FA-Stack: A Fast Array-Based Stack with Wait-Free Progress Guarantee

Yaqiong Peng, *Member, IEEE* and Zhiyu Hao

Abstract—The prevalence of multicore processors necessitates the design of efficient concurrent data structures. Shared concurrent stacks are widely used as inter-thread communication structures in parallel applications. Wait-free stacks can ensure that each thread completes operations on them in a finite number of steps. This characteristic is valuable for parallel applications and operating systems, especially in real-time environments. Unfortunately, because wait-free algorithms are typically hard to design and considered inefficient, practical wait-free stacks are rare. In this paper, we present a practical, fast array-based concurrent stack with wait-free progress guarantee, named FA-Stack. A series of optimizations are proposed to bound the number of steps required to complete every push and pop operation. In addition, FA-Stack adopts a time-stamped scheme to reclaim memory. We use *linearizability*, a correctness condition for concurrent data structures, to prove that FA-Stack is a wait-free linearizable stack with respect to the *Last in First Out* (LIFO) semantics. Our evaluation with representative benchmarks shows that FA-Stack is an efficient wait-free stack. For example, compared to Sim-Stack (a state-of-the-art wait-free stack), FA-Stack improves the throughput of *halfhalf* benchmark by up to 2.4×.

Index Terms—Concurrent stack, wait-free, multicore

1 Introduction

1.1 Motivation

With the prevalence of multicore processors in modern computer systems, it is necessary to develop parallel applications to fully exploit the abundant computing resources of such platforms. Shared concurrent stacks are widely used as inter-thread communication structures, and thus they are major building blocks of parallel applications [1], [2]. To ensure high performance of parallel applications on multicore, it is fundamentally important to design efficient concurrent stacks.

Besides high performance, *progress guarantee* is another important feature of concurrent stacks. Typically, concurrent stacks fall into two classes: *blocking* and *non-blocking*. Before performing an operation on blocking stacks, a thread may need to wait for the completion of an operation run by another thread. Therefore, blocking stacks can introduce problems such as deadlock and priority inversion.

Non-blocking stacks ensure that suspended operations will not block other operations, thus avoiding the aforementioned problems introduced by blocking stacks. Non-blocking stacks provide three levels of progress guarantees from weak to strong, namely *obstruction-free*,[1] *lock-free*,[2] and *wait-free*.[3] Wait-freedom can avoid starvation for every thread. Thus, wait-free stacks are valuable for parallel applications and operating systems, especially in real-time environments.

Unfortunately, the leading concurrent stacks in performance are often the lock-free ones [1], [3], but lock-freedom cannot guarantee non-starvation for every thread. While providing the strongest progress guarantee, wait-free data structures are typically inefficient and hard to design [4], [5]. For example, the state-of-the-art wait-free concurrent stack [2] is even less efficient than a *combining-based* blocking stack [6]. Their inefficiency is mainly attribute to the *helping mechanism*,[4] which is the most common paradigm for guaranteeing wait-freedom. The reason is that this mechanism needs a certain number of expensive atomic operations to control the way threads help each other for ensuring each operation to be applied exactly once [4]. Thus, it still remains an open question how to design concurrent stacks that meet both the high performance and wait-free progress guarantee.

1.2 Key Ideas

To answer the above question, this paper presents FA-Stack, a fast array-based concurrent stack. As shown in Fig. 1, FA-Stack is constructed on an array with *top* index T ($T := 0$ at initial state). In our basic idea of FA-Stack, a push operation involves (1) obtaining a cell index by performing

• *The authors are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China. E-mail: {pengyaqiong, haozhiyu}@iie.ac.cn.*

Manuscript received 13 June 2017; revised 22 Oct. 2017; accepted 29 Oct. 2017. Date of publication 7 Nov. 2017; date of current version 9 Mar. 2018. (Corresponding author: Zhiyu Hao.) Recommended for acceptance by B. He. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TPDS.2017.2770121

1. Ensure one thread to complete an arbitrary operation in a finite number of steps when it runs alone.
2. Ensure some threads to complete an arbitrary operation in a finite number of steps.
3. Ensure every thread to complete an arbitrary operation in a finite number of steps.
4. A formal definition of the helping mechanism is presented in [7].

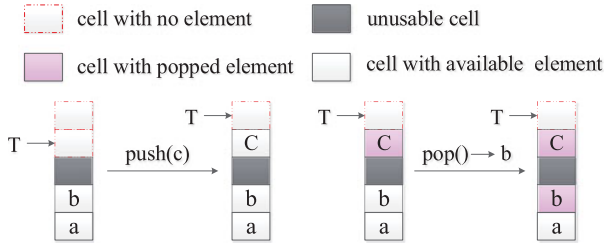

Fig. 1. FA-Stack push and pop operations.

Fetch-and-Add (FAA, see its definition in Table 1) on T, and (2) attempting to deposit its element into the associated cell. This process repeats until the element is deposited into a cell successfully.

To return the latest element, a pop operation involves (1) reading T's current value t, and (2) traversing backward the array from the cell with index $t-1$ to find an available element until successfully popping such an element or locating below the stack bottom (returning *EMPTY*). If a pop operation visits a cell before the matching push operation completes, it marks the cell as *unusable*, which prevents the matching push operation from depositing its element into the cell.

To make FA-Stack practical and wait-free, we present a series of optimizations to bound the number of steps required to complete every push and pop operation in a finite number. As FA-Stack probabilistically spreads concurrent push and pop operations to different cells at the same time, it improves the chance that threads complete their operations without help. In addition, FA-Stack adopts a time-stamped scheme to reclaim memory. Section 4 presents the details.

1.3 Algorithm Correctness

The correctness condition we use for FA-Stack is *linearizability* [8], which guarantees concurrent operations to appear atomic. As analyzed in Section 5, FA-Stack is linearizable, and it conforms to the LIFO semantics of stack. Therefore, FA-Stack is correct.

1.4 Evaluation

To explore the effect of FA-Stack in practice, we compare FA-Stack to the state-of-the-art concurrent stacks including *Sim-Stack* (wait-free) [2], *TS-Stack* (lock-free) [3], *CC-Stack* (combining-based blocking) [6] on a x86 server. Our evaluation with representative benchmarks shows that FA-Stack is an efficient wait-free stack. For example, compared to Sim-Stack (a state-of-the-art wait-free stack) [2], FA-Stack improves the throughput of *halfhalf* benchmark by up to 2.4×.

1.5 Contributions

In summary, this paper makes the following contributions:

- A practical fast array-based stack with wait-free progress guarantee.
- A series of theoretical proofs to show that FA-Stack is a wait-free linearizable stack with respect to the LIFO semantics.
- An experimental evaluation showing the efficiency of FA-Stack by comparing it to the state-of-the-art concurrent stacks.

TABLE 1
Atomic Primitives Used by FA-Stack

$Read(a)$	return $m[a]$
$FAA(a, v)$	return $m[a]$ and store $m[a]+v$ at a
$CAS(a, t, v)$	store v into $m[a]$ if $m[a]=t$ before returning *true*; otherwise, return *false*

1.6 Outline

The rest of the paper is organized as follows. Section 2 introduces some basics that provide a foundation for FA-Stack and overviews the related work. Section 3 presents the high-level design of FA-Stack, followed by the details in Section 4. Section 5 proves the correctness and wait-freedom of FA-Stack. Section 6 provides a comprehensive evaluation of FA-Stack's performance. Section 7 concludes the paper.

2 Preliminaries and Related Work

In this section, we first introduce some basics that provide a foundation for FA-Stack, and then overview the most related work.

2.1 Preliminaries

A concurrent stack is an object that typically supports two types of operations, namely *push* and *pop*, with linearizable LIFO semantics. The linearization point of an operation is the point that the operation takes effect during its execution interval. The correct assigned linearization points of every push and pop operation must match the semantics of a stack. That is, if one were to run a serial stack based on the assigned linearization points of every push and pop operations, the elements returned by every pop operation would be the top of the stack, or *EMPTY* if no elements are left in the stack.

FA-Stack is designed for *asynchronous shared memory systems* [8], where an application runs with n deterministic threads. Similar to previous work [9], [10], we model memory as an array of 64-bit values, and use the notation $m[a]$ for the value stored in address a of the memory. Table 1 describes the 64-bit atomic primitives used by FA-Stack. These primitives are supported by mainstream architectures natively.

2.2 Related Work

Concurrent data structures (especially stacks and queues) have been a topic of active research for decades, which need to be implemented with some synchronization [11]. Treiber proposes the first non-blocking concurrent stack [12], which is faster than lock-based stack. In this work, a stack is represented as a singly-linked list with a *top* pointer. Many non-blocking stacks are constructed based on this structure [1], [13]. In list-based stacks, the top pointer is typically the unique hot spot contended by threads. Concurrent threads atomically update the value of the top pointer by using *CAS* operations in a retry loop, and the *CAS* operations probably fail especially under heavy contention, resulting in significant synchronization cost. Morrison et al. define this problem as *CAS retry problem* [9].

To reduce the synchronization cost, researchers propose concurrent stacks based on the *combining algorithm* [6], [14], [15], which is typically a *universal construction* [16], [17], [18] to design any shared objects. In combining algorithm, a single thread traverses a list of pending operations, and helps

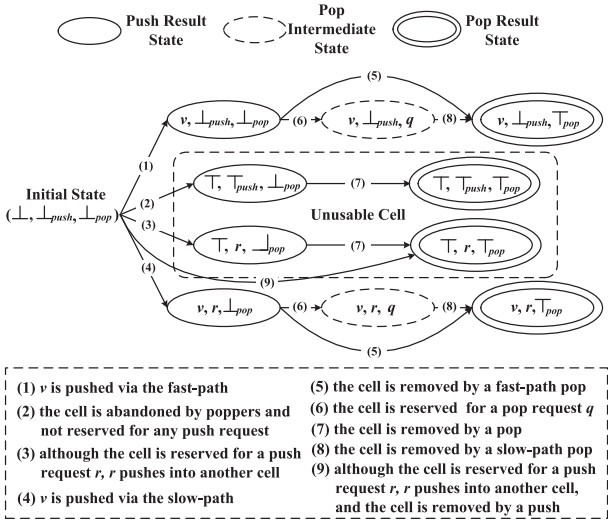

Fig. 2. The state transition diagram of a cell in FA-Stack.

apply them to the object. Although combining-based stacks have lower synchronization cost than that of list-based ones, they sacrifice the parallelism of parallel applications.

As another approach to reduce the synchronization cost, Dodds et al. weak the order of pushed elements and propose an efficient list-based stack named TS-Stack with lock-free progress guarantee [3]. In TS-Stack, every thread executes push operations by independently appending elements into its local buffer, not incurring any synchronization cost. However, the parallelism of push operations comes at the cost of more contentions in the pop operations. Therefore, TS-Stack is relatively weak in pop-dominated workloads.

Wait-freedom is the strongest progress guarantee, and wait-free stacks can limit the *CAS* retry times. However, wait-free progress guarantees often come with a high performance cost [4]. To apply wait-free data structures into practice, many works aim to transform lock-free data structures into wait-free version that are as fast as the lock-free ones [4], [5], [19], [20]. To the best of our knowledge, Sim-Stack is the state-of-the-art wait-free stack [2]. To achieve wait-freedom, Sim-Stack forces each active thread to help execute all pending operations, thus extending the execution time of each operation. As a result, it is less efficient than a *combining-based* blocking stack [6].

Afek et al. propose a wait-free stack based on array [21]. The algorithm is simple and linearizable, but not practical due to two main disadvantages: (1) the array is assumed to be infinite size; (2) the number of elements visited by a pop operation tends to increase monotonically with the number of elements ever inserted. Different from this algorithm, FA-Stack is dynamically resized in segments, and bounds the number of elements visited by a pop operation by making contending poppers help each other and reclaim segments with no available elements.

Yang et al. propose a practical wait-free queue based on array that is as fast as *FAA* [10]. This wait-free queue outperforms LCRQ (the best performing lock-free queue) [9] in most cases, which shows the potential of array-based data structures in meeting both the high performance and wait-free progress guarantee. This work inspires the design of FA-Stack. The differences between this wait-free queue and FA-Stack are as follows:

- In a queue, as enqueue and dequeue operations contend for the tail and head respectively, they naturally work in parallel. Different from queues, both push and pop operations contend for the top of a stack. Therefore, it is more difficult for FA-Stack to improve the parallelism of operations with wait-freedom. To overcome the challenge, FA-Stack scatters the contended hot spots by only allowing push operations to modify the top index. Instead of modifying the top index, a pop operation reads the current value of top index, and walks down the stack until it successfully pops an available element.
- In this wait-free queue, a dequeue operation always obtains a cell index by performing FAA on the head index, and attempts to dequeue the matching cell. As the head index increases monotonically, a dequeue operation can quickly find an available cell to dequeue. Different from a queue, a pop operation always needs to return the top of a stack. Therefore, in the basic idea of FA-Stack, a pop operation probably visits a large number of unusable and popped cells before finding an available element to pop. To solve this problem, FA-Stack adopts a novel segment removing scheme to bound the number of unusable and popped cells visited by a pop operation.

3 High-Level Design of FA-Stack

In this section, we present the key ideas behind FA-Stack, which is shown in Algorithm 1. Logically, a stack object is a 2-tuple (S, T). S is an array divided into segments, and each segment contains an N-vector of cells for storing pushed elements. T is used to index the next top cell of S. For the convenience of description, Algorithm 1 represents the ith cell of S as $S[i]$, and every cell is a 3-tuple $(elem, push, pop)$. $elem$ is used to store a pushed element. $push$ and pop are used to reserve the matching cell for push and pop requests, respectively. For $elem$, we reserve two special values $\perp$ and $\top$ that are distinct from any element pushed into S by threads. For $push$, we reserve $\perp_{push}$ and $\top_{push}$. For pop, we reserve $\perp_{pop}$ and $\top_{pop}$. In addition, pop also acts as a flag indicating whether the matching cell is logically removed. Fig. 2 illustrates the state transition diagram of a cell in FA-Stack. Every cell is initialized with $(\perp, \perp_{push}, \perp_{pop})$. If a cell will never be pushed into any element, it becomes unusable. If all the cells of a segment are logically removed, we call the segment as *retired segment*.

FA-Stack follows the fast-path-slow-path style like previous work on wait-free concurrent data structures [4], [5], [10], [20]. As shown in Algorithm 1, a push operation P first obtains a cell index i by performing FAA on T, and attempts to deposit its element x into $S[i]$ by performing CAS $(\&S[i].elem, \perp, x)$. This process is called *fast-path push*, and it is repeated until the CAS succeeds or the number of CAS failures exceeds a predefined threshold $MAX\_FAILURES$. The CAS fails because $S[i]$ was visited by a contending pop operation (line 24). To ensure wait-free, after $MAX\_FAILURES$ unsuccessful attempts for the fast-path, P switches to the slow-path, where P first announces itself by publishing a push request to enlist help from contending pop operations and then continues to find a candidate cell for storing its

element. Once all contending pop operations become its helpers, P will definitely complete.

Algorithm 1. Key Ideas Behind FA-Stack

```
Shared variables S: an array divided into segments; T: top
    index initialized to 1
 1: procedure PUSH(Element x)
 2:   p := MAX_FAILURES
 3:   // start on fast-path
 4:   i := FAA(&T, 1)
 5:   // try to deposit x into S[i]
 6:   if CAS(&S[i].elem, ⊥, x) then return
 7:   end if
 8:   p := p - 1
 9:   if p > 0 then
10:     goto Line 4
11:   end if
12:   // switch to slow-path
13:   publish my push request to enlist help
14:   continue to find a candidate cell c and tries to deposit x
      into c until I succeed or a contending pop helps me to
      succeed; during this process, remove a retired segment if
      necessary
15: end procedure
16: procedure POP()
17:   p := MAX_FAILURES
18:   t := Read(&T)
19:   // start on fast-path
20:   for c := S[t - 1]downto stack bottom cell do
21:     if p = 0 then
22:       goto Line 38;
23:     end if
24:     if c→elem = ⊤ or CAS(&c→elem, ⊥, ⊤) or c→elem = ⊤
      then
25:       help a push request to deposit its element into c, or
          guarantee c to be permanently unusable
26:     end if
27:     if CAS(&c→pop, ⊥pop, ⊤pop) then
28:       remove the segment of c if necessary
29:       if c→elem ≠ ⊤ then
30:         v := c→elem
31:         goto Line 47
32:       end if
33:     end if
34:     p := p - 1
35:   end for
36:   return EMPTY
37:   // switch to slow-path
38:   publish my pop request to enlist help
39:   for c' := c downto stack bottom cell do
40:     if successfully pop from c' or a contending pop helps
      me to succeed then
41:       v := c'→elem
42:       remove the segment of c' if necessary
43:       goto Line 47
44:     end if
45:   end for
46:   return EMPTY
47:   help a pending pop request; during this process,
      remove some retired segments if necessary
48:   return v
49: end procedure
```

A pop operation Q first obtains an index t by reading T (line 18), and then finds a cell to pop by traversing backward the stack from $S[t-1]$. The traversal is started on the fast-path. At each time point, we say that a cell c can be popped by Q, if c holds a pushed element and c→pop = $\perp_{pop}$. When Q visits c before the matching push operation completes, Q tries to rescue c by helping a pending push request to deposit its element into c, and guarantees that c will never hold a pushed element if failing in such a help. Next, Q attempts to logically remove c by using $CAS(\&c \rightarrow pop, \perp_{pop}, \top_{pop})$. If the CAS succeeds and c holds a pushed element, Q pops from c and stops finding other cells to pop (lines 27-33). No matter which popping thread eventually removes c, the thread always ensures c is in one of the four states before being removed: (v, $\perp_{push}$, $\perp_{pop}$), ($\top$, $\top_{push}$, $\perp_{pop}$), ($\top$, r, $\perp_{pop}$), and (v, r, $\perp_{pop}$). We call these four states as *push result state*. As shown in Fig. 2, once a cell transit into the push result state, its first two fields will always remain unchanged. When a cell is removed by a pop operation, it transit into one of the four states: (v, $\perp_{push}$, $\top_{pop}$), ($\top$, $\top_{push}$, $\top_{pop}$), ($\top$, r, $\top_{pop}$), and (v, r, $\top_{pop}$). We call these four states as *pop result state*.

Similar to push operations, if Q does not pop from a cell (or returns *EMPTY* if no available elements are left in the stack) after visiting *MAX_FAILURES* cells on the fast-path, it switches to the slow-path. Q first announces itself by publishing a pop request q to enlist help from contending pop operations, and then continues to traverse backward the stack to find a cell to pop on its own (lines 39-45). Once Q and its helpers find a cell with pushed element, they try to reserve the cell for q by changing the state of the cell to (v, $\perp_{push}$, q) or (v, r, q). We call these two states as *pop intermediate state*. To ensure every pop operation to be applied exactly once, FA-Stack coordinates Q and all its helpers to reserve one cell for q, and only Q can remove the reserved cell. During the execution interval, Q probably visits cells that are removed by other operations. For the cells that are removed after Q publishes its request, it is easy to bound their number by turning all contending pop operations into Q's helpers. To bound the number of cells that are removed before Q publishes its request, when a push or pop operation removes the last cell of a segment (lines 14, 28, 42, and 47), the operation will remove the segment from S in an efficient manner. Section 4.4 describes the details of our segment removing scheme, and how to safely reclaim retired segments.

4 FA-Stack in Detail

In this section, we present the data structures and supporting operations of FA-Stack in detail.

4.1 Stack Construction

To realize FA-Stack, we design its underlying data structures that are shown in Listing 1. As described in Section 3, the major building block of FA-Stack is an array that can be resized dynamically in segments. We use a doubly-linked list of segments to represent the array, and each segment contains an array *cells* with size N. FA-Stack links these segments by using their $prev$ and $next$ pointers. In the following, the segment with id i is denoted as $segment[i]$, while the jth cell in a segment is denoted as $cell[j-1]$. For the ith

cell of the global array, it corresponds to $cell[i \bmod N]$ in $segment[i/N]$. Different from the key ideas in Section 3, we add a new field $offset$ to every cell for accelerating the performance of fast-path pop. The push and pop requests are represented as *PushReq* and *PopReq*, respectively. Algorithm 2 shows how to construct a stack object s that consists of (1) a top pointer and (2) an index T. Initially, $s \rightarrow top$ points to $segment[0]$, and $s \rightarrow T$ is set to 1.

Listing 1. Data structures

```
1  struct PushReq {
2    Element elem;
3    struct {int pending:1, id:63; } state;
4  }
5
6  struct PopReq {
7    int idx:64;
8    struct {int pending:1, id:63; } state;
9  }
10
11 struct Cell {
12   Element elem; PushReq *push; PopReq *pop;
     int offset:64;
13 }
14
15 struct Segment {
16   int id:64, counter:64, time_stamp:64;
     bool retired;
17   Segment *prev, *next, *real_next, *free_next;
18   Cell cells[N];
19 }
20
21 struct Stack {
22   Segment * top; int T:64;
23 }
24
25 struct Handle {
26   Segment * top, * sp, * free_list;
27   Handle *next;
28   struct { int help_id:64;PushReq req;Handle
     *peer; } push;
29   struct { PopReq req; Handle *peer; } pop;
30   int time_stamp:64;
31 }
```

In Section 3, we have pointed out the problem that a pop operation may visit plenty of removed cells before completion. To solve the problem, FA-Stack needs to dynamically remove and reclaim retired segments. To achieve this goal, we set five variables for each segment: (1) a $counter$ used to count the number of removed cells, (2) a variable $time\_stamped$ indicating when the segment is removed, (3) a flag $retired$ indicating whether the segment retires, (4) a $real\_next$ pointer to the successor segment that is not retired and nearest to it, and (5) a pointer $free\_next$ for inserting the segment into a free list for memory reclamation. When the total number of removed cells in a segment reaches N, the segment retires and thus it can be removed. As described later in Section 4.4, the *remove* function in FA-Stack does not change the $next$ field of any segment, but the $real\_next$ field. Given a segment $segment[i]$, $segment[i]$'s $prev$ points to a predecessor segment that is not retired and nearest to it, or null if $segment[i]$ is the first segment. $segment[i]$'s $next$ always points to $segment[i+1]$, or null if $segment[i]$ is the last segment. The $real\_next$ field of $segment[i]$ is initialized with null. When $segment[i+1]$ retires, $segment[i]$'s $real\_next$ field is changed to a successor segment that is not retired and nearest to it.

Algorithm 2. Stack Construction

```
Shared variables: Stack *s: stack object; int pc: 64: FAA object
    that provides ID for each pop request
 1: procedure NEW_SEGMENT(int id)
 2:   Segment *sg := new Segment
 3:   *sg := (id, 0, ⊥ts, false, null, null, null, null)
 4:   for i := 0 to N - 1 do
 5:     sg→cells[i] := (⊥, ⊥push, ⊥pop, 0)
 6:   end for
 7:   return sg
 8: end procedure
 9: procedure STACK_INIT()
10:   s := new Stack
11:   top := new_segment(0)
12:   next := new_segment(1) // allocate a dummy segment
      as the last segment of the stack
13:   top→next := next
14:   next→prev := top
15:   s→top := top
16:   s→T := 1
17:   pc := 1
18: end procedure
19: procedure FIND_CELL(Segment **sp, int cell_id)
20:   sg := *sp
21:   for (i := sg→id; i < cell_id/N; i++) do
22:     if sg = s→top then
23:       next := sg→next // next is not null because a
          dummy segment is always next to the top segment
24:       nnext := next→next
25:       if nnext = null then
26:         tmp := new_segment(i + 2)
27:         tmp→prev := next
28:         if !CAS(&next→next, null, tmp) then
29:           free(tmp)
30:         end if
31:       end if
32:       CAS(&s→top, sg, next)
33:     else
34:       next := sg→next
35:     end if
36:     sg := next
37:   end for
38:   *sp := sg
39:   return &sg→cells[cell_id mod N]
40: end procedure
```

For the convenience of removing retired segments, FA-Stack always allocates a dummy segment as the last segment of the stack. $s \rightarrow top \rightarrow next$ points to the dummy segment. As shown in Algorithm 2, FA-Stack only allocates two segments during initialization, and the segment list is dynamically extended by visiting threads on demand. To be specific, when a thread T_0 invokes the $find\_cell$ function to find a cell with global index $cell\_id$, T_0 traverses the stack from the input segment sp until it finds the target segment

$segment[cell\_id/N]$. When reaching the top segment sg before finding the target, if $next$ is the last segment of the stack (next := sg→next), T_0 allocates a new dummy segment tmp and tries to append tmp to the stack by using $CAS(\&next \rightarrow next, null, tmp)$. If the CAS fails, the segment list is extended by other threads. Then, T_0 sets $next$ as the new top segment of the stack by using $CAS(\&s \rightarrow top, sg, next)$ (line 32). When the thread finds $segment[cell\_id/N]$, it records $segment[cell\_id/N]$ in the output parameter sp and returns the target cell.

Algorithm 3. Wait-Free Push

```
 1: procedure PUSH(Handle *h, Element x)
 2:   h→time_stamp := get_timestamp()
 3:   h→top := s→top
 4:   for p := MAX_FAILURES downto 1 do
 5:     // fast path
 6:     i := FAA(&s→T, 1)
 7:     c := find_cell(&h→top, i)
 8:     if CAS(&c→elem, ⊥, x) then
 9:       goto Line 13
10:     end if
11:   end for
12:   wf_push(h, x, i) // slow path
13:   h→time_stamp := ⊤ts
14:   h→top := null
15: end procedure
16: procedure WF_PUSH(Handle *h, Element x, int push_id)
17:   // publish my push request
18:   r := &h→push.req
19:   r→elem := x
20:   r→state := (1, push_id)
21:   sp := h→top
22:   repeat
23:     i := FAA(&s→T, 1)
24:     c := find_cell(&sp, i)
25:     if (CAS(&c→push, ⊥push, r) or c→push = r) and (CAS
        (&r→state, (1, push_id), (0, i)) or r→state = (0, i)) then
26:       break;
27:     end if
28:     if c→push = r then
29:       if CAS(&c→pop, ⊥pop, ⊤pop) then
30:         counter := FAA(sp→counter, 1)
31:         if counter = N - 1 then
32:           remove(h, sp)
33:         end if
34:       end if
35:     end if
36:   until r→state.pending = 0
37:   i := r→state.id
38:   find_cell(&h→top, i)
39:   c→elem := x
40: end procedure
```

Because the $remove$ function does not change the $next$ field of any segment, a thread always visits $segment[i+1]$ after $segment[i]$ during a call of the $find\_cell$ function. Assuming that $s \rightarrow top$ points to $segment[i]$ at a time point, any thread extending the segment list afterwards will first try to change $s \rightarrow top$ from $segment[i]$ to $segment[i+1]$. Therefore, there is no need to judge the success of CAS at line 32. If the CAS fails, $next$ was set as the top segment by other threads. If not using the $real\_next$ field, a thread needs a complex CAS retry loop to update $s \rightarrow top$. In addition, the use of the $real\_next$ field also contributes to simply deal with the case that a thread needs to find a cell of a removed segment,[5] because the thread can find the target cell starting from the input segment of the $find\_cell$ function by the $next$ field of visited segments.

Algorithm 4. $help\_push$ Function Called by Push Helpers

```
 1: procedure HELP_PUSH(Handle *h, Cell *c, int i)
 2:   if !CAS(&c→elem, ⊥, ⊤) and c→elem ≠ ⊤ then
 3:     return c→elem
 4:   end if
 5:   if c→push = ⊥push then
 6:     p := h→push.peer
 7:     r := &p→push.req
 8:     s := r→state
 9:     if h→push.help_id ≠ 0 and h→push.help_id ≠
        s.id then
10:       h→push.help_id := 0
11:       h→push.peer := p→next
12:       p := h→push.peer
13:       r := &p→push.req
14:       s := r→state
15:     end if
16:     if s.pending and s.id ≤ i then
17:       CAS(&c→push, ⊥push, r)
18:     end if
19:     if s.pending and c→push ≠ r then
20:       h→push.help_id := s.id
21:     else
22:       h→push.help_id := 0
23:       h→push.peer := p→next
24:     end if
25:     CAS(&c→push, ⊥push, ⊤push)
26:   end if
27:   if c→push = ⊤push then
28:     return ⊤
29:   end if
30:   r := c→push
31:   s := r→state
32:   v := r→elem
33:   if (s.id ≤ i and CAS(&r→state, (1, s.id), (0, i))) or r→
      state = (0, i) then
34:     c→elem := v
35:   end if
36:   return c→elem
37: end procedure
```

To ensure wait-freedom, a thread T_0 checks if a peer thread T_1 needs help by accessing T_1's local state that is maintained by a *handle*. Each thread handle consists of: (1) a top pointer to the top segment in the stack, (2) a sp pointer to the first segment visited by a slow-path pop, (3) a $free\_list$ pointer to a list of retired segments that are removed by this thread but not yet clean up, (4) a $next$ pointer to another thread handle, (5) $push$ state, (6) pop state,

5. For example, when a push operation finds a cell of a segment sp (lines 7 and 24 in Algorithm 3), sp may be retired and thus removed by other operations. FA-Stack adopts a time-stamped memory reclamation scheme to really clean up retired segments after ensuring such segments to be no longer in use.

and (7) a variable $time\_stamp$ indicating the start time point of the current operation run by this thread. To enable any thread to access the state of other threads, all thread handles are linked in a ring using their $next$ pointers like the approach of a wait-free queue [10].

4.2 Wait-Free Push

Algorithm 3 shows the pseudo code for wait-free push of FA-Stack. A push operation invokes the wf_push function after *MAX_FAILURES* unsuccessful attempts for the fast-path. The last cell index i visited on the fast-path serves as the input parameter $push\_id$ of the wf_push function.

When invoking the wf_push function, a push operation first uses its handle's $push.req$ field to publish its push request r to solicit help. At this point, the state of r is (x, 1, $push\_id$), where x is the element to push, and $push\_id$ is an input parameter that serves as the identifier of r. Then, the push operation continues using FAA to obtain indices of additional cells in the stack and trying to reserve each candidate cell c for r by using $CAS(\&c \rightarrow push, \perp_{push}, r)$. In addition, each pop operation invokes the $help\_push$ function on each cell it visits, and tries to help a push request if the cell holds a $\top$. Algorithm 4 shows the pseudo code for the $help\_push$ function. We call any popping thread invoking the $help\_push$ function as *push helper*.

If a push helper visits a cell c before the matching push completes, it changes $c \rightarrow elem$ to $\top$, and then attempts to help a peer push into the cell if the cell is not reserved for any push request. To help push peers, each thread maintains a $peer$ pointer in its push state (Handle.push). The $peer$ pointer is used to take a thread as push peer. To ensure linearizability, a push helper cannot help a request r push into a cell with index i if the identifier of r is greater than i (lines 16-18 in Algorithm 4). Section 5 will explain the reason. Once a push helper Q fails to help its peer, it remembers the push request identifier of the peer in $h \rightarrow push.help\_id$ (lines 19-21 in Algorithm 4),[6] where h is the thread handle of Q. Then, Q continues helping the peer in consequent pop operations until *the peer has published a new request* (lines 9-15 in Algorithm 4), *the peer has no pending request for help* ($s.pending = 0$), or *the helper finds a cell reserved for the peer* ($c \rightarrow push = r$).

The $push$ and $help\_push$ functions maintain the following important invariants:

Invariant 1. *The element of every push operation is definitely deposited into a unique cell, and different push operations cannot deposit their elements into a cell.*

Every push operation visits a cell by using FAA on $s \rightarrow T$ to obtain the cell index, which prevents different push operations from visiting the same cell. If a push operation deposits its element into a cell on the fast-path, this invariant is obviously satisfied. Although a slow-path push and its helpers may reserve different cells for a pending push request, FA-Stack uses $r \rightarrow state$ to synchronize them, and only one cell is claimed for a push request. In addition, each slow-path push and push helper tries to reserve a cell for a given push request r by using $CAS(\&c \rightarrow push, \perp_{push}, r)$. Thus, a cell will not be reserved for multiple push requests. Therefore, this invariant is always satisfied.

Invariant 2. *For all help_push functions invoked on the same cell c, they have the same returned values.*

Once a cell c holds an element ($c \rightarrow elem$!= $\perp$ and $c \rightarrow elem$!=$\top$), any $help\_push$ function invoked on c will return $c \rightarrow elem$ without any changes on it (lines 2-4 in Algorithm 4). Otherwise ($c \rightarrow elem = \top$), before returning $c \rightarrow elem$, $help\_push$ first ensures (1) $c \rightarrow push = \top_{push}$ or (2) $c \rightarrow push$ points to a push request r and tries to help r if necessary. In the first case, c is permanently unusable. In the second case, c is permanently unusable or written into the element of r. Combining with Invariant 1, this invariant is always satisfied.

Invariant 3. *A slow-path push visits at most $2 \cdot (n-1)^2 + 1$ candidate cells before its pending push request closes, where n is the number of threads.*

Suppose that a slow-path push P is run by a thread T_i. Every time T_i fails to reserve a candidate cell c for P due to that a contending popping thread T_j changed $c \rightarrow push$ to (1) a push request r from T_j's push peer, or (2) $\top$. In the first case, T_j will update its push peer to the next thread (lines 21-24 in Algorithm 4). In the second case, only if the push request r from T_j's push peer is still pending and $r \rightarrow id$ is greater than the index of c, T_j will not update its push peer to the next thread (lines 16-20 in Algorithm 4). When T_i tries to reserve next candidate cell c' for P, the index of c' must be greater than $r \rightarrow id$, because they come from the returned values of a later and earlier FAA on the top index respectively. Therefore, every two times T_j prevents T_i from reserving a candidate cell for P, T_j updates its push peer at least one time. In addition, it is obvious that T_j does not take a thread as push peer again until it has taken all other threads as push peers. Therefore, each popping thread prevents T_i from reserving a candidate cell at most $2 \cdot (n-1)$ times before helping T_i to complete P. After T_i fails $2 \cdot (n-1)^2$ times, all other threads will help to complete P. Therefore, this invariant is always satisfied.

4.3 Wait-Free Pop

Algorithm 5 shows the pseudo code for wait-free pop of FA-Stack. Different from the key ideas in Algorithm 1, we optimize the performance of fast-path pop by using FAA to accelerate its stack traversal process. When a fast-path pop Q visits a cell c with index i, it first obtains a value $offset$ by using FAA(&c→offset, 1). If $offset \neq 0$, it indicates that $offset$ fast-path poppers arrive c before Q. In this case, Q gives up $offset$ cells starting from c downto the stack bottom to these earlier poppers, and takes the cell with index $i - offset$ as the next cell to visit or returns *EMPTY* if $i < offset$. Otherwise ($offset = 0$), Q tries to remove c by using $CAS(\&c \rightarrow pop, \perp_{pop}, \top_{pop})$. If the CAS succeeds, Q pops from c; then, Q tries to help a peer to pop by invoking the $help\_pop$ function. Algorithm 6 shows the pseudo code for this function. If Q does not complete after visiting *MAX_FAILURES* cells, it switches to the slow-path. A slow-path pop Q first uses its handle's $pop.req$ field to publish its pop request r to solicit help. The idx field of r remembers the index of the cell that needs to be first visited by Q and

6. Note that the identifier of a push request must not equal to zero because it comes from a cell index. Therefore, if Q does not need to remember the identifier of any push request, $h \rightarrow push.help\_id$ is set to 0.

its helpers. To complete r, Q and all its helpers invoke the *help_pop* function. The *help_pop* function walks down the stack in order to reserve a cell with pushed element for r. When finding such a cell c, this function attempts to reserve c for r by using $CAS(\&c \rightarrow pop, \perp_{pop}, r)$. If the CAS succeeds, r pops from c.

Algorithm 5. Wait-Free Pop

```
 1: procedure POP(Handle *h)
 2:   h→time_stamp := get_timestamp()
 3:   h→top := s→top
 4:   t := Read(&s→T)
 5:   find_cell(&h→top, t)
 6:   sp := h→top
 7:   // search an available element to pop starting at the
      predecessor of the cell with global index t
 8:   if (t mod N) ≠ 0 then
 9:     idx := t - 1
10:   else
11:     sp := sp→prev
12:     if sp ≠ null then
13:       idx := sp→id * N + N - 1
14:     end if
15:   end if
16:   p := 0
17:   while sp ≠ null do
18:     if sp→id ≤ idx/N then
19:       if sp→id < idx/N then // segment with ID idx/N
          was removed
20:         idx := sp→id * N + N - 1
21:       end if
22:       if p = MAX_FAILURES then goto Line 53
23:       end if
24:       i := idx mod N
25:       c := &sp→cells[i]
26:       offset := FAA(&c→offset, 1)
27:       idx := idx - offset // update the index of the
          candidate cell that I really try to pop
28:       if idx < 1 then
29:         v := EMPTY
30:         goto Line 59
31:       end if
32:       p := p + 1
33:       if offset ≠ 0 then continue // some fast-path
          poppers arrive here earlier
34:       end if
35:       v := help_push(h, c, idx)
36:       if CAS(&c→pop, ⊥pop, ⊤pop) then
37:         counter := FAA(&sp→counter, 1)
38:         if counter = N - 1 then
39:           remove(h, sp)
40:         end if
41:         if v ≠ ⊤ then goto Line 55
42:         end if
43:       end if
44:       idx := idx - FAA(&c→offset, 1) // obtain the index
          of next candidate cell
45:     else
46:       sp := sp→prev
47:     end if
48:   end while
49:   if sp = null then // no available elements
50:     v := EMPTY
51:     goto Line 59
52:   end if
53:   h→sp := sp
54:   v := wf_pop(h, idx) // slow path
55:   if v ≠ EMPTY then
56:     help_pop(h, h→pop.peer)
57:     h→pop.peer := h→pop.peer→next
58:   end if
59:   h→time_stamp := ⊤ts
60:   h→top := null
61:   return v
62: end procedure
63: procedure WF_POP(Handle *h, int cid)
64:   // publish my pop request
65:   r := &h→pop.req
66:   r→idx := cid
67:   r→state := (1, FAA(&pc, 1))
68:   help_pop(h, h)
69:   // obtain the target cell index
70:   i := r→state.id
71:   if i = 0 then return EMPTY
72:   end if
73:   find the segment sp with ID i/N, the cell c with
      global index i
74:   v := c→elem
75:   c→pop := ⊤pop
76:   counter := FAA(&sp→counter, 1)
77:   if counter = N - 1 then
78:     remove(h, sp)
79:   end if
80:   return v
81: end procedure
```

When Removing a Segment? The key to make FA-Stack practical is to timely remove retired segments from the stack. Intuitively, only popping threads need to remove cells, so as to cause segments to be retired. In fact, a pushing thread may also cause a segment to be retired by wasting a cell c of the segment in the case that it reserves c for its push request r but r eventually pushes into another cell. Because c is reserved for r, it will never hold the elements of other push operations. To bound the number of cells visited by pop operations, pushing threads need to remove the wasted cells (lines 28-35 in Algorithm 3). To reduce the contention on removing retired segments, only the thread that removes the last remaining cell of a segment has the power to remove the segment (lines 29-34 in Algorithm 3, lines 36-43 and lines 75-79 in Algorithm 5, lines 22-27 in Algorithm 6).

Popping from FA-Stack maintains the following important invariants:

Invariant 4. *Suppose c is the current cell visited by a pop operation (or pop helper) Q, then Q continues to visit the predecessor cell of c only after ensuring that c is (1) unusable, (2) popped by other pop operations, or (3) reserved for other pop requests).*

A slow-path pop operation or its helpers invoke the *help_push* function on every visited cell c, and they can judge whether c holds an element by the returned value. According to Invariant 2, if the *help_push* function returns $\top$, c must be unusable. Once c holds an element and it is popped (or reserved) by Q, Q stops walking down the stack. Therefore, this invariant is satisfied for the slow-path pop

operation and its helpers. In addition, a fast-path pop operation also continues to visit the predecessor cell of c if other fast-path poppers visit c earlier. In this case, c must be either unusable or popped by one of these earlier poppers, and thus this invariant is also satisfied.

Algorithm 6. *help_pop* Function Called by Pop Helpers

```
1: procedure HELP_POP(Handle *h, Handle *helpee)
2:   r := &helpee→pop.req
3:   s := r→state
4:   if !s.pending then
5:     return
6:   end if
7:   idx := r→idx
8:   sp := helpee→sp
9:   h→time_stamp := helpee→time_stamp
10:  for (i := idx mod N; sp ≠ null; sp := sp→prev, i := N - 1)
     do
11:    if r→state != s then
12:      return
13:    end if
14:    while i≥0 do
15:      c := &sp→cells[i]
16:      cid := sp→id * N + i
17:      v := help_push(h, c, cid)
18:      if v ≠ ⊤ and (CAS(&c→pop, ⊥pop, r) or c→pop = r)
         then
19:        CAS(&r→state, s, (0, cid))
20:        return
21:      else
22:        if v = ⊤ and CAS(&c→pop, ⊥pop, ⊤pop) then
23:          counter := FAA(&sp→counter, 1)
24:          if counter = N - 1 then
25:            remove(h, sp)
26:          end if
27:        end if
28:      end if
29:      i := i - 1
30:    end while
31:  end for
32:  if r→state = s then
33:    CAS(&r→state, s, (0, 0))
34:  end if
35: end procedure
```

Invariant 5. *Suppose c is a cell that is deposited into an element by a push operation, then there is at most a single pop operation that pops from c.*

A fast-path pop operation pops from c by successfully performing $CAS(\&c \rightarrow pop, \perp_{pop}, \top_{pop})$, which prevents c from being popped by other operations or reserved for other pop requests. If c is reserved for a pop request r, $c \rightarrow pop$ is first changed to r by the matching pop or its helpers, and eventually changed to $\top_{pop}$, which also prevents c from being popped by other operations or reserved for other pop requests. Therefore, this invariant is always satisfied.

Invariant 6. *Every pop operation is applied exactly once.*

This invariant is obviously satisfied because every pop operation (or its helpers) stops walking down the stack after successfully finding an available cell to pop, or returning $EMPTY$ when locating below the stack bottom.

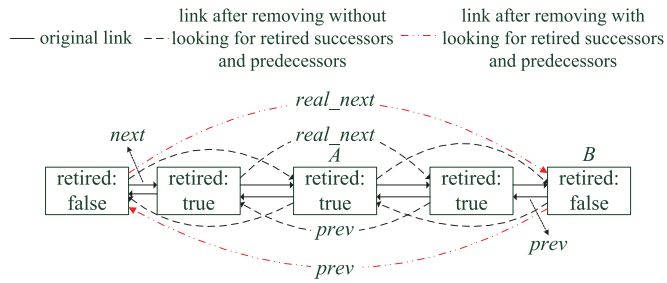

Fig. 3. A segment removing example.

Invariant 7. *A slow-path pop visits at most $(n^2 - n + 2) \cdot N - 1$ cells before its pending pop request closes, where n is the number of threads.*

Given a slow-path pop Q and the first cell $S[i]$ visited by Q, suppose that a logical list L only contains $S[j]$ $(0 < j \leq i)$ that is eventually deposited into an element and not popped before Q publishes its pending pop request, where $S[i]$ represents a cell with index i. Every time a thread pops an element from the stack, it helps a pop peer and updates its pop peer to the next thread. Therefore, every other thread prevents Q from popping a cell in L at most $n - 1$ times, and Q visits at most $(n-1)^2 + 1$ cells in L before its pending pop request closes. In the worst case, these cells locate in different segments, $n - 1$ retired segments is in removing progress when Q publishes its pending pop request, and the segment of $S[i]$ is not retired but every $S[j']$ (i *mod* N = $N - 2$ and $i - N + 1 < j' \leq i$) is not in L. Therefore, Q visits at most $(n-1)^2 \cdot N + N + (n-1) \cdot N + N - 1$ (namely $(n^2 - n + 2) \cdot N - 1$) cells before its pending pop request closes. We can safely draw that this invariant is satisfied.

4.4 Memory Reclamation

Memory management is the key part to make FA-Stack practical. Retired segments are the only garbage that needs reclamation in FA-Stack. Algorithm 7 shows how to remove retired segments and safely reclaim them.

The main problem of removing a retired segment A is how to skip the retired segments around A in a finite number of steps. To make the problem intuitively, Fig. 3 gives an example. In this example, the left and right segments of A are also retired. We assume these retired segments are removed at the same time. If just removing these segments without looking for retired successors and predecessors, a possible link relationship is shown in Fig. 3. In this case, when a subsequent pop operation walks down the stack, it can still visit A by the $prev$ field of the segment B. Therefore, this scheme cannot really remove a retired segment and bound the number of cells visited by a pop operation. To remove A, it is necessary to find the successor and predecessor segments ($next$ and $prev$) of A that are not retired and nearest to A. Then, change $next \rightarrow prev$ and $prev \rightarrow real\_next$ to $prev$ and $next$, respectively. However, the segments around A may continuously become retired. Therefore, it may introduce livelock problem to find such segments $next$ and $prev$.

FA-Stack adopts a wait-free segment removing scheme. Whenever a thread T_i wants to remove a retired segment sp, it first logically removes sp from the stack by changing

$sp \rightarrow retired$ to true. Then, T_i skips at most $NUM\_THREADS - 1$ retired successor segments, and obtains a successor segment $next$, where $NUM\_THREADS$ is the number of threads. Similarly, T_i obtains a predecessor segment $prev$. Next, T_i tries to make $next \rightarrow prev$ point to $prev$ by a CAS till satisfying the three conditions: (1) $next \rightarrow prev$ is null; (2) $next \rightarrow prev$ points to a segment whose $id \leq prev \rightarrow id$; (3) the CAS succeeds. Suppose that a T_i fails in the CAS because another thread T_j makes $next \rightarrow prev$ point to a segment whose id is larger than $prev \rightarrow id$. T_j will not make $next \rightarrow prev$ point to a segment whose id is larger than $prev \rightarrow id$ in subsequent operations, because all of the segments between $prev$ and $next$ are retired. Therefore, this *while loop* definitely stops after failing to make $next \rightarrow prev$ point to $prev$ at most $NUM\_THREADS$ times. Then, if $prev$ is not null, T_i tries to make $prev \rightarrow real\_next$ point to $next$ in a similar way. At last, sp is inserted into $h \rightarrow free\_list$ with a timestamp.

Our segment removing scheme maintains the following invariants:

Invariant 8. *Given a segment sp and push operation P, if P starts after sp is removed by the $remove$ function and does not see sp as the top segment at Line 3 in Algorithm 3, sp is never visited by P during its execution interval.*

P visits segments only when invoking the $find\_cell$ and $remove$ functions. Once P starts, it stores the current timestamp and top segment in the $time\_stamp$ and top field of its handle h in turn. Then, the $find\_cell$ function invoked by P will traverse forward the stack from $h \rightarrow top$. Obviously, sp is behind $h \rightarrow top$. Thus, P cannot visit sp by the $find\_cell$ function. We use contradiction to demonstrate that P cannot visit sp by the $remove$ function. Assuming that sp is one of the predecessor segments visited by P during a call of the $remove$ function on a retired segment sp', there must exist a segment sequence L consisting of $V[i]$ ($0 \leq i \leq m$ and $m \geq 1$; $V[0]$ is sp'; $V[m]$ is sp), and P visits $V[j+1]$ by the $prev$ field of $V[j]$ ($0 \leq j \leq m-1$). According to the characteristic of the $remove$ function, if exist $segment[k]$ ($k < V[m-1] \rightarrow id$ and $k > sp \rightarrow id$), $segment[k]$ must be retired at the time point that the $prev$ field of $V[m-1]$ is set to sp. At each time point, there are at most $NUM\_THREADS$ retired segments in removing progress. Therefore, at most $NUM\_THREADS - 1$ of such $segment[k]$ are visited during the progress of removing sp. Note that sp' becomes retired after P starts. The $remove$ function on sp must ensure that the $prev$ field of a $segment[n]$ ($V[m-1] \rightarrow id \leq n \leq sp' \rightarrow id$) points to a $segment[k']$ ($k' < sp \rightarrow id$), and thus sp cannot belong to L, a contradiction. Therefore, sp cannot be visited as one of sp''s predecessor segments during the removing progress of sp'; similarly, sp cannot be visited as one of sp''s successor segments during the removing progress of sp'. We can safely draw a conclusion that this invariant is satisfied.

Invariant 9. *Given a segment sp and pop operation Q, if Q satisfies the conditions: (1) starting after sp is removed by the $remove$ function and does not see sp as the top segment at Line 3 in Algorithm 5, and (2) not helping a pop operation that starts before sp is removed by the $remove$ function, then sp is never visited by Q during its execution interval.*

Algorithm 7. Time-Stamped Memory Reclamation

```
 1: procedure REMOVE(Handle *h, Segment *sp)
 2:   sp→retired := true
 3:   i := 1
 4:   next := sp→real_next
 5:   if next = null then next := sp→next
 6:   end if
 7:   // next is not null because the last segment of the stack
      is always a dummy segment that is not retired
 8:   while i++ < NUM_THREADS and next→retired do
 9:     rnext := next→real_next
10:     if rnext = null then
11:       next := next→next
12:     else
13:       next := rnext
14:     end if
15:   end while
16:   i := 1
17:   prev := sp→prev
18:   while i++ < NUM_THREADS and prev ≠ null and
      prev→retired do
19:     prev := prev→prev
20:   end while
21:   nprev := next→prev
22:   while nprev ≠ null and (prev = null or nprev→id >
      prev→id) and !CAS(&next→prev, nprev, prev) do
23:     nprev := next→prev
24:   end while
25:   if prev = null then
26:     goto Line 32
27:   end if
28:   pnext := prev→real_next
29:   while (pnext = null or pnext→id < next→id) and !CAS
      (&prev→real_next, pnext, next) do
30:     pnext := prev→real_next
31:   end while
32:   sp→time_stamp := get_timestamp()
33:   insert sp into h→free_list
34:   cleanup(h)
35: end procedure
36: procedure CLEANUP(Handle *h)
37:   for each segment cur in h→free_list do
38:     free := true
39:     for the handle ph of each peer do
40:       if cur = s→top or ph→time_stamp ≤
          cur→time_stamp or ph→top = cur then
41:         free := false
42:         break
43:       end if
44:     end for
45:     if free then
46:       remove cur from h→free_list
47:       free(cur)
48:     end if
49:   end for
50: end procedure
```

Q visits segments in three cases: (1) invoking the $find\_cell$ and $remove$ functions, (2) walking down the stack to find a cell to pop, and (3) helping a pop operation. Similar to Invariant 8, this invariant is satisfied in the first case. In the second case, Q visits segments from $segment[t/N]$ downto

the stack, where t is the value of the top index read by Q. Obviously, sp is behind $segment[t/N]$, and $segment[t/N]$ is not retired at the time point that sp becomes retired. As analyzed in Invariant 8, P cannot visit sp starting from $segment[t/N]$ by the $prev$ field of visited segments in this case. In the third case, Q walks down the stack starting at a segment designated by a pop request. Because this pop request starts after sp is removed, Q will not visit sp according to the aforementioned analysis. We can safely draw a conclusion that this invariant is satisfied.

Listing 2. FA-Stack linearization procedure P. Operations linearized at the same e_j are ordered based on the order of P's steps.

```
1  T_L := 0
2  for j := {0,1,2,...}
3    if e_j is a FAA (line 6 or 23 in Algorithm 3) that
     updates T to t {
4      k := T_L
5      if (f(k) = t − 1) {
6        linearize Push_k at e_j
7        T_L := k + 1
8      }
9    }
10   if e_j is a Read of T performed by a pop (line 4
     in Algorithm 5) {
11     k := T_L - 1;
12     while k >= 0 {
13       if Pop_k has not been linearized at a
         prior event
14         if e_j ≺ Read(Pop_k)
15           break
16         else
17           linearize Pop_k at e_j
18       k := k − 1;
19     }
20     if k < 0 {
21       for every Pop-bar that has not been linearized
         at a prior event
22         if Read(Pop-bar) ≺ e_j
23           linearize Pop-bar at e_j
24     }
25   }
```

To safely reclaim memory for concurrent data structures, it must first ensure that the corresponding memory is no longer used by any thread [22], [23]. According to Invariants 8 and 9, whether a thread visits a removed segment during a push or pop operation, depends on the start time point of the operation and the *top* field of its handle. To give indications for the *cleanup* function, a push or pop operation first obtains the timestamp and stores it in $h \rightarrow timestamp$, and then stores the current top segment in $h \rightarrow top$, where h is the local handle. In addition, before starting to help another pop operation Q, a pop operation updates its timestamp to the timestamp of Q. Therefore, given a removed segment sp and thread T, whenever T runs a push or pop operation, sp will be never visited by T if $sp \neq s \rightarrow top$, $h \rightarrow timestamp > sp \rightarrow timestamp$, and $sp \neq h \rightarrow top$. If sp will be never visited by all threads, it can be safely freed.

5 Correctness Proof

In this section, we prove that FA-Stack is a wait-free linearizable implementation of a stack. We first assign a linearization point to each push and pop operation, and prove that every linearization point happens within the execution interval of the corresponding operation. Next, we prove that the assigned linearization point of every push and pop operation conforms to a correct sequential execution sequence of a LIFO stack. Finally, we reason about the wait-freedom of FA-Stack.

5.1 Linearizability

If a cell is eventually deposited into a pushed element, we define it as an effective cell. Assuming a logical stack L containing only effective cells, the cell index k in L is mapped to the index of the corresponding cell in S by a monotonic function f (e.g., $L[k] = S[f(k)]$, k ∈ {0, 1, 2, ...}). We denote the top indices of L as T_L. We denote a push that deposits an element into $L[k]$ as $Push_k$ and a pop that takes an element from $L[k]$ as Pop_k. We use $\overline{Pop}$ to denote a pop that returns *EMPTY*.

Let $W = \{e_j : j = 0, 1, 2, \ldots\}$ be a possibly infinite execution sequence of FA-Stack. Listing 2 shows the procedure P that assigns a linearization point to each push and pop according to W. For any pop operation Pop_k, we use $Read(Pop_k)$ to represent Pop_k's read of T at line 4 in Algorithm 5, and use $Read_T(Pop_k)$ to represent the returned value of the read. Similarly, if e_j is a *Read* on T, we use $T(e_j)$ to represent its returned value.

Procedure P linearizes each push $Push_k$ at the FAA that updates T to $f(k) + 1$. If $Push_k$ pushes into $L[k]$ via the fast-path, it obtains the index $f(k)$ by its own FAA. This FAA is the linearization point of $Push_k$, and thus definitely happens during the execution interval of $Push_k$. If $Push_k$ fails to complete on the fast-path, it publishes a push request r with an id $r.id$ which is also obtained by its own FAA on T. Then, it and all its helpers ensure $r.id \leq f(k)$. Obviously, when $Push_k$'s push request completes, $T > f(k)$. Since T increments monotonically, the FAA that updates T to $f(k) + 1$ must happen during the execution interval of $Push_k$.

Procedure P linearizes each pop Pop_k at (1) $Read(Pop_k)$ if no such Pop_i exists, where $f(k) < f(i) < Read_T(Pop_k)$ and $Read(Pop_k) \prec Read(Pop_i)$, otherwise (2) the time point that just all such Pop_i have been linearized. In the first case, the linearization point ($Read(Pop_k)$) definitely happens within the execution interval of Pop_k. In the second case, there must exist a Pop_i where $Read(Pop_k) \prec Read(Pop_i)$ and $f(k) < f(i) < Read_T(Pop_k)$, and the procedure P linearizes Pop_k and Pop_i at the same event $Read(Pop_{i'})$ where $Read_T(Pop_k) \leq Read_T(Pop_i) \leq Read_T(Pop_{i'})$ and $f(k) < f(i) \leq f(i')$.

According to Invariant 4, when Pop_k completes, Pop_i must have visited its cell with index $f(i)$ because $f(k) < f(i) < Read_T(Pop_k)$. If Pop_i and $Pop_{i'}$ are the same pop, then $Read(Pop_i)$ is the linearization point of Pop_k, which happens before the completion of Pop_k. We can safely draw that such a linearization point is within the execution interval of Pop_k. If Pop_i and $Pop_{i'}$ are different pop operations, there must exist a Pop_j where $Read(Pop_i) \prec Read(Pop_j)$ and $f(i) < f(j) < Read_T(Pop_i)$, and Pop_j is

also linearized at $Read(Pop_{i'})$. Similarly, Pop_i completes after $Read(Pop_j)$. On the analogy of this, Pop_k completes after $Read(Pop_{i'})$. Therefore, Pop_k's linearization point is always within its execution interval.

Procedure P linearizes each $\overline{Pop}$ at (1) $Read(\overline{Pop})$ if no such Pop_i exists, where $0 \leq f(i) < Read_T(\overline{Pop})$ and $Read(\overline{Pop}) \prec Read(Pop_i)$, otherwise (2) the time point that just all such Pop_i have been linearized. Similar to Pop_k, $\overline{Pop}$'s linearization point is always within its execution interval.

Every Pop is Linearized by P. We show this condition holds using contradiction. Suppose that a pop Pop_k is never linearized. There must exist a Pop_i where $Read(Pop_k) \prec Read(Pop_i)$ and $f(k) < f(i) < Read_T(Pop_k)$, and Pop_i is also never linearized. On the analogy of this, there are infinite number of pop operations that are never linearized. Obviously, all these operations will be active at the same time. In practice, the number of pop operations that are active at the same time is at most n, where n is the number of threads, a contradiction. Therefore, no such Pop_k exists. Similarly, every $\overline{Pop}$ is also linearized. Therefore, this condition is true.

5.2 Meeting Linearizable LIFO Semantics

We now prove that P's linearization of push and pop operations conforms to a correct sequential execution sequence of a LIFO stack. When e_j happens, we define the value of T_L as $T_L(e_j)$.

Lemma 1. *For $i, j \in \{0, 1, 2, \ldots\}$, P's linearization satisfies: (1) if $i < j$, then $Push_i \prec Push_j$; (2) $Push_i \prec Pop_i$.*

Proof. Condition (1) is obviously true since P linearizes $Push_i$ and $Push_j$ at the points that FAA updates T to $i+1$ and $j+1$, respectively.

Condition (2) means that a pop is always linearized after its matching push. This is obviously true since P linearizes Pop_i after $Push_i$ (lines 10-19 in Listing 2). □

Lemma 2. *Every pop operation always returns the latest available element in the stack.*

Proof. Suppose a pop Pop_k that is linearized at e_j. Obviously, any push operation $Push_i$ that is linearized before Pop_k satisfies $0 \leq i < T_L(e_j)$. In addition, P must linearize each Pop_m before Pop_k, where $f(k) < m < T_L(e_j)$ (lines 11-19 in Listing 2). Combining with Lemma 1, Lemma 2 is always true. □

Lemma 3. *When any $\overline{Pop}$ is linearized, the stack has no available elements.*

Proof. Suppose a pop $\overline{Pop}$ that is linearized at e_j. Obviously, any push operation $Push_i$ that is linearized before $\overline{Pop}$ satisfies $0 \leq i < T_L(e_j)$. In addition, P linearizes $\overline{Pop}$ after all Pop_k $(0 \leq k < T_L(e_j))$ (lines 20-24 in Listing 2). Combining with Lemma 1, Lemma 3 is always true. □

Theorem 1. *FA-Stack is a linearizable LIFO stack.*

Proof. Lemmas 1, 2, and 3 show that the assigned linearization point of every push and pop operation on FA-Stack conforms to a correct sequential execution sequence of a LIFO stack. Therefore, FA-Stack is a linearizable LIFO stack. □

5.3 Wait Freedom

We prove that operations supported by FA-Stack are wait-free. For brevity, we refer to the $NUM\_THREADS$ and $MAX\_FAILURES$ constants by n and F, respectively. Note that the number of steps to find a cell by a thread increases with the number of push operations arriving between the time points that the thread reads the top segment and index of the stack respectively. This paper focuses on the case that a thread will not be suspended infinitely during an operation and a finite number of push operations arrive in the interval. Therefore, we claim that the *find_cell* function completes in a finite number of steps in the following analysis.

Lemma 4. *The $remove$ function completes in a finite number of steps.*

Proof Sketch. The $remove$ function contains four *while loop*. Obviously, the number of steps required for a thread to complete the first two *while loop* is bounded by $O(n)$. As analyzed in Section 4.4, the last two *while loop* eventually stops after at most n times of unsuccessful CAS, respectively. In addition, the number of segments to be freed in the $cleanup$ function is finite. Therefore, the $remove$ function completes in a finite number of steps. □

Lemma 5. *Each push operation completes in a finite number of steps.*

Proof Sketch. On the fast path, a push operation P obtains a cell with the *find_cell* function and attempts to deposit its element into the cell by a CAS at most F times. If P switches to the slow path, according to Invariant 3, it repeats the process of attempting to reserve a cell for its request r at most $2 \cdot (n-1)^2 + 1$ times, and invokes the $remove$ function only in the case that it reserves a cell for r but one of its helpers successfully claims another cell for r. In this case, P stops reserving other cells for r. Therefore, P invokes the $remove$ function at most one time. Combining with Lemma 4, P will eventually complete the slow path in a finite number of steps. □

Lemma 6. *Each pop operation completes in a finite number of steps.*

Proof Sketch. Similar to push operations, a pop operation Q will attempt at most F times on the fast path. According to Invariant 7, Q will eventually complete the slow path after visiting $(n^2 - n + 2) \cdot N - 1$ cells. When Q invokes the *help_pop* function to help a pending pop request, it also visits at most $(n^2 - n + 2) \cdot N - 1$ cells before returning from the function. In addition, if Q removes a visited cell and the cell is the last remaining cell of a segment, Q invokes the $remove$ function, and this case happens at most $F + 2 \cdot (n^2 - n + 2) \cdot N - 2$ times during the execution interval of Q. Combining with Lemma 4, each pop operation completes in a finite number of steps. □

Theorem 2. *FA-Stack is wait-free.*

Proof. Lemmas 4, 5, and 6 show that FA-Stack is wait-free because all of its operations are wait-free. □

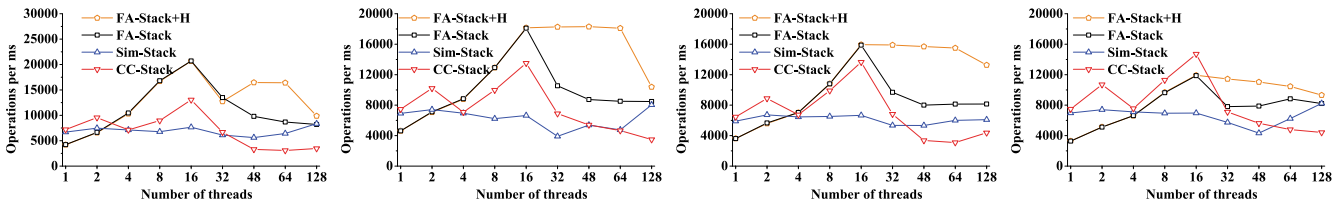

Fig. 4. Throughput of different stacks using Fatourou and Kallimanis's benchmark framework.

6 Evaluation

In this section, we evaluate the performance of FA-Stack by comparing it to the state-of-the-art stack algorithms. All the experiments are conducted on a *Non-Uniform Memory Access* (NUMA) machine with four eight-core 2.0 GHz Intel Xeon E7-4809 v3 chips. Each core has private L1 and L2 caches shared by its two hyper-threads (64 hardware threads in total). All the cores of a chip share a 20 MB shared L3-cache. In the following, we first introduce the selected benchmarks and our experimental methodology, then present the experimental results.

6.1 Benchmarks

We use the following four types of benchmarks to evaluate the performance of stack algorithms:

- *push-only*: Each thread only executes push operations;
- *pairwise*: Each thread executes a push followed by a pop;
- *halfhalf*: Each thread decides uniformly at random to execute a push or pop; The odd of performing a push or pop is 50 percent, respectively;
- *pop-only*: Each thread only executes pop operations.

The first three benchmarks employ an initially-empty stack to which threads apply a series of push and pop operations. In the *pop-only* benchmark, each thread pops from a pre-filled stack with 10,000,000 elements. All the benchmarks execute 10^7 operations (push or pop) that are divided evenly among all threads, and a certain amount of *work* (between 50 and 100 ns) are inserted between each operation to simulate realistic program behaviour. These benchmarks are chosen from previous work [1], [2], [3], [6].

6.2 Experimental Methodology

We first investigate the impact of parameters used by FA-Stack on its performance. Then, we compare FA-Stack to representative stack algorithms in order to answer the two questions: 1) how well does FA-Stack perform against the existing wait-free stacks? 2) how well does FA-Stack perform against *the leading concurrent stacks in performance* (typically not wait-free)? To answer the first question, we compare FA-Stack to Sim-Stack [2] because it is the state-of-the-art wait-free stack. To answer the second question, as the combining algorithm is a well-known technique to improve the performance of concurrent data structures, we use CC-Stack [6] as the representative of stacks based on the combining principle; we also compare FA-Stack to TS-Stack [3] (a state-of-the-art lock-free stack) because lock-free data structures often yield the best performance and many researchers aim to design wait-free data structures as fast as the lock-free ones [4], [5]. Next, we evaluate how many retired segments have been freed at different execution progresses of benchmarks in FA-Stack. Finally, we compare FA-Stack to a representative array-based stack by investigating the number of cells visited by pop operations.

The original TS-Stack is implemented using the Scal benchmark framework in C++ [3], while Sim-Stack and CC-Stack are implemented using the Fatourou and Kallimanis's benchmark framework in C [6]. For the fair comparison with these stacks, we implement FA-Stack on both the benchmark frameworks. We also borrow the idea of LCRQ+H [9] to implement a hierarchical version of FA-Stack named FA-Stack+H for higher performance in NUMA architectures. To be specific, FA-Stack+H has a *cluster* field that identifies the current NUMA node from which most operations should complete. Before starting a push or pop operation, a thread checks if it is running on the *cluster*. If not, the thread waits for a while with a bounded number of iterations, and then CASes *cluster* to be its NUMA node and enters the operation (even if the *CAS* fails). In addition, to show the highest performance and scalability of all the stacks in our NUMA machine, each software thread is mapped to the hardware thread that is closest to previously mapped threads like [10].

6.3 Parameter Sensitivity

FA-Stack uses two parameters N and *MAX_FAILURES*. Their best values can be impacted by the application-level access patterns. For the *push-only* benchmark, a larger N value can help reduce the chance that threads atomically extend the segment list and thus yields a better performance, while varying values of *MAX_FAILURES* ($\geq$1) has the same impact on such benchmark because threads always succeed with 1 attempt for the fast-path push. For other benchmarks, a large N value may improve the worst-case execution time, and increasing *MAX_FAILURES* ($\geq$1) tend to prevent more operations from entering the slow path. Therefore, one value of N or *MAX_FAILURES* that is optimal for a benchmark may be suboptimal for other benchmarks. To select appropriate values for the two parameters, we run all benchmarks with different N and *MAX_FAILURES*. Based on the experimental results, we set N to 120 for ensuring the optimal performance of most benchmarks, and set *MAX_FAILURES* to 100 for making a good trade-off between the performance and progress guarantee of most benchmarks.

6.4 Throughput of Different Stacks

Fig. 4 shows the experimental results with the Fatourou and Kallimanis's benchmark framework. For all the benchmarks, CC-Stack outperforms FA-Stack in sequential executions. This is because CC-Stack incurs lower synchronization cost than FA-Stack in sequential executions. However, this

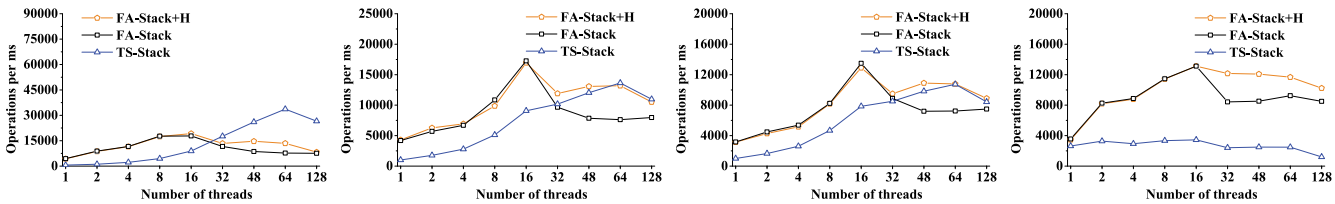

Fig. 5. Throughput of different stacks using Scal benchmark framework.

advantage disappears for the *push-only*, *pairwise*, and *halfhalf* benchmark with 4 or more threads, because CC-stack reduces the synchronization cost by performing push and pop operations serially, while FA-Stack allows threads to perform these operations in parallel. Compared to Sim- Stack and CC-Stack, FA-Stack has relatively less benefits for the *pop-only* benchmark with low number of threads. However, when the number of threads exceeds 32, FA-Stack outperforms CC- Stack by up to 1.9× in the *pop-only* benchmarks.

To achieve wait-freedom, Sim-Stack forces each active thread to execute all pending operations, thus extending the execution time of each operation. Different from Sim-Stack, FA-Stack only forces popping threads to act as helpers, and each popping thread helps only one pending push and pop operation at most, during its execution of a pop operation. As a result, when the number of threads exceeds 4, FA-Stack outperforms Sim-Stack in all the benchmarks. For example, FA-Stack outperforms Sim-Stack by up to 2.4× for the *halfhalf* benchmark.

In our mapping of software threads to hardware threads, when the number of threads exceeds 16, threads cross different NUMA nodes. We can observe that FA-Stack, Sim-Stack and CC-Stack achieve their peak performance at 16 threads, because the expensive communication cost among threads from different NUMA nodes degrades the performance of these stacks. As a hierarchical version of FA-Stack, FA-Stack+H outperforms the original FA-Stack with 32-128 threads by creating batches of operations that complete on the same NUMA node.

Fig. 5 shows the experimental results with the Scal benchmark framework.[7] In TS-Stack, when performing push operations, every thread independently inserts elements into its local buffer, not incurring any synchronization cost among threads. Although FA-Stack allows threads to perform push operations in parallel, they need to first contend for the stack's *top* index by *FAA*. As a result, when the number of threads grows to 32, TS-Stack significantly outperforms FA-Stack and FA-Stack+H in the *push-only* benchmark. In the *pairwise* and *halfhalf* benchmark, when all the threads run on a NUMA node (the number of threads does not exceed 16), FA-Stack outperforms TS-Stack by up to 4.2× and 3.2×, respectively; when the number of threads grows to 32 (threads cross different NUMA nodes), the performance of FA-Stack is lower than that of TS-Stack. The benefit of TS-Stack in NUMA architectures mainly comes from that push operations only access local buffers and thus they do not incur communications among threads from different NUMA nodes. FA-Stack+H alleviates the problem of FA-Stack. When the number of threads is larger than 16, FA-Stack+H matches the performance of TS-Stack or even outperforms it in the *pairwise* and *halfhalf* benchmark. In the *pop-only* benchmark, FA-Stack and FA-Stack+H always outperform TS-Stack, because popping from TS-Stack faces more contentions.

Summary. When the number of threads exceeds 4, FA-Stack always has a higher performance than Sim-Stack that is a state-of-the-art wait-free stack. Although CC-Stack and TS-Stack have a higher performance than FA-Stack and its hierarchical version in some cases, they do not provide a progress guarantee as strong as FA-Stack. Therefore, FA-Stack provides wait-free progress guarantee while attaining high performance.

6.5 Memory Reclamation

Fig. 6 shows the experimental results for the memory reclamation of benchmarks at maximum concurrency. Because the *push-only* benchmark does not remove any segment during its execution interval, it is not included in this experiment. The x-axis presents the *execution progresses* (percentage of completed operations) of the benchmarks, and the y-axis presents the matching percentage of freed retired segments. The experimental results indicate that most retired segments (over 99 percent for the *pairwise* and *pop-only* benchmarks, over 98 percent for the *halfhalf* benchmark) are freed in FA-Stack during the execution interval of the benchmarks.

6.6 Number of Visited Cells

This section experimentally investigates the number of cells visited by pop operations on FA-Stack in comparison with *a representative array-based stack* (referred to as A-Stack here) [21]. We monitor the number of cells visited by each pop operation from FA-Stack and A-Stack in each benchmark that executes 10,000,000, 20,000,000, 30,000,000, 40,000,000, 50,000,000 operations in turn. In each execution, we count the minimum, average, and maximum number of cells visited by pop operations. Due to the space limitation, Fig. 7a only shows the experimental results in the *halfhalf* benchmark at maximum concurrency. As shown in the figure, in each case, the number of cells visited by pop operations

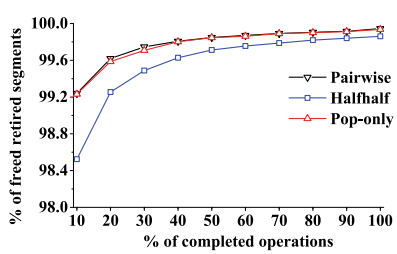

Fig. 6. Memory reclamation of benchmarks in FA-Stack at maximum concurrency.

7. TS-Stack has different implementations. Their differences lie in the timestamping algorithms. In this paper, we compare FA-Stack to the best performing version of TS-Stack that uses the interval timestamp.

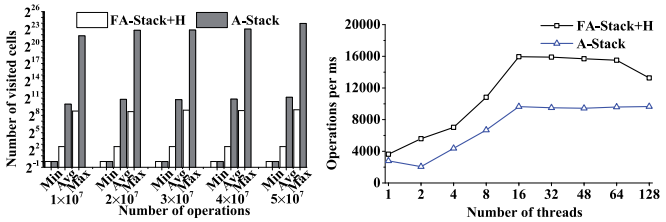

Fig. 7. Comparing FA-Stack to A-Stack in the *halfhalf* benchmark at maximum concurrency: (a) Number of cells visited by pop operations, and (b) throughput.

from A-Stack increases with the increasing of pushed element numbers. In contrast, the number of cells visited by pop operations from FA-Stack does not depend on the number of pushed elements, and it is much lower than that in A-Stack. Therefore, A-Stack is less efficient than FA-Stack. For example, Fig. 7b shows the throughput of FA-Stack and A-Stack with different number of threads in the *halfhalf* benchmark that executes 10,000,000 operations. We can observe that FA-Stack always outperforms A-Stack by over 1.3×.

7 Conclusions and Future Work

This paper presents a fast array-based concurrent stack named FA-Stack. The key idea of FA-Stack is providing opportunities for threads to perform push and pop operations quickly and in parallel. A series of optimizations are proposed to make FA-Stack wait-free. In addition, we prove that FA-Stack is a wait-free linearizable stack with respect to the LIFO semantics. To evaluate the performance of FA-Stack in practice, we compare it to currently state-of-the-art stacks by conducting comprehensive experiments. The experimental results show that FA-Stack provides wait-free progress guarantee while attaining high performance.

Similar to [9], [10], FA-Stack has a limitation that T cannot exceed 2^{63}. In future work, we plan to overcome this limitation. For example, we can use multiple instances of FA-Stack.

Acknowledgments

This work was supported by National Natural Science Foundation of China under grant No. 61702499, National Key Research and Development Program of China under grant No. 2016QY04W0804, Beijing Natural Science Foundation under grant No. 4172069, and Research on Core Technologies of national key infrastructure security supervision platform under grant No. Z161100002616032.

References

[1] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proc. Annu. ACM Symp. Parallelism Algorithms Archit.*, 2004, pp. 206–215.
[2] P. Fatourou and N. D. Kallimanis, "A highly-efficient wait-free universal construction," in *Proc. Annu. ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 325–334.
[3] M. Dodds, A. Haas, and C. M. Kirsch, "A scalable, correct time-stamped stack," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 233–246, Jan. 2015.
[4] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 141–150, Feb. 2012.
[5] S. Timnat and E. Petrank, "A practical wait-free simulation for lock-free data structures," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2014, pp. 357–368.
[6] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 257–266, Feb. 2012.
[7] K. Censor-Hillel, E. Petrank, and S. Timnat, "Help!" in *Proc. ACM Symp. Principles Distrib. Comput.*, 2015, pp. 241–250.
[8] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Languages Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.
[9] A. Morrison and Y. Afek, "Fast concurrent queues for x86 processors," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 103–112, Feb. 2013.
[10] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, pp. 16:1–16:13.
[11] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated," in *Proc. Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2011, pp. 487–498.
[12] R. K. Treiber, "Systems programming: Coping with parallelism," Ohio State Univ., Columbus, OH, USA, 1986.
[13] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *Proc. Annu. ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 2013, pp. 317–328.
[14] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proc. Annu. ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 355–364.
[15] G. Bar-Nissan, D. Hendler, and A. Suissa, "A dynamic elimination-combining stack algorithm," in *Principles of Distributed Systems*. Berlin, Germany: Springer, Jan. 2011.
[16] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Languages Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.
[17] J. Alemany and E. W. Felten, "Performance issues in non-blocking synchronization on shared-memory multiprocessors," in *Proc. ACM Symp. Principles Distrib. Comput.*, 1992, pp. 125–134.
[18] P. Chuong, F. Ellen, and V. Ramachandran, "A universal construction for wait-free transaction friendly data structures," in *Proc. Annu. ACM Symp. Parallelism Algorithms Archit.*, 2010, pp. 335–344.
[19] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2011, pp. 223–234.
[20] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 309–310.
[21] Y. Afek, E. Gafni, and A. Morrison, "Common2 extended to stacks and unbounded concurrency," in *Proc. ACM Symp. Principles Distrib. Comput.*, 2006, pp. 218–227.
[22] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.
[23] O. Balmau, R. Guerraoui, M. Herlihy, and I. Zablotchi, "Fast and robust memory reclamation for concurrent data structures," in *Proc. Annu. ACM Symp. Parallelism Algorithms Archit.*, 2016, pp. 349–359.

Yaqiong Peng received the PhD degree from the Huazhong University of Science and Technology (HUST), in 2016. Currently, she is an assistant professor at the Institute of Information Engineering, Chinese Academy of Sciences. Her current research interests include parallel algorithms, operating systems, and virtualization. She has published over 10 papers in journals and conferences including TPDS, FGCS, CCGRID, IPCCC et al. She is a member of the IEEE.

Zhiyu Hao received the PhD degree in computer system architecture from the Harbin Institute of Technology, in 2007. He is currently a professor in the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include network security, system virtualization, and network emulation.