yehuda Afek and Nir Shavit for allowing us to use their Sun SPARC T5240 machine.

References

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par (2)*, pages 151–162, 2010.

[2] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.

[3] D. Hendler and S. Kutten. Bounded-wait combining: constructing robust and high-throughput shared objects. *Distributed Computing*, 21(6):405–431, 2009.

[4] D. Hendler, S. Kutten, and E. Michalak. An adaptive technique for constructing robust and high-throughput shared objects. In *OPODIS*, pages 318–332, 2010.

[5] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, 2010.

[6] M. Herlihy. Wait-free synchronization. *ACM Transactions On Programming Languages and Systems*, 13(1):123–149, Jan. 1991.

[7] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[8] IBM. *IBM System/370 Extended Architecture, Principles of Operation, publication no. SA22-7085*. 1983.

[9] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared — memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[10] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, (30):645–670, 1997.

[11] K. Taura, S. Matsuoka, and A. Yonezawa. An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. In *Principles Practice of Parallel Programming*, pages 218–228, 1993.

[12] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[13] P. Yew, N. Tzeng, and D. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.

Appendix A: DECS Pseudo-Code Omitted From Paper Body

Figure 6 presents the code performed by a thread when it applies a *push* or a *pop* operation to the DECS stack. A *pop* (*push*) operation starts by initializing a *multiOp* record in line **100** (line **108**). It then attempts to apply the *pop* (*push*) operation to the central stack in line **102** (line **110**). If this attempt fails, the thread then attempts to apply its operation to the elimination-combining layer in line **104** (line **112**). A delegate thread continues these attempts repeatedly until it succeeds. A *pop* operation returns the data stored at the cell that it received either from the central stack (line **103**) or by way of elimination (line **105**).

Figure 6: DECS `Push` and `Pop` functions.

```
  Data pop()
100 multiOp mOp = initMultiOp();
101 while true do
102 |   if cMultiPop(mOp) then
103 |   |   return mOp.cell.data;
104 |   else if collide(mOp) then
105 |   |   return mOp.cell.data;
106 |   end
107 end
```

```
  push(data)
108 multiOp mOp = initMultiOp(data);
109 while true do
110 |   if cMultiPush(mOp) then
111 |   |   return;
112 |   else if collide(mOp) then
113 |   |   return;
114 |   end
115 end
```

Figure 7 presents the pseudo-code of the *multiEliminate* function, which is called by an active collider when the operations of both colliders have reverse semantics. It receives as input pointers to the *multiOp* records of the active and passive colliders. In the loop of lines **118–130**, as many pairs of reverse-semantics operations as possible are matched until at least one of the operation lists is exhausted. All matched operations are signalled by writing the value *FINISHED* to the *cStatus* field of their *multiOp* structure, indicating that they can terminate (lines **124–125**). Note that both lists contain at least one operation, thus at least a single pair of operations are matched. If the lengths of the multi-ops are unequal, then a "residue" sublist remains. In this case, the *length* and *last* fields of the *multiOp* structure belonging to the first waiting thread in the residue sub-list are set. Then that thread is signalled by writing the value *RETRY* to the *cStatus* field of its *multiOp* structure (in line **134** or line **138**). This makes the signaled thread a delegate thread once again and it will retry its multi-op on the central stack.

Figure 7: The `multiEliminate` function.

```
multiEliminate(multiOp: aInf, pInf)
116 aCurr = aInf;
117 pCurr = pInf;
118 repeat
119 |   if aInf.op = POP then
120 |   |   aCurr.cell = pCurr.cell;
121 |   else
122 |   |   pCurr.cell = aCurr.cell;
123 |   end
124 |   aCurr.cStatus = FINISHED;
125 |   pCurr.cStatus = FINISHED;
126 |   aInf.length = aInf.length - 1;
127 |   pInf.length = pInf.length - 1;
128 |   aCurr = aCurr.next;
129 |   pCurr = pCurr.next;
130 until aCurr = null ∨ pCurr = null ;
131 if aCurr ≠ null then
132 |   aCurr.length = aInf.length;
133 |   aCurr.last = aInf.last;
134 |   aCurr.cStatus = RETRY;
135 else if pCurr ≠ null then
136 |   pCurr.length = pInf.length;
137 |   pCurr.last = pInf.last;
138 |   pCurr.cStatus = RETRY;
139 end
```

Appendix B: DECS Correctness Proof Sketch

A concurrent stack is a data structure whose operations are linearizable to those of the sequential stack.

Definition 1 *A Stack S is an object that supports two operation types: pop and push. The state of the stack is a sequence of items* S = $<v_0, ...,v_n>$.
The stack is initially empty. The pop and push operations induce the following state transitions to S with appropriate return values.

- pop()*: if* S *is not empty, returns* v_n *and changes* S *to* $<v_0, ...,v_{n-1}>$*, otherwise returns empty and* S *remains unchanged.*
- push(v_{new})*: changes* S *to* $<v_0,....,v_n, v_{new}>$.

A pool is a relaxation of a stack that does not require LIFO ordering. Similarly to the proofs in [5], we start by proving that DECS has correct pool semantics and then prove that it is linearizable to a sequential

stack. Finally we prove that DECS is deadlock-free.

Correct pool semantics

Definition 2 *A stack algorithm has correct pool semantics if the following requirements are met for all stack operations:*

1. *Let Op be a pop operation, then if the number of push operations preceding Op is larger than the number of pop operations preceding it, Op returns an item (a non-empty value).*
2. *Let Op be a pop operation that returns an item i, then i was previously pushed by a push operation.*
3. *Let Op be a push operation that pushed an item i to the stack, then there is at most a single pop operation that returns it.*

An operation that complies with the above requirements is called a correct pool operation.

Definition 3 *We say that a multi-operation is a* colliding multi-op *if it returns in line* **56***, line* **58** *or line* **64** *(after executing* `activeCollide` *or* `passiveCollide`*).*

Definition 4 *Let op_1 and op_2 two multi-operations. We say that op_1 and op_2 have* collided *if one of the following conditions hold:*

- *op_1 performs a successful CAS in line* **67** *of* `activeCollide` *and pInf points to the multiOp of op_2 at that time.*
- *op_2 performs a CAS in line* **63** *of* `collide` *and the CAS fails because the entry corresponding to op_2 in the location array points to the multiOp of op_1.*

Definition 5 *If op is a colliding multi-op that executes a successful CAS in line* **67** *we say it is an* active colliding multi-op*. If it performs an unsuccessful CAS operation in line* **55** *or line* **63** *we say it is a* passive colliding multi-op*.*

Lemma 1 *Every colliding multi-op is either passive or active, but not both.*

Proof sketch: First, a multi-op cannot collide with itself since it verifies in line **54** that the other multi-op has a different id. If *op* is an active colliding multi-op, then from Definition 5 it executes a successful CAS in line **67**. It follows from the code that an active colliding multi-op executes the `activeCollide` function, implying that it executed a successful CAS in line **55** and by Definition 5 that means *op* is not a passive colliding multi-op.

A similar argument proves that if *op* is a passive colliding multi-op then it is not an active colliding multi-op. ■

Lemma 2 *Every passive colliding multi-op collides with exactly one active colliding multi-op and vice versa.*

Proof sketch: Let op_1 be an active colliding multi-op. According to Lemma 1, op_1 cannot be a passive colliding multi-op. From Definition 5, op_1 performed a successful CAS in line **67**. Let op_2 denote the multiOp that op_1 sets in its corresponding location array entry on line **67**. Only op_2 could have written its multiOp in its corresponding entry in the location array on line **46**. Hence, op_2 is executing `collide` and has not yet finished its execution because the CAS on line **63** was not performed. Since op_1 performed a successful CAS in line **67**, the CAS performed by op_2 on line **55** or on line **63** fails. It follows from Definition 5 that op_2 is a passive colliding multi-op. Finally, any other operation trying to perform the CAS operation in line **67** on the location of the process executing op_2 will fail, hence op_1 is the only active colliding multi-op that collides with op_2. ■

Lemma 3 *Let op_2 be a passive colliding multi-op and let op_1 be the active colliding multi-op it collided with. Then when op_2 enters `passiveCollide`, a pointer to the multiOp of op_1 is written in location$[op_2]$.*

Proof sketch: op_2 enters `passiveCollide` after an unsuccessful CAS on line **55** or line **63**. This could have happened only if op_1 performed a successful CAS on line **67** and set location$[op_2]$ to *pInf*, pointing to op_2's multiOp and setting location$[op_2]$. ■

Definition 6 *let op_1 be an active colliding multi-op and op_2 be the passive colliding multi-op it collided with. If both operations have the same operation semantics, we say the collision is a* combining collision *and we call op_1 and op_2* combining multi-ops*. Otherwise we say the collision is an* eliminating collision *and we call op_1 and op_2* eliminating multi-ops*.*

Definition 7 *We say that a multi-op is a* waiting *multi-op if it executes line* **92***.*

Lemma 4 *A waiting multi-op is a passive colliding multi-op that collided with an active multi-op of the same semantics.*

Proof sketch: From the `collide` function and Definition 5 it follows that an operation executes `passiveCollide` iff it is a passive colliding multi-op. From Lemma 3 and the *passiveCollide* function it follows that a passive colliding multi-op executes line **92** only if the active colliding multi-op it collided with has the same semantics. ■

Definition 8 *We say that a multi-op is a* delegate multi-op *if it is not a waiting multi-op.*
We call the singly linked list of the multiOp structures that starts in the multiOp of the delegate multi-op, the multiOp list *of the delegate multi-op. If op_1 is a delegate multi-op and op_2 is another multi-op whose multiOp structure is in the multiOp list of op_1 then we say that op_2 is in the multiOp list of op_1.*
If the delegate multi-op is push, we call the singly linked list of cells that starts with the cell of the delegate operation the cells list *of the delegate operation.*

Lemma 5 *Let op_1 be a delegate multi-op, then all multi-ops in its multiOp list except for op_1 itself are waiting multi-ops and all the multi-ops in the multiOp list of op_1 have identical operation semantics.*

Proof sketch: Let op_1 be a delegate multi-op. Upon the first invocation of op_1, the only multi-op in op_1's multiOp list is op_1 so the claim holds. In addition, note that the last field in op_1's multiOp points to itself, which is the last multiOp in op_1's multiOp list.

Multi-ops are added to the multiOp list of op_1 only during the execution of the `combine` function which is called during the execution of `activeCollide` and only if some other op_2 is a passive colliding multi-op that collided with op_1 and has identical semantics. According to Lemma 4, a waiting multi-op is a passive colliding multi-op with the same operation semantics as the active colliding multi-op it collided with. Let op_2 denote the waiting multi-op that is added to op_1's multiOp list. Note that, op_1 adds op_2's multiOp to its multiOp list at its end and then updates the last field of its multiOp structure to point to the last element in op_2's multiOp list. As op_2 is a waiting multi-op with the same operation semantics as op_1, the claim holds. ■

Definition 9 *Let op_1 and op_2 be two multi-ops. We say that op_1* signals *SIGNAL to op_2 if op_1 writes SIGNAL to the* cStatus *field of the multiOp structure of op_2.*

Definition 10 *Let op_1 and op_2 be two multi-ops. We say that op_1 is the delegate multi-op of op_2 in time t, if op_1 was not yet eliminated (in the* multiEliminate *function) and one of the following cases holds:*

1. *op_2 is op_1 (that is, every delegate operation is the delegate operation of itself).*
2. *op_1 is an active colliding multi-op, op_2 is the passive colliding multi-op it collided with and op_1 and op_2 have the same operation semantics. In this case, op_1 becomes the delegate multi-op of op_2 when they collide (at which time op_2 ceases to be the delegate multi-op of itself).*
3. *op_3 is the delegate multi-op of op_2 and, in time t, op_1 becomes the delegate multi-op of op_3. In this case, we say that op_1 becomes the delegate multi-op of op_2 instead of op_3 (op_3 ceases to be the delegate multi-op of op_2).*
4. *Assume that op_3 is the delegate multi-op of both op_1 and op_2, the multiOps of both of them are in the multiOp list of op_3 and the multiOp of op_1 appears before the multiOp of op_2 in the multiOp list of op_3. If op_3 signals RETRY to op_1, then op_1 becomes the delegate multi-op of op_2. In this case we say that op_3 ceases being the delegate multi-op of op_2 and that op_1 starts being the delegate multi-op of op_2 (note that in this case op_1 stops executing line* **92** *and by Definition 7 ceases being a waiting multi-op and starts being a delegate multi-op).*

Lemma 6 *Every waiting multi-op has exactly a single delegate multi-op.*

Proof sketch: Let op_2 be a waiting multi-op. By Lemma 4, op_2 is a passive colliding multi-op that collided with an active colliding multi-op with identical semantics, which we denote as op_1. By Definition 10, op_1 is a delegate multi-op of op_2 at the time both of them collide. Also from Definition 10, a multi-op ceases to be the delegate multi-op of op_2 only if it is replaced by a single new delegate multi-op.

■

Lemma 7 *Let op be a delegate multi-op and let* opInfo *be its multiOp structure, then the following properties hold when op begins executing `cMultiPop`, `cMultiPush` or `collide`:*

- *The multiOp list of op is comprised of op's multiOp and of its waiting multi-ops.*
- opInfo.*length is the length of the multiOp list of op.*

- opInfo.*last points to the last multiOp in op's multiOp.*
- *If op is a push operation, then the order of its cells list corresponds to its multiOp list.*

Proof sketch: Upon an invocation of *op*, it is a delegate multi-op with a single multi-op (*op* itself). The multiOp's length field is initialized to 1, and last points to *op*'s multiOp. If the multi-op is a push, its cell field contains the data with which *op* was invoked and the cell's next field points to **null**. We still need to consider the following two cases in which the multiOp list is modified:

1. *op* is the aInf parameter of the `combine` function.
2. *op*'s multiOp list was set on lines **132–134** or lines **136–138** of `multiEliminate` function.

In the first case, *op* is the active colliding multi-op which collides with some other passive colliding multi-op op'. The `combine` function sets the next element of the last element in *op*'s multiOp list to point to op' multiOp list's first element. It then sets *op*'s last pointer to point to the last element in op''s multiOp list, and adds the number of entries of op''s multiOp list to the length field of *op*. In case both operations are of a push semantics, the function concatenates op''s cells list to the end of *op*'s cells list. In the second case, *op* is a delegate multi-op because some other operation, op'', signaled *op* with RETRY in `multiEliminate`. Prior to that signal, the last and length fields of *op* were updated by op''. The last field was set to op'''s last element in its multiOp list, and length was set to the number of multi-ops which were not eliminated from op'' multiOp list. ■

Lemma 8 *Let op be a multi-op returning* **true** *in line* **42** *of* `cMultiPush`*, then the following properties hold:*

1. *Each multi-op in op's multiOp list is a waiting push multi-op except for the first multi-op which is the delegate's multi-op.*
2. *Before returning* **true** *on line* **42***, op signaled FINISHED to each waiting multi-op in its multiOp list.*
3. *All cells in the cells list of op were pushed to the central stack in the reverse order of the cells list.*

Proof sketch:

1. If *op* returns **true** in line **42** then *op* is a push multi-op and, from Lemma 5, all the multi-ops in the delegate multiOp list are waiting push multi-ops except for the first multi-op, which is the delegate multi-op.
2. Clear from the code of the `cMultiPush` function.
3. From Lemma 7, the length field of *op*'s multiOp is equal to the length of *op*'s multiOp list, and the order of the cells list corresponds to the multiOp list. From the code on lines **36–37**, after a successful CAS operation, CentralStack points to the first cell of *op*'s cells list, and the last cell of *op*'s cells list points to the previous top item of the central stack. It is easy to see that the reversed order of the cells list implicates an execution of *op*'s multiOp list push multi-ops starting from its last multi-op and ending with the delegate multi-op.

■

Lemma 9 *Let op be a delegate multi-op that returns* **true** *in line* **13** *or line* **33** *of* `cMultiPop`*, then the following properties hold:*

1. *Each multi-op in op's multiOp list is a waiting pop multi-op except for the first multi-op which is the delegate multi-op.*
2. *Before returning* **true** *on line* **13** *or line* **33***, op signals FINISHED to each waiting multi-op in its multiOp list.*
3. *The number of cells op has popped from the central stack in* `cMultiPop` *is equal to the minimum between the number of multiOp elements in op's multiOp list and the number of cells in the stack.*
4. *If op writes EMPTY_CELL in the cell field of some multiOp in its multiOp list, then the number of elements in the central stack was smaller than the number of elements in op's multiOp list.*
5. *If op writes a cell in the cell field of a multiOp from its multiOp list, then that cell was previously pushed to the stack by some push operation and that multiOp is the only multiOp of a pop operation pointing to that cell.*

Proof sketch:

1. If *op* returns **true** in line **13** or line **33** then *op* is a pop multi-op and from Lemma 5 all the multi-ops in the delegate multiOp list are waiting pop multi-ops except the first multi-op which is the delegate multi-op.
2. Clear from the code of the `cMultiPop` function.
3. From Lemma 7, the length field of the multiOp of *op* is the length of the multiOp list of *op*. Assume that the length of the multiOp list of op_1 is N. If the stack is not empty, on line **20** CentralStack points to the $N+1$'th cell in the stack and if the stack has N items or less, CentralStack points to **null**.
4. Immediate from the code on lines **23**–**32**.
5. Lines **21**–**32** imply that each cell that is assigned to a multi-op is a cell which was taken from the central stack. According to Lemma 8, all cells in the central stack were inserted by a successful push multi-op using `cMultiPush`.

■

Lemma 10 *Let op be a delegate push multi-op returning* **true** *in line* **42** *of* `cMultiPush`*, then each of the cells it inserts to the central stack will be assigned to at most a single pop operation.*

Proof sketch: Assume by way of contradiction that more than two pop operations, op_1 and op_2, were assigned the same cell which was pushed to the central stack. Without loss of generality we assume that op_1 performed the successful CAS on line **20** before op_2. According to Lemma 9 and the code, op_1 assigns this cell to one of the operations in its multiOp list, and accordingly CentralStack then points to a cell which is not one of the removed cells. When op_2 performs the successful CAS on line **20**, all the cells removed by op_2 are different from the cells removed by op_1, in contradiction to the assumption. ■

Definition 11 *We say that an operation is an* eliminated *multi-op if it returns* **true** *on line* **73** *or on line* **90** *or some other thread signals it FINISHED on line* **124** *or on line* **125***.*

Lemma 11 *Let op be an eliminated multi-op, then the following properties hold:*

- *If op is a pop, it obtains the value of a single eliminated push operation.*
- *If op is a push, its value is obtained by a single eliminated pop operation.*

Proof sketch: Let op_1 and op_2 be two delegate multi-ops where WLOG op_1 is an active colliding multi-op and op_2 is the passive colliding multi-op it collided with, and the two multi-ops have reverse operation semantics. We will begin by proving the lemma for the delegate operations themselves. If op_1 returns **true** on line **73**, then it is after it executed `multiEliminate` between multiOps' list of op_1 and op_2. In case op_1 is a push multi-op, op_2 executes `passiveCollide` and obtains the cell of op_1 on line **88**. If, however, op_1 is a pop multi-op, it obtains the cell of op_2 while executing the first iteration on lines **119–123**. Note that op_2 will return **true** on line **90**.
We now prove the elimination of the waiting multi-ops of op_1 and op_2. Let op'_1 be the M'th operation in op_1's multiOp list and op'_2 be the M'th operation in op_2's multiOp list. If op'_1 is a push operation then its value is obtained by op'_2 on line **122**. However, if op'_1 is a pop operation then it obtains op'_2's value on line **120**. Note that op_1 then signals both op'_1 and op'_2 FINISHED on lines **124–125**.

■

Lemma 12 *Let op_2 be an operation returning* **true** *in line* **94** *and let op_1 be the operation that signaled FINISHED to op_2, then:*

- *If op_1 is a pop operation then it obtains the value of a push operation or EMPTY_CELL if the stack was empty when its operation was performed.*
- *If op_1 is a push operation then at most a single pop operation returns its item.*

Proof sketch: Clearly from the code, op_1 could have signaled op_2 FINISHED only in line **10**, line **30**, line **39**, line **124** or line **125**. If op_1 signaled FINISHED in line **10** or line **30**, then op_1 is a pop operation bound to return **true** in line **13** or line **33** and by Lemma 9 it follows that op_2 is a pop operation obtaining the value of a single push operation.
If op_1 signaled FINISHED in line **39**, than op_1 is a push operation bound to return **true** in line **42**. By Lemma 5 op_2 is also a push operation and by Lemma 10 its item will be returned by at most one pop operation. Finally, if op_1 signals op_2 FINISHED in line **124** or line **125**, then by Lemma 11 the proof holds.

■

Theorem 1 *The Dynamic Elimination-Combining Stack has correct pool semantics.*

Proof sketch: A thread can finish its operation by returning **true** in one of the following lines: line **13**, line **33**, line **42**, line **56**, line **58** or line **64**. If a thread finishes its operation in line **13** or line **33**, then it is a pop operation performed on the central stack and, by Lemma 9, the operation is a correct pool operation. If a thread finishes its operation in line **42**, then it is a push operation performed on the central stack and, by Lemma 10, the operation is a correct pool operation.

If a thread finishes its operation in line **56**, then it is an active colliding multi-op that performed an elimination and, by Lemma 11, the operation is a correct pool operation. If a thread finishes its operation in

line **58** or line **64**, then it is a passive colliding multi-op that returned **true** either from line **90** or from line **94** in `passiveCollide`. If the thread return **true** in line **90** then its operation was performed by elimination and, by Lemma 11, it is a correct pool operation. On the other hand, if the thread returned **true** from line **94** then its operation was performed by combining and, by Lemma 12, it is also a correct pool operation. It follows from Definition 2 that DECS has correct pool semantics. ■

Linearizability

Lemma 13 *The linearization points of the DECS algorithm are as follows:*
All active operations are linearized in the following lines, executed in their last iteration of the `push` *(lines* **110–113***) or* `pop` *(lines* **102–105***) operation:*

- *Delegate thread pop operations are linearized in line* **7***, line* **20** *and in line* **67** *when the collision is an eliminating collision.*
- *Delegate thread push operations are linearized in line* **37** *and line* **67** *when the collision is an eliminating collision.*

Eliminating passive operations are linearized in the linearization point of their single matching active colliding multi-op, where push colliding multi-op are linearized before the pop operation.
Waiting operations (combining passive operations) are linearized in the linearization point of their single delegate operation, where:

- *If their delegate operation is a pop operation and its linearization point is line* **7** *or line* **20***, then all the corresponding waiting operations are linearized at that point, according to their order in the multiOp list.*
- *If their delegate operation is a push operation and its linearization point is line* **37***, then all the corresponding waiting operation are linearized at that point, in* reverse *order of the multiOp list.*
- *If their delegate operation linearization point is line* **67***, then the linearization point of each waiting operation occurs at the linearization point of corresponding delegate operation and push colliding multi-op are linearized before pop colliding multi-ops.*

Proof sketch: Line **7**, line **20**, line **37** and line **67** complete by modifying the central stack; in this case, the claim follows by a simple extension of the Treiber [12] algorithm correctness argument for multi-ops.

Next, we consider the linearization points for passive eliminating operations. In order to prove the claim for these operations, we need to prove that the linearization as defined in the lemma statement occurs during the operation's time interval. Let op_2 be a passive colliding multi-op and let op_1 be its matching active colliding multi-op. Assume by way of contradiction that op_2 terminated before the linearization point defined in the lemma statement. From Definitions 5 and 11, op_2 executed lines **84–90**. Specifically, op_2 writes **null** to its entry in the location array and finishes its operation whereas op_1 performs line **67** only later. From the condition of line **54**, it follows that the multiOp of op_2 must be non-**null** for op_1 to succeed in colliding with it. It follows that op_1 fails in the CAS performed in line **67**, in contradiction to Definition 5.

We now consider the linearization points defined for waiting operations. By Lemma 6, every waiting operation has exactly one delegate operation at any point of time, thus the linearization points are well-defined. By the definition of waiting operations, these operation did not yet terminate at their linearization times.

Finally, we consider the linearization points for operations that complete by elimination (these may be active colliding operations, passive colliding operations, or waiting operations). We already showed above that the linearization points for passive colliding operations and waiting operations are well defined. We are left to prove that correct LIFO ordering is maintained between any two linearized colliding operations and between these operations and operations that complete by modifying the central stack.

When linearizing a passive collider in the linearization point of its single active matching operation, no other operations can be linearized between these two operations. Moreover, since the push operation is linearized immediately before the pop operation, this is a legal LIFO ordering that cannot interfere with LIFO matchings of other collisions or with operations completed on the central stack. From Lemma 11, a pop operation obtains the value of the push operation it collides with.

Similar arguments establish the linearizability of the waiting operations in the combining lists of a pair of delegate operations that eliminate with each other. In this case, from Lemma 12, a waiting pop operation returns the value it obtained from the matching push operation. ■

Theorem 2 *The Dynamic Elimination-Combining Stack is a correct linearizable implementation of a stack object.*

Proof sketch: Immediate from Lemma 13.

■

Deadlock Freedom

Theorem 3 *The dynamic elimination-Combining stack algorithm is deadlock-free.*

Proof sketch: Let op be an operation. We show that if each thread is scheduled infinitely often, then in every iteration made by op, some operation is linearized.

If op is an active delegate operation, then it first tries to apply the operations in its multiOp list (which includes its own operation) to the central stack. If op succeeds, then its linearization point has occurred. Otherwise, it must be that op fails the CAS in line **20** or line **37** and this can only occur if another operation applied a successful CAS – and was therefore linearized – in the course of op's iteration.

If op is an active or a passive eliminating operation that succeeds in colliding, then its linearization point has already occurred and in line **90** it returns **true** and finishes its operation.

Otherwise, op is a waiting operation. Let op_1 be the single delegate multi-op such that when op_1 is linearized, it is the delegate multi-op of op. From Lemmma 13, op is linearized at the same time as op_1. More specifically, the linearization point of op_1 is followed by a release of all the waiting operations in its multiOp list. This is done by signaling them a FINISHED value. From Lemma 7, when op_1 is linearized, op is in its multiOp list. Thus, as op_1 continues taking steps after performing its linearization point, op is eventually released from its waiting at line **92** and finishes its operation. To conclude the proof, observe that in every iteration of op, either op's delegate thread or another delegate thread is linearized and will then release all of its waiting operation. ■

Appendix C: The Non-Blocking DECS algorithm

In Section 4, we provided a high-level description of the NB-DECS algorithm. In this section we provide a more detailed description of the differences in the pseudo-codes of DECS and NB-DECS. We only describe the modifications that need to be incorporated into the blocking DECS algorithm presented in Section 2.

In order to highlight the differences between the two algorithms, we marked modified or new pseudo-code lines with red color, and kept the original code written in black color. The headers of new functions and functions that were extensively modified are also marked by red color.

New structure fields added for NB-DECS are shown in Figure 8. An *invalid* flag was added to each *Cell*, indicating whether or not the cell was invalidated by a thread that stopped waiting. Two fields were added to to the `multiOp` structure: (1) *other* - which points to a multiOp structure of a thread with reverse semantics to indicate that an elimination between the two operations occurred; and (2) *invalid* - indicating whether the multiOp structure is still valid (was not invalidated by the waiting thread). *invalid* flags are set using test-and-set to avoid race conditions.

A key change as compared to DECS is that a waiting thread (i.e., a thread that delegates its operation to another thread) stops waiting for its operation to be executed after some period of time. That is, a waiting thread "gives-up" waiting for the delegate thread and retries its operation.

Each thread invokes a *push* or *pop* operation by executing the `push` and `pop` functions respectively (Figure 9). As in DECS, the threads iteratively attempt to apply their operation on the *CentralStack* structure and, if they fail, they access the elimination-combining layer. After accessing the elimination-combining layer, each thread verifies that its multiOp is still valid to use. If the multiOp is marked as invalid (and therefore cannot be used), the thread initializes a new multiOp record and re-attempts its operation on the *CentralStack*.

The code of the `cMultiPush` function is identical to that of the blocking version (Figure 2-(c)). The code of `cMultiPop` changed, however, and is presented in figure 10. A delegate thread which executes the `cMultiPop` operation attempts to execute M (where M is the length of its multiOp list) pop operations, one after the other. If the stack is empty, the delegate thread iterates over its multiOp list, sets the *cStatus* of the threads that are still waiting to FINISHED and their cell to an EMPTY_CELL (lines **167–173**). If the stack is non-empty, the delegate thread removes a single cell from the stack (line **176**) and verifies that this cell is a valid cell (line **177**). When the delegate thread successfully removes a cell from the top of the stack, it searches for a thread in its multiOp list that is still waiting (using the `delegatePop` function). This is required, since, at least theoretically, it is possible that all waiting processes 'gave up' and stopped waiting. If a waiting thread is not found, the removed cell is assigned to the delegate thread itself (lines **179–180**). If the CAS on line **176** fails or the cell is invalid (line **177**), the delegate thread continues and performs additional

Figure 8: Non Blocking DECS Data Structures - modifications

```
140 define Cell: struct { data: Data, next: Cell,
141                       invalid: boolean init false };
142 define multiOp: struct { id, op, length, cStatus: int, cell: Cell, next, last: multiOp;
143                       other: multiOp init null, invalid: boolean init false };
```

Figure 9: Non Blocking push and pop operations

```
Data pop()
144 multiOp mOp = initMultiOp();
145 while true do
146 |   if cMultiPop(mOp) then
147 |   |   return mOp.cell.data;
148 |   else if collide(mOp) then
149 |   |   return mOp.cell.data;
150 |   else if mOp.invalid=true then
151 |   |   mOp = initMultiOp();
152 |   end
153 end
```

```
push(Data: data)
154 multiOp mOp = initMultiOp(data);
155 while true do
156 |   if cMultiPush(mOp) then
157 |   |   return;
158 |   else if collide(mOp) then
159 |   |   return;
160 |   else if mOp.invalid=true then
161 |   |   mOp = initMultiOp(data);
162 |   end
163 end
```

pop operation if required. If no cell was obtained by the delegate thread for itself, the function returns *false*, and the delegate thread enters the elimination-collision layer (as described in the `pop` function).

The `delegatePop` function described in figure 11 receives a delegate thread's pop multiOp and a valid cell that was removed from the stack. The function searches for a waiting thread in the multiOp list and, if such a thread is found, assigns the cell to that waiting thread. The function starts by iterating over the multiOp list, one multiOp at a time, as long as no waiting multiOp is found. Once a waiting thread is found, the cell is assigned to it and the multiOp is removed from the delegate's multiOp list (lines **188**–**191**). In line **193**, the waiting thread's multiOp is checked. If it is still valid (i.e., the thread is still waiting), the `delegatePop` function returns *true*, indicating that a waiting thread which obtained the cell was found. Otherwise, if no waiting thread is found, the function returns *false*.

As in DECS, the `passiveColide` function (presented in figure 12) is invoked by a passive collider after it identifies that it was collided with. The passive collider first reads the multi-op pointer written to its entry in the *location* array by the active collider and initializes its entry in preparation for future operations (lines **198**–**199**). If the multi-ops of the colliding threads-pair are of reverse semantics (line **200**) then the function returns *true* in line **204** because, in this case, it is guaranteed that the colliding delegate threads exchange values. Specifically, if the passive thread's multi-op type is *pop*, the thread copies the cell communicated to it by the active collider (line **202**).

The algorithm in the non blocking version is different for the case where both multi-ops are of identical semantics (lines **205**–**226**). In this case, the passive collider's operations were delegated to the active thread until some time limit expires or the executing thread's *cStatus* is modified, as shown in line **206**. Upon terminating the bounded waiting, the passive collider's operation is either performed by the active thread or has yet to be performed. In the latter case, the *cStatus* field of the passive operation is still INIT, and the function calls the `wakeup` function in line **225** so that the passive thread will retry executing its operation while avoiding a race condition with the active collider.

If, on the other hand, the active collider updated the *cStatus* field, then the executing thread's operation

Figure 10: Non-Blocking central stack operations

```
   boolean cMultiPop(multiOp: mOp)
164 for i = 1 to mOp.length do
165     top = CentralStack;
166     if top = null then
167         repeat
168             mOp.cell = EMPTY_CELL;
169             mOp.cStatus = FINISHED;
170             testAndSet(mOp.invalid);
171             mOp = mOp.next;
172         until mOp = null ;
173         return true;
174     end
175     next = top.next;
176     if CAS(&CentralStack, top, next) then
177         if testAndSet(top.invalid)=false then
178             if delegatePop(mOp, top)=false then
179                 mOp.cell = top;
180                 return true;
181             end
182         end
183     end
184 end
185 return false;
```

is either: (1) executed on the central stack and finished, (2) eliminated with another operation, or (3) an eliminating operation was not found. In the first case, the operation was successfully executed on the central stack by a thread that invoked the `cMultiPush` or the `cMultiPop` functions, and the function returns *true* indicating the operation is terminated (line **210**). The second case, where the operation was eliminated with another operation is dealt with in lines **211–219**, The executing thread invokes a test-and-set operation on the invalid flag of the operation that was assigned to it. If the test-and-set succeeds, then the other operation is a valid operation, an elimination occurs and the function return *true* in line **214**. (Note that if the passive collider's operation is a pop, the passive collider obtains the cell of the other operation on line **213**.) If, on the other hand, the test-and-set fails – implying that the other thread stopped waiting (line **215**) – the executing thread resets its multiOp record fields and returns *false*. The third case, where *cStatus* is set to RETRY, is similar to the corresponding case in the DECS algorithm: The passive operation is notified that an eliminating operation was not found, and so it clears its invalid flag, resets its *cStatus* to INIT and returns *false* indicating that the operation was not executed yet.

Figure 13 shows the `multiEliminate` function which is called by an active collider with two multiOp operations of reverse semantics. As in DECS, the function iterates over the two lists of the given

Figure 11: A pop operation delegation assignment to a waiting thread

```
boolean delegatePop(multiOp: mOp, Cell: cell)
186 next = mOp.next;
187 while next ≠ null do
188 |   next.cell = cell;
189 |   mOp.next = next.next;
190 |   if mOp.next = null then mOp.last = mOp;
191 |   mOp.length = mOp.length-1;
192 |   next.cStatus = FINISHED;
193 |   if testAndSet(next.invalid) = false then
194 |   |   return true;
195 |   else next = mOp.next;
196 end
197 return false;
```

multi-ops on lines **227**–**238** until one of the lists has no more elements. Note that contrary to the blocking `multiEliminate` function, the *cStatus* of the operations is set to EXCHANGE (instead of FINISHED), and the cell is not obtained by the pop operation; instead, the *other* field is set to the matching operation's multiOp record. Setting the *cStatus* to EXCHANGE allows passive threads, upon terminating their bounded await (line **206**), to observe that their operation was eliminated and that the cell of the push operation can be obtained by the pop operation.

After iterating over both lists, one of the lists may contain more elements (recall the "residue" of a multiOp list described in section 2). If the active (passive) collider's multiOp list is longer than the list of the passive (active) collider, lines **239**–**250** (lines **250**–**260**) are executed on the residue list of the active (passive) collider. The delegate thread executing the `multiEliminate` function iterates over the active (passive) residue list, searching for the first operation of a waiting thread, setting its *cStatus* to RETRY after updating its multi-op list on lines **252**–**254** (lines **252**–**254**). Contrary to the blocking algorithm, in this case a thread may no longer be waiting. To cope with such a scenario, after setting *cStatus* to RETRY and updating the multiOp operations list, the delegate thread attempts to test-and-set the multiOp's invalid flag. If the test-and-set operation succeeds, the multiOp is of a thread that is still waiting and the function returns on line **244** (line **255**). If the test-and-set fails, then the thread has stopped waiting and resumed its operation, and the delegate thread updates the length of the multiOp list, and continues to the delegate thread's next multi-op of the active (passive) residue list on lines **246**–**247** (lines **257**–**258**).

The `wakeup` function presented in figure 14 is invoked by a passive collider thread after it gives up waiting. The function is given a pointer to the passive collider's multiOp record and resolves a possible race condition with the active collider. If the operation is a pop (lines **262**–**273**), the invalid flag of the operation is checked. If it is 0, indicating that the active collider did not perform the passive collider's operation, the function returns *false*. If the flag is set, however, then the *cStatus* of the passive collider is now set to another value by the active thread, and the executing thread continues similarly to the code of lines **208**–**224** of `passiveCollide`.

Figure 12: Non blocking passive collide

```
   boolean passiveCollide(multiOp: pInf)
198 aInf = location[pInf.id];
199 location[pInf.id] = null;
200 if pInf.op ≠ aInf.op then
201     if pInf.op = POP then
202         pInf.cell = aInf.cell;
203     end
204     return true;
205 else
206     bounded-await (pInf.cStatus ≠ INIT);
207     if pInf.cStatus ≠ INIT then
208         switch pInf.cStatus do
209             case FINISHED
210                 return true;
211             case EXCHANGE
212                 if testAndSet(pInf.other.invalid) = false then
213                     if pInf.op = POP then pInf.cell = pInf.other.cell;
214                     return true;
215                 else
216                     initFields(pInf);
217                     return false;
218                 end
219             case RETRY
220                 pInf.invalid = false;
221                 pInf.cStatus = INIT;
222                 return false;
223             end
224         end
225     else return wakeup(pInf);
226 end
```

Lines **274**–**283** are executed in case the operation is a push. In line **275**, two test-and-set operations are performed, one on the invalid flag of the passive operation's cell, and the other on the invalid flag of the operation structure itself. If both invalid flags are 0 (i.e., the cell and the multiOp record are both valid, and no other operation needs them), the function returns *false* (line **283**) indicating that the push operation should be restarted. Note that the invalid flag of the cell can only be set by another pop operation, after popping the cell from the central stack (line **177** in `cMultiPop`), and the operation's invalid flag is set when an elimination with another pop operation is done (line **212** in `passiveCollide`). In both cases, it is an indication that the push operation was executed (and its *cStatus* was updated). If the *cStatus* field is

Figure 13: Non blocking elimination of multi operations

```
multiEliminate(multiOp: aInf, pInf)
227 aCurr = aInf;
228 pCurr = pInf;
229 repeat
230     aCurr.other = pCurr;
231     pCurr.other = aCurr;
232     aCurr.cStatus = EXCHANGE;
233     pCurr.cStatus = EXCHANGE;
234     aInf.length = aInf.length - 1;
235     pInf.length = pInf.length - 1;
236     aCurr = aCurr.next;
237     pCurr = pCurr.next;
238 until aCurr ≠ null ∧ pCurr ≠ null ;
239 if aCurr ≠ null then
240     while aCurr ≠ null do
241         aCurr.length = aInf.length;
242         aCurr.last = aInf.last;
243         aCurr.cStatus = RETRY;
244         if testAndSet(aCurr.invalid)=false then return;
245         else
246             aInf.length = aInf.length - 1;
247             aCurr = aCurr.next;
248         end
249     end
250 else if pCurr ≠ null then
251     while pCurr ≠ null do
252         pCurr.length = pInf.length;
253         pCurr.last = pInf.last;
254         pCurr.cStatus = RETRY;
255         if testAndSet(pCurr.invalid)=false then return;
256         else
257             pInf.length = pInf.length - 1;
258             pCurr = pCurr.next;
259         end
260     end
261 end
```

set to RETRY, an elimination attempt was unsuccessful, and the executing thread resets its *cStatus*, clears its invalid flag and returns *false* to retry its push operation (lines **277–279**). Otherwise (the value of *cStatus*

Figure 14: A passive thread wakeup function

```
wakeup(multiOp: mOp)
262 if mOp.op = POP then
263     if testAndSet(mOp.invalid) = true then
264         if mOp.cStatus = RETRY then
265             mOp.invalid = false;
266             mOp.cStatus = INIT;
267             return false;
268         else
269             if mOp.cStatus ≠ null then
270                 mOp.cell = mOp.other.cell;
271             return true;
272         end
273     else return false;
274 else                                                    /* PUSH operation */
275     if testAndSet(mOp.cell.invalid) = true ∨ testAndSet(mOp.invalid) = true then
276         if mOp.cStatus = RETRY then
277             mOp.cStatus = INIT;
278             mOp.invalid = false;
279             return false;
280         else
281             return true;
282         end
283     else return false;
284 end
```

field is either EXCHANGE or FINISHED), the push operation was executed either by elimination or by inserting the element to the central stack, and *true* is returned in line **281**.