A Wait-Free Stack

Seep Goel, Pooja Aggarwal and Smruti R. Sarangi
E-mail: seep.goyal@gmail.com, {pooja.aggarwal, srsarangi}@cse.iitd.ac.in
Indian Institute of Technology, New Delhi, India

Abstract. In this paper, we describe a novel algorithm to create a concurrent wait-free stack. To the best of our knowledge, this is the first wait-free algorithm for a general purpose stack. In the past, researchers have proposed restricted wait-free implementations of stacks, lock-free implementations, and efficient universal constructions that can support wait-free stacks. The crux of our wait-free implementation is a fast pop operation that does not modify the stack top; instead, it walks down the stack till it finds a node that is unmarked. It marks it but does not delete it. Subsequently, it is lazily deleted by a cleanup operation. This operation keeps the size of the stack in check by not allowing the size of the stack to increase beyond a factor of \( W \) as compared to the actual size. All our operations are wait-free and linearizable.

1 Introduction

In this paper, we describe an algorithm to create a wait-free stack. A concurrent data structure is said to be wait-free if each operation is guaranteed to complete within a finite number of steps. In comparison, the data structure is said to be lock-free if at any point of time, at least one operation is guaranteed to complete in a finite number of steps. Lock-free programs will not have deadlocks but can have starvation, whereas wait-free programs are starvation free. Wait-free stacks have not received a lot of attention in the past, and we are not aware of algorithms that are particularly tailored to creating a generalized wait-free stack. However, approaches have been proposed to create wait-free stacks with certain restrictions [1], [3], [7], [8], and with universal constructions [10], [5]. The main reason that it has been difficult to create a wait-free stack is because there is a lot of contention at the stack top between concurrent push and pop operations. It has thus been hitherto difficult to realize the gains of additional parallelism, and also guarantee completion in a finite amount of time.

The crux of our algorithm is as follows. We implement a stack as a linked list, where the top pointer points to the stack top. Each push operation adds an element to the linked list, and updates the top pointer. Both of these steps are done atomically, and the overall operation is linearizable (appears to execute instantaneously). However, the pop operation does not update the top pointer. This design decision has been made to enable more parallelism, and reduce the time per operation. It instead scans the list starting from the top pointer till it reaches an unmarked node. Once, it reaches an unmarked node, it marks it and returns the node as the result of the pop operation. Over time, more and more
nodes get marked in the stack. To garbage collect such nodes we implement a cleanup operation that can be invoked by both the push and pop operations. The cleanup operation removes a sequence of $W$ consecutively marked nodes from the list. In our algorithm, we guarantee that at no point of time the size of the list is more than $W$ times the size of the stack (number of pushes - pops). This property ensures that pop operations complete within a finite amount of time. Here, $W$ is a user defined parameter and it needs to be set to an optimal value to ensure the best possible performance.

The novel feature of our algorithm is the cleanup operation that always keeps the size of the stack within limits. It does not allow the number of marked nodes that have already been popped to indefinitely grow. The other novel feature is that concurrent pop and push operations do not cross each others’ paths. Moreover, all the pop operations can take place concurrently. This allows us to have a linearizable operation. In this paper, we present our basic algorithm along with proofs of important results. Readers can find the rest of the pseudo code, asymptotic time complexities, and proofs in the appendices.

2 Related Work

In 1986, Treiber [15] proposed the first lock-free implementation of a concurrent stack. He employed a linked list based data structure, and in his implementation, both the push and pop operations modified the top pointer using CAS instructions. Subsequently, Shavit et al. [14] and Hendler et al. [6] designed a linearizable concurrent stack using the concept of software combining. Here, they group concurrent operations, and operate on the entire group.

In 2004, Hendler et al. [9] proposed a highly scalable lock-free stack using an array of lock-free exchangers known as an elimination array. If a pop operation is paired with a push operation, then the baseline data structure need not be accessed. This greatly enhances the amount of available parallelism, and is known to be one of the most efficient implementations of a lock-free stack. This technique can be incorporated in our design as well. Subsequently, Bar-Nissan et al. [2] have augmented this proposal with software combining based approaches. Recently, Dodds et al. [4] proposed a fast lock-free stack, which uses a timestamp for ordering the push and pop operations.

The restricted wait-free algorithms for the stack data structure proposed so far by the researchers are summarized in Table 1.

The wait-free stack proposed in [1] employs a semi-infinite array as its underlying data structure. A push operation obtains a unique index in the array (using getAndIncrement()) and writes its value to that index. A pop operation starts from the top of the stack, and traverses the stack towards the bottom. It marks and returns the first unmarked node that we find. Our pop operation is inspired by this algorithm. Due to its unrestricted stack size, this algorithm is not practical.

David et al. [3] proposed another class of restricted stack implementations. Their implementation can support a maximum of two concurrent push operations. Kutten et al. [7,8] suggest an approach where a wait-free shared counter can be adapted to create wait-free stacks. However, their algorithm requires the
DCAS (double CAS) primitive, which is not supported in contemporary hardware.

Wait-free universal constructions are generic algorithms that can be used to create linearizable implementations of any object that has valid sequential semantics. The inherent drawback of these approaches is that they typically have high time and space overheads (creates local copies of the entire (or partial) data structure). A recent proposal by Fatourou et al. [5] can be used to implement stacks and queues. The approach derives its performance improvement over the widely accepted universal construction of Herlihy [10] by optimizing on the number of shared memory accesses.

### 3 The Algorithm

#### 3.1 Basic Data Structures

Algorithm 1 shows the Node class, which represents a node in a stack. It has a value, and pointers to the next (nextDone) and previous nodes (prev) respectively. Note that our stack is not a doubly linked list, the next pointer nextDone is only used for reaching consensus on which node will be added next in the stack.

To support pop operations, every node has a mark field. The pushTid field contains the id of the thread that created the request. The index field and counter is an atomic integer and is used to clean up the stack.

| Author          | Primitives | Remarks |
|-----------------|------------|---------|
| Herlihy 1991 [10] | CAS        | 1. Copies every global update to the private copy of every thread. |
| Afek et al. [1]  | F&A, TAS   | 1. Requires a semi-infinite array (impractical). |
| Hendler et al. [12] | DCAS     | 1. DCAS not supported in modern hardware |
| Fatourou et al. [5] | LL/SC, F&A | 1. Copies every global update to the private copy of every thread. |
| David et al. 2011 [13] | BH Object | 1. Supports at the most two concurrent pop operations |

**Table 1.** Summary of existing restricted wait-free stack algorithms

CAS → compare-and-set, TAS → test-and-set, LL/SC → load linked-store conditional DCAS → double location CAS, F&A → fetch-and-add, BH Object (custom object [3]).
3.2 High level Overview

The push operation starts by choosing a phase number (in a monotonically increasing manner), which is greater than the phase numbers of all the existing push operations in the system. This phase number along with a reference to the node to be pushed and a flag indicating the status of the push operation are saved in the announce array in an atomic step. After this, the thread $t_i$ scans the announce array and finds out the thread $t_j$, which has a push request with the least phase number. Note that, the thread $t_j$ found out by $t_i$ in the last step might be $t_i$ itself. Next, $t_i$ helps $t_j$ in completing $t_j$’s operation. At this point of time, some threads other than $t_i$ might also be trying to help $t_j$, and therefore, we must ensure that $t_j$’s operation is applied exactly once. This is ensured by mandating that for the completion of any push request, the following steps must be performed in the exact specified order:

1. Modify the state of the stack in such a manner that all the other push requests in the system must come to know that a push request $p_i$ is in progress and additionally they should be able to figure out the details required by them to help $p_i$.
2. Update the status flag to DONE in $p_i$’s entry in the announce array.
3. Update the top pointer to point to the newly pushed node.

The pop operation has been designed in such a manner that it does not update the top pointer. This decision has the dual benefit of eliminating the contention between concurrent push and pop operations, as well as enabling the parallel execution of multiple pop operations. The pop operation starts by scanning the linked list starting from the stack’s top till it reaches an unmarked node. Once, it gets an unmarked node, it marks it and returns the node as a result of the pop operation. Note that there is no helping in the case of a pop operation and therefore, we do not need to worry about a pop operation being executed twice. Over time, more and more nodes get marked in the stack. To garbage collect such nodes we implement a clean operation that can be invoked by both the push and pop operations.

3.3 The Push Operation

The first step in pushing a node is to create an instance of the PushOp class. It contains the reference to a Node (node), a Boolean variable pushed that indicates the status of the request, and a phase number (phase) to indicate the age of the request. Let us now consider the push method (Line 14). We first get the phase number by atomically incrementing a global counter. Once the PushOp is created and its phase is initialized, it is saved in the announce array. Subsequently, we call the function help to actually execute the push request.

The help function (Line 19) finds the request with the least phase number that has not been pushed yet. If there is no such request, then it returns. Otherwise it helps that request (minReq) to complete by calling the attachNode method. After helping minReq, we check if the request that was helped is the same as the request that was passed as an argument to the help function (request) in Line 19. If they are different requests, then we call attachNode
for the request request in Line \[26\] This is a standard construction to make a lock-free method wait-free (refer to [11]).

In the attachNode function, we first read the value of the top pointer, and its next field. If these fields have not changed between Lines 31 and 32, then we try to find the status of the request in Line 34. Note that we check that next is equal to null, and mark is equal to false in the previous line (Line 33). The mark field is made true after the top pointer has been updated. Hence, in Line 33 if we find it to be true then we need to abort the current iteration and read the top pointer again.

Algorithm 2: The Push Method

```java
0 class PushOp
1 10 long phase
11 boolean pushed
12 Node node
13 AtomicReferenceArray < PushOp > announce
14 push(tid, value)
15 phase ← globalPhase.getAndIncrement()
16 request ← new PushOp(phase, false, new Node(value, tid))
17 announce[tid] ← request
18 help(request)
19 help(request)
20 (minTid, minReq) ← minReq.phase { (i, req) | 0 ≤ i < N, req = announce[i], !req.pushed } (minReq == null) || (minReq.phase > request.phase) then
21 | return
22 | attachNode(minReq)
23 if minReq ≠ request then
24 | attachNode(request)
25 | while request.pushed do
26 | last ← top.get()
27 | (next, done) ← last.nextDone.get()
28 | if last == top.get() then
29 | if next == null & done = false then
30 | if request.pushed then
31 | myNode ← request.node
32 | res ← last.nextDone.compareAndSet (null, false), (myNode, false)
33 | if res then
34 | updateTop()
35 | last.nextDone.compareAndSet (myNode, false), (null, true)
36 | return
37 | end
38 | end
39 | updateTop()
40 | end
41 | end
```

After, we read the status of the request, and find that it has not completed, we proceed to update the next field of the stack top in Line 36 using a compare-And-Set (CAS) instruction. The aim is to change the pointer in the next field from null to the node the push request needs to add. If we are successful, then we
need to update the top pointer by calling the function, updateTop. After the top pointer has been updated, we do not really need the next field for subsequent push requests. It will not be used. However, concurrent requests need to see that last.nextDone has been updated. The additional compulsion to delete the contents of the pointer in the next field is that it is possible to have references to deleted nodes via the next field. The garbage collector in this case will not be able to remove the deleted nodes. Thus, after updating the top pointer, we set the next field’s pointer to null, and set the mark to true. If a concurrent request reads the mark to be true, then it can be sure, that the top pointer has been updated, and it needs to read it again.

If the CAS instruction fails, then it means that another concurrent request has successfully performed a CAS operation. However, it might not have updated the top pointer. It is thus necessary to call the updateTop function to help the request complete.

Algorithm 3: The updateTop method

| Line | Code |
|------|------|
| 47   | updateTop() |
| 48   | last ← top.get() |
| 49   | (next, mark) ← last.nextDone.get() |
| 50   | if next ≠ null then |
| 51   | request ← announce.get(next.pushTid) |
| 52   | if last == top.get() && request.node == next then |
| 53   | /* Add the request to the stack and update the top pointer */ |
| 54   | next.prev.compareAndSet(null, last) |
| 55   | next.index ← last.index +1 |
| 56   | request.pushed ← true |
| 57   | stat ← top.compareAndSet(last, next) |
| 58   | /* Check if any cleaning up has to be done */ |
| 59   | if next.index % W == 0 && stat == true then |
| 60   | tryCleanUp(next) |
| 61   | end |
| 62   | end |

The updateTop method is shown in Algorithm 3. We read the top pointer, and the next pointer. If next is non-null, then the request has not fully completed. It is necessary to help it complete. After having checked the value of the top pointer, and the value of the next field, we proceed to connect the newly attached node to the stack by updating its prev pointer. We set the value of its prev pointer in Line 54. Every node in the stack has an index that is assigned in a monotonically increasing order. Hence, in Line 55, we set the index of next to 1 plus the index of last. Next, we set the pushed field of the request equal to true. The point of linearizability is Line 57, where we update the top pointer to point to next instead of last. This completes the push operation.

We have a cleanup mechanism that is invoked once the index of a node becomes a multiple of a constant, W. We invoke the tryCleanUp method in Line 60. It is necessary that the tryCleanUp() method be called by only one thread. Hence, the thread that successfully performed a CAS on the top pointer calls the tryCleanUp method if the index is a multiple of W.
3.4 The Pop Operation

Algorithm 4 shows the code for the pop method. We read the value of the top pointer and save it in the local variable, myTop. This is the only instance in this function, where we read the value of the top pointer. Then, we walk back towards the sentinel node by following the prev pointers (Lines 67 – 73). We stop when we are successfully able to set the mark of a node that is unmarked. This node is logically “popped” at this instant of time. If we are not able to find any such node, and we reach the sentinel node, then we throw an EmptyStackException.

Algorithm 4: The Pop Method

| Line | Code                                                                 |
|------|----------------------------------------------------------------------|
| 64   | `pop()`                                                             |
| 65   | `mytop ← top.get()`                                                |
| 66   | `curr ← mytop`                                                     |
| 67   | `while curr ≠ sentinel do`                                         |
| 68   | `mark ← curr.mark.getAndSet(true)`                                |
| 69   | `if !mark then`                                                     |
| 70   | `break`                                                             |
| 71   | `end`                                                               |
| 72   | `curr ← curr.prev`                                                 |
| 73   | `end`                                                               |
| 74   | `if curr == sentinel then`                                          |
| 75   | `/* Reached the end of the stack */`                               |
| 76   | `throw new EmptyStackException()`                                  |
| 77   | `end`                                                               |
| 78   | `/* Try to clean up parts of the stack */`                          |
| 79   | `tryCleanUp(curr)`                                                  |
| 80   | `return curr`                                                      |

After logically marking a node as popped, it is time to physically delete it. We thus call the tryCleanUp method in Line 79. The pop method returns the node that it had successfully marked.

3.5 The CleanUp Operation

The aim of the clean method is to clean a set of $W$ contiguous entries in the list (indexed by the prev pointers). Let us start by defining some terminology. Let us define a range of $W$ contiguous entries, which has four distinguished nodes as shown in Figure 1.

A range starts with a node termed the base, whose index is a multiple of $W$. Let us now define target as base.prev. The node at the end of a range is leftNode. Its index is equal to base.index + $W - 1$. Let us now define a node rightNode such that rightNode.prev = leftNode. Note that for a given range, the base and leftNode nodes are fixed, whereas the target and rightNode nodes keep changing. rightNode is the base of another range, and its index is a multiple of $W$.

The push and the pop methods call the function tryCleanUp. The push method calls it when it pushes a node whose index is a multiple of $W$. This is a valid rightNode. It walks back and increments the counter of the base node of the previous range. We ensure that only one thread (out of all the helpers) does this in Line 59. Similarly, in the pop function, whenever we mark a node, we call
the tryCleanUp function. Since the pop function does not have any helpers, only one thread per node calls the tryCleanUp function. Now, inside the tryCleanUp function, we increment the counter of the base node. Once, a thread increments it to $W + 1$, it invokes the clean function. Since only one thread will increment the counter to $W + 1$, only one thread will invoke the clean function for a range.

![Fig. 1. A range of W entries](image)

| Algorithm 5: The tryCleanUp method |
|-----------------------------------|
| 81 tryCleanUp(myNode) |
| 82 \( \text{temp} \leftarrow \text{myNode.prev} \) |
| 83 while \( \text{temp} \neq \text{sentinel} \) do |
| 84 \( \text{if \( \text{temp.index}() \) \( \% \) \( W \) == 0 then} \) |
| 85 \( \text{if \( \text{temp.counter.incrementAndGet} \) == } W + 1 \text{ then} \) |
| 86 \( \text{clean(getTid(), temp)} \) |
| 87 end |
| 88 \( \text{break} \) |
| 89 end |
| 90 \( \text{temp} \leftarrow \text{temp.prev}() \) |

The functionality of the clean function is very similar to the push function. Here, we first create a DeleteRequest that has four fields: phase (similar to phase in PushOp), threadId, pending (whether the delete has been finished or not), and the value of the base node. Akin to the push function, we add the newly created DeleteRequest to a global array of DeleteRequests. Subsequently, we find the pending request with the minimum phase in the array allDeleteRequests.

Note that at this stage it is possible for multiple threads to read the same value of the request with the minimum phase number. It is also possible for different sets of threads to have found different requests to have the minimum phase. For example, if a request with phase 2 ($R_2$) got added to the array before the request with phase 1 ($R_1$), then a set of threads might be trying to complete $R_2$, and another set might be trying to complete $R_1$. To ensure that our stack remains in a consistent state, we want that only one set goes through to the next stage.

To achieve this, we adopt a strategy similar to the one adopted in the function attachNode. Interested readers can refer to the appendices for a detailed explanation of how this is done. Beyond this point, all the threads will be working on the same DeleteRequest which we term as uniqueRequest. They will then move on to call the helpFinishDelete function that will actually finish the delete request.

Let us describe the helpFinishDelete function in Algorithm 6. We first read the current request from the atomic variable, uniqueRequest in Line 93. If the
request is not pending, then some other helper has completed the request, and we can return from the function. However, if this is not the case, then we need to complete the delete operation. Our aim now is to find the \textit{target}, \textit{leftNode}, and \textit{rightNode}. We search for these nodes starting from the stack top.

The index of the \textit{leftNode} is equal to the index of the node in the current request (\textit{currRequest}) + \(W - 1\). \textit{endIdx} is set to this value in Line 97. Subsequently, in Lines 101–106 we start from the top of the stack, and keep traversing the \textit{prev} pointers till the index of \textit{leftNode} is equal to \textit{endIdx}. Once, the equality condition is satisfied, Lines 101 and 102 give us the pointers to the \textit{rightNode} and \textit{leftNode} respectively. If we are not able to find the \textit{leftNode}, then it means that another helper has successfully deleted the nodes. We can thus return.

\begin{algorithm}
\caption{The \texttt{helpFinishDelete} method}
\begin{algorithmic}[1]
\State \texttt{helpFinishDelete()}
\State \texttt{currRequest ← uniqueRequest.get()}
\If {\texttt{!currRequest.pending}}
\State \texttt{return}
\EndIf
\State \texttt{endIdx ← currRequest.node.index + W - 1}
\State \texttt{leftNode ← rightNode.prev} /* Search for the request from the top */
\While {\texttt{leftNode.index ≠ endIdx} \&\& \texttt{leftNode ≠ sentinel}}
\State \texttt{rightNode ← leftNode}
\State \texttt{leftNode ← leftNode.prev}
\EndWhile
\If {\texttt{leftNode = sentinel}}
\State \texttt{return} /* some other thread deleted the nodes */
\EndIf
\State /* Find the target node */
\State \texttt{target ← leftNode}
\For {\texttt{i = 0; i < W; i++}}
\State \texttt{target ← target.prev}
\EndFor
\State \texttt{rightNode.prev.compareAndSet(leftNode, target)} /* Perform the CAS operation and delete the nodes */
\State \texttt{currRequest.pending ← false} /* Set the status of the delete request to not pending*/
\end{algorithmic}
\end{algorithm}

The next task is to find the \textit{target}. The \textit{target} is \(W\) hops away from the \textit{leftNode}. Lines 108–111 run a loop \(W\) times to find the target. Note that we shall never have any issues with null pointers because \textit{sentinel.prev} is set to \textit{sentinel} itself. Once, we have found the target, we need to perform a CAS operation on the \textit{prev} pointer of the \textit{rightNode}. We accomplish this in Line 112. If the \textit{prev} pointer of \textit{rightNode} is equal to \textit{leftNode}, then we set it to \textit{target}. This operation removes \(W\) entries (from \textit{leftNode} to \textit{base}) from the list. The last step is to set the status of the \textit{pending} field in the current request (\textit{currRequest}) to false (see Line 113).

4 Proof of Correctness

The most popular correctness criteria for a concurrent shared object is \textit{linearizability} [12]. Linearizability ensures that within the execution interval of every
operation there is a point, called the linearization point, where the operation seems to take effect instantaneously and the effect of all the operations on the object is consistent with the object’s sequential specification. By the property of compositional linearizability, if each method of an object is linearizable we can conclude that the complete object is linearizable. Thus, if we identify the point of linearization for both the push and the pop method in our implementation, we can say that our implementation is linearizable and thus establish its correctness.

Interested readers can refer to the appendices, where we show that our implementation is legal and push and pop operations complete in a bounded number of steps.

**Theorem 1.** The push and pop operations are linearizable.

**Proof.** Let us start out by defining the notion of “pass points”. The pass point of a push operation is when it successfully updates the top pointer in the function `updateTop` (Line 57). The pass point of the pop operation, is when it successfully marks a node, or when it throws the `EmptyStackException`. Let us now try to prove by mathematical induction on the number of requests that it is always possible to construct a linearizable execution that is equivalent to a given execution. In a linearizable execution all the operations are arranged in a sequential order, and if request $r_i$ precedes $r_j$ in the original execution, then $r_i$ precedes $r_j$ in the linearizable execution as well.

**Base Case:** Let us consider an execution with only one pass point. Since the execution is complete, we can conclude that there was only one request in the system. An equivalent linearizable execution will have a single request. The outcome of the request will be an `EmptyStackException` if it is a pop request, otherwise it will push a node to the stack. Our algorithm will do exactly the same in the pop and `attachNode` methods respectively. Hence, the executions are equivalent.

**Induction Hypothesis:** Let us assume that all executions with $n$ requests are equivalent to linearizable executions.

**Inductive Step:** Let us now prove our hypothesis for executions with $n + 1$ requests. Let us arrange all the requests in an ascending order of the execution times of their pass points. Let us consider the last ($(n + 1)^{th}$) request just after the pass point of the $n^{th}$ request. Let the last request be a push. If the $n^{th}$ request is also a push, then the last request will use the top pointer updated by the $n^{th}$ request. Additionally, in this case the $n^{th}$ request will not see any changes made by the last request. It will update `last.next` and the top pointer, before the last request updates them. In a similar manner we can prove that no prior push request will see the last request. Let us now consider a prior pop request. A pop request scans all the nodes between the top pointer and the sentinel. None of the pop requests will see the updated top pointer by the last request because their pass points are before this event. Thus, they have no way of knowing about the existence of the last request. Since the execution of the first $n$ requests is linearizable, an execution with the $(n + 1)^{th}$ push request is also linearizable because it takes effect at the end (and will appear last in the equivalent sequential order).
Let us now consider the last request to be a pop operation. A pop operation writes to any shared location only after its pass point. Before its pass point, it does not do any writes, and thus all other requests are oblivious of it. Thus, we can remove the last request, and the responses of the first \( n \) requests will remain the same. Let us now consider an execution fragment consisting of the first \( n \) requests. It is equivalent to a linearizable execution, \( \mathcal{E} \). This execution is independent of the \((n + 1)\)th request.

Now, let us try to create a linearizable execution, \( \mathcal{E}' \), which has an event corresponding to the last request. Since the linearizable execution is sequential, let us represent the request and response of the last pop operation by a single event, \( R \). Let us try to modify \( \mathcal{E} \) to create \( \mathcal{E}' \). Let the sequential execution corresponding to \( \mathcal{E} \) be \( S \).

Now, it is possible that \( R \) could have read the top pointer long ago, and is somewhere in the middle of the stack. In this case, we cannot assume that \( R \) is the last request to execute in the equivalent linearizable execution. Let the state of the stack before the pop reads the top pointer be \( S' \). The state \( S' \) is independent of the pop request. Also note that, all the operations that have arrived after the pop operation have read the top pointer, and overlap with the pop operation. The basic rule of linearizability states that, if any operation \( R_i \) precedes \( R_j \), then \( R_i \) should precede \( R_j \) in the equivalent sequential execution also. Whereas, in case the two operations overlap with each other, then their relative order is undefined and any ordering of these operations is a valid ordering [11].

In this case, we have two possibilities: (I) \( R \) returns the node that it had read as the top pointer as an output of its pop operation, or (II) it returns some other node.

**Case I:** In this case, we can consider the point at which \( R \) reads the top pointer as the point at which it is linearized. \( R \) in this case reads the stack top, and pops it.

**Case II:** In this case, some other request, which is concurrent must have popped the node that \( R \) read as the top pointer. Let \( R \) return node \( N_i \) as its return value. This node must be between the top pointer that it had read (node \( N_{top} \)), and the beginning of the stack. Moreover, while traversing the stack from \( N_{top} \) to \( N_i \), \( R \) must have found all the nodes in the way to be marked. At the end it must have found \( N_i \) to be unmarked, or would have found \( N_i \) to be the end of the stack (returns exception).

Let us consider the journey for \( R \) from \( N_{top} \) to \( N_i \): Let \( N_j \) be the last node before \( N_i \) that has been marked by a concurrent request, \( R_j \). We claim that if \( R \) is linearized right after \( R_j \), and the rest of the sequences of events in \( \mathcal{E} \) remain the same, we have a linearizable execution \( \mathcal{E}' \).

Let us consider request \( R_j \) and its position in the sequential execution, \( S \). At its point of linearization, it reads the top of the stack and returns it (according to \( S \)). This node \( N_j \) is the successor of \( N_i \). At that point \( N_i \) becomes the top of the stack. At this position, if we insert \( R \) into \( S \), then it will read and return \( N_i \) as the stack top, which is the correct value. Subsequently, we can insert the remaining events in \( S \) into the sequential execution. They will still return the same set of values because they are unaffected by \( R \) as proved before.

This proof can be trivially extended to take cleanup operations into account.
5 Conclusion

The crux of our algorithm is the clean routine, which ensures that the size of the stack never grows beyond a predefined factor, $W$. This feature allows for a very fast pop operation, where we need to find the first entry from the top of the stack that is not marked. This optimization also allows for an increased amount of parallelism, and also decreases write-contention on the top pointer because it is not updated by pop operations. As a result, the time per pop operation is very low. The push operation is also designed to be very fast. It simply needs to update the top pointer to point to the new data. To provide wait-free guarantees it was necessary to design a clean function that is slow. Fortunately, it is not invoked for an average of $W - 1$ out of $W$ invocations of push and pop. We can tune the frequency of the clean operation by varying the parameter, $W$ (to be decided on the basis of the workload).

References

1. Afek, Y., Gafni, E., Morrison, A.: Common2 extended to stacks and unbounded concurrency. Distributed Computing 20(4), 239–252 (2007)
2. Bar-Nissan, G., Hendler, D., Sussa, A.: A dynamic elimination-combining stack algorithm. In: Principles of Distributed Systems, pp. 544–561. Springer (2011)
3. David, M., Brodsly, A., Fich, F.: Restricted stack implementations. In: Fraigniaud, P. (ed.) Distributed Computing, Lecture Notes in Computer Science, vol. 3724, pp. 137–151. Springer Berlin Heidelberg (2005)
4. Dodds, M., Haas, A., Kirsch, C.M.: A scalable, correct time-stamped stack (2014)
5. Fatourou, P., Kallimanis, N.D.: A highly-efficient wait-free universal construction. In: SPAA (2011)
6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures, pp. 355–364. ACM (2010)
7. Hendler, D., Kuten, S.: Constructing shared objects that are both robust and high-throughput. In: Distributed Computing, pp. 428–442. Springer (2006)
8. Hendler, D., Kuten, S., Michalak, E.: An adaptive technique for constructing robust and high-throughput shared objects-technical report (2010)
9. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 206–215. SPAA ’04, ACM, New York, NY, USA (2004)
10. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (Jan 1991)
11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Elsevier (2012)
12. Herlihy, M.P., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (Jul 1990)
13. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing 51(1), 1–26 (1998)
14. Shavit, N., Zemach, A.: Combining funnels: a dynamic approach to software combining. Journal of Parallel and Distributed Computing 60(11), 1355–1387 (2000)
15. Treiber, R.K.: Systems programming: Coping with parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center (1986)
Appendices of A Wait-Free Stack

A Asymptotic Worst-Case Time Complexity

Let us now consider the asymptotic worst-case time complexity of the push, pop and clean methods in terms of the number of concurrent threads in the system \((N)\), the actual size of the stack\((S)\) and the parameter \(W\).

A.1 The clean method

The time complexity of the clean method is the same as that of the helpDelete function. The helpDelete function finds the delete request with the minimum phase number, which requires \(O(N)\) steps. After having found the request with the minimum phase number, it calls the uniqueDelete function. The uniqueDelete function contains a while loop. In the worst case this while loop might execute \(N\) times. Now, when we look into the body of this while loop, everything except the call to the helpFinishDelete function has \(O(1)\) time complexity. The helpFinishDelete function contains two loops. The first is a while loop, which traverses the stack from the top to the point it finds the desired node. In the worst possible case, this loop might end up traversing the complete stack. We do not allow the size of the stack to increase by more than a factor of \(W\) as compared to \(S\), the worst case time complexity of this loop is therefore \(O(WS)\). The other loop in the function is a for loop, with time complexity \(O(W)\). So, the worst case time complexity of the helpFinishDelete function is \(O(WS)\) and therefore, the worst case time complexity of the clean function is \(O(NWS)\). The high time complexity of this method is an achilles heel of our algorithm; hence, we are working on reducing its complexity as well as practical run time. However, it should be noted that this function is meant to be called infrequently (1 in \(W\) times).

A.2 The pop method

In the pop method, everything except the while loop and the call to the tryCleanUp function take \(O(1)\) time. The while loop is iterated over till the time an unmarked mode is encountered. In our algorithm, as soon as \(W\) consecutive nodes get marked, we issue a cleanUp request for it and at any point of time there can be at most \(N - 1\) clean requests in the system. Thus after having traversed at most \(WN\) nodes, a pop request is assured to find an unmarked node. Now, if we analyze the tryCleanUp method, in the worst case scenario, the while loop inside the function will be iterated over \(W\) times but the clean method will only be called at most once. In fact, the clean method is called only once for a group of \(W + 1\) operations, and therefore, the worst case time complexity of the tryCleanUp function, which is \(O(NWS)\), will be incurred very infrequently (1 in \(W\)). Nevertheless the worst case time complexity of the pop operation is \(O(NWS)\), and the amortized time complexity (across \(W\) pop operations is \(O(NS)\).
### A.3 The push method

The time complexity of the push method is the same as that of the help function. Since the help function is supposed to find the request with the least phase number, it takes at least $O(N)$ time. After having found the request with the minimum phase number, the help function calls the attachNode function. Note that for any push request, the maximum number of times the while loop in the attachNode function could possibly execute is of $O(N)$. Also note that everything inside the while loop, except the call to the updateTop function requires only a constant amount of time for execution. If the index of the newly pushed node is not a multiple of $W$, the updateTop’s time complexity is $O(1)$, and therefore the time complexity of the push operation is $O(N)$, but if this is not the case, the time complexity of the updateTop function becomes dependent on the time complexity of the tryCleanUp function, which is $O(NWS)$ in the worst case.

All our methods: push, pop and clean are bounded wait-free.

### B Background

A stack is a data structure that provides push and pop operations with LIFO (Last-in-First-Out) semantics. A data structure is said to respect LIFO semantics, if the last element inserted is the first to be removed.

#### B.1 Correctness

The most popular correctness criteria for a concurrent shared object is linearizability [12]. Let us define it formally.

Let us define two kinds of events in a method call namely invocations (inv) and responses (resp). A chronological sequence of invocations (inv) and responses (resp) events in the entire execution is known as a history. Let a matching invocation-response pair with a sequence number $i$ be referred to as request $r_i$. Note that in our system, every invocation has exactly one matching response. A request $r_i$ precedes request $r_j$, if $r_i$’s response comes before $r_j$’s invocation. This is denoted by $r_i \prec r_j$. A history, $H$, is said to be sequential if an invocation is immediately followed by its response. A subhistory $(H|T)$ is the subsequence of $H$ containing all the events of thread $T$. Two histories, $H$ and $H'$, are equivalent if for every thread , $H|T = H'|T$. A complete history - complete($H$) is a history that does not have any pending invocations. The sequential specification of an object constitutes of the set of all sequential histories that are correct. A sequential history is legal if for every object $x$, $H|x$ is in the sequential specification of $x$.

A history $H$ is linearizable if complete$(H)$ is equivalent to a legal sequential history, $S$. Additionally, if $r_i \prec r_j$ in complete$(H)$, then $r_i \prec r_j$ in $S$ also. Alternatively we can say, linearizability ensures that within the execution interval of every operation there is a point, called the linearization point, where the operation seems to take effect instantaneously and the effect of all the operations on the object is consistent with the object’s sequential specification.
B.2 Progress

Generally, there are two kinds of implementations for a concurrent object: blocking and non-blocking. Blocking algorithms use locks. Approaches that protect critical sections with locks unnecessarily limit parallelism and are known to be inefficient.

In comparison, non-blocking implementations can prove to be much faster. Such algorithms rely on atomic primitives such as compare-And-Set (CAS), LL/SC, and getAndIncrement. They do not have critical sections. In this context, lock-freedom is defined as a property that ensures that at any point of time at least one thread makes progress. Or in other words, the system as a whole is always making progress. They can still have problems of starvation.

Wait-free algorithms provide starvation freedom in addition to being lock-free. They ensure that every process completes its operation in a finite number of steps. The wait-free algorithms have a notion of inherent fairness, where fairness measures the degree of imbalance across different threads. We quantify fairness as the ratio of the average number of operations completed by an thread divided by the number of operations completed by the fastest thread. Figure 2 shows a comparison of the fairness of our wait-free stack WF with the lock-free stack LF in [9] and the locked stack in [13] LCK. For the WF version, the average fairness is around 80%, whereas for LF the fairness goes as low as 50%, and for LCK, it even drops to 25%. Also, as shown in figure 3 and 4 in the case of WF, almost all the threads have completed more than 90% of their work, whereas for LF only 11 out of 64 threads have completed more than 90% of their work and for some threads the percentage of work done is as low as 40% only.

C Proof of Correctness

Lemma 1. Every push request is inserted at Line 36 at most once.

Proof. To the contrary, let us assume that the same request is inserted at least twice in Line 36. Let us consider the sequence of steps that need to take place.
Step 0: Read the value of last and next, and observe that next = null.
Step 1: We read the status as start in Line 34.
Step 2: The CAS succeeds in Line 36.
Step 3: Some thread reads next ≠ null in Line 30 (updateTop function).
Step 4: The status of the node is updated to done by some thread in Line 56.
Step 5: The top pointer is changed in Line 57.

Let us now explain the steps and mention why these steps need to be performed in a sequence (not necessarily by the same thread). To insert any node in Line 36 it is necessary to perform steps 0, 1 and 2 because to reach Line 36 it is necessary to satisfy the condition of the if statement in Line 34. At this point, the next pointer has been updated, and the top pointer has not been updated. It is not possible to insert any other request till the next pointer does not become null. This is only possible when we update the top pointer. To update the top pointer some thread – either the thread that is doing the push operation, or some other thread – needs to successfully perform the updateTop operation.

This can only be achieved if a thread executes Lines 60 to 77 or in other words, performs steps 3, 4, and 5. Before the top pointer is updated in Line 57 another concurrent push request cannot be successful because two push requests cannot simultaneously perform a successful CAS operation in step 2.

Now we have proved that if any push operation has successfully completed step 2 (performed the CAS on the next pointer), then till steps 3, 4, and 5 are performed no other push request can be successful. This means that between any two successful push operations, the status of the request needs to be changed (step 4). Let us now assume that two push requests for the same node are successful. Let the request that performs step 2 first be Ri, and let the other request be Rj. We denote this fact as: Ri ≺ Rj.

Rj must read a different value of the stack top in step 0. Otherwise, it will find last.next to be non-null and it will not proceed beyond step 0. Subsequently, it will try to read the status of the request in step 1. Note that this step is preceded by step 4 of Ri. Thus Rj will read the status to done, and it will not be able to proceed to step 2. Consequently, Rj will not be able to do the push operation done by Ri once again.

Hence, the lemma stands proved.

Lemma 2. A push request always adds an entry to the top of the stack in Line 36 in a bounded number of steps.

Proof. Let us assume that a push request, Ri never gets fulfilled. This can happen because it either fails the if conditions in Lines 32 and 33 or the CAS operation in Line 36. This can only happen if some other push request makes progress. Now, let us assume that Ri has the least phase number out of all the push requests that remain unfulfilled for an unbounded amount of time.

Since Ri has the least phase number out of all the unfulfilled requests, all other request with a lower phase number must have gotten fulfilled. This means that there is a point of time at which Ri has the least phase in the announce array. At this point all the threads must be helping Ri to complete its request. One of the threads needs to perform a successful CAS in Line 36. Either that
thread or some other thread can update the top pointer. In this manner, request $R_i$ will get satisfied.

Hence, it is not possible to have a request, $R_i$ that waits for an infinite amount of time to get fulfilled.

**Theorem 2.** A push request adds an entry only once to the stack in a bounded number of steps. Furthermore, if we just consider the next pointers, the stack is always a linked list without duplicate nodes. Thus, the push operation is lock-free.

**Proof.** Lemma 1 and Lemma 2 prove that an entry is added only once (in a bounded number of steps). Secondly, we are allowed to modify the next pointer of a node only once. It cannot only point to another node. Each node points to another node that is pushed to the stack after it because steps 2-5 are executed in sequence. Thus the stack at all times has a structure similar to a linked list. The end of the linked list is the stack top. The pop method does not touch the next pointer; hence, this property is maintained.

**Lemma 3.** Every pop operation pops just one element or returns an EmptyStackException.

**Proof.** We start at the stack top ($mytop ← top.get()$), and proceed towards the bottom of the stack. If we get any unmarked node, then we mark it. After marking a node the pop operation is over. Note that there is no helping in the case of a pop operation. Hence, other threads do not work on behalf of a thread. As a result only one node is marked (or popped). If we are not able to mark a node then we return an EmptyStackException.

### D The clean Method

Let us consider the clean method first. It is called by the tryCleanUp method in Line 86. The aim of the clean method is to clean a set of $W$ contiguous entries in the list (indexed by the prev pointers). Let us start out by defining some terminology. Let us define a range of $W$ contiguous entries, which has four distinguished nodes (see Figure 5).

![Fig. 5. A range of W entries](image)

A range starts with a node termed the base, whose index is a multiple of $W$. Let us now define target as base.prev. The node at the end of a range is
leftNode. Its index is equal to base.index + W − 1. Let us now define a node rightNode such that rightNode.prev = leftNode. Note that for a given range, the base and leftNode nodes are fixed, whereas the target and rightNode nodes keep changing. rightNode is the base of another range, and its index is a multiple of W.

The push and the pop methods call the function tryCleanUp. The push method calls it when it pushes a node whose index is a multiple of W. This is a valid rightNode. It walks back and increments the counter of the base node of the previous range. We ensure that only one thread (out of all the helpers) does this in Line 59. Similarly, in the pop function, whenever we mark a node, we call the tryCleanUp function. Since the pop function does not have any helpers, only one thread per node calls the tryCleanUp function. Now, inside the tryCleanUp function, we increment the counter of the base node. Once, a thread increments it to W + 1, it invokes the clean function. Since only one thread will increment the counter to W + 1, only one thread will invoke the clean function for a range.

Algorithm 7: DeleteRequest

114 class DeleteRequest
115    long phase
116    int threadId
117    AtomicBoolean pending
118    Node node
119    AtomicReferenceArray<DeleteRequest> allDeleteRequests

Let us now consider the clean, helpDelete, and uniqueDelete functions. Their functionality at a high level is very similar to the push and help methods. Here, we first create a DeleteRequest that has four fields: phase (similar to phase in PushOp), threadId, pending (whether the delete has been finished or not), and the value of the base node. Akin to the push function, we add the newly created DeleteRequest to a global array of DeleteRequests in Line 123. Subsequently, we call the helpDelete function. This function finds a pending request with the minimum phase in the array allDeleteRequests, and returns the request as minReq. Subsequently, we invoke uniqueDelete.

Note that at this stage it is possible for multiple threads to read the same value of the request with the minimum phase number. It is also possible for different sets of threads to have found different requests to have the minimum phase. For example, if a request with phase 2 (R2) got added to the array before the request with phase 1 (R1), then a set of threads might be trying to perform uniqueDelete on R1, and another set might be trying to perform uniqueDelete on R1. Our aim in the uniqueDelete function is to ensure that only one set goes through to the next stage. It takes two arguments: req (request) and phase (phase number).
Algorithm 8: clean, helpDelete, and uniqueDelete methods

120 clean(tid, node)
121 phase ← deletePhase.getAndIncrement()
122 request ← new DeleteRequest(phase, tid, true, node)
123 allDeleteRequests[tid] ← request
124 helpDelete(request)
125 helpDelete(request)
126 (minTid, minReq) ← min_{req.phase \{ i, req \mid 0 \leq i < N, req = allDeleteRequests[i], req.pending = true \} }
127 if (minReq == null) || (minReq.phase > request.phase) then
128     break
129 end
130 uniqueDelete(minReq)
131 if minReq ≠ request then
132     uniqueDelete(request)
133 end
134 uniqueDelete(request)
135 while request.pending do
136     currRequest ← uniqueRequest.get()
137     if !currRequest.pending then
138         if request.pending then
139             stat ← (request ≠ currRequest) ? uniqueRequest.compareAndSet(currRequest, request) : true
140             helpFinishDelete()
141             if stat then
142                 return
143             end
144         end
145     end
146 else
147     helpFinishDelete()
148 end
149 end
We adopt a strategy similar to the one adopted in the function `attachNode`. We define a global atomic variable, `uniqueRequest`. If a delete is not pending (Line 137) on `uniqueRequest`, we read its contents, and try to perform a CAS operation on it. We try to atomically replace its current contents with the argument, `req`. Note that at this stage, only one set of threads will be successful. Beyond this point, all the threads will be working on the same `DeleteRequest`. They will then move on to call the `helpFinishDelete` function that will finish the delete request. For threads that are not successful in the CAS operation, or threads that find that the current request contained in `uniqueRequest` has a delete pending will also call the `helpFinishDelete` function. This is required to ensure wait freedom.

**Algorithm 9: The `helpFinishDelete` method**

```plaintext
helpFinishDelete()

150  currRequest ← uniqueRequest.get()
151  if !currRequest.pending then
152      return
153  end

154  endIdx ← currRequest.node.index + W - 1
155  /* Search for the request from the top */
156  rightNode ← top.get()
157  leftNode ← rightNode.prev
158  while leftNode.index ≠ endIdx && leftNode ≠ sentinel do
159      rightNode ← leftNode
160      leftNode ← leftNode.prev
161  end
162  if leftNode = sentinel then
163      return /* some other thread deleted the nodes */
164  end

165  /* Find the target node */
166  target ← leftNode
167  for i=0; i < W; i++ do
168      target ← target.prev
169  end

170  /* Perform the CAS operation and delete the nodes */
171  rightNode.prev.compareAndSet(leftNode, target)

172  /* Set the status of the delete request to not pending*/
173  currRequest.pending ← false
```
Lastly, let us describe the helpFinishDelete function in Algorithm 9. We first read the current request from the atomic variable, uniqueRequest in Line 151. If the request is not pending, then some other helper has completed the request, and we can return from the function. However, if this is not the case, then we need to complete the delete operation. Our aim now is to find the target, leftNode, and rightNode. We search for these nodes starting from the stack top.

The index of the leftNode is equal to the index of the node in the current request (currRequest) + W − 1. endIdx is set to this value in Line 155. Subsequently, in Lines [160][165] we start from the top of the stack, and keep traversing the prev pointers till the index of leftNode is equal to endIdx. Once, the equality condition is satisfied Lines [160] and [161] give us the pointers to the rightNode and leftNode respectively. If we are not able to find the leftNode, then it means that another helper has successfully deleted the nodes. We can thus return.

The next task is to find the target. The target is W hops away from the leftNode. Lines [167][170] run a loop W times to find the target. Note that we shall never have any issues with null pointers because sentinel.prev is set to sentinel itself. Once, we have found the target, we need to perform a CAS operation on the prev pointer of the rightNode. We accomplish this in Line 172. If the prev pointer of rightNode is equal to leftNode, then we set it to target. This operation removes W entries (from leftNode to base) from the list. The last step is to set the status of the pending field in the current request (currRequest) to false (see Line 174).