# A Scalable Lock-free Stack Algorithm

Danny Hendler
*Ben-Gurion University*

Nir Shavit
*Tel-Aviv University*

Lena Yerushalmi
*Tel-Aviv University*

---

The literature describes two high performance concurrent stack algorithms based on combining funnels and elimination trees. Unfortunately, the funnels are linearizable but blocking, and the elimination trees are non-blocking but not linearizable. Neither is used in practice since they perform well only at exceptionally high loads. The literature also describes a simple lock-free linearizable stack algorithm that works at low loads but does not scale as the load increases. The question of designing a stack algorithm that is non-blocking, linearizable, and scales well throughout the concurrency range, has thus remained open.

This paper presents such a concurrent stack algorithm. It is based on the following simple observation: that a single elimination array used as a backoff scheme for a simple lock-free stack is lock-free, linearizable, and scalable. As our empirical results show, the resulting *elimination-backoff stack* performs as well as the simple stack at low loads, and increasingly outperforms all other methods (lock-based and non-blocking) as concurrency increases. We believe its simplicity and scalability make it a viable practical alternative to existing constructions for implementing concurrent stacks.

---

[0]A preliminary version of this paper appeared in the proceedings of the *16th Annual ACM Symposium on Parallelism in Algorithms*, Barcelona, Spain, 2004, pages 206-215.

# 1. INTRODUCTION

Shared stacks are widely used in parallel applications and operating systems. As shown in [28], LIFO-based scheduling not only reduces excessive task creation, but also prevents threads from attempting to dequeue and execute a task which depends on the results of other tasks. A concurrent shared stack is a data structure that supports the usual `push` and `pop` operations with linearizable LIFO semantics. Linearizability [16] guarantees that operations appear atomic and can be combined with other operations in a modular way.

When threads running a parallel application on a shared memory machine access the shared stack object simultaneously, a synchronization protocol must be used to ensure correctness. It is well known that concurrent access to a single object by many threads can lead to a degradation in performance [1; 14]. Therefore, in addition to correctness, synchronization methods should offer efficiency in terms of scalability, and robustness [12] in the face of scheduling constraints. Scalability at high loads should not, however, come at the price of good performance in the more common low contention cases.

Unfortunately, the two known methods for parallelizing shared stacks do not meet these criteria. The combining funnels of Shavit and Zemach [27] are linearizable LIFO stacks that offer scalability through combining, but perform poorly at low loads because of the combining overhead. They are also blocking and thus not robust in the face of scheduling constraints [18]. The elimination trees of Shavit and Touitou [24] are non-blocking and thus robust, but the stack they provide is not linearizable, and it too has large overheads that cause it to perform poorly at low loads. On the other hand, the results of Michael and Scott [21] show that the best known low load method, the simple linearizable lock-free stack introduced by IBM [17] (a variant of which was later presented by Treiber [29]), scales poorly due to contention and an inherent sequential bottleneck.

This paper presents the *elimination backoff stack*, a new concurrent stack algorithm that overcomes the combined drawbacks of all the above methods. The algorithm is linearizable and thus easy to modularly combine with other algorithms; it is lock-free and hence robust; it is parallel and hence scalable; and it utilizes its parallelization construct adaptively, which allows it to perform well at low loads. The *elimination backoff stack* is based on the following simple observation: that a single elimination array [24], used as a backoff scheme for a lock-free stack [17], is both lock-free and linearizable. The introduction of elimination into the backoff process serves a dual purpose of adding parallelism and reducing contention, which, as our empirical results show, allows the *elimination-backoff stack* to outperform all algorithms in the literature at both high and low loads.

We believe its simplicity and scalability make it a viable practical alternative to existing constructions for implementing concurrent stacks.

## 1.1 Background

Generally, algorithms for concurrent data structures fall into two categories: blocking and non-blocking. There are several lock-based concurrent stack implementations in the literature. Typically, lock-based stack algorithms are expected to offer limited robustness as they are susceptible to long delays and priority inversions [10].

The first non-blocking implementation of a concurrent list-based stack appeared in the IBM System 370 principles of operation manual in 1983 [17] and used the double-width compare-and-swap (CAS) primitive. A variant of that algorithm in

which push operations use unary CAS instead of double-width compare-and-swap appeared in a report by Treiber in 1986 [29]. We henceforth refer to that algorithm as the IBM/Treiber algorithm. The IMB/Treiber algorithm represented a stack as a singly-linked list with a top pointer and used (either unary or double) compare-and-swap (CAS) to modify the value of the top atomically. No performance results were reported by Treiber for his non-blocking stack. Michael and Scott in [21] compared the IBM/Treiber stack to an optimized non-blocking algorithm based on Herlihy's general methodology [13], and to lock-based stacks. They showed that the IBM/Treiber algorithm yields the best overall performance, and that the performance gap increases as the amount of multiprogramming in the system increases. However, from their performance data it is clear that because of its inherent sequential bottleneck, the IBM/Treiber stack offers little scalability.

Shavit and Touitou [24] introduced elimination trees, scalable tree like data structures that behave "almost" like stacks. Their elimination technique (which we will elaborate on shortly as it is key to our new algorithm) allows highly distributed coupling and execution of operations with reverse semantics like the pushes and pops on a stack. Elimination trees are lock-free, but not linearizable. In a similar fashion, Shavit and Zemach introduced combining funnels [27], and used them to provide scalable stack implementations. Combining funnels employ both combining [8; 9] and elimination [24] to provide scalability. They improve on elimination trees by being linerarizable, but unfortunately they are blocking. As noted earlier, both [24] and [27] are directed at high-end scalability, resulting in overheads which severely hinder their performance under low loads.

The question of designing a practical lock-free linearizable concurrent stack that will perform well at both high and low loads has thus remained open.

### 1.2 The New Algorithm

Consider the following simple observation due to Shavit and Touitou [24]: if a `push` followed by a `pop` are performed on a stack, the data structure's state does not change. This means that if one can cause pairs of pushes and pops to meet and pair up in separate locations, the threads can exchange values without having to touch a centralized structure since they have "eliminated" each other's effect on it. Elimination can be implemented by using a collision array in which threads pick random locations in order to try and collide. Pairs of threads that "collide" in some location run through a lock-free synchronization protocol, and all such disjoint collisions can be performed in parallel. If a thread has not met another in the selected location or if it met a thread with an operation that cannot be eliminated (such as two `push` operations), an alternative scheme must be used. In the elimination trees of [24], the idea is to build a tree of elimination arrays and use the diffracting tree paradigm of Shavit and Zemach [26] to deal with non-eliminated operations. However, as we noted, the overhead of such mechanisms is high, and they are not linearizable.

The new idea (see Figure 1) in this paper is strikingly simple: use a single elimination array as a backoff scheme on a shared lock-free stack. If the threads fail on the stack, they attempt to eliminate on the array, and if they fail in eliminating, they attempt to access the stack again and so on. The surprising result is that this structure is linearizable: any operation on the shared stack can be linearized at the access point, and any pair of eliminated operations can be linearized when they meet.

Because it is a backoff scheme, it delivers the same performance as the simple

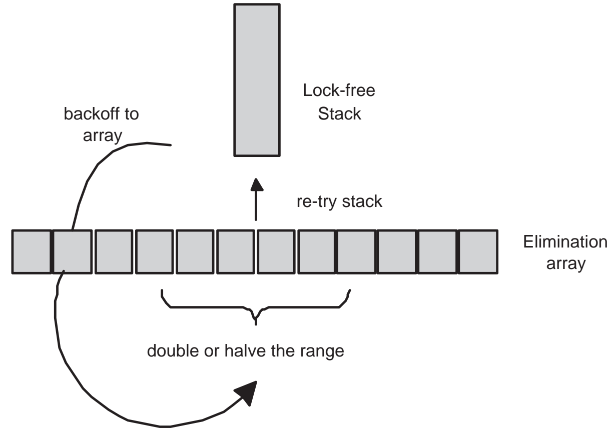


Fig. 1. Schematic depiction of the elimination-backoff cycle.

stack at low loads. However, unlike the simple stack it scales well as load increases because: (1) the number of successful eliminations grows, allowing many operations to complete in parallel; and (2) contention on the head of the shared stack is reduced beyond levels achievable by the best exponential backoff schemes [1] since scores of backed off operations are eliminated in the array and *never* re-attempt to access the shared structure.

### 1.3 Performance

We compared our new *elimination-backoff stack* algorithm to a lock-based implementation using Mellor-Crummey and Scott's MCS-lock [19] and to several non-blocking implementations: the linearizable IBM/Treiber [17; 29] algorithm with and without backoff, and the elimination tree of Shavit and Touitou [24]. Our comparisons were based on a collection of synthetic microbenchmarks executed on a 14-node shared memory machine. Our results, presented in Section 4, show that the elimination-backoff stack outperforms all three methods, and specifically the two lock-free methods, exhibiting almost three times the throughput at peak load. Unlike the other methods, it maintains constant latency throughout the concurrency range, and performs well also in experiments with unequal ratios of `push`s and `pop`s.

The remainder of this paper is organized as follows. In the next section we describe the new algorithm in depth. In Section 3, we give the sketch of adaptive strategies we used in our implementation. In Section 4, we present our empirical results. In Section 5, we provide a proof that our algorithm has the required properties of a stack, is linearizable, and lock-free. We conclude with a discussion in Section 6.

## 2. THE ELIMINATION BACKOFF STACK

### 2.1 Data Structures

We now present our elimination backoff stack algorithm. Figure 2 specifies some type definitions and global variables.

```
struct Cell {
    Cell *pnext;
    void *pdata;
};

struct Simple_Stack {
    Cell *ptop;
};

struct AdaptParams {
    int count;
    float factor;
};

 struct ThreadInfo {
    u_int id;
    char  op;
    Cell  *cell;
    AdaptParams *adapt;
 };

Simple_Stack S;
void **location;
int *collision;
```

Fig. 2. Types and Structures

Our central stack object follows IBM/Treiber [17; 29] and is implemented as a singly-linked list with a top pointer. The elimination layer follows Shavit and Touitou [24] and is built of two arrays: a global `location`[1..n] array has an element per thread $t \in \{1..n\}$, holding the pointer to the ThreadInfo structure, and a global `collision`[1..size] array, that holds the IDs of the threads trying to collide. The elements of both these arrays are initialized to *NULL*. Each ThreadInfo record contains the thread id, the type of the operation to be performed by the thread (`push` or `pop`), a pointer to the cell for the operation, and a pointer to the `adapt` structure that is used for dynamic adaptation of the algorithm's behavior (see Section 3).

## 2.2 Elimination Backoff Stack Code

We now provide the code of our algorithm. It is shown in Figures 3 and 4. As can be seen from the code, first each thread tries to perform its operation on the central stack object (line P1). If this attempt fails, a thread goes through the collision layer in the manner described below.

Initially, thread $t$ announces its arrival at the collision layer by writing its current information to the `location` array (line S2). It then chooses a random location in the `collision` array (line S3). Thread $t$ reads into `him` the id of the thread written at `collision[pos]` and tries to write its own id in place (lines S4 and S5). If it fails, it retries until it succeeds (lines S5 and S6).

After that, there are three main scenarios for thread actions, according to the information the thread has read. They are illustrated in Figure 5. If $t$ reads an id of another thread $q$ (i.e., `him!=EMPTY`), $t$ attempts to collide with $q$. The collision

```
void StackOp(ThreadInfo* p) {
P1: if(TryPerformStackOp(p)==FALSE)
P2:   LesOP(p);
P3: return;
}
void LesOP(ThreadInfo *p) {
S1: while (1) {
S2:   location[mypid]=p;
S3:   pos=GetPosition(p);
S4:   him=collision[pos];
S5:   while(!CAS(&collision[pos],him,mypid))
S6:     him=collision[pos];
S7:   if (him!=EMPTY) {
S8:     q=location[him];
S9:     if(q!=NULL&&q->id==him&&q->op!=p->op) {
S10:      if(CAS(&location[mypid],p,NULL)) {
S11:        if(TryCollision(p,q)==TRUE)
S12:          return;
S13:        else
S14:          goto stack;
          }
S15:      else {
S16:        FinishCollision(p);
S17:        return
      }  }  }
S18:  delay(spin);
S19:  AdaptWidth(SHRINK);
S20:  if (!CAS(&location[mypid],p,NULL)) {
S21:      FinishCollision(p);
S22:      return;
      }
stack:
S23:  if (TryPerformStackOp(p)==TRUE)
S24:      return;
      }
}
```

```
boolean TryPerformStackOp(ThreadInfo* p){
    Cell *ptop,*pnxt;
T1: if(p->op==PUSH) {
T2:    ptop=S.ptop;
T3:    p->cell->pnext=ptop;
T4:    if(CAS(&S.ptop,ptop,p->cell))
T5:      return TRUE;
T6:    else
T7:      return FALSE; }
T8: if(p->op==POP) {
T9:    ptop=S.ptop;
T10:   if(ptop==NULL) {
T11:     p->cell=EMPTY;
T12:     return TRUE;
       }
T13:   pnxt=ptop->pnext;
T14:   if(CAS(&S.ptop,ptop,pnxt)) {
T15:      p->cell=ptop;
T16:      return TRUE;
      }
T17:   else
T18:      return FALSE; }
}
```

Fig. 3. Elimination Backoff Stack Code - part 1

is accomplished by $t$ first executing a read operation (line S8) to determine the type of the thread being collided with. As two threads can collide only if they have opposing operations, if `q` has the same operation as $t$, $t$ waits for another collision (line S18). If no other thread collides with `t` during its waiting period, $t$ calls the $A$daptWidth procedure (line S19) that dynamically changes the width of $t$'s collision layer according to the perceived contention level (see Section 3). Thread $t$ then clears its entry in the `location` array (line S20), and tries once again to perform its operation on the central stack object (line S23). If $p$'s entry cannot be cleared, it follows that $t$ has been collided with, in which case $t$ completes its operation and returns.

If `q` does have a complementary operation, $t$ tries to eliminate by performing two CAS operations on the `location` array. The first (line S10) clears $t$'s entry, assuring no other thread will collide with it during its collision attempt (this eliminates race

```
void TryCollision(ThreadInfo*p,ThreadInfo *q) {
C1:  if(p->op==PUSH) {
C2:    if(CAS(&location[him],q,p))
C3:      return TRUE;
C4:    else
         {
C5:      adaptWidth(ENLARGE)
C6:      return FALSE;
         }
     }
C7:  if(p->op==POP) {
C8:    if(CAS(&location[him],q,NULL)){
C9:      p->cell=q->cell;
C10:     location[mypid]=NULL;
C11:     return TRUE
       }
C12:   else
         {
C13:     adaptWidth(ENLARGE)
C14:     return FALSE;
         }
     }
}

void FinishCollision(ThreadInfo *p) {
F1: if (p->op==POP_OP) {
F2:     p->cell=location[mypid]->cell;
F3:     location[mypid]=NULL;
    }
}
```

Fig. 4. Elimination Backoff Stack Code - part 2

conditions). The second (lines C2 or C8, see Figure 4) attempts to mark $q$'s entry as "collided with $t$". If both CAS operations succeed, the collision is successful. Therefore $t$ can return (in case of a `pop` operation it stores the value of the popped cell).

If the first CAS fails, it follows that some other thread has already managed to collide with $t$. In this case, thread $t$ acts as in case of a successful collision, mentioned above. If the first CAS succeeds but the second fails, then the thread with which $t$ is trying to collide is no longer available for collision. In that case, $t$ calls the $A$daptWidth procedure (lines C5 or C13, see section 3), and then tries once more to perform its operation on the central stack object; $t$ returns in case of success, and repeatedly goes through the collision layer in case of failure.

### 2.3 Memory Management and ABA Issues

As our algorithm is based on the compare-and-swap (CAS) operation, it must deal with the "ABA problem" [17]: if a thread reads the top of the stack, computes a new value, and then attempts a CAS on the top of the stack, the CAS may succeed when it should not, if between the read and the CAS some other thread changes the value to the previous one again. Similar scenarios are possible with CAS operations

that access the `location` array.

Since the only dynamic-memory structures used by our algorithms are Cell and ThreadInfo structures, the ABA problem can be prevented by making sure that no Cell or ThreadInfo structure is recycled (freed and returned to a pool so that it can be used again) while a thread performing a stack operation holds a pointer to it which it will later access.

Some runtime environments, such as that provided by Java®, implement automatic mechanisms for dynamic memory reclamation. Such environments seamlessly prevent ABA problems in algorithms such as ours. In environments that do not implement such mechanisms, the simplest and most common ABA-prevention technique is to include a tag with the target memory location, such that both the application value and the tag are manipulated together atomically, and the tag is incremented with each update of the target memory location [17]. The CAS operation is sufficient for such manipulation, as most current architectures that support CAS (Intel x86, Sun SPARC®) support their operation on aligned 64-bit blocks. One can also use general techniques to eliminate ABA issues through memory management schemes such as Safe Memory Reclamation (SMR) [20] or ROP [15].

We now provide a brief description of the SMR technique and then describe in detail how it can be used to eliminate ABA issues in our algorithm.

2.3.1 *ABA Elimination Using SMR.* The SMR technique [20] uses *hazard pointers* for implementing safe memory reclamation for lock-free recycable structures. Hazard pointers are single-writer multi-reader registers. The key idea is to associate a (typically small) number of hazard pointers with each thread that intends to access lock-free recycable structures. A hazard pointer is either null or points to a structure that may be accessed later by that thread without further validation that the structure was not recycled in the interim; such an access is called a *hazardous reference.*

Hazard pointers are used in the following manner. Before making an hazardous reference, the address of the structure about to be referenced is written to an available hazard pointer; this assignment announces to other threads that the structure about to be accessed must not be recycled. Then, the thread must validate that the structure about to be accessed is still *safe*, that is, that it is still logically part of the algorithm's data-structure. If this is not the case, then the access is not made; otherwise, the thread is allowed to access the structure. After the hazardous access is made, the thread can safely nullify the hazard pointer or re-use it to protect against additional hazardous references.

When a thread logically removes a structure from the algorithm's data-structure, it calls the *RetireNode* method with a pointer to that structure. The structure will not be recycled, however, until after no hazard pointers are pointing to it. The *RetireNode* method scans a list of structures that were previously removed from the data-structure and only recycles structures that are no longer pointed at by hazard pointers. For more details, please refer to [20].

The SMR technique can be used by our algorithm as follows. Each thread maintains two pools: a pool of Cell structures and a pool of ThreadInfo structures. In addition, a single hazard pointer is allocated per thread. We now describe the changes that are introduced to the code of Figures 3, 4 for ensuring safe memory reclamation.

It is easily seen that the only hazardous references that exist in the code are CAS

operations that attempt to swap structure pointers. Specifically, these are lines T4, T14, S10, S20, C2 and C8. This is how these hazardous references are dealt with.

(1) Line T4: Immediately after line T2, the executing thread's hazard pointer is set to point to *ptop*. Then, the condition `S.ptop==ptop` is checked. If the condition is not satisfied, the procedure returns FALSE, since the CAS of line T4 must fail. Otherwise the code proceeds as in the pseudo-code of Figure 3. It is now ensured that the structure pointed at by *ptop* cannot be recycled before the CAS of T4 is performed.

(2) Line T14: Immediately before line T13, the executing thread's hazard pointer is set to point to *ptop*. Then, the condition `S.ptop==ptop` is checked. If the condition is not satisfied, the procedure returns FALSE, since the CAS of line T14 must fail. Otherwise the code proceeds as in the pseudo-code of Figure 3. It is now ensured that the structure pointed at by *ptop* cannot be recycled before the CAS of T14 is performed.

(3) Line S10: Although the CAS of line S10 is technically a hazardous reference, there is no need to protect it: if the ThreadInfo structure pointed by $p$ is recycled between when the executing thread $t$ starts performing the `LesOP` procedure and when it performs the CAS of line S10, then it must be that another operation collided with this operation, changed the corresponding value of the location array to a value other than $p$, and nullified the entry of the collision array where $t$'s ID was written. Since no other `LesOP` operation can write $t$'s ID again to the collision array before $t$ performs the CAS of line S10, no operation will write to the entry of the *location* array corresponding to $t$ before $t$ executes the CAS of line S10. Hence this CAS will fail.

(4) Line S20: By exactly the same rationale applied in the case of line S10, there is no need to protect line S20.

(5) Lines C2, C8: Immediately after line S8, the executing thread's hazard pointer is set to point to $q$. Then, the condition `q==location[him]` is checked. If the condition is not satisfied, the collision attempt has failed and a `goto stack` command is executed. Otherwise the procedure proceeds as in the pseudo-code of Figure 3. We are now ensured that the structure pointed at by $q$ cannot be recycled before the CAS of line C2 or C8 is performed.

The following additional changes are introduced in the code of Figures 3, 4 for correct memory recycling.

(6) Before starting a *push* operation, a ThreadInfo structure $i$ is removed from the ThreadInfo pool and a Cell structure $c$ is removed from the Cell pool. Both are initialized, and *i->cell* is initialized to point to $c$.

(7) After successfully pushing a Cell to the central stack in line T4 and before returning, the ThreadInfo structure used by the operation is initialized and returned to the ThreadInfo pool from which it originated. There is no need to call the *RetireNode* method here, since at this point the executing thread's ThreadInfo structure is not accessible for collisions.

(8) After returning from a *pop* operation on an empty stack (line T12), the ThreadInfo and Cell structures used by the *pop* operation are initialized and returned to the respective pools from which they originated. There is no need to call the *RetireNode* method here, since at this point the ThreadInfo structure is not accessible for collisions and hence also the Cell structure will not be accessed.

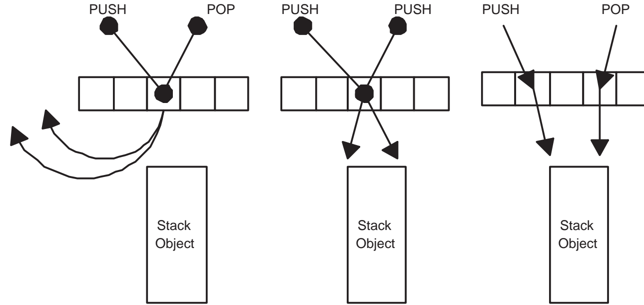


Fig. 5. Collision scenarios

(9) After completing a *pop* operation that returned a cell (in line S12, S17 or T16), and upon consuming the application value pointed at from that cell, The *RetireNode* method is called to indicate that both the ThreadInfo structure used by the *pop* operation and the Cell structure it returned are now logically not part of the data-structure.

From [20], Theorem 1, we get:

LEMMA 1. *The following holds for all algorithm lines that perform a CAS operation to swap a structure pointer (lines T4, T14, S10, S20, C2 and C8): a structure is not recycled between when its old value is read and when the CAS operation using this old value is performed.*

Quoting [14]: "the expected amortized time complexity of processing each retired node until it is eligible for reuse is constant". Thus the expected effect of the SMR technique on performance is small. Specifically, in the absence of contention, time complexity remains constant.

## 3. ADAPTATIVE ELIMINATION BACKOFF

The classical approach to handling load is backoff, and specifically exponential backoff [1]. In a regular backoff scheme, once contention in detected on the central stack, threads back off in time. Here, threads will back off in both *time* and *space*, in an attempt to both reduce the load on the centralized data structure and to increase the probability of concurrent colliding. Our backoff parameters are thus the width of the collision layer, and the delay at the layer.

The elimination backoff stack has a simple structure that naturally fits with a localized *adaptive* policy for setting parameters similar to the strategy used by Shavit and Zemach for combining funnels in [27]. Decisions on parameters are made locally by each thread, and the collision layer does not actually grow or shrink. Instead, each thread independently chooses a sub-range of the collision layer it will map into, centered around the middle of the array, and limited by the maximal array width. It is possible for threads to have different ideas about the collision layer's width, and particularly bad scenarios might theoretically lead to bad performance, but as we will show, in practice the overall performance is superior to that of exponential backoff schemes [1]. Our policy is to first attempt to access the central stack object, and only if that fails to back off to the elimination

array. This allows us, in case of low loads, to avoid the collision array altogether, thus achieving the latency of a simple stack (in comparison with this, [27] are at best three times slower than a simple stack).

Our algorithm adaptively changes the width of the collision layer in the following way. Each thread $t$ keeps a local variable called `factor`, $0 <$`factor`$\leq 1$, by which it multiplies the collision layer width to choose the interval into which it will randomly map to try and collide (e.g., if `factor`=0.5 only half the width is used). The dynamic adaptation of the collision layer's width is performed by the `AdaptWidth` procedure (see Figure 6). If a thread $t$ fails to collide because it did not encounter any other thread, then it calls the `AdaptWidth` procedure with the `SHRINK` parameter to indicate this situation (line S19). In this case, the `AdaptWidth` procedure decrements a counter local to thread $t$. If the counter reaches 0, it is reset to its initial value (line A5), and the width of $t$'s collision layer is being halved (line A6). The width of the collision layer is not allowed to decrease bellow the `MIN_FACTOR` parameter. The rationale of shrinking the collision layer's width is the following. If $t$ fails to perform its operation on the central stack object, but does not encounter other threads to collide with, then $t$'s collision layer's width should eventually be decreased so as to increase the probability of a successful collision.

In a symmetric manner, if a thread $t$ does encounter another thread, but fails to collide with it because of contention, then $t$ calls the `AdaptWidth` procedure with the `ENLARGE` parameter to indicate this situation (lines C5, C13). In this case, the `AdaptWidth` procedure increments the local counter. If it reaches the value `MAX_COUNT`, it is reset to its initial value (line A10), and the width of $t$'s collision layer is being doubled (line A11). The width of the collision layer is not allowed to surpass the `MAX_FACTOR` parameter. The rational of enlarging the collision layer's width is the following. If $t$ fails to collide with some thread $u$ because $u$ collides with some other thread $v$, then the width of $t$'s collision layer should eventually be enlarged so as to decrease the contention on $t$'s collision layer and increase the probability of a successful collision. Finally, the `GetPosition` procedure, called on line S3, uses the value of the thread's `factor` variable to compute the current width of its collision layer.

```
void AdaptWidth(enum direction) {
A1:  if (direction==SHRINK)
A2:    if (p->adapt->count > 0)
A3:      p->adapt->count--
A4:    else {
A5:      p->adapt->count=ADAPT_INIT
A6:      p->adapt->factor=max(p->adapt->factor/2, MIN_FACTOR);
         }
A7:  else if (p->adapt->count < MAX_COUNT)
A8:    p->adapt->count++;
A9:  else {
A10:   p->adapt->count=ADAPT_INIT;
A11:   p->adapt->factor=min(2*p->adapt->factor, MAX_FACTOR);
       }
}
```

Fig. 6. Pseudo-code of the `AdaptWidth` procedure. This procedure dynamically changes the width of a thread's collision layer according to the level of contention it encounters when trying to collide.

The second part of our strategy is the dynamic update of the delay time for attempting to collide in the array, a technique used by Shavit and Zemach for diffracting trees in [25; 26]. This is being done by the `delay` function called at line S18, which simply implements exponential backoff. Note, that whereas the changes in the collision layer width are kept between invocations of `StackOp`, the updates to the delay time are internal to `StackOp`, and so the delay time is reset to its default value whenever `StackOp` is called.

There are obviously other conceivable ways of adaptively updating these two parameters and this is a subject for further research.

## 4. PERFORMANCE

We evaluated the performance of our *elimination-backoff stack* algorithm relative to other known methods by running a collection of synthetic benchmarks on a 14 node Sun Enterprise™ E6500, an SMP machine formed from 7 boards of two 400MHz UltraSPARC® processors, connected by a crossbar UPA switch, and running Solaris™ 9 Operating Environment. Our C code was compiled by a Sun *cc* compiler 5.3, with flags `-xO5 -xarch=v8plusa`. All our tests use kernel-space threads rather than user-space threads.

### 4.1 The Benchmarked Algorithms

We compared our stack implementation to the lock-free but non-linearizable elimination tree of Shavit and Touitou [24] and to two linearizable methods: a serial stack protected by MCS lock [19], and the non-blocking IBM/Treiber algorithm [17; 29].

—**MCS** A serial stack protected by an MCS-queue-lock [19]. Each processor locks the top of the stack, changes it according to the type of the operation, and then unlocks it. The lock code was taken directly from the article.

—**IBM/Treiber** Our implementation of the IBM/Treiber non-blocking stack followed the code given in [29]. We added to it exponential backoff scheme, as introduced in [2].

—**ETree** An elimination tree [24] based stack. Its parameters were chosen so as to optimize its performance, based on empirical testing.

### 4.2 The Produce-Consume Benchmark

In the produce-consume benchmark, each thread alternately performs a push or pop operation and then waits for a period of time whose length is chosen uniformly at random from the range: [0 . . . workload]. The waiting period simulates the local work that is typically done by threads in real applications between stack operations (see Figure 7). In all our experiments the stack was initialized as sufficiently filled to prevent it from becoming empty during the run.

### 4.3 Measuring the performance of benchmarked algorithms

We ran the produce-consume benchmark specified above varying the number of threads and measuring *latency*, the average amount of time spent per operation, and *throughput*, the number of operations per second. We compute throughput and latency by measuring the total time required to perform the specific amount of operations by each thread. We refer to the longest time as the time needed to complete the specified amount of work.

```
repeat
  op:=random(push,pop)
  perform op
  w:=random(0..workload)
  wait w millisecs
until 500000 operations performed
```

Fig. 7. Produce-Consume benchmark

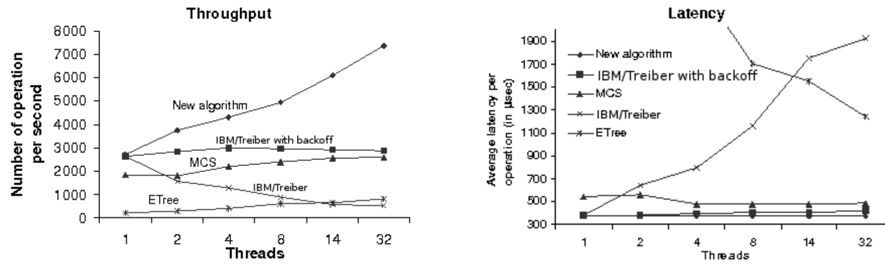


Fig. 8. Throughput and latency of different stack implementations with varying number of threads. Each thread performs 50% `push`s, 50% `pop`s.

To counteract transient startup effects, we synchronized the start of the threads (i.e., no thread can start before all other threads finished their initialization phase). Each data point is the average of three runs, with the results varying by at most 1.4% throughout all our benchmarks.

### 4.4 Empirical Results

Figure 8 shows the results of a benchmark in which half a million operations were performed by every working thread, with each thread performing 50% `push`s and 50% `pop`s on average. Figure 9 provides a detailed view of the three best performers. From Figure 8 it can be seen that our results for known structures generally conform with those of [21; 23], and that the IBM/Treiber algorithm with added exponential backoff is the best among known techniques. It can also be seen that the new algorithm provides superior scalable performance at all tested concurrency levels. The throughput gap between our algorithm and the IBM/Treiber algorithm with backoff grows as concurrency increases, and at 32 threads the new algorithm is almost three times faster. Such a significant gap in performance can be explained by reviewing the difference in latency for the two algorithms.

Table 1 shows latency measured on a single dedicated processor. The new algorithm and the IBM/Treiber algorithm with backoff have about the same latency, and outperform all others. The reason the new algorithm achieves this good performance is due to the fact that elimination backoff (unlike the elimination used in structures such as combining funnels and elimination trees) is used only as a backoff scheme and introduces no overhead. The gap of the two algorithms, with respect to MCS and ETree, is mainly due to the fact that a push or a pop in our algorithm and in the IBM/Treiber algorithm typically needs to access only two cache lines in the data structure, while a lock-based algorithm has the overhead of accessing lock variables as well. The ETree has an overhead of travelling through the tree.

As Figure 9 shows, as the level of concurrency increases, the latency of the

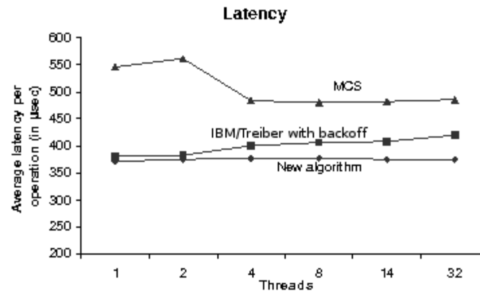


Fig. 9. Detailed graph of latency with threads performing 50% `push`s, 50% `pop`s.

IBM/Treiber algorithm grows since the head of the stack, even with contention removed, is a sequential bottleneck. On the other hand, the new algorithm has increased the rate of successful collisions on the elimination array as concurrency increases. As Table 2 shows, the fraction of successfully eliminated operations increases from only 11% for two threads up to 43% for 32 threads. The increased elimination level means that increasing numbers of threads complete their operations quickly and in parallel, keeping latency fixed and increasing overall throughput.

We also tested the robustness of the algorithms under workloads with an imbalanced distribution of `push` and `pop` operations. Such imbalanced workloads are not favorable for the new algorithm because of the smaller chance of successful collision. From Figure 10 it can be seen that the new algorithm still scales, but at a slower rate. The slope of the latency curve for our algorithm is 0.13 $\mu$sec per thread, while the slope of the latency curve for the IBM/Treiber algorithm is 0.3 $\mu$sec per thread, explaining the difference in throughput as concurrency increases.

In Figure 11 we compare the various methods under sparse access patterns and low load, by setting `workload` = 1000. In these circumstances, all the algorithms (with the exception of the elimination tree) maintain an almost constant latency as the level of concurrency increases because of the low contention. The decrease in the latency of elimination tree w.r.t. the case of `workload` = 0 is smaller, because it achieves lower levels of elimination. In contrast, the adverse effect of the sparse

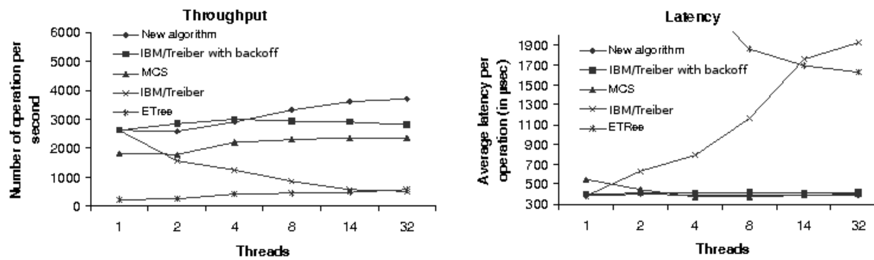


Fig. 10. Throughput and latency under varying distribution of operations: 25% push, 75%pop

Table 1. Latency on a single processor (no contention).

| | |
|---|---|
| New algorithm | 370 |
| IBM/Treiber with backoff | 380 |
| MCS | 546 |
| IBM/Treiber | 380 |
| ETree | 6850 |

Table 2. Fraction of successfully eliminated operations per concurrency level

| | |
|---|---|
| 2 threads | 11% |
| 4 threads | 24% |
| 8 threads | 32% |
| 14 threads | 37% |
| 32 threads | 43% |

access pattern on our algorithm's latency is small, because our algorithm uses the collision layer only as a backup if it failed to access the central stack object, and the rate of such failures is low when the overall load is low.

To further test the effectiveness of our policy of using elimination as a backoff scheme, we measured the fraction of operations that failed on their first attempt to change the top of the stack. As seen in Figure 12, this fraction is low under low loads (as can be expected) and grows together with load, and, perhaps unexpectedly, is lower than in the IBM/Treiber algorithm. This is a result of using the collision layer as the backoff mechanism in the new algorithm as opposed to regular backoff, since in the new algorithm some of the failed threads are eliminated and do not interfere with the attempts of newly arrived threads to modify the stack. These results further justify the choice of elimination as a backoff scheme.

To study the behavior of our adaptation strategy we conducted a series of experiments to hand-pick the "optimized parameter set" for each level of concurrency. We then compared the performance of elimination backoff with an adaptive strategy to an optimized elimination backoff stack. These results are summarized in Figure 13. Comparing the latency of the best set of parameters to those achieved using adaptation, we see that the adaptive strategy is about 2.5% - 4% slower.

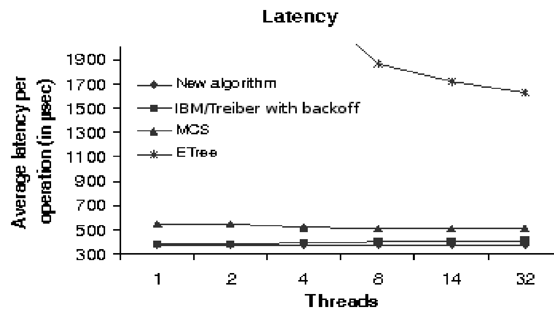


Fig. 11. Workload=1000

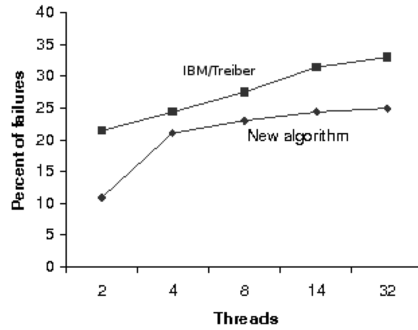


Fig. 12. Fraction of failures on first attempt

From these results we conclude that our adaptation techniques appear to work reasonably well. Based on the above benchmarks, we conclude that for the concurrency range tested, elimination backoff is the algorithm of choice for implementing linearizable stacks.

## 5. CORRECTNESS PROOFS

This section contains a formal proof that our algorithm is a lock-free linearizable implementation of a stack. It is organized as follows. In Section 5.1, we describe the model used by our proofs. In section 5.2, we prove basic correctness properties of our algorithm. Proofs of linearizability and lock-freedom are then provided in Sections 5.3 and 5.4, respectively.

### 5.1 Model

Our model for multithreaded computation follows [16], though for brevity and accessibility we will use operational style arguments. A concurrent system models an asynchronous shared memory system where A set $P$ of $n$ deterministic *threads* communicate by executing atomic *operations* on shared *variables* from some finite set

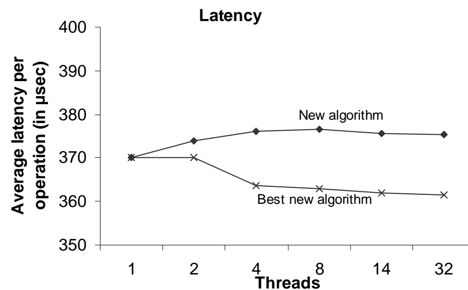


Fig. 13. Comparison of algorithm latency achieved by hand-picked parameters with that achieved by using an adaptive strategy

$V$. Each thread is a sequential execution path that performs a sequence of *steps.* In each step, a thread may perform some local computation and may invoke at most a single atomic operation on a shared variable. The atomic operations allowed in our model are *read*, *write*, and *compare-and-swap.* The compare-and-swap operation (abbreviated CAS) is defined as follows: *CAS(v,expected,new)* changes the value of variable $v$ to *new* only if its value just before CAS is applied is *expected*; in this case, the *CAS* operation returns *true* and we say it is *successful.* Otherwise, CAS does not change the value of $v$ and returns *false*; in this case, we say that the *CAS* was *unsuccessful.*

A *configuration* of the system is a vector of size $n + |V|$, that stores the states of all threads of $P$ and the values of all variables of $V$.[1] We say that *thread t is enabled to execute line L at configuration s* if $t$'s state in $s$ implies that, when next scheduled, $t$ executes line L.

An *execution* is a (finite or infinite) sequence of steps that starts from an *initial configuration.* This is a configuration in which all variables in $\mathbf{V}$ have their initial values and all threads are in their initial states. If $o \in \mathbf{B}$ is a base object and $E$ is a finite execution, then $value(E, o)$ denotes the value of $o$ at the end of $E$. If no event in $E$ changes the value of $o$, then $value(E, o)$ is the initial value of $o$. In other words, in the configuration resulting from executing $E$, each base object $o \in \mathbf{B}$ has value $value(E, o)$. For any finite execution fragment $E$ and any execution fragment $E'$, the execution fragment $EE'$ denotes the concatenation of $E$ and $E'$.

A concurrent stack is a data structure whose operations are linearizable [16] to those of the sequential stack as defined in [6]. The following is a sequential specification of a stack object.

Definition 1. *A stack S is an object that supports two types of operations: push and pop. The state of a stack is a sequence of items* $S = \langle v_0, ..., v_k \rangle$. *The stack is initially empty. The push and pop operations induce the following state transitions of the sequence* $S = \langle v_0, ..., v_k \rangle$, *with appropriate return values:*

—*push(*$v_{new}$*), changes S to be* $\langle v_0, ..., v_k, v_{new} \rangle$

—*pop(), if S is not empty, changes S to be* $\langle v_0, ..., v_{k-1} \rangle$ *and returns* $v_k$*; if S is empty, it returns empty and S remains unchanged.*

We note that a *pool* is a relaxation of a stack that does not require LIFO ordering. We start by proving that our algorithm implements a concurrent pool, without considering a linearization order. We then prove that our stack implementation is linearizable to the sequential stack specification of Definition 1. Finally we prove that our implementation is lock-free.

## 5.2 Correct Pool Semantics

We first prove that our algorithm has correct pool semantics, i.e., that *pop* operations can only pop items that were previously pushed, and that items pushed by *push* operations are not duplicated and can be popped out. This is formalized in the following definition. [2]

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

[1] The state of each thread consists of the values of the thread's local variables, registers and program-counter.

[2] For simplicity we assume all items are unique, but the proof can easily be modified to work without this assumption.

(1) *Let Op be a* pop *operation that returns an item $i$, then $i$ was previously pushed by a* push *operation.*

(2) *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 $i$.*

(3) *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 a value.*

*We call any operation that complies with the above requirement a* correct pool operation.

Lemma 2. *Operations that modify the central stack object are correct pool operations.*

Proof. Follows from the correctness of Treiber's algorithm [29]. □

In the following, we prove that operations that exchange their values through collisions are also correct pool operations, thus we show that our algorithm has correct pool semantics. We first need the following definitions.

Definition 3. *We say that an operation op is a* colliding operation *if it returns in line S12, S17 or S22 of `LesOP`. If op performs a push then we say it is a* push colliding operation, *otherwise we say that it is a* pop colliding operation.

Definition 4. *Let $op_1$ be a push operation and $op_2$ be a pop operation. We say that $op_1$ and $op_2$ have* collided *if $op_2$ obtains the value pushed by $op_1$ without accessing the central stack object. More formally, we require that one of the following conditions hold:*

—*Operation $op_2$ performs a successful CAS in line C8 of `TryCollision` and $q$ points to the ThreadInfo structure representing $op_1$ at that time.*

—*Operation $op_2$ performs a CAS operation in line S20 of `TryPerformStackOp` and the CAS fails because the entry of the location array corresponding to the thread executing $op_2$ points at that time to the ThreaInfo structure representing $op_1$.*

Definition 5. *We say that a colliding operation op is active if it executes a successful CAS in lines C2 or C8 of `TryCollision`. We say that a colliding operation is passive if op performs an unsuccessful CAS operation in lines S10 or S20 of `LesOP`.*

Definition 6. *Let op be an operation performed by thread $t$ and let $s$ be a configuration. If $t$ is enabled to execute a line of `LesOP`, `TryCollision` or `FinishCollision` in $s$, then we say that $t$ is* trying to collide at s. *Otherwise, we say that op is* not trying to collide at $s$.

We next prove that operations can only collide with operations of the opposite type. In the proofs that follow, we let $l_t$ denote the element corresponding to $t$ in the `location` array. First we need the following technical lemma.

Lemma 3. *Every colliding operation op is either active or passive, but not both.*

Proof. Let *Op* be a colliding operation. From Definition 3, we only need to consider the following cases.

—*Op* returns in line S12. In this case, *op* performed a successful CAS in line C2 or C8 of `TryCollision`. Thus, from Definition 5, *Op* is active. To obtain a contradiction, assume that *Op* is also passive. It follows from Definition 5, that

*Op* performed an unsuccessful CAS operation in line S10 or S20 before calling `TryCollision`. In each of these cases, however, *Op* returns after performing `FinishCollision` and does not call `TryCollision` after that. This is a contradiction.

—*Op* returns in line S17. It follows that *Op* performed an unsuccessful CAS operation in line S10. Hence, from Definition 5, *Op* is passive. To obtain a contradiction, assume that *Op* is also active. It follows from Definition 5, that *Op* executed a successful CAS operation in lines C2 or C8 of `TryCollision` before its unsuccessful CAS in line S10. In this case, however, `TryCollision` returns *true* and so *Op* immediately returns in line S12. This is a contradiction.

—*Op* returns in line S22. It follows that *Op* performs an unsuccessful CAS operation in line S20. Hence, from Definition 5, *Op* is passive. The proof proceeds in a manner identical to that of the corresponding proof for line S17.

□

LEMMA 4. *Operations can only collide with operations of the opposite type: an operation that performs a* `push` *can only collide with operations that perform a* `pop`*, and vice versa.*

PROOF. Let *op* be a colliding operation. From the code and from Definition 3, *op* either returns *true* from TryCollision or executes FinishCollision. We now examine both these cases.

(1) `TryCollision` can succeed only in case of a successful CAS in line C2 (for a push operation) or in line C8 (for a pop operation). Such a CAS changes the value of the other thread's cell in the *location* array, thus exchanging values with it and returns without modifying the central stack object. From the code, before calling `TryCollision` *op* has to execute line S9, thus verifying that it collides with an operation of the opposite type. Finally, from Lemma 1, $q$ points to the same ThreadInfo structure starting from when it is assigned in line S8 by *Op* and until either line C2 or C8 is performed by *Op*.

(2) If *op* is a passive colliding operation, then op performs FinishCollision, which implies that *op* failed in resetting its entry in the location array (in line S10 or s20). Let *op*1 be the operation that has caused *op*'s failure by writing to its entry. From the code, *op*1 must have succeeded in `TryCollision`. The proof now follows from case (1).

□

LEMMA 5. *An operation terminates without modifying the central stack object if and only if it is a colliding operation.*

PROOF. If *Op* modifies the central stack object, then its call of `TryPerformStackOp` returns *true* and it returns in line S24. It follows from Definition 3 that *Op* is not a colliding operation. As for the other direction, if *Op* is a colliding operation, then it returns in lines S12, S17 or S22. It follows that it does not return in line S24, hence it could not have changed the central stack object. □

LEMMA 6. *Let s be a configuration, and let* t *be a thread . Then* t *is trying to collide in s if and only if* $l_t \neq NULL$ *holds in s.*

PROOF. In the initial configuration, $l_t$=*NULL* holds. In the beginning of each collision attempt performed by operation *op*, the value of $l_t$ is set to a non-NULL

value in line S2. We need to show that *op* changes the value of $l_t$ back to NULL before completing the collision attempt, either successfully or unsuccessfully. We now check both these cases.

—Suppose that *op* fails in its collision attempt, that is, *op* reaches line S23. Clearly from the code, to reach line S23, *op* has to perform a successful CAS in either line S10 or in line S20, thus it sets the value of $l_t$ to NULL upon finishing its unsuccessful collision attempt.

—Otherwise, *op* succeeds in its collision attempt. Consequently, it exits `LesOp` in the same iteration, in lines S12, S17 or S22. Clearly from the code, to reach line S12 *op* has to perform a successful CAS in line S10, thus setting the value of $l_t$ to NULL. If *op* returns in lines S17 or S22, then it returns after it executes *FinishCollision*. From the code of *FinishCollision*, if *op* is a `pop` operation, then *FinishCollision* sets $l_t$ to NULL at line F3. Finally, if *op* is a `push` operation and reaches *FinishCollision*, then *op* must have failed to perform a CAS in lines S10 or S20. This failure implies that some other operation changed the value of $l_t$. As *op* is a `push` operation, we have by Lemma 4, that the value of $l_t$ was changed by another operation, $op_1$, that performed a pop operation; thus $op_1$ changed $l_t$ to NULL in line C8.

□

LEMMA 7. *Let Op be a* `push` *operation by some thread* t*, and let s be a configuration. If it holds in s that* $l_t \neq$NULL*, then Op is trying in s to push the value* $l_t$*->cell.pdata.*

PROOF. Clearly from the code and from Lemma 1, only *Op* can write a value different than `NULL` to $l_t$. From Lemma 6, $l_t$ is $NULL$ after *Op* exits, hence *Op* is in the midst of a collision attempt in configuration *s*. From Lemma 1, it follows that the value of $l_t$ in *s* is a a pointer to *t*'s `ThreadInfo` structure written on line S2. As *Op* is a `push` operation, the cell *Op* is trying to push is pointed at from the `cell` field of that structure. □

LEMMA 8. *Every passive colliding operation collides with exactly one active colliding operation and vice versa.*

PROOF. Immediate from Definition 4 and from Lemma 1. □

LEMMA 9. *Every colliding operation op collides with exactly one operation of the opposite type.*

PROOF. Follows from Lemmas 3 and 8. □

We now prove that, upon colliding, opposite operations exchange values in a proper way.

LEMMA 10. *If a* `pop` *operation collides, it obtains the value of the single* `push` *operation it collided with.*

PROOF. Let $op_1$ denote a pop operation performed by thread *t*. If $op_1$ is a passive colliding operation, then, from Lemma 9 and Definition 5, it collides with a single active push colliding operation, $op_2$. As $op_1$ succeeds in colliding, from Definition 4 and from Lemma 1, it obtains in line F2 the cell that was written to $l_t$ by $op_2$.

Assume now that $op_1$ is a an active colliding operation, then, from Lemma 9 it collides with a single passive push colliding operation, $op_2$. As $op_1$ succeeds in

colliding, it succeeds in the CAS of line C8 and thus, from Lemma 1, returns the cell that was written by $op_2$. □

LEMMA 11. *If a* push *operation collides, its value is obtained by the single* pop *operation it collided with.*

PROOF. Let $op_1$ denote a push operation performed by thread $t$. If $op_1$ is a passive colliding operation, then, from Lemma 9 and Definition 5, it collides with a single active pop colliding operation, $op_2$. As $op_1$ is passive, from Definition 5, $op_1$ performs an unsuccessful CAS in line S10 or S20. From Lemma 1, it follows that the value of $l_t$ was previously set to NULL by $op_2$ as it obtained in line F2 the cell that was written by $op_1$ to $l_t$ .

Assume now that $op_1$ is a an active colliding operation, then, from Lemma 9 and Definition 5, it collides with a single passive pop colliding operation, $op_2$. Let $q$ be the thread performing $op_2$. As $op_1$ is active, from Definition 5, it performs a successful CAS operation in line C2, thus writing a pointer to its ThreadInfo structure to $l_q$. From Lemma 1, it follows that, if and when $op_2$ returns, it returns the cell field of this structure. □

We can now prove that our algorithm has correct pool semantics.

THEOREM 1. *The elimination-backoff stack has correct pool semantics.*

PROOF. From Lemma 2, all operations that modify the central stack object are correct pool operations. From Lemmas 10 and 11, all colliding operations are correct pool operations. Thus, all operations on the elimination-backoff stack are correct pool operations. It follows from Definition 2 that the elimination-backoff stack has correct pool semantics. □

### 5.3 Linearizability

Given a sequential specification of a stack, we provide specific linearization points mapping operations in our concurrent implementation to sequential operations so that the histories meet the specification.

DEFINITION 7. *The* elimination backoff stack linearization points *are selected as follows. All operations, except for passive colliding operations, are linearized in the following lines, executed in their (single) successful iteration:*

—push *operation are linearized in lines T4 or C2.*

—pop *operations are linearized in lines T10, T14 or C8.*

*For passive colliding operations, we set the linearization point to be at the time of linearization of the matching active colliding operation, and the push colliding operation is linearized before the pop colliding operation.*

Each push or pop operation consists of a while loop that repeatedly attempts to complete the operation. We say that an iteration is a *successful iteration* if the operation returns at that iteration; otherwise, another iteration will be performed. Every completed operation has exactly one successful iteration (its last one), and the linearization point of the operation occurs in the course of performing that iteration.

From definition 1, it follows that a successful collision does not change the state of the central stack object. Consequently, at any point in time, the state of the stack is determined solely by the state of its central stack object. We proceed by

proving that the aforementioned code lines are correct linearization points both for operations that complete by modifying the central stack object, and for operations that exchange values through collisions.

LEMMA 12. *The lines specified in Definition 7 are correct linearization points for operations that complete by modifying the central stack object.*

PROOF. The linearization points specified in Definition 7 for operations that complete by modifying the central stack object are:

—Line T4 (for a push operation)

—Line T10 (in case of empty stack) or line T14 (for a pop operation)

Since colliding operations do not change the state of the stack, the claim follows directly from the linearizability of Treiber's algorithm [29]. □

Before establishing the correctness of the linearization points for colliding operations, we need the following technical lemma.

LEMMA 13. *Let $op_1$ be an active colliding operation and let $op_2$ be the passive colliding operation with which it collides. Then the linearization point of $op_1$, as specified in Definition 7, is within the time interval of $op_2$.*

PROOF. From definition 5, $op_1$ performs a successful CAS in line C2 (if it is a *push* operation) or in line C8 (if it is a *pop* operation). From Definition 7, this is $op_1$'s linearization point. From Lemma 9, $op_1$ collides only with $op_2$ and these two operations have opposite types.

Let $s$ be the configuration immediately preceding the execution of the linearization line of $op_1$. The success of the CAS in line C2 or C8 and Lemma 1 imply that the value of $op_2$'s entry in the location array is non-NULL in $s$ (otherwise the check at line S9 would have failed). Thus, from Lemma 6 and definition 6, $op_2$ is trying to collide in configuration $s$. □

LEMMA 14. *The lines specified in Definition 7 are correct linearization points for colliding operations.*

PROOF. To simplify the proof and avoid the need for backward simulation style arguments, we consider only complete execution histories, that is, ones in which all abstract operations have completed, so we can look "back" at the execution and say for each operation where it happened.

We first note that according to Lemma 13, the linearization point of passive colliding operations is well-defined (it is obviously well-defined for active colliding operations). We need to prove that correct LIFO ordering is maintained between any two linearized colliding operations and between these operations and operations that modify the central stack object.

As we linearize a passive colliding operation in the linearization point of its (single) counterpart active colliding operation, no other operations can be linearized between these two operations. Moreover, since the *push* operation is linearized just before the *pop* operation, this is a legal LIFO matching that cannot interfere with the LIFO matching of other pairs of colliding operations or with that of operations that modify the central stack object. Finally, from Lemma 10, the pop operation indeed obtains the value of the push operation it collides with. □

THEOREM 2. *The elimination-backoff stack is a correct linearizable implementation of a stack object.*

PROOF. Immediate from Lemmas 12 and 14. □

### 5.4 Lock Freedom

Theorem 3. *The elimination-backoff stack algorithm is lock-free.*

Proof. Let *op* be some operation. We show that in every iteration made by *op*, some operation performs its linearization point, thus the system as a whole makes progress. If *op* manages to collide, then *op*'s linearization has occurred, as have the linearization of the operation *op* collided with.

Otherwise, *op* calls *TryPerformStackOp*. If *TryPerformStackOp* returns TRUE, *op* immediately returns, and its linearization has occurred. If, on the other hand, *TryPerformStackOp* returns FALSE, then it must be that the CAS operation it applied to the central stack object was unsuccessful. This implies, in turn, that a CAS operation applied to *S.ptop* by some other operation $op_1$ was successful. Hence, $op_1$ was linearized. It follows that whenever *op* completes a full iteration, some operation is linearized. □

## 6. DISCUSSION

Shared stacks are widely used in parallel applications and operating systems. In this paper, we presented the *elimination backoff stack*, the first concurrent stack algorithm that is both linearizable, lock-free and can achieve high throughput in high contention executions. The *elimination backoff stack* is based on the following simple observation: that a single elimination array [24], used as a backoff scheme for a lock-free stack [29], is both lock-free and linearizable. The introduction of elimination into the backoff process serves a dual purpose of adding parallelism and reducing contention, which, as our empirical results show, allows the *elimination-backoff stack* to outperform all algorithms in the literature at both high and low loads.

We observe that, unlike the simple algorithm of [29] in which threads can be anonymous, our algorithm requires that all threads that concurrently perform collision attempts have unique identifiers. The same thread can use different identifiers, however, in different collision attempts. It is therefore easy to support applications in which threads are created and deleted dynamically: threads can get and release unique identifiers from a small name space by using any long-lived renaming algorithm (see, e.g., [3; 4]); since accessing the central stack does not require an identifier, there is no adverse effect on time complexity in the absence of contention. There is also no need of a-priori knowledge of the maximum number of concurrently participating threads, as a lock-free dynamically resizable array (see [7]) can be used instead of the static `location` array.

Our algorithm includes a tight "busy-waiting" loop in lines S5-S6, performed by a process as it tries to apply a successful CAS operation to an entry of the `collision` array. In general, long busy-waiting loops have adverse effect on performance; as our empirical results establish, however, this is not the case with our algorithm. The reason for this is that our algorithm uses a mechanism for dynamically adapting the width of the `collision` array; when the load is high, this mechanism increases the width of the `collision` array and reduces the probability of CAS failures. This ensures that the loop of lines S5-S6 is short in practice.

Several related works have appeared since the preliminary version of this paper was published. Colvin and Grobes [5] presented a somewhat simplified version of our algorithm and proved its correctness by using the PVS [22] theorem prover. Recently, Hendler and Kutten [11] introduced *bounded-wait combining*, a technique by which asymptotically high-throughput lock-free linearizable implementations of

objects that support combinable operations (such as counters, stacks, and queues) can be constructed.

## Acknowledgments

We would like to thank the anonymous referees for many valuable comments on a previous draft of this article, which helped improve its quality.

## REFERENCES

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Symposium on Computer Architecture*, pages 41–55, June 1989.

[2] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[3] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[4] A. Brodsky, F. Ellen, and P. Woelfel. Fully-adaptive algorithms for long-lived renaming. In *DISC*, pages 413–427, 2006.

[5] R. Colvin and L. Groves. A scalable lock-free stack algorithm and its verification. In *SEFM*, pages 339–348, 2007.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2002.

[7] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *OPODIS*, pages 142–156, 2006.

[8] R. Goodman and M. K. V. P. J. Woest. Efficient synchronisation primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-III*, pages 64–75, 1989.

[9] A. Gottleib, B. D. Lubachevsky, and L. Rudolph. Efficient techniques for coordinating sequential processors. *ACM TOPLAS*, 5(2):164–189, April 1983.

[10] M. Greenwald. *Non-Blocking Synchronization and System Design*. PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 8 1999.

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

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

[13] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[14] M. Herlihy, B.-H. Lim, and N. Shavit. Scalable concurrent counting. *ACM Transactions on Computer Systems*, 13(4):343–364, 1995.

[15] M. Herlihy, V. Luchangco, and M. Moir. The repeat-offender problem, a mechanism for supporting dynamic-sized,lock-free data structures. Technical Report TR-2002-112, Sun Microsystems, September 2002.

[16] 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.

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

[18] B.-H. Lim and A. Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Transactions on Computer Systems*, 11(3):253–294, August 1993.

[19] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[20] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.

[21] 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.

[22] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *CADE*, pages 748–752, 1992.

[23] M. Scott and W. Schrerer. User-level spin locks for large commercial applications. In *SOSP, Work-in-progress talk*, 2001.

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

[25] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. *Theory of Computing Systems*, 31(4):403–423, 1998.

[26] N. Shavit and A. Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, 1996.

[27] N. Shavit and A. Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, (60):1355–1387, 2000.

[28] 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.

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