A Dynamic Elimination-Combining Stack Algorithm

Gal Bar-Nissan, Danny Hendler, and Adi Suissa[⋆]

Department of Computer Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel

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.

[⋆] Supported by a grant from the Lynne and William Frankel Center for Computer Sciences and by the *Israel Science Foundation* (grant number 1227/10).

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 back-off 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

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

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 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 Fig. 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 4 times its throughput.

For some applications, a nonblocking [6] stack may be preferable to a blocking one because nonblocking implementations are 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 is provided in Section 4. We conclude the paper in Section 5 with a short discussion of our results. Correctness proofs and the pseudo-code of the NB-DECS algorithm appear in the full paper.

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.

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

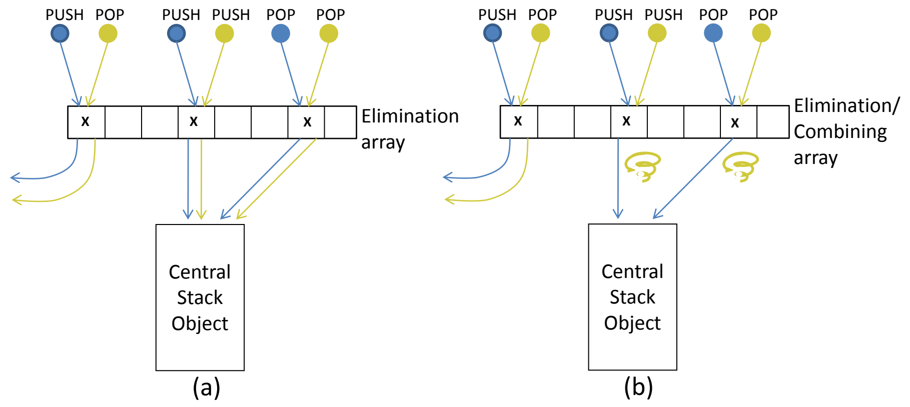


Fig. 1: Collision-attempt scenarios: (a) Collision scenarios in the elimination-backoff stack; (b) Collision scenarios in DECS.

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 Fig. 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.

Fig. 2: (a): Data structures, (b) and (c): DECS `Push` and `Pop` functions.

(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) **Data** `pop`()

```
6  multiOp mOp = initMultiOp();
7  while true do
8      if cMultiPop(mOp) then
9          return mOp.cell.data;
10     else if collide(mOp) then
11         return mOp.cell.data;
12     end
13 end
```

(c) `push`(Data: *data*)

```
14 multiOp mOp =
   initMultiOp(data);
15 while true do
16     if cMultiPush(mOp) then
17         return;
18     else if collide(mOp) then
19         return;
20     end
21 end
```

Figures 2-(b) and 2-(c) present 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 **6** (line **14**). It then attempts to apply the *pop* (*push*) operation to the central stack in line **8** (line **16**). If this attempt fails, the thread then attempts to apply its operation to the elimination-combining layer in line **10** (line **18**). A 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 **9**) or by way of elimination (line **11**).

Fig. 3: (a) and (b): Central stack operations.

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

```
22 top = CentralStack;
23 if top = null then
24     repeat
25         mOp.cell = EMPTY_CELL;
26         mOp.cStatus = FINISHED;
27         mOp=mOp.next;
28     until mOp = null ;
29     return true;
30 end
31 nTop = top.next;
32 m = 1;
33 while nTop ≠ null ∧ m < mOp.length do
34     nTop = nTop.next, m++;
35 end
36 if CAS(&CentralStack, top, nTop) then
37     mOp.cell = top;
38     top = top.next;
39     while mOp.next ≠ null do
40         if top = null then
41             mOp.next.cell = EMPTY_CELL;
42         else
43             mOp.next.cell = top;
44             top = top.next;
45         end
46         mOp.next.cStatus = FINISHED;
47         mOp.next = mOp.next.next;
48     end
49     return true;
50 else return false;
```

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

```
51 top = CentralStack;
52 mOp.last.cell.next=top;
53 if CAS(&CentralStack, top, mOp.cell) then
54     while mOp.next ≠ null do
55         mOp.next.cStatus = FINISHED;
56         mOp.next = mOp.next.next;
57     end
58     return true;
59 else
60     return false;
61 end
```

Central Stack Functions Figures 3-(a) and 3-(b) 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 **22**). If the stack is empty, then all the *pop* operations in the list are linearized in line **22** and will return an *empty* indication. In lines **25–30**, 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 **31**–**35**); 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 **36**). If the CAS fails, *false* is returned (line **50**) 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 **39**–**48**). It assigns the m cells that were popped from the central stack to the first m pop operations (line **43**) and assigns an EMPTY_CELL to the rest of the pop operations, if any (line **41**). It then sets the *cStatus* of all these operations to FINISHED (line **46**), signalling all waiting threads that their response is ready. The `cMultiPop` function then returns *true*, indicating that it was successful (line **49**).

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 **52**) and applies a *CAS* operation in an attempt to atomically chain the list to the central stack (line **53**). 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 **54**–**57**). It then returns *true* in line **58**, indicating its success. If the CAS fails, `cMultiPush` returns *false* (line **60**) indicating that the multi-push should now be applied to the elimination-combining layer.

Elimination-Combining Layer Functions The `collide` function, presented in Fig. 4, implements the elimination-combining backoff algorithm performed after a multi-op fails on the central stack.[2] 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 **62**) 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 **63**) and repeatedly attempts to swap the value in the corresponding entry with its own ID by using *CAS* (lines **64**–**66**).

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 **68**), 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

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

multiOp structure *oInfo* (line **69**) and then verifies that the other thread may still be collided with (line **70**).[3]

Fig. 4: The `collide` function.

```
(c) boolean collide(multiOp: $mOp$)
62 $location[id] = mOp$;
63 $index$ = randomIndex();
64 $him = collision[index]$;
65 while $CAS(\&collision[index], him, id)$=false do
66     $him = collision[index]$;
67 end
68 if $him \neq EMPTY$ then
69     $oInfo = location[him]$;
70     if $oInfo \neq NULL \wedge oInfo.id \neq id \wedge oInfo.id = him$ then
71         if $CAS(\&location[id], mOp, NULL)$=true then
72             return activeCollide($mOp$, $oInfo$);
73         else
74             return passiveCollide($mOp$);
75         end
76     end
77 end
78 wait();
79 if $CAS(\&location[id], mOp, NULL)$=false then
80     return passiveCollide(mOp);
81 end
82 return false;
```

If the tests of line **70** succeed, the delegate thread attempts to *deregister* by CAS-ing its *location* entry back to *NULL* (line **71**). If the CAS is successful, the thread calls the `activeCollide` function (line **72**) 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 **74**).

If the tests of line **70** fail, the thread attempts to become a passive collider and waits for a short period of time in line **78** 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 **80**) 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.

[3] Some of the tests of line **70** are required because *location* array entries are not re-initialized when operations terminate (for optimization reasons) and thus may contain outdated values.

Fig. 5: (a) The `activeCollide`, (b) `passiveCollide`, (c) `combine` and (d) `multiEliminate` functions.

```
(a) boolean activeCollide(multiOp: aInf, pInf)
83  if CAS(&location[pInf.id], pInf, aInf) then
84      if $aInf.op = pInf.op$ then
85          combine(aInf,pInf);
86          return false;
87      else
88          multiEliminate(aInf,pInf);
89          return true;
90      end
91  else
92      return false;
93  end
```

```
(b) boolean passiveCollide(multiOp: pInf)
94  $aInf = location[pInf.id]$;
95  $location[pInf.id] =$ null;
96  if $pInf.op \neq aInf.op$ then
97      if $pInf.op = POP$ then
98          $pInf.cell = aInf.cell$;
99      end
100     return true;
101 else
102     await($pInf.cStatus \neq$ INIT);
103     if $pInf.cStatus = FINISHED$ then
104         return true;
105     else
106         $pInf.cStatus =$ INIT;
107         return false;
108     end
109 end
```

```
(c) combine(multiOp: aInf, pInf)
110 if $aInf.op = PUSH$ then
111     $aInf.last.cell.next = pInf.cell$;
112 end
113 $aInf.last.next = pInf$;
114 $aInf.last = pInf.last$;
115 $aInf.length = aInf.length + pInf.length$;
```

```
(d) 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 $\vee$ $pCurr =$ null ;
131 if $aCurr \neq$ null then
132     $aCurr.length = aInf.length$;
133     $aCurr.last = aInf.last$;
134     $aCurr.cStatus =$ RETRY;
135 else if $pCurr \neq$ null then
136     $pCurr.length = pInf.length$;
137     $pCurr.last = pInf.last$;
138     $pCurr.cStatus =$ RETRY;
139 end
```

The `activeCollide` function (Figure 5-(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 **83**. If the CAS fails then the passive collider is no longer eligible for collision and the function returns *false* (line **92**), 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 **84**) and calls either the `combine` or the `multiEliminate` function, depending on whether the multi-ops have identical or reverse semantics, respectively (lines **84**–**89**). 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 5-(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 **94**–**95**). If the multi-ops of the colliding threads-pair are of reverse semantics (line **96**) then the function returns *true* in line **100** 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 **98**).

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: (1) *cStatus = FINISHED* holds in line **103**. In this case, the thread's operation response is ready and it returns *true* in line **104**. (2) *cStatus = RETRY* holds (line **105**) 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 **106**) and returns *false*, indicating that the operation should be retried on the central stack.

The `combine` function (Figure 5-(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 **113**–**115**). In addition, if the type of both multi-ops is *push*, then their cell-lists are also concatenated (line **111**); this allows the delegate thread to push all its operations to the central stack by using a single *CAS* operation.

The `multiEliminate` function (Figure 5-(d)) 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.

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. According to common practice, 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.

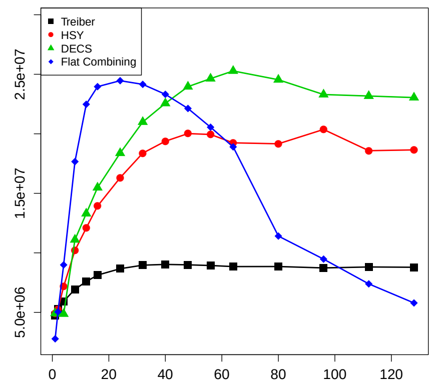
We compare DECS with the Treiber stack[4] and with the most effective known stack implementations: the HSY elimination-backoff stack, and a flat-combining based stack.[5] [6]

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 number of threads 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.

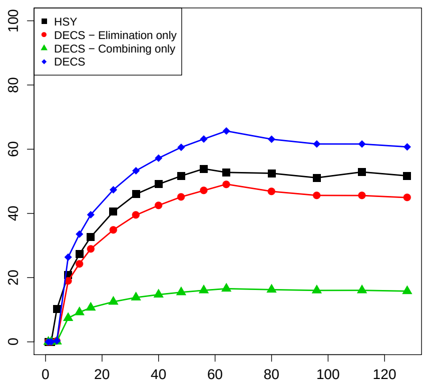
[4] We evaluated two variants of the Treiber algorithm - with and without exponential backoff. The variant using exponential backoff performed consistently better and is the version we compare with.

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

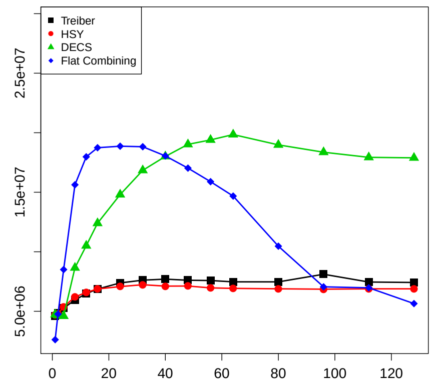
[6] The Treiber, HSY and DECS algorithms need to cope with the "ABA problem" [8], since they use dynamic-memory structures that may need to be recycled and perform CAS operations on pointers to these structures. We implemented the simplest and most common ABA-prevention technique that includes a tag 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].



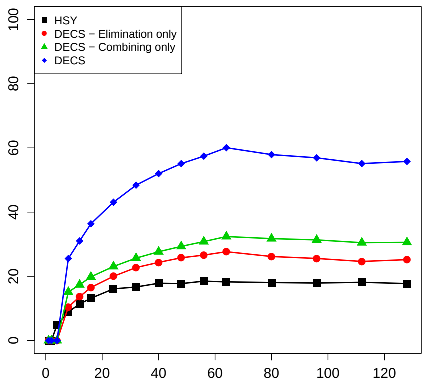
(a) Throughput: 50% push, 50% pop



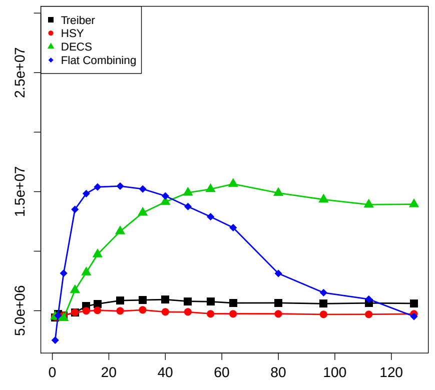
(d) Collision success (%): 50% push, 50% pop



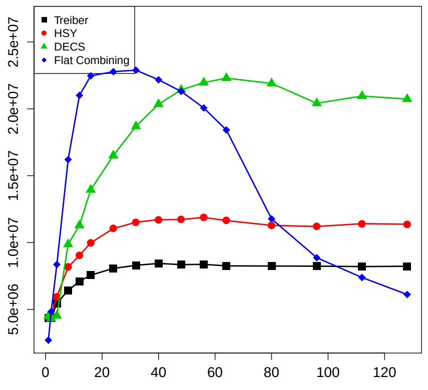
(b) Throughput: 25% push, 75% pop



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



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



(f) Throughput, phased execution: 75%, 50%, 25% push

Fig. 6: Throughput and collision success rates. X-axis: threads #; Y-axis in (a)-(c), (f): throughput.

Figures 6-(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 Fig. 6-(a), even here the HSY stack is outperformed by DECS by a margin of up to 31% (when the number of 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 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 6-(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.[7] The curves titled "Elimination only" and "Combining only" show a finer partition of the DECS operations ac-

[7] 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.

cording 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 6-(b) and 6-(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 Fig. 6-(e), the ratio of DECS operations that complete via elimination is significantly reduced for the 25% push workload. This ratio drops to 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 6-(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 6-(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.

The asymmetric workloads we considered above serve for highlighting the performance tradeoffs between the algorithms we evaluate, as a function of the ratio of *push* and *pop* operations in the workload. A more realistic scenario is when the stack goes through phases, in each of which it is initially filled with items, then it is accessed by a symmetric workload and finally it is being emptied. Figure 6-(f) compares throughput for a phased execution that lasts 3 seconds. In the first second, workload consists of 75% push operations. In the following second, workload is symmetric. Finally, in the third second, the workload consists of 75% pop operations. It can be seen that the performance tradeoffs between the evaluated algorithms are the same as for the non-phased workloads. DECS is consistently better than the HSY stack by a margin of up to 94% (for 80 threads). It is outperformed by FC by a margin of up to 86% (for 12 threads) but, due to its better scalability, provides 21% higher performance for 64 threads and outperforms FC by a factor of approximately 3.4 for 128 threads. The picture is similar for fully asymmetric phased workloads.

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 NB-DECS algorithm appears in the full paper, where we also present a comparative evaluation of this new algorithm.

Recall that waiting threads await a signal from their delegate thread in the `passiveCollide` function (line **102** in Fig. 5-(b)). 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.

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. However, NB-DECS provides better performance than the HSY stack for all workload types, providing up to 2 times the throughput for push-dominated workloads.

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 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.