A Scalable Lock-free Stack Algorithm

Danny Hendler *
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel 69978
hendlerd@post.tau.ac.il

Nir Shavit
Tel-Aviv University &
Sun Microsystems
Laboratories
shanir@sun.com

Lena Yerushalmi
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel 69978
lenay@post.tau.ac.il

ABSTRACT

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.

Categories and Subject Descriptors

C.1.4.1 [**Computer Systems Organization**]: Processor Architectures—*Parallel Architectures,Distributed Architectures*; E.1.4.1 [**Data**]: Data Structures—*lists, stacks and queues*

General Terms

Algorithms, theory, lock-freedom, scalability

1. INTRODUCTION

Shared stacks are widely used in parallel applications and operating systems. As shown in [21], 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 [11] 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, 9]. Therefore, in addition to correctness, synchronization methods should offer efficiency in terms of scalability and robustness 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 [20] are linearizable [11] 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 [12]. The elimination trees of Shavit and Touitou [17] 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 [15] show that the best known low load method, the simple linearizable lock-free stack of Treiber [22], 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 [17], used as a backoff scheme for a lock-free stack [22], 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.

*This work was supported in part by a grant from Sun Microsystems.

SPAA'04, June 27–30, 2004, Barcelona, Spain.
Copyright 2004 Sun Microsystems, Inc. All rights reserved.
ACM 1-58113-840-7/04/0006.

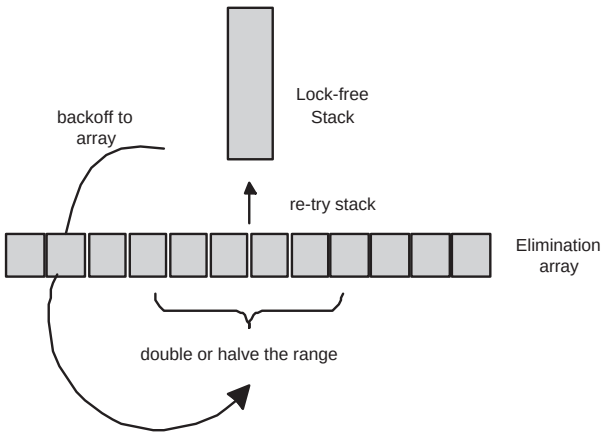

Figure 1: Schematic depiction of the elimination-backoff cycle.

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 [7].

Treiber [22] proposed the first non-blocking implementation of concurrent list-based stack. He represented the stack as a singly-linked list with a top pointer and used 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 [15] compared Treiber's stack to an optimized non-blocking algorithm based on Herlihy's general methodology [8], and to lock-based stacks. They showed that Treiber's 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 Treiber stack offers little scalability.

Shavit and Touitou [17] 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 [20], and used them to provide scalable stack implementations. Combining funnels employ both combining [5, 6] and elimination [17] to provide scalability. They improve on elimination trees by being linerarizable, but unfortunately they are blocking. As noted earlier, both [17] and [20] 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 [17]: if a `push` followed by a `pop` are performed on a stack, the data structure's state does not change (similarly for a `pop` followed by a `push`). 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 anyhow "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 [17], the idea is to build a tree of elimination arrays and use the diffracting tree paradigm of Shavit and Zemach [19] 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 met. Because it is a backoff scheme, it delivers the same performance as the simple 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 [13] and to several non-blocking implementations: the linearizable Treiber [22] algorithm with and without backoff and the elimination tree of Shavit and Touitou [17]. 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. Finally, in Section 5, we provide a proof that our algorithm has the required properties of a stack, is linearizable, and lock-free.

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 {                  struct Simple_Stack {
    Cell *pnext;                   Cell *ptop;
    void *pdata;               };
};
 struct ThreadInfo {           Simple_Stack S;
    u_int id;                  void **location;
    char  op;                  int *collision;
    Cell  cell;
    int   spin;
 };
```

Figure 2: Types and Structures

Our central stack object follows Treiber [22] and is implemented as a singly-linked list with a top pointer. The elimination layer follows Shavit and Touitou and is built of two arrays: a global `location`[1..n] array has an element per thread $p \in \{1..n\}$, holding the pointer to the ThreadInfo structure, and a global `collision`[1..size] array, that holds the ids of the threads trying to collide. Each ThreadInfo record contains the thread id, the type of the operation to be performed by the thread (`push` or `pop`), and the node for the operation. The `spin` variable holds the amount of time the thread should delay while waiting to collide.

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 p announces its arrival at the collision layer by writing its current information to the `location` array (line S2). It then chooses the random location in the `collision` array (line S3). Thread p 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 success (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 p reads an id of the existing thread q (i.e., `him!=EMPTY`), p attempts to collide with q. The collision is accomplished by p 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 p, p waits for another collision (line S18). If no other thread collides with `p` during its waiting period, p clears its entry in the `location` array and tries to perform its operation on the central stack object. If p's entry cannot be cleared, it follows that p has been collided with, in which case p completes its operation and returns.

If `q` does have a complementary operation, p tries to eliminate by performing two CAS operations on the `location` array. The first clears p's entry, assuring no other thread will collide with it during its collision attempt (this eliminates race conditions). The second attempts to mark q's

```
void StackOp(ThreadInfo* pInfo) {
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(p->spin);
S19:  if (!CAS(&location[mypid],p,NULL)) {
S20:      FinishCollision(p);
S21:      return;
      }
stack:
S22:  if (TryPerformStackOp(p)==TRUE)
          return;
      }
}

boolean TryPerformStackOp(ThreadInfo*p){
    Cell *phead,*pnext;
T1: if(p->op==PUSH) {
T2:    phead=S.ptop;
T3:    p->cell.pnext=phead;
T4:    if(CAS(&S.ptop,phead,&p->cell))
T5:      return TRUE;
T6:    else
T7:      return FALSE;
   }
T8: if(p->op==POP) {
T9:    phead=S.ptop;
T10:   if(phead==NULL) {
T11:     p->cell=EMPTY;
T12:     return TRUE;
       }
T13:   pnext=phead->pnext;
T14:   if(CAS(&S.ptop,phead,pnext)) {
T15:      p->cell=*phead;
T16:      return TRUE;
      }
T17:   else {
T18:      p->cell=EMPTY;
T19:     return FALSE;
       }
}
void FinishCollision(ProcessInfo *p) {
F1: if (p->op==POP_OP) {
F2:     p->pcell=location[mypid]->pcell;
F3:     location[mypid]=NULL;
    }
}
```

Figure 3: Elimination Backoff Stack Code - part 1

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

Figure 4: Elimination Backoff Stack Code - part 2

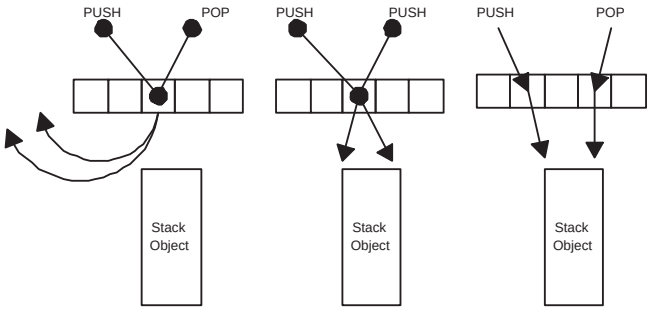

Figure 5: Collision scenarios

entry as "collided with p". If both CAS operations succeed, the collision is successful. Therefore p 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 r has already managed to collide with p. In that case the thread p acts as in case of a successful collision, mentioned above. If the first CAS succeeds but the second fails, then the thread with whom p is trying to collide is no longer available for collision. In that case, p tries to perform the operation on the central stack object, returns in case of success, and repeatedly goes through the collision layer in case of failure.

2.3 Memory Management and ABA Issues

In our implementation we use a very simple memory management mechanism - a pool of cells available for restricted use (similar to the pool introduced in [22]). When a thread needs a cell to perform a push operation on a stack, it removes a cell from the pool and uses it. When a thread pops a cell from the stack, it returns the cell to the pool. Note that the cells are returned only by threads that performed `pop` operations, thus insuring correctness in lines C8 and F2. Without this assumption we would need to copy the contents of the cell and not just its address. Though outside the scope of this paper, we note that one can use techniques such as those of Trieber [22], or more general techniques such as SMR [14] or ROP [10], to detect when a cell in the pool can be reused.

As our algorithm is based on the compare-and-swap (CAS) operation, it must deal with the "ABA problem" [4]. 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(s) change the value to the previous one again. The simplest and most common ABA-prevention mechanism is to include a tag with the target memory location such that both are manipulated together atomically, and the tag is incremented with updates of the target location [4]. 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 managements such as SMR [14] or ROP [10].

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 [20]. 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 subrange 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 particulary bad scenarios might lead to bad performance, but as we will show, 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, [20] are at best three times slower than a simple stack).

One way of adaptively changing the width of the collision layer is the following. Each thread t keeps a value, $0<factor<1$, 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). When t fails to collide because it did not encounter another thread, it increments a private counter. When the counter exceeds some limit, *factor* is halved, and the counter is being reset to its initial value. If, on the other hand, t encountered some other thread u, performing an opposite operation-type, but fails to collide with it (the most probable reason being that some other thread v succeeded in colliding with u before t), the counter is being decremented, and when it reaches 0, *factor* is doubled, and the counter is being reset to its initial value.

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 [18, 19]. One way of doing that is the following. Each

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

Figure 6: Produce-Consume benchmark

thread t keeps a value `spin` which holds the amount of time that t should delay while waiting to be collided. The `spin` value may change within a predetermined range. When t successfully collides, it increments a local counter. When the counter exceeds some limit, t doubles `spin`. If t fails to collide, it decrements the local counter. When the counter decreases bellow some limit, `spin` is halved. This localized version of exponential backoff serves a dual role: it increases the chance of successful eliminations, and it plays the role of a backoff mechanism on the central stack structure.

There are obviously other conceivable ways of adaptively updating these 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. Our C code was compiled by a Sun *cc* compiler 5.3, with flags `-xO5 -xarch=v8plusa`.

4.1 The Benchmarked Algorithms

We compared our stack implementation to the lock-free but non-linearizable elimination tree of Shavit and Touitou [17] and to two linearizable methods: a serial stack protected by MCS lock [13], and a non-blocking implementation due to Treiber [22].

- **MCS** A serial stack protected by an MCS-queue-lock [13]. 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.
- **Treiber** Our implementation of Treiber's non-blocking stack followed the code given in [22]. We added to it exponential backoff scheme, as introduced in [2].
- **ETree** An elimination tree [17] 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 or 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 6). In all our experiments the stack was initialized as sufficiently filled to prevent it from becoming empty during the run.

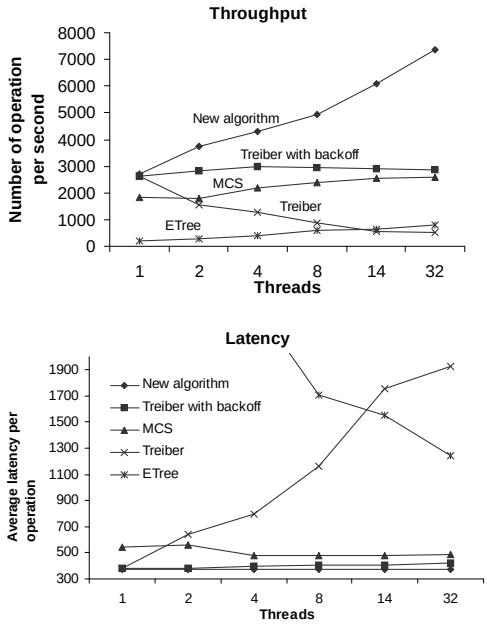

Figure 7: Throughput and latency of different stack implementations with varying number of threads. Each thread performs 50% `push`s, 50% `pop`s.

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.

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 7 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 7 it can be seen that our results for known structures generally conform with those of [15, 16], and that Treiber's 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 Treiber's 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 pro-

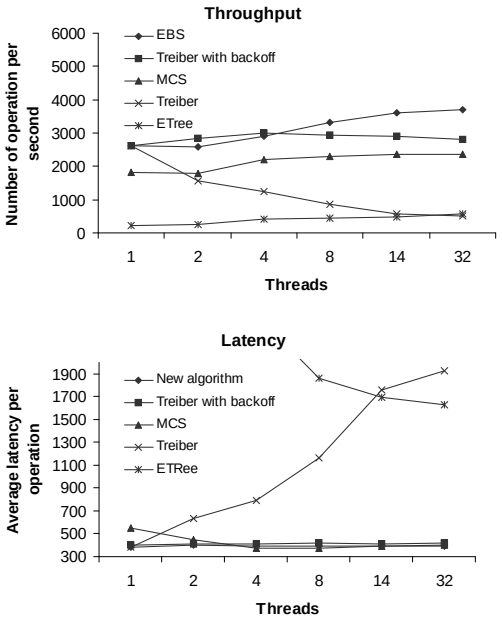

Figure 8: Throughput and latency under varying distribution of operations: 25% push, 75%pop

cessor. The new algorithm and Treiber's 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 Treiber's 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 Treiber's 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 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 8 it can be sees that the new algorithm still scales, but at a slower rate. The slope of the latency curve for our algorithm is 0.13 μsec per thread, while the slope of the latency curve for Treiber's algorithm is 0.3

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

New algorithm	370
Treiber with backoff	380
MCS	546
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%

μsec per thread, explaining the difference in throughput as concurrency increases.

In Figure 10 we compare the various methods as access patterns become sparse and the load decreases. Under low load, when `workload` = 1000, all the algorithms (except 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 of the lower levels of elimination. In contrast, the adverse effect of the sparse 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 11, this fraction is low under low loads (as can be expected) and grows together with load, and, perhaps unexpectedly, is lower than in Trieber's 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 12. Comparing the latency of the best set of parameters to those achieved using adaptation we see that adaptive strategy is about 2.5% - 4% slower.

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 we tested, elimination backoff is the algorithm of choice for implementing linearizable stacks.

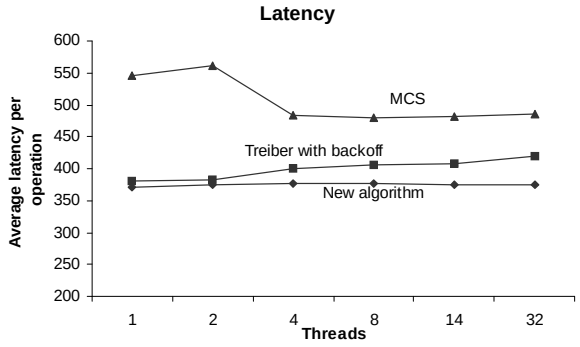

Figure 9: Detailed graph of latency with threads performing 50% pushs, 50% pops.

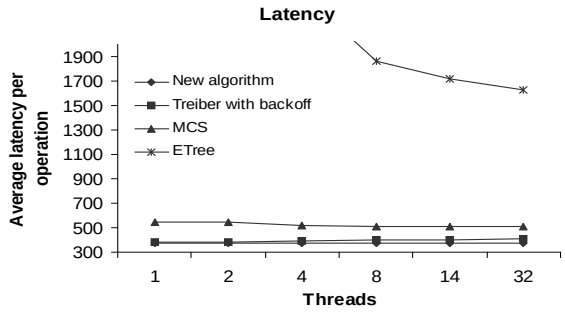

Figure 10: Workload=1000

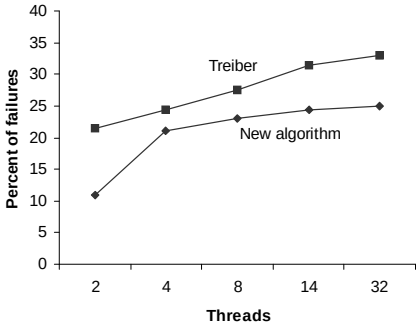

Figure 11: Fraction of failures on first attempt

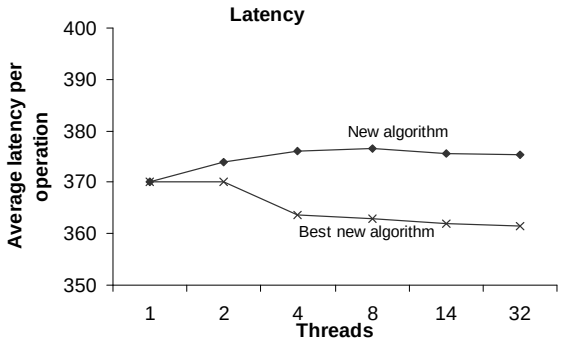

Figure 12: Comparison of algorithm latency achieved by hand-picked parameters with that achieved by using an adaptive strategy

5. CORRECTNESS PROOF

This section contains a formal proof that our algorithm is a lock-free linearizable implementation of a stack. For lack of space, proofs of a few lemmata are omitted and would appear in the full paper.

Our model for multithreaded computation follows [11], though for brevity and accessibility we will use operational style arguments. In our proof we will ignore issues relating to the ABA problem typical of implementations using the CAS operation. As described earlier (Section 2.3), there are several standard techniques for overcoming the ABA problem [10, 14]. A concurrent stack is a data structure whose operations are linearizable [11] to those of the sequential stack as defined in [3]. The following is a sequential specification of a stack object.

DEFINITION 5.1. *A stack S is an object that supports two types of operations on it: 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 *set* is a relaxation of a stack that does not require LIFO ordering. We begin by proving that our algorithm implements a concurrent set, without considering a linearization order. We then prove that our stack implementation is linearizable to the sequential stack specification of Definition 5.1. Finally we prove that our implementation is lock-free.

5.1 Correct Set Semantics

We now prove that our algorithm has correct set semantics, i.e. that *pop* operations can only pop items that were previously pushed, and that items pushed by *push* operations are not duplicated. This is formalized in the following definition [1].

DEFINITION 5.2. *A stack algorithm has* correct set semantics *if the following requirements are met for all stack operations:*

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.*

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

LEMMA 5.1. *Operations that modify the central stack object are correct set operations.*

PROOF. Follows from the correctness of Treiber's algorithm [22]. □

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

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

DEFINITION 5.3. *We say that two operations op_1 and op_2 have collided if they have exchanged their values and have not modified the central stack object; we say that each of op_1, op_2 is a colliding operation.*

DEFINITION 5.4. *We say that a colliding operation op is active if it executes a successful CAS in lines C2 or C7. We say that a colliding operation is passive if op fails in the CAS of line S10 or S19.*

DEFINITION 5.5. *A* state *s of the algorithm in an n-thread system is a vector of size n, with entry i, $1 \leq i \leq n$, representing the state of thread i. The state of thread i in s consists of the values of thread i's data structures and of the value of thread i'th program-counter.*

DEFINITION 5.6. *Let op be an operation performed by thread t. We say that op is* trying to collide at state s, *if, in s, the value of t's program counter is pointing at a statement of one of the following procedures:* `LesOP`, `TryCollision`, `FinishCollision`. *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. First we need the following technical lemma.

LEMMA 5.2. *Every colliding operation op is either active or passive, but not both.*

PROOF. Clearly from the code, a colliding operation is active and/or passive. We have to show that it cannot be both. Suppose that the operation *op* is passive, then *op* fails the CAS of line S10 or that of line S19; clearly from the code, *op* then calls *FinishCollision* and exits, therefore *op* cannot play an active-collider role after playing a passive-collider role. Suppose now that *op* is active. From definition 5.4, it executes a successful CAS in lines C2 or C7. It is clear from the code that in this case *op* returns TRUE from *TryCollision* and does not reach line S10 or S19 afterwards (it returns in line S12). So *op* cannot play a passive-collider role after playing an active-collider role. □

LEMMA 5.3. *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 us consider some operation, *op*, that collides. From the code, in order to successfully collide, *op* must either succeed in performing `TryCollision` or execute `FinishCollision`. We now examine both cases.

- `TryCollision` can succeed only in case of a successful CAS in line C2 (for a `push` operation) or in line C7 (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.
- 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 s19). 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`, thus, it has verified in line S9 that its type is opposite to that of *op*.

□

The proofs of the following three technical lemmata are omitted for lack of space.

LEMMA 5.4. *An operation terminates without modifying the central stack object, if and only if it collides with another operation.*

LEMMA 5.5. *For every thread* p *and in any state s, if p is not trying to collide in s, then it holds in s that the element corresponding to p in the* `location` *array is NULL.*

LEMMA 5.6. *Let op be a* `push` *operation by some thread* p*; if location[p] $\neq$ NULL, then op is trying to push the value* location[p]->cell.pdata.

In the next three lemmata, we show that *push* and *pop* operations are paired correctly during collisions.

LEMMA 5.7. *Every passive collider collides with exactly one active collider.*

PROOF. Assume by contradiction that some passive collider, op_1, collides with multiple other operations, and let op_2, op_3 be the last two operations that succeed in colliding with op_1. We denote the element written by op_1 to the `location` array by l_{op_1}. We consider the following two possibilities.

- Assume op_1 is a passive-collider performing a *pop* operation. From Lemma 5.3, both op_2, op_3 are *push* operations. From Lemma 5.2, op_1 cannot be both active and passive. Thus op_1 exchanges values only in line F2, with the last operation that has written to its entry in the *location* array. As both op_2 and op_3 are active colliders performing a *push*, both succeed in the CAS of line C2. As op_3 succeeds in colliding with op_1 after op_2 does, the q parameter used in the CAS of op_3 at line C2 must be the value written by op_2 in its successful CAS of line C2. This is impossible, because in line S9 op_3 verifies that q is of type *pop*, but op_2 is performing a *push*.
- Otherwise, assume op_1 is a passive-collider performing a *push* operation. From Lemma 5.3, both op_2, op_3 perform a *pop* operation. Thus it must be that both op_2 and op_3 succeed in the CAS of line C7. This implies that both succeed in writing *NULL* to the entry of op_1's thread in the *location* array. This, however, implies that the q parameter used by op_3 in line C7 is *NULL*, which is impossible since in this case op_3 would have failed the check in line S9.

□

LEMMA 5.8. *Every active collider op_1 collides with exactly one passive collider.*

PROOF. The proof is by contradiction. Assume that an active-collider, op_1, collides with two operations op_2 and op_3. From Lemma 5.3, both op_2 and op_3 are passive, hence both op_2 and op_3 have failed while executing CAS in lines S10 or S19. It follows that op_1 must have written its value to the `location` array twice. From the code, this is impossible, because op_1 can perform such a write only in line C2 or C9, and it exits immediately after. □

LEMMA 5.9. *Every colliding operation op participates in exactly one collision with an operation of the opposite type.*

PROOF. Follows from Lemmata 5.2, 5.7 and 5.8. □

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

LEMMA 5.10. *If a* `pop` *operation collides, it obtains the value of the single* `push` *operation it collided with.*

PROOF. Let op_1, op_2 respectively denote the `pop` operation and the `push` operation that collided with it. Also, let p_1 and p_2 respectively denote the threads that perform op_1 and op_2. We denote the entry corresponding to p_1 in the `location` array as l_{p_1}. We denote the entry corresponding to p_2 in the `location` array as l_{p_2}. Assume that op_1 is a passive collider, then from Lemma 5.9 it collides with a single active push collider, op_2. As op_1 succeeds in colliding, it obtains in line F2 the cell that was written to its entry in the `location` array by op_2.

Assume that op_1 is a an active collider, then from Lemma 5.9 it collides with a single passive push collider, op_2. As op_1 succeeds in colliding, it succeeds in the CAS of line C7 and thus returns the cell that was written by op_2. □

LEMMA 5.11. *If a* `push` *operation collides, its value is obtained by the single* `pop` *operation it collided with.*

PROOF. Symmetric to the proof of Lemma 5.10. □

We can now finally prove that our algorithm has correct set semantics.

THEOREM 5.12. *The elimination-backoff stack has correct set semantics.*

PROOF. From Lemma 5.1, all operations that modify the central stack object are correct set operations. From Lemmata 5.10 and 5.11, all colliding operations are correct set operations. Thus, all operations on the elimination-backoff stack are correct set operations and so, from Definition 5.2, the elimination-backoff stack has correct set semantics. □

5.2 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. Specifically, we choose the following linearization points for all operations, except for passive-colliders:

- Lines T4, C2 (for a `push` operation)
- Lines T10, T14, C7 (for a `pop` operation)

For a passive-collider operation, we set the linearization point to be at the time of linearization of the matching active-collider 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. An iteration is successful if its attempt succeeds, in which case the operation returns at that iteration; otherwise, another iteration is performed . Each completed operation has exactly one successful attempt (its last attempt), and the linearization of the operation occurs in that attempt. In other words, the operations are linearized in the aforementioned lineanirazation points only in case of a successful CAS, which can only be performed in the last iteration of the while loop.

We note that, from definition 5.1, a successful collision does not change the state of the central stack object. It follows that at any point of time, the state of the stack is determined solely by the state of its central stack object.

To prove that the aforementioned lines are correct linearization points of our algorithm, we need to prove that these are correct linearization points for the two types of operations: operations that complete by modifying the central stack object, and operations that exchange values through collisions.

LEMMA 5.13. *For operations that do not collide, we can choose the following linearization points:*

- *Line T4 (for a* `push` *operation)*
- *Line T10 (in case of empty stack) or line T14 (for a* `pop` *operation)*

PROOF. Follows directly from the linearizability of Treiber's algorithm [22]. □

We still have to prove that the linearization points for collider-operations are consistent, both with one another, and with non-colliding operations. We need the following technical lemma, whose proof is omitted for lack of space.

LEMMA 5.14. *Let op_1, op_2, be a colliding operations-pair, and assume w.l.o.g. that op_1 is the active-collider and op_2 is the passive collider, then the linearization point of op_1 (as defined above) is within the time interval of op_2.*

LEMMA 5.15. *The following are legal linearization points for collider-operations.*

- *An active-collider, op_1, is linearized at either line C2 (in case of a* `push` *operation) or at line C7 (in case of a* `pop` *operation).*
- *A passive-collider, op_2, is linearized at the linearization time of the active-collider it collided with. If op_2 is a push operation, it is linearized immediately before op_1, otherwise it is linearized immediately after op_1.*

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 5.14, the linearization point of the passive-collider is well-defined (it is obviously well-defined for the active-collider). We need to prove the correct LIFO ordering between two linearized collided operations.

As we linearize the passive-collider in the linearization point of its counterpart active-collider, no other operations can be linearized between op_1 and op_2; as 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 collider-pairs or that of non-collider operations. Finally, from Lemma 5.10, the pop operation indeed obtains the value of the operation it collided with. □

THEOREM 5.16. *The elimination-backoff stack is a correct linearizable implementation of a stack object.*

PROOF. Immediate from Lemmata 5.13, 5.15 □

5.3 Lock Freedom

THEOREM 5.17. *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 occued, and *op* does not iterate anymore before returning. Otherwise, *op* calls *TryPerformStackOp*; if *TryPerformStackOp* returns TRUE, *op* immediately returns, and its linearization has occured; if, on the other hand, *TryPerformStackOp* returns FALSE, this implies that the CAS performed by it has failed, and the only possible reason for the failure of the CAS by *op* is the success of a CAS on *phead* by some other operation, thus whenever *op* completes a full iteration, some operation is linearized. □

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

[4] I. Corporation. *IBM System/370 Extended Architecture, Principles of Operation*. IBM Publication No. SA22-7085, 1983.

[5] 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.

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

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

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

[21] 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.

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