A Dynamic-Sized Nonblocking Work Stealing Deque

**Danny Hendler, Yossi Lev,
Mark Moir, and Nir Shavit**

A Dynamic-Sized Nonblocking Work Stealing Deque

Danny Hendler, Yossi Lev,* Mark Moir, and Nir Shavit

SMLI TR-2005-144 November 2005

Abstract:

The non-blocking work-stealing algorithm of Arora, Blumofe, and Plaxton [2] (henceforth *ABP work-stealing*) is on its way to becoming the multiprocessor load balancing technology of choice in both industry and academia. This highly efficient scheme is based on a collection of array-based double-ended queues (deques) with low cost synchronization among local and stealing processes. Unfortunately, the algorithm's synchronization protocol is strongly based on the use of fixed size arrays, which are prone to overflows, especially in the multi programmed environments for which they are designed. This is a significant drawback since, apart from memory inefficiency, it means that the size of the deque must be tailored to accommodate the effects of the hard-to-predict level of multiprogramming, and the implementation must include an expensive and application-specific overflow mechanism.

This paper presents the first *dynamic memory* work-stealing algorithm. It is based on a novel way of building non-blocking dynamic-sized work stealing deques by detecting synchronization conflicts based on "pointer-crossing" rather than "gaps between indexes" as in the original ABP algorithm. As we show, the new algorithm dramatically increases robustness and memory efficiency, while causing applications no observable performance penalty. We therefore believe it can replace array-based ABP work stealing deques, eliminating the need for application-specific overflow mechanisms.

---

*This work was conducted while Yossi Lev was a student at Tel Aviv University, and is derived from his MS thesis [1].

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email addresses:
hendlerd@post.tau.ac.il
levyossi@cs.brown.edu
mark.moir@sun.com
nir.shavit@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

A Dynamic-Sized Non-Blocking Work-Stealing Deque

Danny Hendler
Tel-Aviv University

Yossi Lev*
Brown University & Sun Microsystems Laboratories

Mark Moir
Sun Microsystems Laboratories

Nir Shavit
Sun Microsystems Laboratories & Tel-Aviv University

March 2005

Abstract

The non-blocking work-stealing algorithm of Arora, Blumofe, and Plaxton [2] (henceforth *ABP work-stealing*) is on its way to becoming the multiprocessor load balancing technology of choice in both industry and academia. This highly efficient scheme is based on a collection of array-based double-ended queues (deques) with low cost synchronization among local and stealing processes. Unfortunately, the algorithm's synchronization protocol is strongly based on the use of fixed size arrays, which are prone to overflows, especially in the multiprogrammed environments for which they are designed. This is a significant drawback since, apart from memory inefficiency, it means that the size of the deque must be tailored to accommodate the effects of the hard-to-predict level of multiprogramming, and the implementation must include an expensive and application-specific overflow mechanism.

This paper presents the first *dynamic memory* work-stealing algorithm. It is based on a novel way of building non-blocking dynamic-sized work-stealing deques by detecting synchronization conflicts based on "pointer-crossing" rather than "gaps between indexes" as in the original ABP algorithm. As we show, the new algorithm dramatically increases robustness and memory efficiency, while causing applications no observable performance penalty. We therefore believe it can replace array-based ABP work-stealing deques, eliminating the need for application-specific overflow mechanisms.

*This work was conducted while Yossi Lev was a student at Tel-Aviv University, and is derived from his MS thesis [1].

1 Introduction

Scheduling multithreaded computations on multiprocessor machines is a well-studied problem. To execute multithreaded computations, the operating system runs a collection of kernel-level *processes*, one per processor, and each of these processes controls the execution of multiple computational *threads* created dynamically by the executed program. The scheduling problem is that of dynamically deciding which thread is to be run by which process at a given time, so as to maximize the utilization of the available computational resources (processors).

Most of today's multiprocessor machines run programs in a *multiprogrammed mode*, where the number of processors used by a computation grows and shrinks over time. In such a mode, each program has its own set of processes, and the operating system chooses in each step which subset of these processes to run, according to the number of processors available for that program at the time. Therefore the scheduling algorithm must be dynamic (as opposed to static): at each step it must schedule threads onto processes, without knowing which of the processes are going to be run.

When a program is executed on a multiprocessor machine, the threads of computation are dynamically generated by the different processes, implying that the scheduling algorithm must have processes load balance the computational *work* in a distributed fashion. The challenge in designing such distributed work scheduling algorithms is that performing a re-balancing, even between a pair of processes, requires the use of costly synchronization operations. Re-balancing operations must therefore be minimized.

Distributed work scheduling algorithms can be classified according to one of two paradigms: *work-sharing* or *work-stealing*. In work-sharing (also known as *load-distribution*), the processes continuously re-distribute work so as to balance the amount of work assigned to each [3]. In work-stealing, on the other hand, each process tries to work on its newly created threads locally, and attempts to steal threads from other processes only when it has no local threads to execute. This way, the computational overhead of re-balancing is paid by the processes that would otherwise be idle.

The ABP work-stealing algorithm of Arora, Blumofe, and Plaxton [2] has been gaining popularity as the multiprocessor load-balancing technology of choice in both industry and academia [2, 4, 5, 6]. The scheme implements a provably efficient work-stealing paradigm due to Blumofe and Leiserson [7] that allows each process to maintain a local work deque,[1] and steal an item from others if its deque becomes empty. It has been extended in various ways such as stealing multiple items [9] and stealing in a locality-guided way [4]. At the core of the ABP algorithm is an efficient scheme for stealing an item in a non-blocking manner

[1] Actually, the work-stealing algorithm uses a *work-stealing deque*, which is like a deque [8] except that only one process can access one end of the queue (the "bottom"), and only Pop operations can be invoked on the other end (the "top"). For brevity, we refer to the data structure as a deque in the remainder of the paper.

from an array-based deque, minimizing the need for costly Compare-and-Swap (CAS)[2] synchronization operations when fetching items locally.

Unfortunately, the use of fixed size arrays[3] introduces an inefficient memory-size/robustness tradeoff: for n processes and total allocated memory size m, one can tolerate at most $\frac{m}{n}$ items in a deque. Moreover, if overflow does occur, there is no simple way to malloc additional memory and continue. This has, for example, forced parallel garbage collectors using work-stealing to implement an application-specific blocking overflow management mechanism [5, 10]. In multiprogrammed systems, the main target of ABP work-stealing [2], even inefficient over-allocation based on an application's maximal execution-DAG depth [2, 7] may not always work. If a small subset of non-preempted processes end up queuing most of the work items, since the ABP algorithm sometimes starts pushing items from the middle of the array even when the deque is empty, this can lead to overflow.[4]

This state of affairs leaves open the question of designing a dynamic memory algorithm to overcome the above drawbacks, but to do so while maintaining the low-cost synchronization overhead of the ABP algorithm. This is not a straightforward task, since the the array-based ABP algorithm is unique: it is possibly the only real-world algorithm that allows one to transition in a lock-free manner from the common case of using loads and stores to using a costly CAS *only* when a potential conflict requires processes to synchronize. This transition rests on the ability to detect these boundary synchronization cases based on the relative gap among array indexes. There is no straightforward way of translating this algorithmic trick to the pointer-based world of dynamic data structures.

1.1 The New Algorithm

This paper introduces the first lock-free[5] *dynamic-sized* version of the ABP work-stealing algorithm. It provides a near-optimal memory-size/robustness tradeoff: for n processes and total pre-allocated memory size m, it can potentially tolerate up to $O(m)$ items in a single deque. It also allows one to malloc additional memory beyond m when needed, and as our empirical data shows, it is far more robust than the array-based ABP algorithm in multiprogrammed environments.

An ABP-style work-stealing algorithm consists of a collection of *deque* data structures with each process performing pushes and pops on the "bottom" end of its local deque and multiple thieves performing pops on the "top" end. The new algorithm implements each deque as a doubly linked list of nodes, each of

[2]The *CAS(location, old-value, new-value)* operation atomically reads a value from *location*, and writes *new-value* in *location* if and only if the value read is *old-value*. The operation returns a boolean indicating whether it succeeded in updating the location.

[3]One may use cyclic array indexing but this does not help in preventing overflows.

[4]The ABP algorithm's built-in "reset on empty" mechanism helps in some, but not all, of these cases.

[5]Our abstract deque definition is such that the original ABP algorithm is also lock-free.

which is a short array that is dynamically allocated from and freed to a shared pool; see Figure 1. It can also use malloc to add nodes to the shared pool in case its node supply is exhausted.

The main technical difficulties in the design of the new algorithm arise from the need to provide performance comparable to that of ABP. This means the doubly linked list must be manipulated using only loads and stores in the common case, resorting to using a costly CAS *only* when a potential conflict requires it; it is challenging to make this transition correctly while maintaining lock-freedom.

The potential conflict that requires CAS-based synchronization occurs when a pop by a local process and a pop by a thief might both be trying to remove the same item from the deque. The original ABP algorithm detects this scenario by examining the gap between the `Top` and `Bottom` array indexes, and uses a CAS operation only when they are "too close." Moreover, in the original algorithm, the empty deque scenario is checked simply by checking whether `Bottom` $\leq$ `Top`.

A key algorithmic feature of our new algorithm is the creation of an equivalent mechanism to allow detection of these boundary situations in our linked list structures using the relations between the `Top` and `Bottom` pointers, even though these point to entries that may reside in different nodes. On a high level, our idea is to prove that one can restrict the number of possible ways the pointers interact, and therefore, given one pointer, it is possible to calculate the different possible positions for the other pointer that imply such a boundary scenario.

The other key feature of our algorithm is that the dynamic insertion and deletion operations of nodes into the doubly linked-list (when needed in a push or pop) are performed in such a way that the local thread uses only loads and stores. This contrasts with the more general linked-list deque implementations [11, 12] which require a double-compare-and-swap synchronization operation [13] to insert and delete nodes.

1.2 Performance Analysis

We compared our new dynamic-memory work-stealing algorithm to the original ABP algorithm on a 16-node shared memory multiprocessor using the benchmarks of the style used by Blumofe and Papadopoulos [14]. We ran several standard *Splash2* [15] applications using the Hood scheduler [16] with the ABP and new work-stealing algorithms. Our results, presented in Section 3, show that the new algorithm performs as well as ABP, that is, the added dynamic-memory feature does not slow the applications down. Moreover, the new algorithm provides a better memory/robustness ratio: the same amount of memory provides far greater robustness in the new algorithm than the original array-based ABP work-stealing. For example, running Barnes-Hut using ABP work-stealing with an 8-fold level of multiprogramming causes a failure in 40% of the executions if one uses the deque size that works for stand-alone (non-multiprogrammed) runs. It causes *no failures* when using the new dynamic memory work-stealing algorithm.

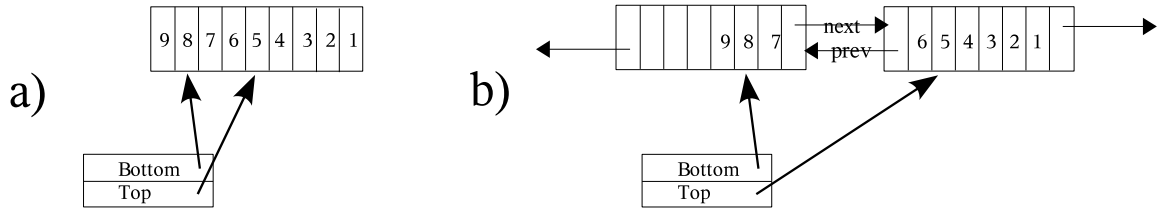

Figure 1: The original ABP deque structure (a) vs. that of the new dynamic deque (b). The structure is after 9 `PushBottom` operations, 4 successful `PopTop` operations, and 2 `PopBottom` operations. (In practice the original ABP deque uses cell indexes and not pointers as in our illustration.)

2 The Algorithm

2.1 Basic Description

Figure 1(b) presents our new deque data-structure. The doubly-linked list's nodes are allocated from and freed to a shared pool, and the only case in which one may need to malloc additional storage is if the shared pool is exhausted. The deque supports the `PushBottom` and `PopBottom` operations for the local process, and the `PopTop` operation for the thieves.

The first technical difficulty we encountered was in detecting the conflict that may arise when the local `PopBottom` and a thief's `PopTop` operations concurrently try to remove the last item from the deque. Our solution is based on the observation that when the deque is empty, one can restrict the number of possible scenarios among the pointers. Given one pointer, we show that the "virtual" distance of the other, ignoring which array it resides in, cannot be more than 1 if the deque is empty. We can thus easily test for each of these scenarios. (Several such scenarios are depicted in parts (a) and (b) of Figure 2).

The next problem one faces is the maintenance of the deque's doubly-linked list structure. We wish to avoid using CAS operations when updating the `next` and `previous` pointers, since this would cause a significant performance penalty. Our solution is to allow only the local process to update these fields, thus preventing `PopTop` operations from doing so when moving from one node to another. We would like to keep the deque dynamic, which means freeing old nodes when they're not needed anymore. This restriction immediately implies that an active list node may point to an already freed node, or even to a node which was freed and reallocated again, essentially ruining the list structure. As we prove, the algorithm can overcome this problem by having a `PopTop` operation that moves to a new node free only the node *preceding* the old node and not the old node itself. This allows us to maintain the invariant that the doubly-linked list structure

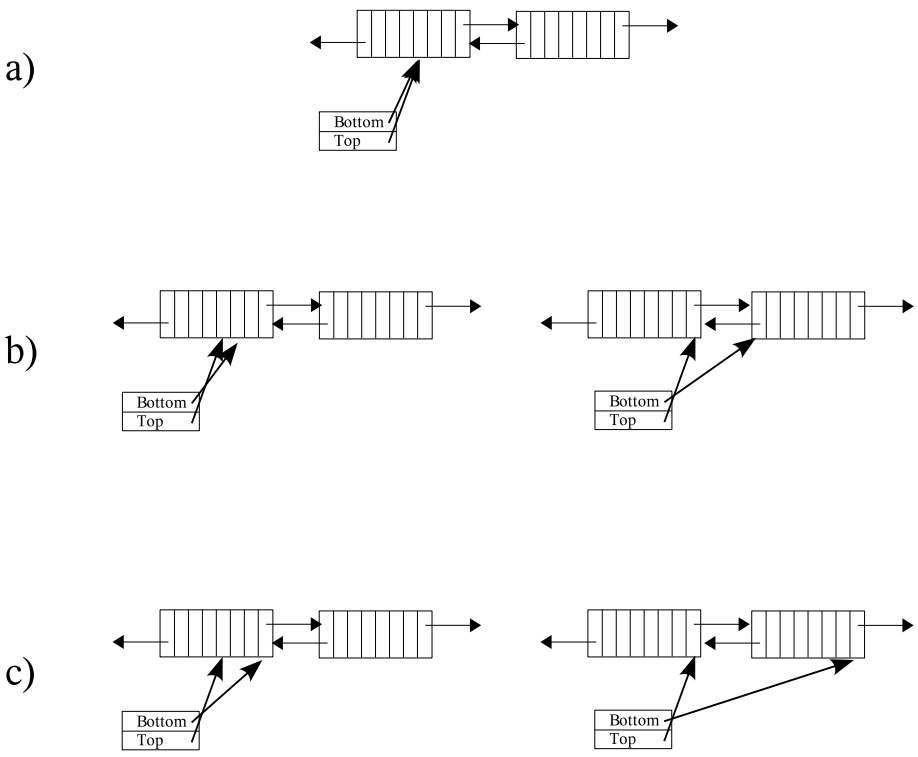

Figure 2: The different types of empty deque scenarios. (a) Simple: `Bottom` and `Top` point to the same cell. (b) Simple Crossing: both the left and right scenarios are examples where `Bottom` passed over `Top` by one cell, but they still point to neighboring cells. (c) Non-Simple Crossing (with the reset-on-empty heuristic): both the left and right scenarios are examples of how pointers can cross given the reset-on-empty heuristic, between the reset of `Bottom` to the reset of Top.

between the `Top` and `Bottom` pointers is preserved. This is true even in scenarios such as that depicted in parts b and c of Figure 2 where the pointers cross over.

2.2 The Implementation

`C++`-like pseudocode for our deque algorithm is given in Figures 3-5. As depicted in Figure 3, the deque object stores the `Bottom` and `Top` pointers information in the `Bottom` and `Top` data members. This information includes the pointer to a list's node and an offset into that node's array. For the `Top` variable, it also includes a tag value to prevent the ABA problem [17]. The deque methods uses the *EncodeBottom*, *DecodeBottom*, *EncodeTop* and *DecodeTop* macros to encode/decode this information to/from a value that fits in a CAS-able size

```
struct BottomStruct {
     DequeNode* nodeP;
     int cellIndex;
};
struct TopStruct {
     DequeNode* nodeP;
     int cellIndex;
     int tag;
};

class DynamicDeque {
     void PushBottom(ThreadInfo theData);
     ThreadInfo PopTop();
     ThreadInfo PopBottom();

     BottomStruct Bottom;
     TopStruct Top;
};
```

```
struct DequeNode {
     enum{ArraySize=/*Size of array*/};
     ThreadInfo itsDataArr[ArraySize];
     DequeNode* next;
     DequeNode* prev;
};
```

Figure 3: Data types and classes used by the dynamic deque algorithm.

word.[6] Underlined procedures in the pseudocode represent code blocks which are presented in the detailed algorithm presentation used for the correctness proof in Section 4. We now describe each of the methods.

2.2.1 PushBottom

The `PushBottom` method begins by reading `Bottom` and storing the pushed value in the cell it's pointing to (Lines 1-2). Then it calculates the next value of `Bottom` linking a new node to the list if necessary (Lines 3-14). Finally the method updates `Bottom` to its new value (Line 15). As in the original ABP algorithm, this method is executed only by the owner process, and therefore regular writes suffice (both for the value and `Bottom` updates). Note that the new node is linked to the list *before* `Bottom` is updated, so the list structure is preserved for the nodes between `Bottom` and `Top`.

2.2.2 PopTop

The `PopTop` method begins by reading the `Top` and `Bottom` values, in that order (Lines 16-18). Then it tests whether these values indicate an empty deque, and returns `EMPTY` if they do[7] (Line 19). Otherwise, it calculates the next position for `Top` (Lines 20-31). Before updating `Top` to its new value, the method must read the value which should be returned if the steal succeeds (Line 32) (this read

[6] If the architecture does not support a 64-bit CAS operation, we may not have the space to save the whole node pointer. In this case, we might use the offset of the node from some base address given by the shared memory pool. For example, if the nodes are allocated continuously, the address of the first node can be such a base address.

[7] This test may also return ABORT if `Top` was modified, since then it is not guaranteed that the tested values represent a consistent view of the memory.

```
void DynamicDedeque::PushBottom(ThreadInfo theData)
{
1  <currNode, currIndex> = DecodeBottom(Bottom); // Read Bottom data
2  currNode->itsDataArr[currIndex] = theData;    // Write data in current bottom cell
3  if (currIndex!=0)
4  {
5              newNode = currNode;
6              newIndex = currIndex-1;
7  }
8  else
9  {  // Allocate and link a new node:
10             newNode = AllocateNode();
11             newNode->next = currNode;
12             currNode->prev = newNode;
13             newIndex = DequeNode::ArraySize-1;
14 }
15 Bottom = EncodeBottom(newNode,newIndex);   // Update Bottom
}


ThreadInfo DynamicDedeque::PopTop()
{
16 currTop = Top;    // Read Top
17 <currTopTag, currTopNode, currTopIndex> = DecodeTop(currTop);
18 currBottom = Bottom; // Read Bottom
19 if (EmptinessTest(currBottom,currTop)) {
      if (currTop == Top) {return EMPTY;} else {return ABORT;}
   }
20 if (currTopIndex!=0)    // if deque isn't empty, calculate next top pointer:
21 {  // stay at current node:
22             newTopTag = currTopTag;
23             newTopNode = currTopNode;
24             newTopIndex = currTopIndex-1;
25 }
26 else
27 {  // move to next node and update tag:
28             newTopTag = currTopTag+1;
29             newTopNode = currTopNode->prev;
30             newTopIndex = DequeNode::ArraySize-1;
31 }
32 retVal = currTopNode->itsDataArr[currTopIndex]; // Read value
33 newTopVal = Encode(newTopTag,newTopNode,newTopIndex);
34 if (CAS(&Top, currTop, newTopVal))  //Try to update Top using CAS
35 {
36             FreeOldNodeIfNeeded();
37             return retVal;
38 }
39 else
40 {
41             return ABORT;
42 }
}
```

Figure 4: Pseudocode for the PushBottom and PopTop operations.

cannot be done after the update of `Top` because by then the node may already be freed by some other concurrent `PopTop` execution). Finally the method tries to update `Top` to its new value using a CAS operation (Line 34), returning the popped value if it succeeds, or `ABORT` if it fails. (In the work-stealing algorithm, if a thief process encounters contention with another, it may be preferable to try stealing from a different deque; returning `ABORT` in this case provides the opportunity for the system to decide between retrying on the same deque or doing something different.) If the CAS succeeds, the method also checks whether there is an old node that needs to be freed (Line 36). As explained earlier, a node is released only if Top moved to a new node, and the node released is not the old top node, but the preceding one.

2.2.3 PopBottom

The `PopBottom` method begins by reading `Bottom` and updating it to its new value (Lines 43-55) after reading the value to be popped (Line 54). Then it reads the value of `Top` (Line 56), to check for the special cases of popping the last entry of the deque, and popping from an empty deque. If the `Top` value read points to the old `Bottom` position (Lines 58-63), then the method rewrites `Bottom` to its old position, and returns `EMPTY` (since the deque was empty even without this `PopBottom` operation). Otherwise, if `Top` is pointing to the new `Bottom` position (Lines 64-78), then the popped entry was the last in the deque, and as in the original ABP algorithm, the method updates the `Top` tag value using a CAS, to prevent a concurrent `PopTop` operation from popping out the same entry. Otherwise there was at least one entry in the deque after the `Bottom` update (lines 79-83), in which case the popped entry is returned. Note that, as in the original ABP algorithm, most executions of the method will be short, and will not involve any CAS-based synchronization operations.

2.2.4 Memory Management

We implement the shared node pool using a variation of Scott's shared pool [18]. It maintains a local group of g nodes per process, from which the process may allocate nodes without the need to synchronize. When the nodes in this local group are exhausted, it allocates a new group of g nodes from a shared LIFO pool using a CAS operation. When a process frees a node, it returns it to its local group, and if the size of the local group exceeds $2g$, it returns g nodes to the shared LIFO pool. In our benchmarks we used a group size of 1, which means that in case of a fluctuation between pushing and popping, the first node is always local and CAS is not necessary.

2.3 Enhancements

We briefly describe two enhancements to the above dynamic-memory deque algorithm.

```
ThreadInfo DynamicDedeque::PopBottom()
{
43 <oldBotNode,oldBotIndex > = DecodeBottom(Bottom); // Read Bottom Data
44 if (oldBotIndex != DequeNode::ArraySize-1)
45 {
46             newBotNode = oldBotNode;
47             newBotIndex = oldBotIndex+1;
48 }
49 else
50 {
51             newBotNode = oldBotNode->next;
52             newBotIndex = 0;
53 }
54 retVal = newBotNode->itsDataArr[newBotIndex];      // Read data to be popped
55 Bottom = EncodeBottom(newBotNode,newBotIndex);     // Update Bottom
56 currTop = Top;                                     // Read Top
57 <currTopTag,currTopNode,currTopIndex> = DecodeTop(currTop);

58 if (oldBotNode == currTopNode &&          // Case 1: if Top has crossed Bottom
59     oldBotIndex == curTopIndex  )
60 {
      //Return bottom to its old possition:
61             Bottom = EncodeBottom(oldBotNode,oldBotIndex);
62             return EMPTY;
63 }
64 else if ( newBotNode == currTopNode &&    // Case 2: When popping the last entry
65           newBotIndex == currTopIndex )   //         in the deque (i.e. deque is
66 {                                         //         empty after the update of bottom).

      //Try to update Top's tag so no concurrent PopTop operation will also pop the same entry:
67             newTopVal = Encode(currTopTag+1, currTopNode, currTopIndex);
68             if (CAS(&Top, currTop, newTopVal))
69             {
70                FreeOldNodeIfNeeded();
71                return retVal;
72             }
73             else  // if CAS failed (i.e. a concurrent PopTop operation already popped the last entry):
74             {
               //Return bottom to its old possition:
75                Bottom = EncodeBottom(oldBotNode,oldBotIndex);
76                return EMPTY;
77             }
78 }
79 else // Case 3: Regular case (i.e. there was at least one entry in the deque after bottom's update):
80 {
81                FreeOldNodeIfNeeded();
82                return retVal;
83 }
}
```

Figure 5: Pseudocode for the PopBottom operation.

2.3.1 Reset-on-Empty

In the original ABP algorithm, the `PopBottom` operation uses a mechanism that resets `Top` and `Bottom` to point back to the beginning of the array every time it de-

tects an empty deque (including the case of popping the last entry by `PopBottom`). This reset operation is necessary in ABP since it is the only "anti-overflow" mechanism at its disposal.

Our algorithm does not need this method to prevent overflows, since it works with the dynamic nodes. However, adding a version of this resetting feature gives the potential of improving our space complexity, especially when working with large nodes.

There are two issues to be noted when implementing the reset-on-empty mechanism in our dynamic deque. The first issue is that while performing the reset operation, we create another type of empty deque scenario, in which `Top` and `Bottom` do not point to the same cells nor to neighboring ones (see part *c* of Figure 2). This scenario requires a more complicated check for the empty deque scenario by the `PopTop` method (Line 19). The second issue is that we must be careful when choosing the array node to which `Top` and `Bottom` point after the reset. In case the pointers point to the same node before the reset, we simply reset to the beginning of that node. Otherwise, we reset to the beginning of the node pointed to by Top. Note, however, that `Top` may point to the same node as `Bottom` and then be updated by a concurrent `PopTop` operation, which may result in changing on-the-fly the node to which we direct `Top` and `Bottom`.

2.3.2 Using a Base Array

In the implementation described, all the deque nodes are identical and allocated from the shared pool. This introduces a trade-off between the performance of the algorithm and its space complexity: small arrays save space but cost in allocation overhead, while large arrays cost space but reduce the allocation overhead.

One possible improvement is to use a large array for the initial `base` node, allocated for each of the deques, and to use the pool only when overflow space is needed. This base node is used only by the process/deque it was originally allocated to, and is never freed to the shared pool. Whenever a Pop operation frees this node, it raises a boolean flag, indicating that the base node is now free. When a `PushBottom` operation needs to allocate and link a new node, it first checks this flag, and if true, links the base node to the deque (instead of a regular node allocated from the shared pool).

3 Performance

We evaluated the performance of the new dynamic memory work-stealing algorithm in comparison to the original fixed-array based ABP work-stealing algorithm in an environment similar to that used by Blumofe and Papadopoulos [14] in their evaluation of the ABP algorithm. Our results include tests running several standard Splash2 [15] applications using the *Hood Library* [16] on a 16 node Sun Enterprise™ 6500, an SMP machine formed from 8 boards of two 400MHz

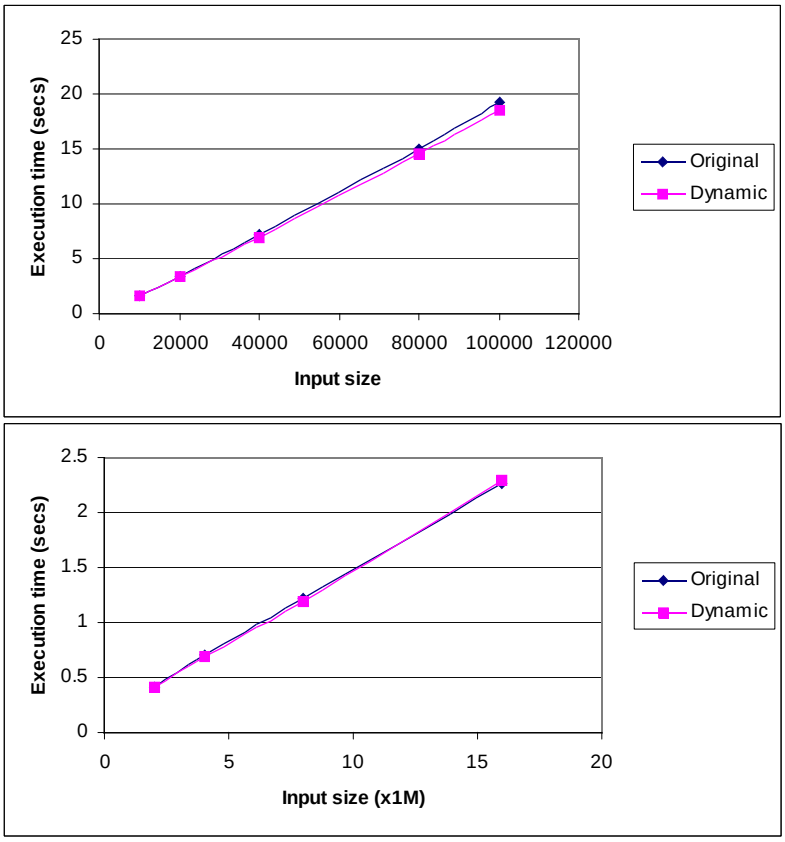

Figure 6: *Barnes Hut* Benchmark on top and *MergeSort* on the bottom

UltraSPARC® processors, connected by a crossbar UPA switch, and running the Solaris™ 9 Operating System.

Our benchmarks used the work-stealing algorithms as the load balancing mechanism in Hood. The Hood package uses the original ABP deques for the scheduling of threads over processes. We compiled two versions of the Hood library, one using an ABP implementation, and the other using the new implementation. In order for the comparison to be fair, we implemented both algorithms in `C++`, using the same tagging method.

We present here our results running the *Barnes Hut* and *MergeSort* Splash2 [15] applications. Each application was compiled with the minimal ABP deque size needed for a stand-alone run with the biggest input tested. For our deque algorithm we chose a base-array size of about 75% of the ABP deque size, a node array size of 6 items, and a shared pool size such that the total memory used (by the deques and the shared pool together) is no more than the total memory used by all ABP deques. In all our benchmarks the number of processes equaled the number of processors on the machine.

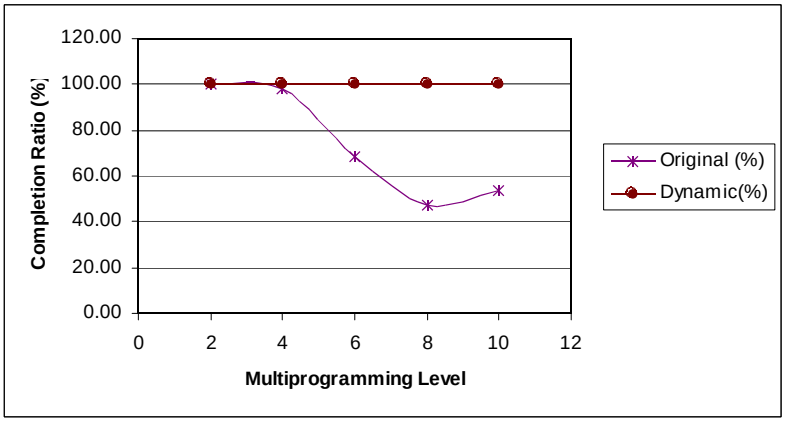

Figure 7: *Barnes Hut* completion ratio vs. level of multiprogramming

Figure 6 shows the total execution time of both algorithms, running stand-alone, as we vary the input size. As can be seen, there is no real difference in performance between the two approaches. This is in spite of the fact that our tests show that the deque operations of the new algorithm take as much as 30% more time on average than those of ABP. The explanation is simple: work-stealing accounts for only a small fraction of the execution time in these (and in fact in most) applications. In all cases both algorithms had a 100% completion rate in stand-alone mode, i.e., none of the deques overflowed.

Figure 7 shows the results of running the *Barnes Hut* [15] application (on the largest input) in a multiprogrammed fashion by running multiple instances of Hood in parallel. The graph shows the completion rate of both algorithms as a function of the multiprogramming level (i.e. the number of instances run in parallel). One can clearly see that while both versions perform perfectly at a multiprogramming level of 2, ABP work-stealing degrades rapidly as the level of multiprogramming increases, while the new algorithm maintains its 100% completion rate. By checking Hood's statistics regarding the amount of work done by each process, we noticed that some processes complete 0 work, which means much higher workloads on the others. This, we believe, caused the deque size which worked for a stand-alone run (in which the work was more evenly distributed between the processes), to overflow in the multiprogrammed run. We also note that as the workload on individual processes increases, the chances of a "reset-on-empty" decrease, and the likelihood of overflow increases. In the new dynamic version, because 25% of the memory is allocated in the common shared pool, there is much more flexibility in dealing with the work imbalance between the deques, and no overflow occurs.

Our preliminary benchmarks clearly show that for the same amount of memory, we get significantly more robustness with the new dynamic algorithm than with the original ABP algorithm, with a virtually unnoticeable effect on the ap-

plication's overall performance. It also shows that the deque size depends on the maximal level of multiprogramming in the system, an unpredictable parameter which one may want to avoid reasoning about by simply using the new memory version of the ABP work-stealing algorithm.

4 Correctness Proof

4.1 Overview

This section contains a detailed proof that the algorithm in Section 2 implements a lock-free linearizable deque.[8] While a work-stealing system generally uses several deques, our proof concentrates on a single deque.

We first define notation and terminology and present a detailed version of the algorithm's pseudocode that we use throughout the proof. In Sections 4.2-4.6, we prove various properties of the algorithm, which are used later in the linearizability proof. Section 4.7 specifies the sequential semantics of the implemented deque and then shows that the deque is linearizable to this specification. Finally, Section 4.8 shows that the algorithm is lock-free.

4.1.1 Notation

Formally, we model the algorithm by a labelled state-transition system, where the labels are called *actions*. We write $s \xrightarrow{a} s'$ if the system has a transition from s to s' labelled a; s is called the *pre-state*, s' the *post-state*. We say that an action a is *enabled* in a state s if there exists another state s' such that $s \xrightarrow{a} s'$. An execution is a sequence of transitions such that the pre-state of the first transition is the initial state, and the post-state of any transition (except the last) is the pre-state of the next transition.

We use s and s' for states, a for actions, and p, p' and p'' for processes. We use $p@X$ to mean that process p is ready to execute statement number X. We use $p@\langle X_1, X_2, ..., X_n\rangle$ to denote $p@X_1 \vee p@X_2 \vee ... \vee p@X_n$. If process p is not executing any operation then $p@0$, holds. Thus, $p@0$ holds initially for all p, statements of process that return from any of the operations establish $p@0$, and if $p@0$ holds, then an action of process p is enabled that nondeterministically chooses a legal operation and parameters, and invokes the operation, thereby setting p's program counter to the first line of that operation, and establishing legal values for its parameters. We denote a private variable v of process p by $p.v$.

For any variable v, shared or local, $s.v$ denotes the value of v is state s. For any logical expression E, $s.E$ holds if and only if E holds in state s.

[8] As noted previously, the data structure we implement is not strictly speaking a deque. The precise semantics of the implemented data structure is specified in Section 4.7.1.

4.1.2 Pseudocode

```
   DynamicDeque:DynamicDeque()
   nodeA = AllocateNode();
   nodeB = AllocateNode();
   nodeA→next = nodeB;
   nodeB→prev = nodeA;
   Bottom= EncodeBottom(nodeA,DequeNode::ArraySize-1);
   Top= EncodeTop(0 ,nodeA, DequeNode::ArraySize-1);
   Ordered= {nodeA, nodeB};
```

Figure 8: Deque Constructor

```
   void DynamicDedeque::PushBottom(ThreadInfo theData)
1  <currNode, currIndex> = DecodeBottom(Bottom);
2  currNode→itsDataArr[currIndex] = theData;
   if (currIndex!=0) then
3      newNode = currNode;
       newIndex = currIndex-1;
   else
4      newNode = AllocateNode();
5      newNode→next = currNode;
6      currNode→prev = newNode;
       newIndex = DequeNode::ArraySize-1;
7  Bottom= EncodeBottom(newNode,newIndex);
   if (currNode != newNode) then Ordered.AddLeft(newNode);
```

Figure 9: The PushBottom Method

Figures 8, 9, 10, 11 and 12 present the detailed pseudocode of the deque implementation. The `EncodeTop`, `DecodeTop`, `EncodeBottom` and `DecodeBottom` macros are described in Section 4.2.2. The `Ordered` variable used in the pseudocode is an auxiliary variable; its use is described in Section 4.3. Auxiliary variables do not affect the behavior of the algorithm, and are used only for the proof: they are not included in a real implementation.

We consider execution of the algorithm beginning at one numbered statement and ending immediately before the next numbered statement to be one atomic action. This is justified by the fact that no such action accesses more than one shared variable (except auxiliary variables), and atomicity of the actions is consistent with available operations for memory access. Note that we can include accesses to auxiliary variables in an atomic action because they are not included in a real implementation, and therefore are not required to comport with the atomicity constraints of a target architecture.

As a concrete example, consider the action labelled 17 in Figure 10, when executed by process p. This action atomically does the following. First, the action reads `Top` and compares the value read to $p.currTop$. If `Top` $\neq p.currTop$,

```
   ThreadInfo DynamicDedeque::PopTop()
 8 currTop = Top;
   <currTopTag, currTopNode, currTopIndex> = DecodeTop(currTop);
 9 currBottom = Bottom;
10 if (IndicateEmpty(currBottom, currTop)) then
11     if (currTop == Top) then return EMPTY;
       return ABORT;
12 if (currTopIndex!=0) then
13     nodeToFree = NULL;
       newTopTag = currTopTag;
       newTopNode = currTopNode;
       newTopIndex = currTopIndex-1;
       newTopVal = EncodeTop(newTopTag,newTopNode,newTopIndex);
   else
14     nodeToFree = currTopNode→next;
15     newTopTag = currTopTag+1;
       newTopNode = currTopNode→prev;
       newTopIndex = DequeNode::ArraySize-1;
       newTopVal = EncodeTop(newTopTag,newTopNode,newTopIndex);
16 retVal = currTopNode→itsDataArr[currTopIndex];
17 if (CAS(&Top, currTop, newTopVal)) then
       if (nodeToFree != NULL) then Ordered.RemoveRight();
18     if (nodeToFree != NULL) then FreeNode(nodeToFree);
19     return retVal;
   else
20     return ABORT;
```

Figure 10: The PopTop Method

then the action changes p's program counter to 20. Otherwise, the action stores $p.newTopVal$ to `Top`, removes the rightmost element from `Ordered` if $p.nodeToFree \neq NULL$, and changes p's program counter to 18.

4.1.3 Proof Method

Most of the invariants in the proof are proved by induction on the length of an arbitrary execution of the algorithm. That is, we show that the invariant holds initially, and that for any transition $s \xrightarrow{a} s'$, if the invariant holds in the pre-state s, then it also holds in the post-state s'. For invariants of the form $A \Rightarrow B$, we often find it convenient to prove this by showing that the consequent (B) holds after any statement execution establishes the antecedent (A), and that no statement execution falsifies the consequent while the antecedent holds.

It is convenient to prove the conjunction of all of the properties, rather than proving them one by one. This way, we can assume that all properties hold in the pre-state when proving that a particular property holds in the post-state. It is also convenient to be able to use other properties in the post-state. However,

```
   ThreadInfo DynamicDedeque::PopBottom()
21 oldBotVal = Bottom;
   <oldBotNode,oldBotIndex> = DecodeBottom(oldBotVal);
22 if (oldBotIndex != DequeNode::ArraySize-1) then
23     newBotNode = oldBotNode;
       newBotIndex = oldBotIndex+1;
       newBotVal = EncodeBottom(newBotNode,newBotIndex);
   else
24     newBotNode = oldBotNode→next;
       newBotIndex = 0;
       newBotVal = EncodeBottom(newBotNode,newBotIndex);
25 Bottom= newBotVal;
26 currTop = Top;
   <currTopTag,currTopNode,currTopIndex> = DecodeTop(currTop);
27 retVal = newBotNode→itsDataArr[newBotIndex];
28 if (oldBotNode == currTopNode && oldBotIndex == curTopIndex) then
29     Bottom= EncodeBottom(oldBotNode,oldBotIndex);
30     return EMPTY;
31 else if (newBotNode == currTopNode && newBotIndex == currTopIndex)
   then
32     newTopVal = EncodeTop(currTopTag+1, currTopNode, currTopIndex);
33     if (CAS(&Top, currTop, newTopVal)) then
34         if (oldBotNode != newBotNode) then
               FreeNode(oldBotNode);
               Ordered.RemoveLeft();
35         return retVal;
       else
36         Bottom= EncodeBottom(oldBotNode,oldBotIndex);
37         return EMPTY;
   else
38     if (oldBotNode != newBotNode) then
           FreeNode(oldBotNode);
           Ordered.RemoveLeft();
39     return retVal;
```

Figure 11: The PopBottom Method

this must be done with some care, in order to avoid circular reasoning. It is important that there is a single order in which we prove all properties hold in the post-state, given the inductive assumption that they all hold in the pre-state, without using any properties we have not yet proved. However, presenting the proofs in this order would disturb the flow of the proof from the reader's point of view. Therefore, we now present some rules that we adopted that imply that such an order exists.

The properties of the proof (Invariants, Claims, Lemmas and Corollaries) are indexed by the order in which they are proved. In some cases, we state an invari-

```
bool IndicateEmpty(BottomStruct bottomVal, TopStruct topVal)
<botNode, botCellIndex> = DecodeBottom(bottomVal);
<topTag, topNode, topCellIndex> = DecodeTop(topVal);
if ((botNode==topNode) && (botCellIndex==topCellIndex ||
botCellIndex==(topCellIndex+1))) then
    return true;
else if ((botNode==topNode→next) && (botCellIndex==0) &&
(topCellIndex==(DequeNode::ArraySize-1))) then
    return true;
return false;
```

Figure 12: The `IndicateEmpty` Macro

ant without proving it immediately, and only provide its proof after presenting and proving some other properties. We call such invariants *Conjectures* to clearly distinguish them from regular properties, which are proved as soon as they are stated. To avoid circular reasoning, our proofs obey the following rules:

1. The proof of Property i can use any Property j in the pre-state.
2. If Property i is not a Conjecture, its proof can use the following properties in the post-state:
   (a) All Conjectures.
   (b) Property j if and only if $j < i$.
3. The proof of Conjecture i can use Conjecture j in the post-state if and only if $i < j$.

Informally, these rules simply state that the proof of a non-Conjecture property can use in the post-state any other property that was already stated (because the only properties that were stated before it but with higher index are Conjectures), and that the proof of a Conjecture can use in the post-state any other Conjecture that was not already proven. The following shows that our proof method is sound.

Soundness of the proof method: By considering the proof of each property in the following order, we see that each property can be proved without using a property not already proved. We assume all properties hold in the pre-state s, and we want to show that all of them hold in the post-state s'. We begin with the highest indexed Conjecture m (that is, for all Conjecture j, $m \geq j$). Because the proof of Conjecture m does not use any other property in the induction post-state s', we can infer that Conjecture m holds in s'. Similarly, because Conjectures use only higher numbered Conjectures in the post-state, by considering the proofs of all Conjectures in reverse order, none of the proofs depends on a property that has not already been proved.

Having proved all Conjectures, we can now prove all other properties in order, starting with the lowest indexed one. When considering Property i, since Property j for $j < i$ and all conjectures were already shown to hold in s', we can infer that Property i holds in s' as well.

To make the proof more readable, we also avoid using in the pre-state properties that were not yet stated.

4.2 Basic Notation and Invariants

4.2.1 The Deque Data Structure

Our deque is implemented using a doubly linked list. Each list node contains an array of deque entries. The structure has `Bottom` and `Top` variables that indicate cells at the two ends of the deque; these variables are discussed in more detail in Section 4.2.2. We use the following notation:

- We let N_j denote a (pointer to a) deque node, and C_i denotes a cell at index i in a node. $C_i \in N_j$ denotes that C_i is the ith cell in node N_j.
- We let $Node(C_i)$ denote the node of cell C_i. That is: $Node(C_i) = N_j \Leftrightarrow C_i \in N_j$.
- The *Bottom cell* of the deque, denoted by C_B, is the cell indicated by `Bottom`. The *Bottom node* is the node in which C_B resides, and is denoted by N_B.
- The *Top cell* of the deque, denoted by C_T, is the cell indicated by `Top`. The *Top node* is the node in which C_T resides, and is denoted by N_T.
- If N is a deque node, than $N \rightarrow next$ is the node pointed to by N's next pointer, and $N \rightarrow prev$ is the node pointed to by its previous pointer.

The following property models the assumption that only one process calls the PushBottom and PopBottom operations.

Invariant 1. *If* $p@\langle 1 \ldots 7, 21 \ldots 39 \rangle$ *then:*

1. *p is the owner process of the deque.*
2. *There is no $p' \neq p$ such that $p'@\langle 1 \ldots 7, 21 \ldots 39 \rangle$.*

Proof. The invariant follows immediately from the requirement that only the owner process may call the PushBottom or PopBottom procedures. □

The following lemma states that various variables are not modified by various transitions.

Lemma 2. *Consider a transition* $s \xrightarrow{a} s'$. *Then:*

1. $p@\langle 2 \dots 7 \rangle \Rightarrow (s.p.currNode = s'.p.currNode \wedge s.p.currIndex = s'.p.currIndex)$.
2. $p@\langle 9 \dots 20 \rangle \Rightarrow (s.p.currTop = s'.p.currTop \wedge s.p.currTopTag = s'.p.currTopTag \wedge s.p.currTopNode = s'.p.currTopNode \wedge s.p.currTopIndex = s'.p.currTopIndex)$.
3. $p@\langle 16 \dots 20 \rangle \Rightarrow (s.p.newTopVal = s'.p.newTopVal \wedge s.p.nodeToFree = s'.p.nodeToFree \wedge s.p.newTopTag = s'.p.newTopTag \wedge s.p.newTopNode = s'.p.newTopNode \wedge s.p.newTopIndex = s'.p.newTopIndex)$.
4. $p@\langle 22 \dots 39 \rangle \Rightarrow (s.p.oldBotVal = s'.p.oldBotVal \wedge s.p.oldBotNode = s'.p.oldBotNode \wedge s.p.oldBotIndex = s'.p.oldBotIndex)$.
5. $p@\langle 27 \dots 39 \rangle \Rightarrow (s.p.currTop = s'.p.currTop \wedge s.p.currTopTag = s'.p.currTopTag \wedge s.p.currTopNode = s'.p.currTopNode \wedge s.p.currTopIndex = s'.p.currTopIndex)$.

Proof. Straightforward by examining the code. □

4.2.2 The `Top` and `Bottom` Variables

The `Top` and `Bottom` shared variables store information about C_T and C_B, respectively, and they are both of a CASable size. The `Top` variable also contains an unbounded Tag value, to avoid the ABA problem as we describe in Section 4.2.3. The structure of `Top` and `Bottom` variables is detailed in Figure 3 on page 7.

In practice, in order to store all the information on a CASable word size even if only a 32-bit CAS operation is available, we represent the node's pointer by its *offset* from some base address given by the nodes' memory manager. In this case, if the size of the node is of a power of two, we can even save only the offsets to C_T and C_B, and calculate the offsets of N_T and N_B by simple bitwise operations. That way we save the space used by the `cellIndex` variable, and leave enough space for the tag value.

In the rest of the proof we use the *Cell* operator to denote the cell to which a variable of type `BottomStruct` or `TopStruct` points (for example, $Cell(\texttt{Top}) = C_T$ and $Cell(\texttt{Bottom}) = C_B$):

Definition 3. *If* `TorBVal` *is a variable of type* `TopStruct` *or* `BottomStruct` *then:* Cell(TorBVal) = TorBVal.nodeP → itsDataArr[TorBVal.cellIndex].

Note that the *Cell* operator points to a real array cell only if the *cellIndex* field indicates a valid cell index. The following invariant states that this is true for all values of variables of type `TopStruct` or `BottomStruct` used in the algorithm.

Invariant 4. *For any variable* V *of type* `TopStruct` *or* `BottomStruct` *that is used by the implementation, we have:* $0 \leq V.cellIndex < DequeNode :: ArraySize$.

Proof. Straightforward by examining all statements in the code that modify variables of type `TopStruct` or `BottomStruct`. □

Our implementation uses the `EncodeTop` and `EncodeBottom` macros to construct values of type `TopStruct` and `BottomStruct`, respectively, and similarly uses the `DecodeBottom` and `DecodeTop` macros to extract the components from values of these types. For convenience, we use processes' private variables to hold the different fields of values read from `Top` and `Bottom`. For example, after executing the code segment:

```
oldBotVal = Bottom;
<oldBotNode,oldBotIndex> = DecodeBottom(oldBotVal);
```

using `oldBotNode` and `oldBotIndex`, as long as they are not modified, is equivalent to using `oldBotVal.nodeP` and `oldBotVal.cellIndex`, respectively. The following invariant formally states these equivalences:

Invariant 5.

1. $p@\langle 2 \dots 7 \rangle \Rightarrow (p.currNode = \texttt{Bottom}.nodeP \wedge p.currIndex = \texttt{Bottom}.cellIndex)$.
2. $p@\langle 9 \dots 20 \rangle \Rightarrow (p.currTopTag = p.currTop.tag \wedge$
   $p.currTopNode = p.currTop.nodeP \wedge p.currTopIndex = p.currTop.cellIndex)$.
3. $p@\langle 16 \dots 20 \rangle \Rightarrow (p.newTopTag = p.newTopVal.tag \wedge$
   $p.newTopNode = p.newTopVal.nodeP \wedge$
   $p.newTopIndex = p.newTopVal.cellIndex)$.
4. $p@\langle 22 \dots 39 \rangle \Rightarrow (p.oldBotNode = p.oldBotVal.nodeP \wedge$
   $p.oldBotIndex = p.oldBotVal.cellIndex)$.
5. $p@\langle 25 \dots 39 \rangle \Rightarrow (p.newBotNode = p.newBotVal.nodeP \wedge$
   $p.newBotIndex = p.newBotVal.cellIndex)$.
6. $p@\langle 27 \dots 39 \rangle \Rightarrow (p.currTopTag = p.currTop.tag \wedge$
   $p.currTopNode = p.currTop.nodeP \wedge p.currTopIndex = p.currTop.cellIndex)$.

Proof. The invariant is immediately derived from Invariant 1, Lemma 2 and examination of the code. □

Lemma 2 and Invariants 4 and 5 are used very frequently in the proof. For brevity, we often use these invariants implicitly.
The following invariant describes the values of `Bottom` in terms of the private variables:

Invariant 6.

1. $p@\langle 2 \dots 7 \rangle \Rightarrow (N_B = p.currNode \wedge C_B = C_{p.currIndex} \in N_B)$.
2. $p@\langle 22 \dots 25, 30, 37 \rangle \Rightarrow \texttt{Bottom} = p.oldBotVal$.
3. $p@\langle 26 \dots 29, 31 \dots 36, 38 \dots 39 \rangle \Rightarrow \texttt{Bottom} = p.newBotVal$.

Proof. Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s.

- If a is an execution of Line 1, 21, 25, 29 or 36: Then Invariant 5 and a simple examination of the code implies that the invariant holds in s'.
- Otherwise, by Invariant 1:
  $p@\langle 1 \dots 7, 21 \dots 39\rangle \Rightarrow \forall p' \neq p \ \neg p'@\langle 1 \dots 7, 21 \dots 39\rangle$. Also,
  $\neg s.p@\langle 1, 21, 25, 29, 36\rangle \wedge s'.p@\langle 2 \dots 7, 22 \dots 25, 30, 37, 26 \dots 29, 31 \dots 36, 38 \dots 39\rangle$
  implies that $s'.\texttt{Bottom} = s.\texttt{Bottom}$, and by Lemma 2 the transition does not modify any of $p.currNode$, $p.newBotVal$ or $p.oldBotVal$ variables either. Therefore a does not falsify the invariant.

□

4.2.3 The ABA Problem

Our implementation uses the CAS operation for all updates of the `Top` variable. The CAS synchronization primitive is susceptible to the *ABA problem*: Assume that the value A is read from some variable v, and later a CAS operation is done on that variable, with the value A supplied as the old-value parameter of the CAS. If, between the read and the CAS, the variable v has been changed to some other value B and then to A again, the CAS would still succeed.

In this section, we prove some properties concerning mechanisms used in the algorithm to avoid the ABA problem. We start by defining an order between different `Top` values:

Definition 7. *Let $TopV_1$ and $TopV_2$ be two values of type `TopStruct`. $TopV_1 \ll TopV_2$ if and only if:*

1. $TopV_1.tag \leq TopV_2.tag$, *and*
2. $(TopV_1.tag = TopV_2.tag) \Rightarrow (TopV_1.cellIndex > TopV_2.cellIndex)$.

Clearly if two `Top` values are equal $TopV_1 = TopV_2$, then neither $TopV_1 \ll TopV_2$ nor $TopV_2 \ll TopV_1$. The following claim shows that the $\ll$ operator is transitive:

Claim 8. $TopV_1 \ll TopV_2 \ll TopV_3 \Rightarrow TopV_1 \ll TopV_3$.

Proof. By the definition of $\ll$ we have: $TopV_1 \ll TopV_2 \ll TopV_3 \Rightarrow TopV_1.tag \leq TopV_3.tag$. If $TopV_1.tag = TopV_3.tag$, then: $TopV_1.tag = TopV_2.tag = TopV_3.tag$, which implies $TopV_1.cellIndex > TopV_2.cellIndex > TopV_3.cellIndex$, and therefore $TopV_1 \ll TopV_3$. □

Invariant 9.

1. *If $p@\langle 16 \dots 17\rangle$ then $p.currTop \ll p.newTopVal$.*

2. *If* $p@33$ *then* $p.currTop \ll p.newTopVal$.

Proof. Initially $p@0$ so the invariant holds. Since no statement modifies $p.currTop$ or $p.newTopVal$ while $p@\langle 16 \ldots 17, 33\rangle$ holds, the only transition a that might falsify the invariant is:

1. An execution of Line 13: In this case $s'.p@16$, $s'.p.newTopVal.tag = s'.p.currTop.tag$ and $s'.p.newTopVal.cellIndex < s'.p.currTop.cellIndex$ so the invariant holds.
2. An execution of Line 15: In this case $s'.p@16$, $s'.p.newTopVal.tag > s'.p.currTop.tag$ so the invariant holds.
3. An execution of Line 32: In this case $s'.p@33$, $s'.p.newTopVal.tag > s'.p.currTop.tag$ so the invariant holds.

□

Lemma 10. *Let* $s \xrightarrow{a} s'$ *be a step of the algorithm, and suppose* a *writes a value to* `Top`. *Then* $s.\texttt{Top} \ll s'.\texttt{Top}$.

Proof. Only statements p.17 and p.33 for some process p may write a value to the `Top` variable. In both cases, $s'.\texttt{Top} = s.p.newTopVal$ if and only if $s.\texttt{Top} = s.p.currTop$. By Invariant 9 $s.\texttt{Top} \ll s'.\texttt{Top}$. □

Invariant 11. $p@\langle 9 \ldots 20, 27 \ldots 39\rangle \Rightarrow (p.currTop = \texttt{Top} \vee p.currTop \ll \texttt{Top})$

Proof. Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$. The only statements which may establish the antecedent are $p.8$ and $p.26$. In both cases, the statement reads `Top`, and therefore $s'.p.currTop = \texttt{Top}$, so the consequent holds.

Since no statement modifies $p.currTop$ while $p@\langle 9 \ldots 20, 27 \ldots 39\rangle$ holds, the only transition that might falsify the consequent while the antecedent holds is one that modifies `Top`. In this case, by Lemma 10 $s.\texttt{Top} \ll s'.\texttt{Top}$, and by the transitive property of the $\ll$ operator (Claim 8), $p.currTop \ll s'.\texttt{Top}$. □

Corollary 12. *Consider a transition* $s \xrightarrow{a} s'$ *where* a *writes a value to* `Top`. *Then:* $\forall p\ s.p@\langle 9 \ldots 20, 27 \ldots 39\rangle \Rightarrow s'.p.currTop \neq s'.\texttt{Top}$

Proof. Straightforward from Invariant 11, Lemma 10 and Claim 8. □

4.2.4 Memory Management

Our algorithm uses an external linearizable *shared pool* module, which stores the available list nodes. The shared pool module supports two operations: `AllocateNode` and `FreeNode`. The details of the shared pool implementation are not relevant to our algorithm, so we simply model a linearizable shared pool that supports atomic `AllocateNode` and `FreeNode` operations.

We model the shared pool using an auxiliary variable `Live`, which models the set of nodes that have been allocated from the pool and not yet freed:

1. Initially `Live` $= \emptyset$.
2. An `AllocateNode` operation atomically adds a node that is not in `Live` to `Live` and returns that node.
3. A `FreeNode`(N) operation with $N \in$ `Live` atomically removes N from `Live`.
4. While $N \in$ `Live`, the shared pool implementation does not modify any of N's fields.

The shared pool behaves according to the above rules provided our algorithm uses it properly. The following conjecture states the rules for proper use of the shared pool. We prove that the conjecture holds in Section 4.4.

Conjecture 30. Consider a transition $s \xrightarrow{a} s'$ of our algorithm.

- If $N \notin s.$`Live`, then a does not modify any of N's fields.
- If a is an execution of `FreeNode`(N), then $N \in s.$`Live`.

Definition 13. *A node N is* live *if and only if $N \in$ `Live`.*

4.3 Ordered Nodes

We introduce an auxiliary variable `Ordered`, which consists of a sequence of nodes. We regard the order of the nodes in `Ordered` as going from *left* to *right*. Formally, the variable `Ordered` supports four operations: *AddLeft*, *AddRight*, *RemoveLeft* and *RemoveRight*. If $|$`Ordered`$| = l$, `Ordered` $= \{N_1, \ldots, N_l\}$ then:

- N_1 is the leftmost node and N_l is the rightmost one.
- An `Ordered`.AddLeft(N) operation results in `Ordered` $= \{N, N_1, \ldots, N_l\}$.
- An `Ordered`.AddRight(N) operation results in `Ordered` $= \{N_1, \ldots, N_l, N\}$.
- A `Ordered`.RemoveLeft() operation results in `Ordered` $= \{N_2, \ldots, N_l\}$, and returns N_1.
- A `Ordered`.RemoveRight() operation results in `Ordered` $= \{N_1, \ldots, N_{l-1}\}$, and returns N_l.

Definition 14. *A node N is* ordered *if and only if $N \in$ `Ordered`.*

The following conjecture describes the basic properties of the nodes in `Ordered`:

Conjecture 55. Let $|$`Ordered`$| = n + 2$, `Ordered` $= \{N_0, \ldots, N_{n+1}\}$. Then:

1. $\forall_{0 \le i \le n}\ N_i \rightarrow next = N_{i+1} \wedge N_{i+1} \rightarrow prev = N_i$.
2. Exactly one of the following holds:

(a) $n \geq 0,\ N_0 = N_B, N_n = N_T$.

(b) $n > 0,\ N_1 = N_B, N_n = N_T$.

(c) $n = 0,\ N_0 = N_T, N_1 = N_B$.

Corollary 15.

1. $|s.\texttt{Ordered}| \geq 2$.
2. *N_T is ordered and is the second node from the right in* `Ordered`.
3. *N_B is ordered and is either the first or the second node from the left in* `Ordered`.
4. *$N_T \rightarrow next$ is* `Ordered`.

Proof. Straightforward from Conjecture 55. □

The proof of Conjecture 55 is given in section 4.6.The following invariants and lemmas state different properties of the nodes in `Ordered`.

Invariant 16. *Exactly one of the following holds:*

1. *N_B is the leftmost node in* `Ordered` $\wedge$ $(p@\langle 26 \ldots 34, 36 \ldots 38\rangle \Rightarrow p.oldBotNode = N_B)$.
2. $\exists p$ *such that* $p@\langle 26 \ldots 29, 31 \ldots 34, 36, 38\rangle \wedge N_B \neq p.oldBotNode \wedge$ *$p.oldBotNode$ is the leftmost node in* `Ordered`.

Proof. Initially $\forall p, p@0$ holds and N_B is the leftmost node in `Ordered`, as depicted in Figure 8 on page 15, so the invariant holds. Consider a transition $s \xrightarrow{a} s'$ that falsifies the invariant.

Since no statement modifies $p.oldBotNode$ while $p@\langle 26 \ldots 34, 36 \ldots 38\rangle$ holds, we need only consider statements that might modify N_B or `Ordered`, statements that might establish $p@\langle 26 \ldots 34, 36 \ldots 38\rangle$ while N_B is the leftmost node in `Ordered`, and statements that might falsify $p@\langle 26 \ldots 29, 31 \ldots 34, 36, 38\rangle$ for some process p while N_B is not the leftmost node in `Ordered`. Therefore a is either:

1. An execution of line $p'.7$ for some process p': By Invariant 1, $\forall p \neq p'$ $\neg s.p@\langle 26 \ldots 29, 31 \ldots 34, 36, 38\rangle$ and therefore, since the invariant holds in s, $s.N_B$ is the leftmost node in $s.\texttt{Ordered}$. By Invariant 6 $p'.currNode = s.N_B$. If $p'.newNode = p'.currNode$ then $s'.N_B = p'.newNode = p'.currNode = s.N_B$ is the leftmost node in $s.\texttt{Ordered}$. Otherwise, $p'.newNode$ is added to `Ordered` by the AddLeft operation and therefore $s'.N_B = p'.newNode$ is the leftmost node in $s'.\texttt{Ordered}$. Finally, since a is the execution of Line 7, by Invariant 1 $\forall p\ \neg s'.p@\langle 26 \ldots 34, 36 \ldots 38\rangle$, and therefore the invariant holds in s'.

2. An execution of line $p.25$: By Invariant 1,
   $\forall p' \neq p \neg s.p'@\langle 26 \dots 29, 31 \dots 34, 36, 38 \rangle$ and therefore, since the invariant holds in s, $s.N_B$ is the leftmost node in $s.\texttt{Ordered}$. Also note that $s'.p.oldBotNode = s.p.oldBotNode$, and by Invariant 6 $p.oldBotNode = s.N_B$.

   If $p.newBotNode = p.oldBotNode$, then $s'.N_B = p.newBotNode = s.N_B$ is the leftmost node in `Ordered`, and the invariant holds in s'. Otherwise, $s'.p@26$ and $s'.N_B = p.newBotNode \neq p.oldBotNode$, and $p.oldBotNode = s.N_B$ is the leftmost node in $s'.\texttt{Ordered}$, since the transition does not modify `Ordered`.

3. An execution of p.29 or p.36: Note that $s.\texttt{Ordered} = s'.\texttt{Ordered}$, $s.p@\langle 26 \dots 29, 31 \dots 34, 36, 38 \rangle$, and $p.oldBotNode$ is not modified by a. Since the invariant holds in s then $p.oldBotNode$ is the leftmost node in `Ordered`. Since $s'.N_B = p.oldBotNode$ it follows that $s'.N_B$ is the leftmost node in `Ordered`, so the invariant holds in s'.

4. An execution of line $p.34$ or line $p.38$: By Invariant 1 $\forall p' \neg s'.p'@\langle 26 \dots 34, 36 \dots 38 \rangle$, and therefore it suffices to show that $s'.N_B$ is the leftmost node in $s'.\texttt{Ordered}$.
   By Invariant 6 $p.newBotNode = s.N_B$. Since $s.p@\langle 26 \dots 29, 31 \dots 34, 36, 38 \rangle$ and the invariant holds in s, it follows by Invariant 1 that either:

   - $p.oldBotNode \neq s.N_B$ and $p.oldBotNode$ is the leftmost node in `Ordered`: Then $p.oldBotNode \neq p.newBotNode$, and therefore the leftmost node is removed from `Ordered`. By Corollary 15 N_B is the second node from the left in $s.\texttt{Ordered}$ (since it is not the leftmost one), and therefore the leftmost node in $s'.\texttt{Ordered}$.
   - $p.oldBotNode = s.N_B = p.newBotNode$ is the leftmost node in `Ordered`. In this case, the transition does not modify `Ordered` or N_B, and therefore $s'.N_B = s.N_B$ is the leftmost node in $s'.\texttt{Ordered}$.

5. An execution of p'.17 for some process p': In this case a can falsify the invariant only by modifying `Ordered`. However, since a may only remove the rightmost node from `Ordered`, and by Corollary 15 $|s.\texttt{Ordered}| \geq 2$, a does not change the leftmost node in `Ordered`, and therefore does not falsify the invariant.

□

Conjecture 29. All ordered nodes are live.

Corollary 17. $N \in \texttt{Ordered} \Rightarrow N \neq \texttt{NULL}$.

Proof. By Conjecture 29, $N \in \texttt{Ordered} \Rightarrow N \in Live$. By the properties of the shared pool stated in Section 4.2.4, the only module that modifies $Live$ is the shared pool module, which adds to $Live$ the new allocated nodes. Therefore NULL is never being added to $Live$, which implies that $N \neq \texttt{NULL}$. □

We now prove various properties about the ordered nodes, which we use later to prove Conjecture 29. The proof of Conjecture 29 appears in Section 4.3.1.

The following lemma states that no node becomes ordered while the owner process executes any statement but Line 7:

Lemma 18. *Consider a transition* $s \xrightarrow{a} s'$, *where* $s.p@\langle 1 \dots 7, 21 \dots 39\rangle \wedge s'.p@\langle 1 \dots 7, 21 \dots 39\rangle$, *then* $n \in s'.\texttt{Ordered} \rightarrow n \in s.\texttt{Ordered}$.

Proof. By examining the code, only statement p'.7 for some process p' adds nodes to Ordered. For any process $p' \neq p$, by Invariant 1 $s.p@\langle 1 \dots 7, 21 \dots 39\rangle \Rightarrow \neg s.p'@7$, and therefore a cannot be an execution of p'.7. Finally, if a is an execution of p.7 then $\neg s'.p@\langle 1 \dots 7, 21 \dots 39\rangle$. □

Invariant 19. *If* $p@\langle 5 \dots 7\rangle$, *then* $(p.newNode \notin \texttt{Ordered}) \vee (p@7 \wedge p.newNode = N_B)$.

Proof. Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$ and suppose the invariant holds in s. Only statements p@3 and p@4 can establish the antecedent:

- If a is an execution of Line 3, then $s'.p@7$, and by Invariant 6 $s'.N_B = s.p.currNode = s'.p.newNode$.
- If a is an execution of Line 4 by process p, then the node returned by the AllocateNode was not live in s: $s'.p.newNode \notin s.\texttt{Live}$ and therefore by Conjecture 29 $s'.p.newNode \notin s.\texttt{Ordered}$. Since a does not add any node to Ordered, $s'.\texttt{Ordered} = s.\texttt{Ordered}$ and the invariant holds.

It remains to show that no statement falsifies the consequent while the antecedent holds. No statement modifies $p.newNode$ while the antecedent holds, and by Invariant 1 no statement adds a node to Ordered or modifies N_B while the antecedent holds. Therefore no statement falsifies the consequent while the antecedent holds. □

Invariant 20. *Suppose* $\texttt{Ordered} = \{N_0, \dots N_{n+1}\}$. *Then* $\forall_{0 \leq i,j \leq n+1}$, $i \neq j \Rightarrow N_i \neq N_j$.

Proof. Initially the deque is constructed with $\texttt{Ordered} = \{nodeA, nodeB\}$, and $nodeA \neq nodeB$, so the invariant holds. The only statement that may falsify the invariant is one that adds a node to Ordered which is already there. By

Lemma 18, the only transition $s \xrightarrow{a} s'$ that may add a node to `Ordered` is an execution of Line 7. By examining the code, Line 7 adds $s.p.newNode$ to `Ordered` if and only if $s.p.newNode \neq s.p.currNode$, and by Invariant 6 $s.p.currNode = s.N_B$. Therefore by Invariant 19: $s.p.newNode \notin s.\texttt{Ordered}$, which implies that a does not falsify the invariant. □

The following lemma states that the next and previous pointers of nodes are not modified while the nodes are ordered.

Lemma 21. *If $N \in s.\texttt{Ordered}$ and $s \xrightarrow{a} s'$ then:*

1. $s'.N \rightarrow next = s.N \rightarrow next$.
2. *If $(s'.N \rightarrow prev \neq s.N \rightarrow prev)$, then $N = N_B$ and it is the leftmost node in `Ordered`.*

Proof. By Conjecture 29 $N \in \texttt{Ordered} \Rightarrow N \in \texttt{Live}$, and therefore only executions of the deque methods' statements may update the *prev* or *next* fields. By examining the code, the only statement that updates a node's *next* field is Line 5. By Invariant 19 $p.newNode$ is not ordered when being updated.

As for the *prev* field, the only statement that updates a node's *prev* field is Line 6. By Invariant 6 $p@6$ implies $p.currNode = N_B$, and by Invariant 16 it is the leftmost node in `Ordered`. □

Invariant 22. *If $p@\langle 15 \dots 17\rangle \wedge p.nodeToFree \neq \texttt{NULL} \wedge p.currTop == \texttt{Top}$ then $p.nodeToFree \in \texttt{Ordered}$, it is the rightmost node there, and $p.nodeToFree = N_T \rightarrow next$.*

Proof. Note that it is enough to show that $p.nodeToFree = N_T \rightarrow next$ since by Corollary 15 it immediately implies that $p.nodeToFree \in \texttt{Ordered}$ and that it is the rightmost node there.

Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s.

- If a modifies `Top`: No statement in $\langle 15 \dots 17\rangle$ modifies `Top`, and if a modification of `Top` is done by some process p, $\neg s'.p@\langle 15 \dots 17\rangle$. If some process p' modifies `Top` while the antecedent holds, then by Corollary 12 $s'.p.currTop \neq s'.\texttt{Top}$, so the antecedent does not hold in s'. Therefore a does not falsify the invariant.
- Otherwise, we consider statements that may establish the antecedent. Because $p.nodeToFree$ is not modified while $p@\langle 15 \dots 17\rangle$ holds, then the only statements that may establish the antecedent are $p.13$ and $p.14$.
  1. If a is an execution of p.13: By examining the code, $s'.p.nodeToFree =$ `NULL`, so the invariant holds in s'.

2. If a is an execution of p.14: By examining the code, $s'.p.nodeToFree = p.currTopNode \rightarrow next$, and since a does not modify $p.currTop$ or `Top`, then: $(p.currTopNode = N_T) \vee (s'.p.currTop \neq \texttt{Top})$ holds, which implies that the invariant holds in s'.

- Otherwise, we consider statements that might falsify the consequent while the antecedent holds. Since a does not modify `Top`, $s'.N_T = s.N_T$. Since no statement modifies $p.nodeToFree$ and $p.currTop$ while $p@\langle 15 \dots 17 \rangle$ holds, we only need to consider statements that modifies $N_T \rightarrow next$. But by Corollary 15 $N_T \in s.\texttt{Ordered}$, and therefore by Lemma 21 $s'.N_T \rightarrow next = s.N_T \rightarrow next$. Therefore no statement falsifies the consequent while the antecedent holds.

□

Lemma 23. *If $s \xrightarrow{a} s'$ and a is a successful CAS operation at Line 17 by some process p, then $s'.p.nodeToFree \notin s'.\texttt{Ordered}$.*

Proof. Note that the statement at Line 17 does not modify $p.nodeToFree$, and that it removes the rightmost node in `Ordered` if and only if $p.nodeToFree \neq$ `NULL`. If $p.nodeToFree =$ `NULL`, then by Corollary 17 $p.nodeToFree \notin s'.\texttt{Ordered}$. Otherwise, since the CAS is successful $s.p.currTop = s.\texttt{Top}$, and by Invariant 22 $p.nodeToFree \in s.\texttt{Ordered}$ and it is the rightmost node there. Therefore after the execution of p.17, which removes the rightmost node from `Ordered`, by Invariant 20 we get that $p.nodeToFree \notin s'.\texttt{Ordered}$. □

Invariant 24. *If $p@\langle 22 \dots 34, 36 \dots 38 \rangle$ then $p.oldBotNode \in \texttt{Ordered}$, and it is the leftmost node there.*

Proof. Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$that falsifies the invariant. Note that $p.oldBotNode$ is not modified while $p@\langle 22 \dots 34, 36 \dots 38 \rangle$, and that by Lemma 18 no node is added to `Ordered` while the antecedent holds. Therefore the only statements we need to consider are p.21 which establishes the antecedent, and any statement which removes a node from `Ordered` while the antecedent holds.

1. If a is an execution of p.21: Then $s'.oldBotNode = s.N_B$ and therefore by Conjecture 55, $s'.oldBotNode \in \texttt{Ordered}$. Because $s.p@21$, by Invariant 1 $\forall p' \neq p \ \neg p'@\langle 20 \dots 39 \rangle$, and therefore by Invariant 16 $s.N_B$ is the leftmost node in `Ordered`, so the invariant holds in s'.

2. If a is a removal of a node from `Ordered` while the antecedent holds: Since the antecedent holds in s and s', then $p.34$ and $p.38$ are not enabled, and by Invariant 1 $\forall p' \neq p$ $p'.34$ and $p'.38$ are not enabled either.

   Therefore a must be an execution of a *RemoveRight* operation by statement p'.17 for some process $p' \neq p$. Since the invariant and the antecedent holds in s, then $p.oldBotNode$ is the leftmost node in $s.\texttt{Ordered}$,

and by Corollary 15 $|s.\texttt{Ordered}| \geq 2$ which implies that $p.oldBotNode \in s'.\texttt{Ordered}$ (since a *RemoveRight* operation cannot remove the leftmost node if there is more than one node in `Ordered`).

□

Conjecture 27. If $p@18 \wedge p.nodeToFree \neq \texttt{NULL}$ then $p.nodeToFree \in \texttt{Live} \wedge p.nodeToFree \notin \texttt{Ordered}$.

Invariant 25. *If* $p@\langle 5 \ldots 7\rangle \wedge p'@18 \wedge p' \neq p \wedge p'.nodeToFree \neq \texttt{NULL}$ *then* $p.newNode \neq p'.nodeToFree$.

Proof. Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$ that falsifies the invariant. Because $p.newNode$ and $p'.nodeToFree$ are not modified while $p@\langle 5 \ldots 7\rangle \wedge p'@18$ holds, then it suffices to consider statements $p.3$, $p.4$ and $p'.17$.

1. If a is an execution of $p.3$: By Invariant 6, $s'.p.newNode = N_B$ in this case, which implies by Corollary 15 that $s'.p.newNode \in \texttt{Ordered}$. If $p'.nodeToFree = \texttt{NULL} \vee \neg p'@18$ then the antecedent does not hold in s'. Otherwise, by Conjecture 27, $p'.nodeToFree \notin \texttt{Ordered}$ and therefore $p'.nodeToFree \neq s'.p.newNode$.

2. If a is an execution of $p.4$: Since a is an execution of `AllocateNode` operation, then $s'.p.newNode \notin s.\texttt{Live}$. If $p'.nodeToFree = \texttt{NULL} \vee \neg p'@18$ then the antecedent does not hold in s'. Otherwise by Conjecture 27 $p'.nodeToFree \in s.\texttt{Live}$ and therefore $p'.nodeToFree \neq s'.p.newNode$.

3. If a is an execution of $p'.17$: In this case a does not modify $p'.nodeToFree$ or $p.newNode$. If $p'.nodeToFree = \texttt{NULL}$ then the antecedent does not hold in s', and if $p'.currTop \neq s.\texttt{Top}$, then the CAS fails and the antecedent does not hold in state s'. Otherwise, by Invariant 22 $p'.nodeToFree \in s.\texttt{Ordered}$ and by Lemma 23 $p'.nodeToFree \notin s'.\texttt{Ordered}$.

   - If $s.p@\langle 5 \ldots 7\rangle$ then by Invariant 19 $p.newNode \notin s.\texttt{Ordered} \vee p.newNode = N_B$. If $p.newNode \notin s.\texttt{Ordered}$, since $p'.nodeToFree \in s.\texttt{Ordered}$ then $p.newNode \neq p'.nodeToFree$. If $p.newNode = N_B$, then by Conjecture 55 $p.newNode \in s'.\texttt{Ordered}$, and since $p'.nodeToFree \notin s'.\texttt{Ordered}$ then $p.newNode \neq p'.nodeToFree$.
   - Otherwise the antecedent does not hold in state s'.

□

Invariant 26. *If* $p@18 \wedge p'@18$, *then* $(p'.nodeToFree \neq p.nodeToFree) \vee (p.nodeToFree = p'.nodeToFree = \texttt{NULL})$.

Proof. Initially p@0 so the invariant holds. Consider a transition $s \xrightarrow{a} s'$ that falsifies the invariant. Then:

1. If $s.p@18 \wedge s.p'@18$ then $p.nodeToFree$ and $p'.nodeToFree$ are not modified by a, and therefore a cannot falsify the invariant.

2. If $\neg s.p@18 \wedge \neg s.p'@18$, then $\neg s'.p@18 \vee \neg s'.p'@18$ and therefore the antecedent does not hold in s', and the invariant is not falsified by a.

3. Otherwise, a is a successful execution of the CAS statement at Line 17. W.l.o.g we assume that the statement is executed by p', and therefore $s.p@18$. Note that a does not modify $p.NodeToFree$ or $p'.NodeToFree$. If $p.nodeToFree = \texttt{NULL} \vee p'.nodeToFree = \texttt{NULL}$ then the invariant clearly holds. Otherwise, by Conjecture 27 we get $p.nodeToFree \notin s.\texttt{Ordered}$ and by Invariant 22 we get $p'.nodeToFree \in s.\texttt{Ordered}$, and therefore $p.nodeToFree \neq p'.nodeToFree$.

□

Using Invariants 25 and 26 we now prove Conjecture 27:

Conjecture 27. *If* $p@18 \wedge p.nodeToFree \neq \texttt{NULL}$ *then* $p.nodeToFree \in \texttt{Live} \wedge p.nodeToFree \notin \texttt{Ordered}$.

Proof. Initially p@0 so the invariant holds. Consider a transition $s \xrightarrow{a} s'$ that falsifies the invariant. Since $p.nodeToFree$ is not modified while p@18 holds, the only statement that might establish the antecedent is $p.17$.

If a is an execution of $p.17$, then a does not modify $p.nodeToFree$ or `Live`, and a establishes the antecedent only if $p.nodeToFree \neq NULL \wedge p.currTop = s.\texttt{Top}$. By Lemma 23 $p.nodeToFree \notin s'.\texttt{Ordered}$ holds in this case. Also, by Invariant 22 $p.nodeToFree \in s.\texttt{Ordered}$ which by Conjecture 29 implies that $p.nodeToFree \in s.\texttt{Live} = s'.\texttt{Live}$. Therefore the invariant holds in s'.

The only statements that might falsify the consequent while the antecedent holds are $p'.18$, $p'.34$, $p'.38$ and $p'.7$ for some process $p' \neq p$ (that is, deallocation of a node or addition of a node to `Ordered`).

1. If a is an execution of $p'.7$: Then a does not modify $p.nodeToFree$ or `Live`. Since the antecedent holds in s, Invariant 25 implies that $s.p'.newNode \neq p.nodeToFree$. Therefore, since the only node that may be added to `Ordered` by a is $s.p'.newNode$, a does not falsify the consequent.

2. If a is an execution of $p'.18$: Then a does not modify $p.nodeToFree$ or `Ordered`. Since the antecedent holds in s, Invariant 26 implies that $s.p'.nodeToFree = NULL \vee s.p'.nodeToFree \neq p.nodeToFree$. Since a can only deallocate $s.p'.nodeToFree$ and does so only if $s.p'.nodeToFree \neq NULL$, it follows that a does not falsify the invariant.

3. If a is an execution of $p'.34$ or $p'.38$: Then a does not modify $p.nodeToFree$. Since the consequent holds in s, and Invariant 24 implies that $s.p'.oldBotNode \in s.\texttt{Ordered}$, it follows that that $s.p'.oldBotNode \neq p.nodeToFree$. Since a can only deallocate $s.p'.oldBotNode$, it does not falsify the invariant.

□

Invariant 28. *If* $p@\langle 5 \dots 7 \rangle$*, then* $p.newNode \in \texttt{Live}$.

Proof. Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose $s \xrightarrow{a} s'$ falsifies the invariant. There are three cases to consider:

- If a is an execution of p.4, then the node returned by the `AllocateNode` is guaranteed to be live, that is: $s'.p.newNode \in s'.\texttt{Live}$.
- Otherwise, if a is an execution of p.3, then $s'.\texttt{Live} = s.\texttt{Live}$, and by Invariant 6 $s'.p.newNode = s.p.currNode = N_B \in \texttt{Live}$.
- Otherwise a must falsify the consequence while the antecedent holds. Since no statement modifies $p.newNode$ while $p@\langle 5 \dots 7 \rangle$ holds, then a must deallocate a node. By Invariant 1 the only enabled statement that might do that is p'.18 for some process $p' \neq p$. If $s.p'.nodeToFree = \texttt{NULL}$ then no node is deallocated. Otherwise, by Invariant 25 we get $p.newNode \neq s.p'.nodeToFree$ and therefore $p.newNode \in s'.\texttt{Live}$.

□

4.3.1 Proof of Conjecture 29

Using the above invariants, we can now give the proof of Conjecture 29.

Conjecture 29. *All ordered nodes are live.*

Proof. Initially there are two live nodes in `Ordered` (deque constructor pseudo code, depicted in Figure 8 on page 15), so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s, that is: $s.\texttt{Ordered} \subseteq s.\texttt{Live}$. Clearly, the only operations that may falsify the invariant are deallocation of a node, or addition of a node to `Ordered`. Therefore, there are three statements to consider:

1. p.7 for some process p: Then $s'.\texttt{Live} = s.\texttt{Live}$ and by Invariant 28 $p.newNode \in s.\texttt{Live}$. Therefore the invariant still holds in s'.
2. p.18 for some process p: Then a deallocates $p.nodeToFree$ if and only if $p.nodeToFree \neq \texttt{NULL}$. By Conjecture 27 $p.nodeToFree \neq \texttt{NULL} \Rightarrow p.nodeToFree \notin s.\texttt{Ordered}$, and since $s'.\texttt{Ordered} = s.\texttt{Ordered}$, the invariant still holds in s'.

3. p.34 or p.38 for some process p: In this case, the statement deallocates $p.oldBotNode$ if and only if it also removes the leftmost node from s.`Ordered`. By Invariant 24, $p.oldBotNode \in s.\texttt{Ordered}$ and it is the leftmost node there, and therefore $p.oldBotNode \notin s'.\texttt{Live} \Rightarrow p.oldBotNode \notin s'.\texttt{Ordered}$.

□

4.4 Legality of Shared Pool Usage

In this section we show that our algorithm uses the shared pool properly, as stated by Conjecture 30.

Conjecture 30. *Consider a transition $s \xrightarrow{a} s'$.*

- *If $N \notin s.\texttt{Live}$, then a does not modify any of N's fields.*
- *If a is an execution of `FreeNode`(N), then $N \in s.\texttt{Live}$.*

Proof. We first show that a `FreeNode` operation is always called on a live node. Suppose a is a `FreeNode`(N) operation. The only statements which call the `FreeNode` operation are $p.18$, $p.34$ and $p.38$ for some process p. By Conjecture 27, if a is an execution of $p.18$ then $s.p.nodeToFree \in s.\texttt{Live}$, and therefore a does not falsify the invariant. Otherwise if a is an execution of $p.34$ or $p.38$, then by Invariant 24 $s.p.oldBotNode \in s.\texttt{Ordered}$, and therefore by Conjecture 29 $s.p.oldBotNode \in s.\texttt{Live}$, and therefore a does not falsify the invariant.

Next we show that a does not modify a field of node N if $N \notin s.\texttt{Live}$. The only statements that might modify node's fields are $p.2$, $p.5$ and $p.6$ for some process p. By Invariant 28 $p@5$ implies that $s.p.newNode \in s.\texttt{Live}$, and by Invariant 6 $s.p@\langle 2, 6\rangle$ implies that $s.p.currNode = s.N_B$. By Corollary 15 $s.N_B \in s.\texttt{Ordered}$ which implies by Conjecture 29 that $s.N_B \in s.\texttt{Live}$, and therefore a does not falsify the invariant. □

4.5 Order On Cells

Section 4.3 introduced the `Ordered` sequence, which defines an order between a subset of these nodes. This section defines an order between the cells of these nodes, and proves some properties regarding this order.

Definition 31. *For a node $N \in$ `Ordered`, $Pos(\texttt{Ordered}, N)$ denotes the index of N in `Ordered`, where the leftmost node in `Ordered` is indexed as 0. (Note that by Invariant 20, $Pos(\texttt{Ordered}, N)$ is well defined.)*

Definition 32. *For two nodes M and N, and two cells $C_i \in M$ and $C_j \in N$, we define the* order $\prec_s$ *between these cells to be the lexicographic order $\langle Node, CellIndex\rangle$, where the nodes are ordered by the `Ordered` series, and the indices by the whole numbers order. Formally, $C_i \prec_s C_j$ if and only if $M \in s.\texttt{Ordered} \wedge N \in s.\texttt{Ordered} \wedge (Pos(s.\texttt{Ordered}, M) < Pos(s.\texttt{Ordered}, N) \vee (M = N \wedge i < j))$.*

Note that the $\prec_s$ operator depends on the state s, since it depends on the membership and order of the nodes in `Ordered`. The following lemma implies that the order between two cells cannot be changed unless the node of one of the cells is removed or added to `Ordered`:

Lemma 33. *For any step of the algorithm $s \xrightarrow{a} s'$: $(C_i \prec_s C_j) \Rightarrow \neg(C_j \prec_{s'} C_i)$.*

Proof. If C_i and C_j belongs to the same node this is obvious, since the order of cells inside a node is never changed. Otherwise, suppose $C_i \in N \wedge C_j \in M \wedge (N \neq M)$. The only way the order between C_i and C_j can be changed is if the order between N and M in `Ordered` is changed. Since the `Ordered` series only supports addition and removal of nodes (and does not support any swap operation), the order of nodes inside `Ordered` cannot be changed unless one of the nodes is removed from `Ordered` first. Therefore $N \in s'.\texttt{Ordered} \wedge M \in s'.\texttt{Ordered} \Rightarrow (C_i \prec_{s'} C_j)$. Otherwise by the definition of $\prec$ we have: $\neg(C_i \prec_{s'} C_j) \wedge \neg(C_j \prec_{s'} C_i)$. □

In the remainder of the proof we sometimes omit the s subscript from the $\prec_s$ operator when considering transitions that do not modify `Ordered`. We are still required to show, however, that cells' nodes are in `Ordered` in order to claim that the cells are ordered by $\prec$.

Definition 34. *We define: $C_i \preceq C_j \equiv (C_i = C_j \vee C_i \prec C_j)$.*

Definition 35. *Let $C_i \in N_k$ and $C_j \in N_l$ be two cells such that $N_k \in$ `Ordered` and $N_l \in$ `Ordered`.*

- *C_i and C_j are* neighbors *if and only if they are adjacent with respect to $\prec$. We will use the predicate $Neighbors(C_i, C_j)$ to indicate if C_i and C_j are neighbors. $Neighbors(C_i, C_j)$ is false if the order between C_i and C_j is not defined.*
- *C_i is the* left neighbor *of C_j, denoted by $C_i = LeftNeighbor(C_j)$, if and only if $Neighbors(C_i, C_j) \wedge (C_i \prec C_j)$.*
- *C_i is the* right neighbor *of C_j, denoted by $C_i = RightNeighbor(C_j)$, if and only if $C_j = LeftNeighbor(C_i)$.*

Note that the $LeftNeighbor$ and $RightNeighbor$ are only partial functions, that is they are not defined for all cells. By the definition of Neighbors and the $\prec$ order, it is easy to see that:

1. $RightNeighbor(C_i)$ is defined if and only if $C_i \in N \in$ `Ordered` and either N is not the rightmost node in `Ordered`, or $i \neq DeqeuNode :: ArraySize - 1$.
2. $LeftNeighbor(C_i)$ is defined if and only if $C_i \in N \in$ `Ordered` and either N is not the leftmost node in `Ordered`, or $i \neq 0$.

4.5.1 The `IndicateEmpty` Macro

The `IndicateEmpty` macro, called at Line 10, takes values of type `BottomStruct` and `TopStruct` and indicates whether the deque would be empty if these were the values of `Top` and `Bottom`, respectively, in the state in which the macro is invoked. The code for the macro is depicted in Figure 12 on page 18. The following Lemma describe the properties of the macro:

Lemma 36. *Let $bottomVal$ and $topVal$ be two variables of type $BottomStruct$ and $TopStruct$, respectively, and suppose $Cell(topVal) \in N \in$ `Ordered` and that N is not the rightmost node in* `Ordered`*.*
Then `IndicateEmpty`*$(bottomVal, topVal) = true$ if and only if $(Cell(topVal) = Cell(bottomVal)) \vee (Cell(topVal) = LeftNeighbor(Cell(bottomVal)))$.*

Proof. If $botNode \in$ `Ordered`, then by examination of the code, Conjecture 55 and Invariant 4 it is obvious that the lemma is correct. The only interesting case is if $botNode \notin$ `Ordered`, in which case the macro should return false (since then $\neg(Cell(topVal) \preceq Cell(bottomVal))$). Since $topNode = N \in$ `Ordered` and it is not the rightmost node there, by Conjecture 55 we have: $topNode \rightarrow next \in$ `Ordered`, and therefore if $botNode \notin$ `Ordered IndicateEmpty` indeed returns false. □

We later prove why this property captures exactly the empty deque scenario. Note that as long as $bottomVal$ and $topVal$ are local process' variables, the `IndicateEmpty` macro does at most one read from the shared memory (that is, the read of the next pointer of the node indicated by $topVal$), and therefore is regarded as one atomic operation when called at Line 10. Finally, there is no guarantee on the return value of `IndicateEmpty` if $Node(Cell(topVal))$ is not in `Ordered`, or if it is the rightmost node there.

4.5.2 Crossing States

We say that the deque is in a crossing state when the cell pointed by `Top` is to the left of the cell pointed by `Bottom`, as depicted in Figure 2(b) on page 6. As we later explain, these states correspond to an empty deque.

Definition 37. *The deque is in a* crossing state *if and only if $C_T \prec C_B$.*

Note that if s.`Ordered` is in the state described by part c of Conjecture 55, then the deque is in a crossing state (since N_T precedes N_B in `Ordered`). The following is the main invariant describing when and under what conditions the deque may be in a crossing state:

Conjecture 54. If the deque is in a crossing state then:

1. C_T and C_B are neighbors.

2. $\exists p$ such that $p@\langle 26 \ldots 29, 31 \ldots 33, 36\rangle$.

Note that by Conjecture 55 we already know that if the deque is in a crossing state, then N_T and N_B are either the same node, or adjacent nodes in `Ordered`. The following invariants will be used for the proof of Conjecture 54, which is given in Section 4.6.

Invariant 38. $p@\langle 4 \ldots 6\rangle \Rightarrow (p.currIndex = 0)$.

Proof. Straightforward by examination of the code and by the observation that $s.p@\langle 4 \ldots 6\rangle \Rightarrow (s.p.currIndex = s'.p.currIndex)$ for any step $s \xrightarrow{a} s'$. □

Invariant 39. $p@6 \Rightarrow p.newNode \rightarrow next = p.currNode$.

Proof. Initially $p@0$ so the invariant holds. Let $s \xrightarrow{a} s'$be a step of the algorithm.

- If a is an execution of $p.5$: Then by examining the code $s'.p.newNode \rightarrow next = s'.p.currNode$.
- Otherwise, $s.p@6 \wedge s'.p@6$ implies by Invariant 1 that $\forall p'\ \neg s.p'@5$. By Invariant 28 we have $p.newNode \in s.\texttt{Live}$. Since the only statement that writes the $next$ pointer of a live node is $p'.5$ which is not enabled in s, we have: $(s.p.newNode \rightarrow next = s.p.currNode) \Rightarrow (s'.p.newNode \rightarrow next = s'.p.currNode)$, and therefore a does not falsify the consequence while the antecedent holds.

□

Invariant 40. *If $p@7$, then exactly one of the following is true:*

1. $p.newNode = p.currNode \wedge p.newIndex = p.currIndex - 1$.
2. $(p.newNode \neq p.currNode) \wedge (p.newNode \rightarrow next = p.currNode) \wedge (p.currNode \rightarrow prev = p.newNode) \wedge (p.currIndex = 0) \wedge (p.newIndex = DequeNode :: ArraySize - 1)$.

Proof. Initially $p@0$ so the invariant holds. Let $s \xrightarrow{a} s'$be a step of the algorithm.

- If a is an execution of p.3: By examining the code $s'.p.newNode = p.currNode \wedge s'.p.newIndex = p.currIndex - 1$.
- If a is an execution of p.6: By Invariant 38 we have $p.currIndex = 0$. By examining the code $(s'.p.currNode \rightarrow prev = s'.p.newNode) \wedge (s'.p.newIndex = DequeNode :: ArraySize - 1)$. Since the transition does not modify either a node's $next$ pointer, $p.currNode$ or $p.newNode$, then by Invariant 39 we have: $s'.p.newNode \rightarrow next = s'.p.currNode$. Finally, by Invariant 19 we have $p.newNode \notin \texttt{Ordered}$ and by Invariant 6 we have: $p.currNode = N_B$ which implies by Corollary 15 that $p.currNode \in \texttt{Ordered}$, and therefore $p.newNode \neq p.currNode$.

- Otherwise, by examining the code, no statement modifies $p.currNode$, $p.newNode$ or $p.currIndex$ while p@7.
  By Invariant 28 we have $p.newNode \in (s.\texttt{Live} \cap s'.\texttt{Live})$. By Invariant 6 we have: $p.currNode = N_B$ which implies by Corollary 15 that $p.currNode \in (s.\texttt{Ordered} \cap s'.\texttt{Ordered})$ and therefore by Conjecture 29 $p.currNode \in (s.\texttt{Live} \cap s'.\texttt{Live})$. Lines 5 and 6 are the only ones that writes the *next* or *prev* pointers of a live node, which by Invariant 1 are not enabled while p@7. Therefore if the invariant holds at s it also holds in s'.

□

Lemma 41. *Consider a transition $s \xrightarrow{a} s'$ where a is an execution of p.7, then: $s'.C_B = LeftNeighbor_{s'}(s.C_B)$.*

Proof. $s.N_B \in s.\texttt{Ordered}$ by Corollary 15, and $s.N_B \in s'.\texttt{Ordered}$ since a did not remove any node from `Ordered`. If $p.currNode = p.newNode$ then clearly $s'.N_B \in s'.\texttt{Ordered}$ (because $s.N_B = s'.N_B$ and $s.\texttt{Ordered} = s'.\texttt{Ordered}$). Otherwise $p.newNode$ is pushed to `Ordered` which implies $s'.N_B \in s'.\texttt{Ordered}$.

Since $s'.N_B \in s'.\texttt{Ordered}$ and $s.N_B \in s'.\texttt{Ordered}$, by Invariants 4, 40 and 55 we get $s'.C_B = LeftNeighbor_{s'}(s.C_B)$. □

Invariant 42. $p@33 \Rightarrow Cell(p.newTopVal) = Cell(p.currTop)$.

Proof. Straightforward by examination of Line 32. □

Corollary 43. *If $s \xrightarrow{a} s'$, where a is an execution of p.33, then: $s'.C_T = s.C_T$.*

Proof. The CAS operation at Line 33 modifies `Top` to $p.newTopVal$ if and only if $s.\texttt{Top} = p.currTop$, and by Invariant 42 $Cell(p.newTopVal) = Cell(p.currTop)$. □

Based on Corollary 43, in the rest of proof we do not regard Line 33 as one of the statements that may modify C_T.

Invariant 44. *If $p@\langle 25 \dots 29, 31 \dots 34, 36 \dots 38 \rangle$, then $Cell(p.newBotVal) = RightNeighbor(Cell(p.oldBotVal))$.*

Proof. Initially $p@0$ so the invariant holds. Let $s \xrightarrow{a} s'$be a step of the algorithm, and suppose the invariant holds in s.

- If a is the execution of Line p.23: Then a does not modify $p.oldBotNode$, $p.oldBotIndex$, or `Ordered`. By examining the code, $s'.p.newBotNode = p.oldBotNode$, and by Invariant 6 $p.oldBotNode = s.N_B = s'.N_B$, which by Corollary 15 implies that $p.oldBotNode \in \texttt{Ordered}$. By examining the code we also have: $s'.p.newBotIndex = p.oldBotIndex + 1$, and therefore by Invariant 4 and the definition of neighbors: $Cell(s'.p.newBotVal) = RightNeighbor(Cell(s'.p.oldBotVal))$.

- If a is the execution of Line p.24: Then a does not modify $p.oldBotNode$, $p.oldBotIndex$, or `Ordered`. By examining the code, $s'.p.newBotNode = p.oldBotNode \rightarrow next$. By Invariant 6 $p.oldBotNode = s.N_B = s'.N_B$, which by Corollary 15 implies that $p.oldBotNode \in$ `Ordered`. By Invariants 54 and 1 the deque is not in a crossing state at s, and therefore by Conjecture 55: $N_B \rightarrow next \in$ `Ordered`. Therefore $s'.p.newBotNode = p.oldBotNode \rightarrow next$ implies by Conjecture 55 that $s'.p.newBotNode$ and $s'.p.oldBotNode$ are adjacent in s'.`Ordered` (since both nodes are in s'.`Ordered`). Finally, by examining the code we have: $s'.p.oldBotIndex = DequeNode :: ArraySize - 1 \wedge s'.p.newBotIndex = 0$, which implies that: $Cell(s'.p.newBotVal) = RightNeighbor(Cell(s'.p.oldBotVal))$.

- Otherwise, we consider statements that might falsify the consequence while the antecedent holds. Since the invariant holds in s we have: $Cell(s.p.newBotVal) = RightNeighbor_s(Cell(s.p.oldBotVal))$ which also implies: $s.p.newBotNode \in s.$`Ordered` $\wedge\ s.p.oldBotNode \in s.$`Ordered`. Note that $p.oldBotNode$ and $p.newBotNode$ are not modified while $p@\langle 25 \ldots 29, 31 \ldots 34, 36 \ldots 38 \rangle$. By Lemma 33, and because `Ordered` only supports addition of nodes to its left and right ends, we only need to show that $p.oldBotNode$ and $p.newBotNode$ are not removed from `Ordered`. There are two cases to consider:

  - If $s.p@25$:
    - If a is an execution of p.25, then $s'.$`Ordered` $= s.$`Ordered`.
    - Otherwise, since $s'.p@25$, by Invariant 6: $p.oldBotNode = s'.N_B \in s'.$`Ordered`, and by Conjecture 54 the deque is not in a crossing state at s', which implies by Conjecture 55 that $p.newBotNode \in s'.$`Ordered`.
  - Otherwise, by Invariant 6 $p.newBotNode = N_B$ and by Invariant 16 $p.oldBotNode \in s'.$`Ordered`.

  Therefore the consequence is not falsified while the antecedent holds, and the invariant holds in s'.

□

Lemma 45. *Let $s \xrightarrow{a} s'$ be a step of the algorithm, and suppose a is an execution of Line 29 or Line 36, then:*
$s'.C_B = LeftNeighbor_s(s.C_B) = LeftNeighbor_{s'}(s.C_B)$.

Proof. Let p be the process executing Line 29 or 36. By Invariant 44, $s.p@\langle 29, 36 \rangle \Rightarrow Cell(p.newBotVal) = RightNeighbor_s(Cell(p.oldBotVal))$. By Invariant 6: $s.p@\langle 29, 36 \rangle \Rightarrow s.p.newBotVal = s.$`Bottom`, and by examining the code $s'.$`Bottom` $= s.p.oldBotVal$, which implies: $s.C_B = RightNeighbor_s(s'.C_B)$, and therefore by the definition of the neighboring relation $s'.C_B = LeftNeighbor_s(s.C_B)$. Finally,

since $s.\texttt{Ordered} = s'.\texttt{Ordered}$, $LeftNeighbor_s(s.C_B) = LeftNeighbor_{s'}(s.C_B)$. □

Invariant 46. *If* $p@\langle 10, 12 \dots 17\rangle \wedge p.currTop = \texttt{Top} \wedge C_T \preceq C_B$ *then:*

1. $(p@10 \wedge$
   $(Cell(p.currBottom) = C_T \vee Cell(p.currBottom) = RightNeighbor(C_T)))$,
   or
2. (a) $C_T = C_B$, *and*
   (b) $\exists p'$ *such that:* $p'@26 \vee (p'@\langle 27, 28, 31 \dots 33\rangle \wedge p'.currTop = \texttt{Top})$.

Proof. Initially $p@0$ so the invariant holds. We use the following notations for the antecedent and consequent:

- $s.P \equiv (s.p@\langle 10, 12 \dots 17\rangle \wedge s.p.currTop = s.\texttt{Top} \wedge s.C_T \preceq s.C_B)$
- $s.R \equiv (s.R1 \vee s.R2)$, where:
  $s.R1 \equiv (s.p@10 \wedge (Cell(s.p.currBottom) = s.C_T \vee Cell(s.p.currBottom) = RightNeighbor_s(s.C_T)))$,
  $s.R2 \equiv (s.C_T = s.C_B \wedge$
  $(\exists p'\ s.p'@26 \vee (s.p'@\langle 27, 28, 31 \dots 33\rangle \wedge s.p'.currTop = s.\texttt{Top})))$.

Let $s \xrightarrow{a} s'$ be a step of the algorithm and suppose the invariant holds in s. We should show then that $s'.P \Rightarrow s'.R$:

1. If a is an execution of line $p'.7$: By Conjecture 54 $s.C_T \succeq_s s.C_B$ and since a does not remove any node from $\texttt{Ordered}$, we have $s.C_T \succeq_{s'} s.C_B$. By Lemma 41 $s.C_B \succ_{s'} s'.C_B$. Finally, $s'.C_T = s.C_T$ and therefore $\neg(s'.C_T \preceq_{s'} s'.C_B)$, which implies that $s'.P$ does not hold.
2. If a writes a value to $\texttt{Top}$: Then by Corollary 12 $s'.p@\langle 10, 12 \dots 17\rangle \Rightarrow s'.p.currTop \neq s'.\texttt{Top}$ which implies that $s'.P$ does not hold.
3. Otherwise, suppose a establishes the antecedent (that is, $s'.P \wedge \neg s.P$ holds):
   - If a is an execution of $p.9$: Then $Cell(s'.p.currBottom) = s.C_B = s'.C_B$. By applying Conjecture 54 to s', $s'.C_T \preceq s'.C_B \Rightarrow s'.C_B = s'.C_T \vee s'.C_B = RightNeighbor(s'.C_T)$. Therefore $s'.R1$ holds, which implies that $s'.R$ holds.
   - If a modifies $p.currTop$: No statement modifies $p.currTop$ while $p@\langle 10, 12 \dots 17\rangle$ holds, and therefore P cannot be established by a.
   - Otherwise, note that the $\prec$ relation is used in P only between C_T and C_B, which by Conjecture 55 are always ordered. Therefore, because $\texttt{Ordered}$ does not support operations that reorder its elements, $C_T \preceq C_B$ cannot be established if $s'.C_T = s.C_T \wedge s'.C_B = s.C_B$. Since a

does not modify `Top`, then only statements that modify `Bottom` may establish P.

Because a modifies `Bottom` and it is not an execution of Line 7 we have: $s'.\texttt{Ordered} = s.\texttt{Ordered}$, and therefore: $\prec_s \equiv \prec_{s'}$. Since P does not hold in s but holds in s' we have: $(s.C_B \prec C_T \wedge s'.C_B \succeq C_T)$, which implies $s.C_B \prec s'.C_B$.

- If a is an execution of $p'.25$ for some process $p' \neq p$: Then $s'.p'@26$. Since $s.C_B \prec C_T$, by Invariants 6 and 44 $s'.C_B \preceq C_T$, and because we also have $s'.C_B \succeq C_T$, then $s'.C_B = C_T$. Therefore $s'.R2$ holds, which implies that $s'.R$ holds.
- If a is an execution of $p'.29$ or $p'.36$ for some process $p' \neq p$: Then $s'.C_B = Cell(p'.oldBotVal)$ and by Invariant 6: $s.C_B = Cell(p'.newBotVal)$. Therefore by Lemma 45 $s'.C_B \prec s.C_B$, a contradiction.

We showed then that if $s.P$ does not hold, then $s'.P$ does not hold, unless a is an execution of Line 9 or 25, in which case $s'.R$ holds.

4. Otherwise $s.P$ holds, and since the invariant holds in s, then $s.R$ holds as well. There are two cases two be considered:

   - If $s.R1$ holds: $p@10 \Rightarrow s'.p.currBottom = s.p.currBottom$, and we already showed that if a writes a value to `Top` then it does not falsify the invariant, and therefore $s'.C_T = s.C_T$. Therefore $Cell(p.currBottom) = C_T$ is not falsified while $p@10$.

     By Corollary 15 $RightNeighbor_{s'}(C_T)$ is defined (since N_T is not the rightmost node in s'.`Ordered`), and since no statement atomically removes and adds a node to `Ordered`, we get: $RightNeighbor_s(C_T) = RightNeighbor_{s'}(C_T)$, and therefore $Cell(p.currBottom) = RightNeighbor(C_T)$ is not falsified while $p@10$. Therefore the only transition that can falsify $R1$ is an execution of $p.10$.

     If a is an execution of $p.10$, since $s.P$ holds we have $s.p.currTop = s.\texttt{Top}$, which implies by Corollary 15 that $s.p.currTopNode \in s.\texttt{Ordered}$ and that it is not the rightmost node there. Since $s.p.currTop = s.\texttt{Top}$, then by Lemma 36: $Cell(s.p.currBottom) = RightNeighbor(s.C_T) \Rightarrow s'.p@11$. Therefore $s'.P$ does not hold.
   - Otherwise, $(\neg s.R1 \wedge s.R2)$ holds: Only statements that modify `Top` or `Bottom`, and statements of process p' can falsify $R2$. Since a does not modify `Top`, by Invariant 1 no statement modifies `Bottom` while $p'@26 \vee (s.p'@\langle 27, 28, 31 \ldots 33\rangle$ holds, and no statement modifies $p.currTop$ while $s.p'@\langle 27, 28, 31 \ldots 33\rangle$ holds, then we only need to consider statements $p'.26, p'.28, p'.31$ and $p'.33$:

- If a is an execution of $p'.26$: Then clearly $s'.p'.currTop = \texttt{Top}$ and since a does not modify `Top` or `Bottom` we have $s'.C_T = s'.C_B$, and R still holds in s'.
- If a is an execution of $p'.28$: Since $R2$ holds in s we have $C_T = C_B$ and $p'.currTop = \texttt{Top}$. By Invariant 6 $C_B = Cell(p'.newBotVal)$, and by Invariant 44 $Cell(p'.newBotVal) \neq Cell(p'.oldBotVal)$. Thus: $\underline{Cell(p'.currTop)} = C_T = C_B = \underline{Cell(p'.newBotVal) \neq Cell(p'.oldBotVal)}$, and therefore the test at Line 28 fails, which implies that $s'.p'@31$ and R holds in s'.
- If a is an execution of $p'.31$: As in the previous case, we have: $\underline{Cell(p'.currTop)} = C_T = C_B = \underline{Cell(p'.newBotVal)}$, and therefore the test at Line 31 returns true, which implies that $s'.p'@32$ and R holds in s'.
- If a is an execution of $p'.33$: Since $R2$ holds in s we have $p'.currTop = \texttt{Top}$, and therefore the CAS operation of Line 33 succeeds. But we already claimed that if a is an update of `Top` then P does not hold at s', which implies that the invariant holds in s'.

We showed then that if $s.P$ and $s.R$ hold, then $s'.R$ also holds unless a is the execution of Line 33, in which case $s'.P$ does not hold.

□

Corollary 47. $p@\langle 12 \dots 17\rangle \wedge C_T \prec C_B \Rightarrow p.currTop \neq Top.$

Proof. Since $p@\langle 12 \dots 17\rangle \wedge C_T \prec C_B$, if $p.currTop = \texttt{Top}$ we can apply Invariant 46 and get $C_T = C_B$, which contradict the assumption that $C_T \prec C_B$. □

Invariant 48. *If* $p@\langle 12 \dots 17\rangle$, $p.currTop = \texttt{Top}$, *and* $p.currTopIndex == 0$ *then* $p.currTopNode \rightarrow prev \in \texttt{Ordered}$.

Proof. We will prove this invariant without using induction. If $p.currTop = \texttt{Top}$, by Corollary 15 we have: $p.currTopNode \in \texttt{Ordered}$. By Corollary 47 $p@\langle 12 \dots 17\rangle \wedge p.currTop = \texttt{Top}$ implies that the deque is not in a crossing state, and therefore $C_B \preceq C_T$. We consider two cases:

- If $C_B \neq C_T$: then $C_B \prec C_T$, and since $p.currTopIndex = 0$, then N_T is not the leftmost node in `Ordered`, and therefore by Conjecture 55 $N_T \rightarrow prev \in \texttt{Ordered}$. If $p.currTop = \texttt{Top}$ then $p.currTopNode = N_T$, which implies that $p.currTopNode \rightarrow prev \in \texttt{Ordered}$.
- Otherwise $C_B = C_T$ and by Invariant 46 $\exists p'\ p'@\langle 26 \dots 28, 31 \dots 33\rangle$. By Invariant 6 we have: $C_B = Cell(p'.newBotVal)$, and by Invariant 44: $\underline{Cell(p'.oldBotVal)} = LeftNeighbor(Cell(p'.newBotVal)) = \overline{LeftNeighbor(C_B)} = \underline{LeftNeighbor(C_T)}$. Since $p.currTop = \texttt{Top}$ and

$p.currTopIndex = 0$, by Conjecture 55 and the definition of Neighbors we have:
$LeftNeighbor(C_T) = Cell(p'.oldBotVal) \Rightarrow$
$p.currTopNode \rightarrow prev = p'.oldBotNode \in$ `Ordered`.

□

Invariant 49.

1. $s.p@13 \Rightarrow s.p.currTopIndex \neq 0$.
2. $s.p@\langle 14, 15\rangle \Rightarrow s.p.currTopIndex = 0$.

Proof. Straightforward by examination of the code. □

Invariant 50. *If* $p@\langle 16, 17\rangle \wedge (p.currTop =$ `Top`*), then* $Cell(p.newTopVal) = LeftNeighbor(Cell(p.currTop))$.

Proof. Initially $\neg p@\langle 16, 17\rangle$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$ that falsifies the invariant.

1. If a is a write operation to `Top`: Then by Corollary 12 $s'.p.currTop \neq s'.$`Top`, so the invariant holds in s'.
2. Consider statements that may establish the antecedent. Since no statement in $\langle 16, 17\rangle$ modifies any of p's private variables, and a does not modify `Top`, we only need to consider statements that establish $p@\langle 16, 17\rangle$. If $s'.p.currTop \neq s'.$`Top`, then a does not establish the antecedent. Otherwise by Conjecture 55 we have: $s'.p.currTopNode \in s'.$`Ordered`, and there are two statements to consider: Line 13 and Line 15:
   - If a is an execution of Line 13:
     Then $(s'.p.newTopNode = s.p.currTopNode = s'.p.currTopNode) \wedge$ $(s'.p.newTopIndex = s.p.currTopIndex - 1)$, and by Invariant 49 $s.p.currTopIndex \neq 0$. Since $s'.p.newTopNode = s'.p.currTopNode \in$ $s'.$`Ordered`, then by Invariant 4:
     $Cell(s'.p.newTopVal) = LeftNeighbor_{s'}(Cell(s'.p.currTop))$.
   - If a is an execution of Line 15:
     Then $s'.p.newIndex = DequeNode :: ArraySize - 1$, and since $s'.p.currTop = s.p.currTop$ we have:
     $s'.p.newTopNode = s'.p.currTopNode \rightarrow prev$.
     By Invariant 49 $s.p.currTopIndex = 0$, and since $s'.p.currTopIndex =$ $s.p.currTopIndex$ and $s'.p.currTop = s'.$`Top`, by Invariant 48 we get: $s'.p.newTopNode \in s'.$`Ordered`. Finally by Invariants 55 we get:
     $Cell(s'.p.newTopVal) = LeftNeighbor_{s'}(Cell(s'.p.currTop))$.

3. Consider statements that may falsify the consequent while the antecedent holds. No statement modifies any of p's private variables while $p@\langle 16, 17\rangle$ holds (except $p.retVal$ which is irrelevant for this invariant). Since the antecedent holds in s we have $p.currTop = s.\texttt{Top}$, and since a does not modify `Top`, we get: $p.currTop = s'.\texttt{Top}$, which implies by Conjecture 55 that $p.currTopNode \in s'.\texttt{Ordered}$.
Therefore if $s.p.currTopNode = s.p.newTopNode$, no statement can falsify the consequent.

   Otherwise, since the consequence holds in s and $p.newTopNode \neq p.currTopNode$, we have: $p.newTopNode = p.currTopNode \rightarrow s.prev \wedge p.currTopIndex = 0$, and since the antecedent holds in s' we have: $s'.p@\langle 16, 17\rangle$, which implies by Invariant 48 that:
$p.currTopNode \rightarrow s'.prev \in s'.\texttt{Ordered}$. Therefore the only transition a that might falsify the consequent is one that modifies the $prev$ field of $p.currTopNode$. There are two cases to consider:

   (a) If $p.currTopNode \neq s.N_B$ then by Lemma 21: $p.currTopNode \rightarrow s.prev = p.currTopNode \rightarrow s'.prev$.

   (b) Otherwise $p.currTopNode = s.N_B$, and since $p.currTopIndex = 0$ and $Cell(p.newTopVal) = LeftNeighbor_s(Cell(p.currTop))$, $s.N_B$ is not the leftmost node in s.`Ordered`. By Invariant 16 and Invariant 1 $\forall p\ \neg p@6$, and since execution of Line 6 is the only transition that writes the $prev$ field of a node, we get: $p.currTopNode \rightarrow s.prev = p.currTopNode \rightarrow s'.prev$.

□

Invariant 51. *If* $p@\langle 27, 28, 31 \ldots 33\rangle \wedge (p.currTop \neq Top)$ *then* $\neg(C_T \prec C_B) \vee (Cell(p.currTop) = C_B)$.

Proof. Initially $p@0$. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s.

- If a modifies `Top`: If $\neg s'.p@\langle 27, 28, 31 \ldots 33\rangle$ then the invariant clearly holds in s'. Otherwise, the only statement that might modify `Top` is $p'@17$ for some process $p' \neq p$.

  In this case a falsifies the invariant only if $s'.C_T \prec_{s'} s'.C_B = s.C_B$ and $s.p@\langle 27, 28, 31 \ldots 33\rangle$. By Conjecture 54, $s'.C_T \prec_{s'} C_B$ implies $s'.C_T = LeftNeighbor_{s'}(C_B)$, and since the CAS at Line 17 modifies `Top` to be $p'.newTopVal$ if and only if $p'.currTop = s.\texttt{Top}$, by Invariant 50 it implies that $s.C_T = C_B$.
  By Invariant 46 we get: $\exists p''\ s.p''@26 \vee (s.p''@\langle 27, 28, 31 \ldots 33\rangle \wedge s.p''.currTop = s.\texttt{Top})$, and by Invariant 1, $p'' = p$. Since $\neg p@26$, $s'.p.currTop = s.p.currTop = s.\texttt{Top}$, which implies that $Cell(s'.p.currTop) = s.C_T = s.C_B = s'.C_B$.
  Therefore the invariant holds in s'.

- Otherwise, since a does not modify `Top`, the only statement that might establish the antecedent is p.26. In this case $s'.p.currTop = s'.\texttt{Top}$ and the invariant holds in s'.
- Otherwise, we consider statements that may falsify the consequence while the antecedent holds. Sine `Bottom` and $p.currTop$ are not modified while $p@\langle 27, 28, 31 \dots 33\rangle$ holds, we only need to consider statements that modify `Top`. But we already showed that such statements cannot falsify the invariant, and therefore no statement can falsify the consequence while the antecedent holds.

□

Invariant 52.

1. $s.p@\langle 29, 30\rangle \Rightarrow (Cell(s.p.currTop) = Cell(s.p.oldBotVal))$.
2. $s.p@\langle 31 \dots 39\rangle \Rightarrow (Cell(s.p.currTop) \neq Cell(s.p.oldBotVal))$.
3. $s.p@\langle 32 \dots 37\rangle \Rightarrow (Cell(s.currTop) = Cell(s.p.newBotVal))$.
4. $s.p@\langle 38, 39\rangle \Rightarrow (Cell(s.p.currTop) \neq Cell(s.p.newBotVal))$.

Proof. Straightforward by examination of the code and from the observation that $p@\langle 27 \dots 39\rangle \Rightarrow (s.p.currTop = s'p.currTop) \wedge (s.p.oldBotVal = s'.p.oldBotVal) \wedge (s.p.newBotVal = s'.p.newBotVal)$. □

Invariant 53. *If* $p@\langle 15 \dots 17\rangle \wedge p.currTop = \texttt{Top}$ *then:*
$(p.nodeToFree \neq \texttt{NULL}) \Leftrightarrow (p@15 \vee p.currTopNode \neq p.newTopNode)$.

Proof. Initially $p@0$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s.

1. If a modifies `Top`, then by Corollary 12, $\neg s'.p@\langle 15 \dots 17\rangle \vee s'.p.currTop \neq s'.\texttt{Top}$, so the invariant holds in s'.
2. Otherwise, consider statements that may establish the antecedent. Because no statement modifies $p.currTop$ while $s.p@\langle 15 \dots 17\rangle$ holds, we only need to consider statements that may establish $p@\langle 15 \dots 17\rangle$.
   - If a is an execution of $p.13$: Then $s'.p.nodeToFree = \texttt{NULL} \wedge \neg s'.p@15 \wedge s'.p.newTopNode = s.p.currTopNode = s'.p.currTopNode$ and therefore the consequent holds in s'.
   - If a is an execution of $p.14$: Then $s'.p@15$ and $s'.p.nodeToFree = s.p.currTopNode \rightarrow next$. Note that a does not modify $p.currTop$ or `Top`. Therefore if $s.p.currTop \neq s.\texttt{Top}$ then a does not establish the antecedent. Otherwise, by Invariant 5 and Corollary 15 it follows that $s'.p.nodeToFree \in \texttt{Ordered}$, and therefore by Corollary 17, $s'.p.nodeToFree \neq \texttt{NULL}$. Therefore the consequent holds in s'.

3. We now consider statements that may falsify the consequent while the antecedent holds. Since no statement modifies $p.nodeToFree$ or $p.currTopNode$ while the antecedent holds, it suffices to consider statements that falsify $p@15$ or modify $p.newTopNode$ while the antecedent holds. The only such statement is $p.15$.

   If a is an execution of $p.15$, then $s.p@15$ and since the consequent holds in s, it follows that $s.p.nodeToFree \neq \texttt{NULL}$. Since $s'.p.nodeToFree = s.p.nodeToFree$, it remains to show that $s'.p.newTopNode \neq s'.p.currTopNode$.

   By examining the code $s'.p.newTopNode = s.p.currTopNode \rightarrow prev$, and since the antecedent holds in s we have $s.p.currTopNode = s.N_T$. Therefore by Conjecture 55 and Invariant 20: $s.p.currTopNode \rightarrow prev \neq s.p.currTopNode$, and since $p.currTopNode$ is not modified by a we can conclude that $s'.p.newTopNode \neq s'.p.currTopNode$.

□

4.6 Proofs of Conjectures 54 and 55

In this section we prove Conjectures 54 and 55, which are the two main invariants of the algorithm. Later these invariants will be useful in the linearizability proof.

As explained in Section 4.1.3, we must be careful to avoid circular reasoning, because the proofs of some of the invariants proved so far use Conjectures 55 and 54 in the post-state of their inductive step. Accordingly, in the proof of Conjecture 54, we use Conjecture 55 both in the induction pre-state and post-state, but all other invariants and conjectures are used only in the induction pre-state. In the proof of Conjecture 55, we only use other invariants and conjectures in the induction pre-state.

Conjecture 54. *If the deque is in a crossing state then:*

1. *C_T and C_B are neighbors.*
2. *$\exists p$ such that $p@\langle 26 \ldots 29, 31 \ldots 33, 36 \rangle$.*

Proof. Initially the deque is constructed such that $Cell(\texttt{Bottom}) = Cell(\texttt{Top})$ so the invariant holds. Consider a transition $s \xrightarrow{a} s'$, and suppose the invariant holds in s. Note that by Conjecture 55, $N_T \in \texttt{Ordered} \wedge N_B \in \texttt{Ordered}$ and therefore: $\neg(C_T \prec C_B) \Leftrightarrow C_B \preceq C_T$. That also means that if we have, for example, $s.C_B \preceq_s s.C_T \wedge s'.C_B \prec_{s'} s.C_B \wedge s'.C_T = s.C_T$, it implies that $s'.C_B \prec_{s'} s'.C_T$, since neither $s.C_T$ nor $s.C_B$ can be removed from $\texttt{Ordered}$ by the transition, and therefore by Lemma 33 the order between them cannot be changed.

- We first consider transitions that may establish the antecedent. Then we have $s.C_B \preceq_s s.C_T$ and $s'.C_T \prec_{s'} s'.C_B$, which implies by Lemma 41 that a

cannot be the execution of Line 7, and by Lemma 45 that it cannot be the execution of Line 29 or 36. Therefore we have two statements to consider: Line 25 and Line 17.

1. If a is an execution of p.25: Then by Invariant 44 $s'.C_B = RightNeighbor(s.C_B)$. Since $s'.C_T = s.C_T \wedge s'.\texttt{Ordered} = s.\texttt{Ordered}$, then if the deque is in crossing state in s': $((s'.C_T \prec_{s'} s'.C_B) \wedge (s.C_B \preceq_s s.C_T) \wedge (s'.C_B = RightNeighbor(s.C_B))) \Rightarrow s'.C_T = LeftNeighbor(s'.C_B)$. Also, $s'.p@26$ and therefore the consequent holds in s'.
2. If a is an execution of p.17: Then by Invariant 50 $s'.C_T = LeftNeighbor_s(s.C_T)$. Therefore if the deque is in crossing state in s': $((s.C_B \preceq_s s.C_T) \wedge (s'.C_T \prec_{s'} s'.C_B)) \Rightarrow (s.C_T = s.C_B = s'.C_B \wedge s'.C_T = LeftNeighbor_s(s.C_B)) \Rightarrow s'.C_T = LeftNeighbor_{s'}(s'.C_B)$. By Invariant 46, $(s.p@17 \wedge s.C_T = s.C_B) \Rightarrow ((s.p.currTop \neq s.\texttt{Top}) \vee (\exists p' \neq p\ p'@\langle 26 \ldots 29, 31 \ldots 33, 36\rangle))$. Therefore it is either that the CAS fails and a does not modify `Top` (and therefore does not establish the antecedent), or that the consequent holds in s'.

- We now consider transitions that may falsify the consequent while the antecedent holds. Because the antecedent and the invariant holds in s, $\exists p\ s.p@\langle 26 \ldots 29, 31 \ldots 33, 36\rangle \wedge s.C_T = LeftNeighbor_s(s.C_B)$, and since the antecedent holds in s', $s'.C_T \prec_{s'} s'.C_B$. By Conjecture 55 if a falsifies $C_T = LeftNeighbor(C_B)$ then $s'.C_B \neq s.C_B \vee s'.C_T \neq s.C_T$. Therefore there are two types of transitions that might falsify the consequent: A modification of C_T or C_B that results in: $s'.C_T \neq LeftNeighbor_{s'}(s'.C_B))$, or an execution of a statement that results in $\forall p\ \neg s'.p@\langle 26 \ldots 29, 31 \ldots 33, 36\rangle$.

  1. If a is a modification of C_T or C_B: Then $s.C_T = LeftNeighbor_s(s.C_B) \wedge s'.C_T \prec_{s'} s'.C_B$ implies by Lemma 41 that a is not an execution of Line 7, and by Lemma 45 that it is not an execution of Line 29 or 36. Therefore it is left to consider executions of Line 25 and 17.
     By Invariant 1, $s.p@\langle 26 \ldots 29, 31 \ldots 33, 36\rangle \Rightarrow \forall p'\ \neg s.p'@25$, and therefore a is not an execution of Line 25. If a is an execution of p'.17, then by Corollary 47 $s.C_T \prec_s s.C_B \wedge s.p'.17 \Rightarrow s.p'.currTop \neq s.\texttt{Top}$, and therefore $s'.C_T = s.C_T \wedge s'.C_B = s.C_B$, a contradiction.
  2. Otherwise, since a is not a modification of C_T or C_B, then the only transitions that may falsify $p@\langle 26 \ldots 29, 31 \ldots 33, 36\rangle$ are executions of p.31 or p.33.
     - If a is an execution of p.31: Then a can falsify the consequent only if $s.C_T \prec s.C_B \wedge s'.p@38$.
       By Invariant 6 $s.\texttt{Bottom} = s.p.newBotVal$. There are two cases to be considered:

(a) If $s.p.currTop \neq s.\texttt{Top}$, then by Invariant 51 $s.C_T \prec s.C_B \Rightarrow Cell(s.p.currTop) = s.C_B = Cell(s.p.newBotVal)$, and therefore $s'.p@32$ and the consequent holds in s'.

(b) Otherwise, $Cell(s.p.currTop) = s.C_T$. By Invariant 52: $Cell(s.p.currTop) \neq Cell(s.p.oldBotVal)$. By Invariant 44: $Cell(s.p.oldBotVal) = LeftNeighbor_s(Cell(s.p.newBotVal)) = LeftNeighbor_s(s.C_B)$, and therefore $s.C_T \neq LeftNeighbor_s(s.C_B)$, a contradiction.

– If a is an execution of p.33: Then $s'.C_B = s.C_B$ and by Corollary 43: $s'.C_T = s.C_T$. By Invariant 52 $Cell(s.p.currTop) = Cell(s.p.newBotVal)$. If $s.p.currTop \neq s.\texttt{Top}$ then the CAS fails and $s'.p@36$, so a does not falsify the consequent. Otherwise, $s.C_T = Cell(s.p.currTop) = Cell(s.p.newBotVal) = s.C_B$. Therefore $s.C_T = s.C_B$, which implies $s'.C_T = s'.C_B$, so the antecedent does not hold in s'.

□

Conjecture 55. *Let $n = |\texttt{Ordered}| - 2$, and $\texttt{Ordered} = \{N_0, \ldots, N_{n+1}\}$. Then:*

1. $\forall_{0 \leq i \leq n}\ N_i \rightarrow next = N_{i+1} \wedge N_{i+1} \rightarrow prev = N_i$.
2. *Exactly one of the following holds:*
   (a) $n \geq 0,\ N_0 = N_B, N_n = N_T$.
   (b) $n > 0,\ N_1 = N_B, N_n = N_T$.
   (c) $n = 0,\ N_0 = N_T, N_1 = N_B$.

Proof. We use the following notations in the proof:

- $WellLinked \equiv (0 \leq i \leq n) \Rightarrow (N_i \rightarrow next = N_{i+1} \wedge N_{i+1} \rightarrow prev = N_i)$.
- $WellOrdered \equiv Order1 \vee Order2 \vee Order3$, where:
  – $Order1 \equiv (n \geq 0) \wedge (N_0 = N_B) \wedge (N_n = N_T)$.
  – $Order2 \equiv (n > 0) \wedge (N_1 = N_B) \wedge (N_n = N_T)$.
  – $Order3 \equiv (n = 0) \wedge (N_0 = N_T) \wedge (N_1 = N_B)$.

Note that if the invariant holds, then $n \geq 0$. Therefore, because $|\texttt{Ordered}| = n + 2$, N_0, N_1, and N_n are all well defined. We use $s.N_i$, to denote node N_i in $s.\texttt{Ordered}$, and $s.n$ to denote the value of n in $s.\texttt{Ordered}$ (that is, $n = |s.\texttt{Ordered}| - 2$).

We need to show that: $WellLinked \wedge WellOrdered$ holds. Note that by Invariant 20 we have:

- $s.Order1 \Rightarrow (\neg s.Order2 \wedge \neg s.Order3)$.

- $s.Order2 \Rightarrow (\neg s.Order1 \wedge \neg s.Order3)$.
- $s.Order3 \Rightarrow (\neg s.Order1 \wedge \neg s.Order2)$.

That is, $Order1$, $Order2$, and $Order3$ are mutually exclusive.

The Initial State: By examining the code of the deque constructor depicted in Figure 8 on page 15, we see that right after the construction of the deque:

1. $\texttt{Ordered} = \{nodeA, nodeB\}$.
2. $nodeA \rightarrow next = nodeB$.
3. $nodeB \rightarrow prev = nodeA$.
4. $N_B = N_T = nodeA$.

Thus we have $s.WellLinked \wedge s.Order1$ holds (with $n = 0$), and therefore the invariant holds.

The Inductive Step: Consider a transition $s \xrightarrow{a} s'$ in the algorithm, and suppose the invariant holds in s.

We first consider transitions that might falsify $WellLinked$. There are two types of such transitions:

- If a does not add a node to `Ordered`: Then since the only operations `Ordered` supports are removal and addition of nodes at the ends of the series, a might falsify $WellLinked$ only if it modifies a $next$ or $prev$ field of a node $N \in s.\texttt{Ordered}$. By Lemma 21 we have:

  1. $(N \in s.\texttt{Ordered}) \Rightarrow (s'.N \rightarrow next = s.N \rightarrow next)$.
  2. $(N \in s.\texttt{Ordered} \wedge s'.N \rightarrow prev \neq s.N \rightarrow prev) \Rightarrow (N = s.N_0)$.

  and by Invariant 20, a does not falsify $WellLinked$.

- Otherwise, if a adds a node to `Ordered`, then Lemma 18 implies that a is an execution of p.7 by some process p. Line 7 adds $p.newNode$ to `Ordered` only if $p.newNode \neq p.currNode$, which by Invariant 40 implies that: $(p.newNode \rightarrow next = p.currNode) \wedge (p.currNode \rightarrow prev = p.newNode)$. Finally, by Invariant 6: $p.currNode = s.N_B$, and by Invariants 1 and 16, $s.N_B = s.N_0$. Since a adds $p.newNode$ to `Ordered` by performing an `AddLeft` operation, then a does not falsify $WellLinked$.

Next, we consider transitions that might falsify $WellOrdered$. There are two types of such transitions:

- If a does not modify `Top` or `Bottom`: Then a might falsify $WellOrdered$ only if it removes or adds a node to `Ordered` (without modifying `Top` or `Bottom`). By Lemma 41 an execution of Line 7 always modifies `Bottom`, and by Invariant 50 an execution of Line 17 does not modify `Top` only if the CAS fails, in which case it also does not modify `Ordered`. Therefore we only need to consider the execution of p.34 or p.38 for some process p, which removes the leftmost node from `Ordered` if and only if $s.p.oldBotNode \neq s.p.newBotNode$.

  By Invariant 6 we have: $N_B = p.newBotNode$, and by Invariant 16 we have: $p.oldBotNode = s.N_0$. By Invariants 1 and 54 the deque is not in a crossing state in s, and since the invariant holds in s, $s.p.oldBotNode \neq s.p.newBotNode$ implies that $s.Order2$ holds. Therefore after a removes the leftmost node, $s'.Order1$ holds (note that $s'.n = s.n - 1$), which implies that a does not falsify $WellOrdered$.

- Otherwise, a might falsify $WellOrdered$ only if it modifies N_B or N_T, that is: $s'.N_B \neq s.N_B \vee s'.N_T \neq s.N_T$. By Invariant 42 Line 33 does not modify N_T, and therefore we only need to consider executions of: p.7, p.17, p.25, p.29 or p.36 for some process p.

  - If a is an execution of p.7: Then $s'.N_T = s.N_T$, by Invariants 1 and 16 $s.N_B = s.N_0$, and since the invariant holds in s, by Invariant 20 $s.Order1$ holds.
    - If $p.newNode = p.currNode$ then $(s'.N_B = s.N_B) \wedge (s'.\texttt{Ordered} = s.\texttt{Ordered})$, which implies that $s'.Order1$ holds.
    - Otherwise $s'.N_0 = p.newNode = s'.N_B$, which also implies that $s'.Order1$ holds.

    Therefore a does not falsify $WellOrdered$.
  - If a is an execution of p.25: Then $s'.N_T = s.N_T$, by Invariants 1 and 16 $s.N_B = s.N_0$, and since the invariant holds in s, by Invariant 20 $s.Order1$ holds. By Invariant 6 we have $s.N_B = p.oldBotNode$, and therefore $(p.newBotNode = p.oldBotNode) \Rightarrow s'.N_B = s.N_B$. Otherwise by Invariant 44 $p.newBotNode = p.oldBotNode \rightarrow next$, which implies: $s'.N_B = p.newBotNode = p.oldBotNode \rightarrow next = s.N_0 \rightarrow next$. Since $s.WellLinked$ holds, $s.N_0 \rightarrow next = s.N_1 = s'.N_1$, which implies that $s'.Order2 \vee s'.Order3$ holds. Therefore a does not falsify $WellOrdered$.
  - If a is an execution of p.29 or p.36: Then $s'.N_T = s.N_T$, by Invariants 1 and 16 $p.oldBotNode = s.N_0$, and since the invariant holds in s we have $s.N_T = s.N_{s.n}$. Since `Ordered` is not modified by a, $s.N_{s.n} = s'.N_{s'.n}$ which implies that $s'.Order1$ holds, and $WellOrdered$ is not falsified by a.

– If a is an execution of p.17: Then since $s.WellOrdered$ holds, we have $s.N_T = s.N_{s.n}$. Since $s.WellLinked$ holds, by Invariant 50 we get that $s'.N_T \neq s.N_T \Rightarrow s'.N_T = s.N_{s.n-1}$, and therefore it is enough to show that $s'.N_T \neq s.N_T$ if and only if a removes the rightmost node from `Ordered` (since the removal of the rightmost node from `Ordered` results in: $s'.n = s.n - 1$, and therefore $s.N_{s.n-1} = s'.N_{s'.n}$). The last is implied immediately by Invariant 53, since the transition removes the rightmost node from `Ordered` if and only if $s.p.currTop = s.\texttt{Top} \wedge s.p.nodeToFree \neq \texttt{NULL}$. Therefore a does not falsify $WellOrdered$.

□

4.7 Section: Linearizability

In this section we show that our implementation is linearizable to a sequential deque. We assume a sequentially consistent shared-memory multiprocessor system.[9] For brevity we will consider only *complete execution histories*:

Definition 56. *A* complete execution history *is an execution in which any operation invocation has a corresponding response (that is, the history does not contain any partial execution of an operation).*

Since we later show that our algorithm is wait-free, linearizability of all complete histories implies linearizability of all histories as well.

The linearizability proof is structured as follows: In Section 4.7.1 we give the sequential specification of a deque, to which our implementation is linearized. In Section 4.7.2 we specify the linearization points, and in Section 4.7.3 we give the proof itself.

4.7.1 The Deque Sequential Definition

The following is the sequential specification of the implemented deque:

1. Deque state: A deque is a sequence of values, called the *deque elements*. We call the two ends of the sequence the *left end* and the *right end* of the sequence.

2. Supported Operations: The deque supports the PushBottom, PopTop and PopBottom operations.

3. Operations' Sequential Specifications:
   The following two operations may be invoked *only* by one process, which we'll refer to as the *deque's owner process*:

[9] In practice, we have implemented our algorithm for machines providing only a weaker memory model, which required insertion of some memory barrier instructions to the code.

- *PushBottom(v):* This operation adds the value v to the left end of the deque, and does not return a value.
- *PopBottom:* If the deque is not empty, then this operation removes the leftmost element in the deque and returns it. Otherwise, it returns EMPTY and does not modify the state of the deque.

The following operation may be invoked by *any process*:

- *PopTop:* This operation can return ABORT, given the rule stated by Property 57; if the operation returns ABORT it does not modify the deque state. Otherwise, if the deque is empty, the operation returns EMPTY and does not modify the state of the deque, and if the deque is not empty, the operation removes the rightmost value from the deque and returns it.

Property 57. *In any sequence of operations on the deque, for any PopTop operation that has returned ABORT, there must be a corresponding Pop operation (i.e., a PopTop or PopBottom operation), which has returned a deque element. For any two different PopTop operations executed by the same process that return ABORT, the corresponding successful Pop operations are different.*

We have permitted the PopTop operation to return ABORT because in practical uses of work-stealing deques, it is sometimes preferable to give up and try stealing from a different deque if there is contention. As we prove later, our algorithm is wait-free. We also show that if the ABORT return value is not allowed (that is, if the PopTop operation retries until it returns either EMPTY or the rightmost element in the deque), then our algorithm is lock-free.

4.7.2 The Linearization Points

Before specifying the linearization points of our algorithm we must define the *Physical Queue Content* (henceforth PQC): a subset of the ordered nodes' cells (Section 4.3, Definition 14), which as we later show, at any given state stores exactly the deque elements.

Definition 58. *The* PQC *is the sequence of cells that lie in the half-open interval* $(C_B \cdots C_T]$ *according to the order* $\prec$.

By the definition of the order $\prec$ (Definition 32), $C \in PQC \Rightarrow Node(C) \in$ `Ordered`. By Corollary 15 $Node(C_B) = N_B \in$ `Ordered` $\wedge\, Node(C_T) = N_T \in$ `Ordered`, and therefore $(C_B \preceq C_T) \vee (C_T \prec C_B)$ holds. Also note that the PQC is empty if and only if $C_T \preceq C_B$. Specifically, the PQC is empty if the deque is in a crossing state (Definition 37).

The following claim is needed for the definition of the linearization points:

Claim 59. *Suppose that an execution of the PopTop operation does not return ABORT or EMPTY. Then the PQC was not empty right after the operation executed Line 9.*

Proof. Suppose that the PopTop operation is executed by process p. Let denote the state right before and right after the execution of $p.x$ as s_x and s'_x respectively. Note that $s_{10}.p.currTop = s'_8.\texttt{Top}$, $s_{10}.p.currTopNode = s'_8.N_T$, and $s_{10}.p.currBottom = s'_9.\texttt{Bottom}$.

1. By examining the code, the PopTop operation does not return ABORT or EMPTY if and only if it executes the CAS operation in Line 17 and this CAS succeeds. Therefore $s_{17}.p.currTop = s_{17}.\texttt{Top}$, which implies by Corollary 12 that for all states s between s'_8 and s_{17}, $s.\texttt{Top} = s'_8.\texttt{Top}$, and therefore $s.C_T = s'_8.C_T \wedge s.N_T = s'_8.N_T$.

2. Specifically $s_{10}.p.currTopNode = s'_8.N_T = s_{10}.N_T$ which implies by Corollary 15 that $s_{10}.p.currTopNode \in s_{10}.\texttt{Ordered}$ and it is not the rightmost node there.

3. Therefore, since the IndicateEmpty call at Line 10 returns false, by Lemma 36: $Cell(s_{10}.p.currBottom) \neq Cell(s_{10}.p.currTop) \wedge$ $Cell(s_{10}.p.currTop) \neq LeftNeighbor_{s_{10}}(Cell(s_{10}.p.currBottom))$.

4. Note that $s'_8.\texttt{Top} = s'_9.\texttt{Top}$. Therefore we can substitute $s_{10}.p.currBottom$ and $s_{10}.p.currTop$ with $s'_9.\texttt{Bottom}$ and $s'_9.\texttt{Top}$ respectively, and get: $s'_9.C_B \neq s'_9.C_T \wedge s'_9.C_B \neq RightNeighbor_{s_{10}}(s'_9.C_T)$.

5. Next we show that $RightNeighbor_{s'_9}(s'_9.C_T) = RightNeighbor_{s_{10}}(s'_9.C_T)$. By Corollary 15, N_T is always in $\texttt{Ordered}$, and is never the rightmost node there. Therefore:

   (a) $RightNeighbor_s(s'_9.C_T)$ is defined for all states s between s'_9 and s_{10} (since $s'_9.C_T = s.C_T$ in these states).

   (b) $s'_9.N_T$ and $Node(RightNeighbor_{s'_9}(s'_9.C_T))$ are both in $s'_9.\texttt{Ordered}$, and are either the same node or adjacent nodes in $s'_9.\texttt{Ordered}$. Also, $s'_9.N_T$ and $Node(RightNeighbor_s(s'_9.C_T))$ are both in $s.\texttt{Ordered}$ for all states s between s'_9 and s_{10}. Therefore, since any transition that modifies $\texttt{Ordered}$ either adds a node to one end or removes a node from one end (but not both), $s'_9.N_T$ and $Node(RightNeighbor_s(s'_9.C_T))$ are both in $s.\texttt{Ordered}$, and are either the same node or adjacent nodes in $s.\texttt{Ordered}$, for all states s between s'_9 and s_{10}. Therefore $RightNeighbor_{s'_9}(s'_9.C_T) = RightNeighbor_{s_{10}}(s'_9.C_T)$.

6. Therefore: $s'_9.C_B \neq s'_9.C_T \wedge s'_9.C_B \neq RightNeighbor_{s_{10}}(s'_9.C_T) = RightNeighbor_{s'_9}(s'_9.C_T)$, and therefore by Conjecture 54 $\neg(s'_9.C_T \preceq s'_9.C_B)$, which implies that the PQC is not empty in s'_9.

□

Definition 60. Linearization Points:

PushBottom *The linearization point of this method is the update of* `Bottom` *at Line 7.*

PopBottom *The linearization point of this method depends on its returned value, as follows:*

- `EMPTY`*: The linearization point here is the read of* `Top` *at Line 26.*
- *A deque entry: The linearization point here is the update of* `Bottom` *at Line 25.*

PopTop *The linearization point of this method depends on its return value, as follows:*

- `EMPTY`*: The linearization point here is the read of* `Bottom` *pointer at Line 9.*
- `ABORT`*: The linearization point here is the statement that first observed the modification of* `Top`*. This is either the CAS operation at Line 17, or the read of* `Top` *at Line 11.*
- *A deque entry: If the PQC was not empty right before the CAS statement at Line 17, then the linearization point is that CAS statement. Otherwise, it is the first statement whose execution modified the PQC to be empty, in the interval after the execution of 9, and right before the execution of the CAS operation at Line 17.*[10]

Claim 61. *The linearization points of the algorithm are well defined. That is, for any PushBottom, PopTop, or PopBottom operation, the linearization point statement is executed between the invocation and response of that operation.*

Proof. By examination of the code, all the linearization points except the one of a PopTop operation that returns a deque entry are well defined, since they are statements that are always executed by the operation being linearized. In the case of a PopTop operation, if the linearization point is the CAS statement, then it is obvious. Otherwise, the PQC was empty right before the execution of this successful CAS operation, and by Claim 59 the PQC was not empty right after the PopTop operation executed Line 9. Therefore there must have been a transition that modified the PQC to be empty in this interval, and this transition corresponds to the linearization point of the PopTop operation. □

[10]Note that the linearization point of the PopTop operation in this case might be the execution of a statement by a process other then the one executing the linearized PopTop operation. The existence of this point is justified in Claim 59.

4.7.3 The Linearizability Proof

In this section we show that our implementation is linearizable to the sequential deque specification given in Section 4.7.1. For this we need several lemmas, including one that shows how the linearization points of the deque operations modify the PQC, and one that shows that the PQC is not modified except at the linearization point of some operation. We begin with the following claim:

Definition 62. *For any Cell C, $Data(C)$ is the value stored in that cell.*

Claim 63. *Consider a transition $s \xrightarrow{a} s'$. Then $C \in s.PQC \Rightarrow s.Data(C) = s'.Data(C)$.*

Proof. If $C \in s.PQC$ then $Node(C) \in s.\texttt{Ordered}$, and therefore by Conjecture 29 $N \in s.\texttt{Live}$. Therefore the content of a cell can be modified only by a statement of a deque operation. The only statement that modifies the data in a cell is Line 2. By Invariant 6 $C = s.C_B$, and therefore $C \notin s.PQC$, a contradiction. □

Lemma 64. *Consider a transition $s \xrightarrow{a} s'$ where a is a statement execution corresponding to a linearization point of a deque operation. Then:*

- *Case 1: If a is the linearization point of a PushBottom(v) operation, then it adds a cell containing v to the left end of the PQC.*
- *Case 2: If a is the linearization point of a PopBottom operation: let R be the operation returned value:*[11]
  - *If R is EMPTY, then the $s.PQC = s'.PQC = \emptyset$.*
  - *If R is a deque entry, then R is stored in the leftmost cell of $s.PQC$, and this cell does not belong to $s'.PQC$.*
- *Case 3: If a is the linearization point of a PopTop operation: let R be the operation return value:*
  - *If R is EMPTY, then the $s.PQC = s'.PQC = \emptyset$.*
  - *If R is a deque entry, then R is stored in the rightmost cell of $s.PQC$, and this cell does not belong to $s'.PQC$.*

Proof. Let p be the process that executes the linearized deque operation, and p' be the process executes a. Note that $p = p'$ unless maybe in the case where the deque operation is a PopTop one (Case 3). The proof proceeds by considering the type of the deque operation for which a is a linearization point.

[11] We can refer to the returned value of an operation since we're dealing only with complete histories.

- Case 1 - PushBottom:

  The value pushed by the PushBottom operation is written in a cell by Line 2, and by Invariant 1 no process $p'' \neq p$ executes Line 2 while p executes the PushBottom operation. Let denote this cell by C. By Invariant 6 $C = C_B$ as long as $p \in \langle 2 \dots 7 \rangle$. Therefore right before the transition $s \xrightarrow{a} s'$ (the execution of Line 7), $C = s.C_B$ which implies that $C \notin s.PQC$, and by Invariant 41 $C = RightNeighbor_{s'}(s'.C_B)$ which implies that C is the leftmost cell in $s'.PQC$. Therefore the transition adds C to the left end of the PQC.

- Case 2 - PopBottom:

  If the PopBottom operation return value is a deque entry: then this value was read from $C = Cell(p.newBotVal)$ (Line 27), and a is the `Bottom` update at Line 25. Since $C = s'.C_B$ then $C \notin s'.PQC$, and since by Invariant 44 $C = RightNeighbor(s.C_B)$, $C \in s.PQC$ unless $s.PQC = \emptyset$. If $s.PQC = \emptyset$ then the deque must be in a crossing state in s', because $[C_T \preceq s.C_B \wedge s'.C_B = RightNeighbor(s.C_B)] \Rightarrow C_T \prec s'.C_B$ (note that C_T and `Ordered` are not modified by a). By Conjecture 54, the deque cannot be in a crossing state when p returns from the PopBottom operation. By Invariant 50, an update of `Top` cannot fix the crossing state. By Corollary 15 C_T and C_B are always in `Ordered`, and therefore $\neg(C_B \prec C_T) \Leftrightarrow C_B \succeq C_T$, which implies by Lemma 33 that a modification of `Ordered` also cannot fix the crossing state. Therefore the transition that fixes the crossing state must be a modification of `Bottom`, and since by Invariant 1 no process $p'' \neq p$ modifies `Bottom` while p is executing the PopBottom operation, then p must modify `Bottom` by executing Line 29 or 36. Therefore the PopBottom operations returns EMPTY, a contradiction.

  If the return value is EMPTY: then a is the `Top` read operation at Line 26. By examining the code, the operation returns EMPTY only if:

  - $Cell(p.newBotVal) = Cell(p.currTop)$, or
  - $Cell(p.oldBotVal) = Cell(p.currTop)$, which by Invariant 44 implies that $p.currTop$ is a left neighbor of $p.newBotVal$.

  Clearly $s'.C_T = s.C_T = s'.p.currTop$ and by Invariant 6 $s'.p.newBotVal = s.p.newBotVal = s.\texttt{Bottom} = s'.\texttt{Bottom}$. Therefore $s.PQC = s'.PQC = \emptyset$.

- Case 3 - PopTop:

  If the value returned by PopTop is a deque entry: then this entry was read from $C = Cell(p.currTop)$ at Line 16. If the PQC was not emptied before the execution of the CAS statement at Line 17, then a is the execution of this CAS. Also, since the CAS was successful, then $p.currTop = s.\texttt{Top}$, and therefore:

1. C is the rightmost cell in $s.PQC$.
2. By Corollary 12, $p.currTop = \texttt{Top}$ while $p \in \langle 9 \dots 17 \rangle$. Therefore, because the PQC was not emptied before the execution of $p.17$, $C \in PQC$ while $p \in \langle 16 \dots 17 \rangle$, and therefore by Claim 63 C still stores the return value in s.
3. By Invariant 50, $C = s.C_T = RightNeighbor(s'.C_T)$, and therefore $C \notin s'.PQC$.

Otherwise if the PQC was emptied before the execution of the CAS, then a is the transition that emptied the PQC. Since the CAS is successful, by Corollary 12 $p.currTop = s.\texttt{Top}$, which implies that C is the rightmost cell in $s.PQC$. Right after a the PQC is empty, and therefore $C \notin s'.PQC$.

If the return value is EMPTY: then a is the execution of Line 9. Because the operation returns EMPTY, it must be the case that p executed Line 11, and that $p.currTop = \texttt{Top}$ at this point. Therefore by Corollary 12, $p.currTop = \texttt{Top}$ while $p \in \langle 9 \dots 11 \rangle$. Specifically, $s'.p.currTop = s'.\texttt{Top} = s.\texttt{Top} \wedge s'.p.currBottom = s'.\texttt{Bottom}$. Because Line 11 is executed, we know that the `IndicateEmpty` macro at Line 10 returned true, which implies by Lemma 36 that $s.PQC = s'.PQC = \emptyset$.

□

For the proof of the second lemma, we need the following claims:

Claim 65. *Let $s \xrightarrow{a} s'$ be an execution of Line 29 or 36. Then $s'.PQC$ is empty.*

Proof. $(s'.PQC = \emptyset) \Leftrightarrow (s'.C_T \preceq s'.C_B)$, and by Lemma 45 $(s'.C_T \preceq s'.C_B) \Leftrightarrow (s.C_T \prec s.C_B)$. Therefore $s'.PQC$ is empty if and only if the deque is in a crossing state in s. Let p be the process executing a.

1. If a is an execution of Line 29: Let denote by s_X and s'_X the pre-state and post-state of the transition executing statement $p.X$, respectively. We first show that the deque is in a crossing state in s'_{26}:
   (a) By Invariant 52, $Cell(s_{29}.p.currTop) = Cell(s_{29}.p.oldBotVal)$.
   (b) $(s'_{26}.p.oldBotVal = s_{29}.p.oldBotVal) \wedge (s'_{26}.p.newBotVal = s_{29}.p.newBotVal)$, and by Invariant 6 and 1, $Cell(s_{29}.p.newBotVal) = s_{29}.C_B = s'_{26}.C_B$.
   (c) By Invariant 44, $Cell(s'_{26}.p.oldBotVal) \prec_{s'_{26}} Cell(s'_{26}.p.newBotVal)$.
   (d) Therefore, $Cell(s_{29}.p.currTop) \prec_{s'_{26}} s'_{26}.C_B$.
   (e) Since $s_{29}.p.currTop = s'_{26}.p.currTop = s'_{26}.\texttt{Top}$, $s'_{26}.C_T \prec_{s'_{26}} s'_{26}.C_B$, which implies that the deque is in crossing state in s'_{26}.

By Corollary 47 C_T is not modified by Line 17 while the deque is in a crossing state, which also implies that `Ordered` is not modified by any statement executed by a concurrent PopTop operation while the deque is in crossing state. By Invariant 1 no other transition modifies `Top`, `Bottom`, or `Ordered` between states s'_{26} and s_{29}, and therefore $s_{29}.C_T \prec_{s_{29}} s_{29}.C_B$, which implies that the deque is in crossing state in $s = s_{29}$.

2. If a is an execution of Line 36: By Invariant 52, $Cell(s.p.currTop) = Cell(s.p.newBotVal)$, and by Invariant 6, $Cell(s.p.newBotVal) = s.C_B$, which implies: $Cell(s.p.currTop) = s.C_B$. By examining the code, p@36 only if the CAS at Line 33 was executed, and failed, which implies that `Top` was modified between its read at Line 26 and the execution of the CAS. Since the only concurrent operations that may modify `Top` are PopTop operations, by Invariant 50 $s''.C_T \prec_{s''} Cell(s.p.currTop) = s.C_B$, where s'' is the post-state of the first transition that modifies `Top` after its read at Line 26. By Invariant 1, $s.C_B = s''.C_B$, which implies that the deque is in crossing state at s''. By Corollary 47 C_T is not modified by Line 17 while the deque is in a crossing state, and therefore `Ordered` is not modified either by any concurrent PopTop operation between the states s'' and s. By Invariant 1 no other process modifies `Ordered` between these states and therefore $s.C_T \prec_s s.C_B$.

□

Claim 66. *Consider a transition $s \xrightarrow{a} s'$. Then: $s.PQC \neq s'.PQC \Rightarrow s.C_T \neq s'.C_T \vee s.C_B \neq s'.C_B$.*

Proof. We prove that the claim holds by showing that: $s.C_T = s'.C_T \wedge s.C_B = s'.C_B \Rightarrow s.PQC = s'.PQC$.

Suppose that $s.C_T = s'.C_T \wedge s.C_B = s'.C_B$. Then by Invariant 55 $s.N_T = s'.N_T$ and $s.N_B = s'.N_B$ are not removed from `Ordered` by a (Note that no single statement both removes and adds a node to `Ordered`, and therefore it cannot be the case that N_B or N_T are removed and then returned to `Ordered`). Therefore, since the `Ordered` sequence only supports removal or addition of nodes to the ends of the sequence, the subsequence of nodes lying between N_B and N_T is not changed by a, which implies by the definitions of the PQC and the $\prec$ operator that $s.PQC = s'.PQC$.

□

Claim 67. *Suppose $s \xrightarrow{a} s'$ is an execution of p.25 that corresponds to a linearization point of a PopTop operation executed by process p'. Then:*

1. *$s'.p'@\langle 10, 12 \ldots 17\rangle$, and*
2. *$p@\langle 26 \ldots 28, 31 \ldots 33\rangle$ is not falsified while $p'@\langle 10, 12 \ldots 17\rangle$ holds.*

Proof. By the definition of the linearization points, a could be a linearization point of a PopTop operation only if it is executed while $p'@\langle 10, 12 \dots 17\rangle$, where p' is the process executing that PopTop operation. Therefore $s'.p'@\langle 10, 12 \dots 17\rangle$ holds. Also, it must be the case that this PopTop operation executes a successful CAS at Line 17 (otherwise a would not be its linearization point), which implies by Corollary 12 that `Top` is not mosified while $p'@\langle 10, 12 \dots 17\rangle$. Finally note that in order for a to be the linearization point of the PopTop operation, a must empty the PQC, which implies by Invariant 44 that $s'.C_B = s'.C_T$.
Right after the execution of p.25, $p@26$ holds. Suppose that $p@\langle 26 \dots 28, 31 \dots 33\rangle$ is falsified while $p'@\langle 10, 12 \dots 17\rangle$ holds:

- If $p@\langle 26 \dots 28, 31 \dots 33\rangle$ is falsified by an execution of p.28: Since a is the linearization point of the PopTop operation and `Top` is not modified while $p'@\langle 10, 12 \dots 17\rangle$, $p.currTop = \texttt{Top} \wedge C_T = C_B$ right before p.28 is executed. By Invariant 6, `Bottom` $= p.newBotVal$ at that point, and by Invariant 44 $p.newBotVal \neq p.oldBotVal$. Therefore $Cell(p.currTop) \neq Cell(p.oldBotVal)$ right before the execution of p.28, which implies that $p@\langle 26 \dots 28, 31 \dots 33\rangle$ is not falsified by that transition.

- Otherwise, $p@\langle 26 \dots 28, 31 \dots 33\rangle$ must be falsified by an execution of p.33. Because `Top` was not modified since the execution of p.25, then $p.currTop =$ `Top` right before p.33 is executed, which implies that the CAS at this line succeeds. But we already showed that no transition modifies `Top` while $p'@\langle 10, 12 \dots 17\rangle$ holds, a contradiction.

□

Definition 68. *A* successful Pop operation *is either a PopTop or a PopBottom operation that did not return EMPTY or ABORT.*

Lemma 69. *Consider a transition $s \xrightarrow{a} s'$ that modifies the PQC (that is, $s.PQC \neq s'.PQC$), then a is the execution of a linearization point of either a PushBottom operation, a PopBottom operation, or a successful PopTop operation.*

Proof. We first show that a either adds or removes a cell from the PQC,[12] but not both: If $s.PQC$ is empty, then it is clear that a can only add a cell to the PQC. Otherwise, by Claim 66 a modification to the PQC is done only by a modification of `Bottom` or `Top`. Since no transition writes them both, a must be a modification of exactly one of them, which implies that a either adds or removes a cell from the PQC.

There are two cases to be considered:

1. If a adds a cell to the PQC: Then a is not the CAS operation at Line 17, since by Invariant 50 this statement can only remove cells from the PQC. By Corollary 43, a cannot be an execution of Line 33 since then: $s'.C_T = s.C_T$.

[12] Note that if a adds more than one cell to the PQC, this statement still holds.

Therefore a must be a modification of `Bottom`, and $s'.C_B \prec s.C_B$ must hold, which implies by Invariant 44 that a is not an execution of Line 25. Therefore a must be the execution of one of three statements: Line 29, Line 36 or Line 7. Finally, by Claim 65 an execution of Line 29 or 36 results in an empty PQC (which contradicts the assumption that a adds a cell to the PQC), which implies that a must be an execution of Line 7. By the linearization points specification (Definition 60), this is the linearization point of the PushBottom operation that executes a.

2. If a removes a cell from the PQC: Then a is either an execution of a successful CAS by a PopTop operation at Line 17, or the modification of `Bottom` by the PopBottom operation at Line 25, because these are the only statements that modify `Bottom` or `Top` while satisfying: $s'.C_B \succ s.C_B \vee s'.C_T \prec s.C_T$ (Lemmas 41, 45 and Invariants 50 and 44).

   (a) *If a is an execution of Line 17:* Since $s'.PQC \neq s.PQC$, and $s'.\texttt{Top} \prec s.\texttt{Top}$, then $s.PQC$ is not empty. Therefore by the linearization points specification (Definition 60), a is the linearization point of the PopTop operation that executes it.

   (b) *If a is an execution of Line 25:* This is the more tricky case, since the `Bottom` update operation may be a linearization point of both a successful PopTop operation and a successful PopBottom operation. We must show then that it is a linearization point of exactly one of them. Let p be the process that executes Line 25. There are three cases to be considered:

   - *If $s'.PQC$ is not empty*, then by the linearization points specification (Definition 60) it cannot be the linearization point of a PopTop operation, and it is the linearization point of the PushBottom operation that executes it as long as this PopBottom operation does not return EMPTY.
     Suppose that the PopBottom operation returns EMPTY. The PushBottom operation returns EMPTY if and only if it executes Line 29 or 36, and by Claim 65 and Lemma 45, the deque is in a crossing state right before executing this line. Since `Bottom` is not modified while $p@\langle 25 \dots 29, 31 \dots 36\rangle$ (Invariant 1), and $s'.PQC$ is not empty, then by Invariant 50 there must be at least two executions of successful CAS in Line 17 after s' but before the execution of Line 29 or 36: one that empties the PQC, and the other that results in the crossing state.
     Let p' be the process executing the successful CAS operation that empties the PQC, and p'' be the process executing the CAS operation that results in the crossing state (it might be that $p'' = p'$). Note that $p'.17$ is executed before $p''.17$, and both are executed after s' but before p modifies `Bottom` at Line 29 or 36. Since both

CAS are successful, then by Corollary 12, $p''.8$ was executed after $p'.17$. Since `Bottom` is not modified until the execution of $p''.17$, then $Cell(p''.currTop) = Cell(p''.currBottom)$ when $p''.10$ is executed (the execution of the `InidicateEmpty` macro by p''), which implies by Lemma 36 that the macro returns true, and the CAS at Line 17 is not executed: a contradiction. Therefore the PopBottom operation does not return EMPTY, and a is the linearization point of that operation.

- *If $s'.PQC$ is empty, and `Top` is not modified by a process $p' \neq p$ while $p@\langle 26 \dots 33 \rangle$:* Since a empties the PQC, $s'.C_T = s'.C_B$. Since `Top` is not modified, right before executing $p.31$ $p.currTop =$ `Top` and by Invariant 6 $p.newBotVal =$ `Bottom`, which implies that the test in Line 31 succeeds. Therefore the CAS in Line 33 is executed, and since `Top` was not modified it is successful. Therefore the PopBottom operation does not return EMPTY, which implies that a is its linearization point. It is left to show then that a is not the linearization point of a concurrent PopTop operation.

  Note that a might be a linearization point of a concurrent PopTop operation executed by $p' \neq p$ only if $s.p'@\langle 10 \dots 17 \rangle$, which implies that $p'.currTop \neq$ `Top` right after the successful CAS operation at Line 33. Therefore p' fails the CAS at Line 17, which implies that a is not the linearization point of this PopTop operation.
- *Otherwise,* Let $p' \neq p$ be the process that is the first to modify `Top` before the PopBottom operation terminates. Then it must be an execution of $p'.17$ that modifies `Top` and since $s'.PQC$ is empty, then by Invariant 50 the deque is in a crossing state right after $p'.17$ is executed. By Conjecture 54 the deque is not in a crossing state if $\neg p@\langle 26 \dots 29, 31 \dots 33, 36 \rangle$, and therefore p must modify `Bottom` before it leaves this interval. This implies that $p.29$ or $p.36$ are executed, and therefore the PopBottom operation returns EMPTY. Therefore a is not a linearization point of the PopBottom operation.

  It is left to show that a is the linearization point of the PopTop executed by p'. Since the PQC was empty right before the execution of $p'.17$, then the PopTop operation is linearized on *some* execution of Line 25 that empties the PQC. By Claim 67 and Invariant 1, this execution must be a, because if it is an execution of Line 25 by another PopBottom operation, by Claim 67 that PopBottom operation could not have been terminated before $p'.17$ is executed, which contradicts the assumption that $p'.17$ is executed after $s \xrightarrow{a} s'$ takes place.

□

The following lemma shows that the ABORT property holds:

Lemma 70. *In any complete execution history, for any PopTop operation that has returned ABORT, there is a corresponding Pop operation (that is, a PopTop or PopBottom operation), which has returned a deque element. For any two different PopTop operations executed by the same process that returned ABORT, the corresponding successful Pop operations are different.*

Proof. Let p be a process that executes a PopTop operation that returned ABORT. By examining the code, the PopTop operation returns ABORT only if `Top` has been modified by another process after p read it at Line 8. There are only two possibilities: either it was modified by the CAS of a concurrent PopTop operation, or by the CAS of a concurrent PopBottom operation. In both case the concurrent Pop operation returns a deque element.

For any subsequent PopTop operation by the same process that returns ABORT, we know that `Top` underwent another modification, since the first modification took place before the invocation of the subsequent PopTop operation (and specifically before it read `Top` at Line 8). Since no deque operation modifies `Top` more than once, it follows that different successful Pop operations are responsible for these two modifications of `Top`. □

The Linearizablity Theorem: Using Lemmas 64, 69, and 70, we now show that our implementation is linearizable to the sequential deque specification given in Section 4.7.1:

Theorem 71. *Any complete execution history of our algorithm is linearizable to the sequential deque specification given in Section 4.7.1.*

Proof. Given an arbitrary complete execution history of the algorithm, we construct a total order of the deque operations by ordering them in the order of their linearization points. By Claim 61, each operation's linearization point occurs after it is invoked and before it returns, and therefore the total order defined respects the partial order in the concurrent execution.

It is left to show that the sequence of operations in that total order respects the sequential specification given in Section 4.7.1. We begin with some notation. For each state s in the execution, we assign an *abstract deque state*, which is achieved by starting with an empty abstract deque, and applying to it all of the operations whose linearization points occurs before s in the order in which they occur in the execution.

We say that the PQC sequence *matches the abstract deque sequence* in a state s, if and only if the length of the abstract deque state and the length of the PQC (denoted *length(PQC)*) are equal, and for all $i \in [0..length(PQC))$, the data stored in the ith cell of the PQC sequence is the ith deque element in the abstract deque state.

We now show that in any state s of the execution, the PQC matches the abstract deque sequence in s:

1. Both sequences are empty at the beginning of the execution.

2. By Lemma 69, any transition that modifies the PQC is the linearization point of a successful PushBottom, PopBottom, or PopTop operation, and therefore it also modifies the abstract deque state.

3. By Lemma 64, the linearization point of a PushBottom operation adds a cell containing the pushed element to the left end of the PQC, the linearization point of a successful PopBottom operation removes the cell containing the popped element from the left end of the PQC, and the linearization point of a successful PopTop operation removes the cell containing the popped element from the right end of the PQC.

Therefore by induction on the length of the execution, and the abstract deque operation specification given in Section 4.7.1, the PQC sequence matches the abstract deque sequence in any state s of the execution.

By Lemma 70 the ABORT property holds. By Lemma 64 a PopBottom operation returns the leftmost value in the PQC if it is not empty or EMPTY otherwise, and if the PopTop operation does not return ABORT, then it returns the rightmost value in the PQC if it is not empty, or EMPTY otherwise. Therefore, since the PQC sequence matches the abstract deque sequence, the operations return the correct values according to the sequential specification given in Section 4.7.1, which implies that our implementation is linearizable to this sequential specification.

□

4.8 The Progress Properties

Theorem 72. *Our deque implementation is wait-free.*

Proof. Our implementation of the deque does not contain any loops, and therefore each operation must eventually complete. □

The reason our algorithm is wait-free is that we have defined the ABORT return value as a legitimate one. However, in many cases we may want to keep executing the PopTop operation until we gets either a deque element or the EMPTY return value. The following theorem shows that our implementation is lock-free even if the PopTop operation is executed until it returns such a value.

Definition 73. *A* legitimate value *returned by a Pop operation is either a deque element or EMPTY.*

Theorem 74. *Our deque implementation, where the PopTop operation retries until it returns a legitimate value, is lock-free.*

Proof. The Abort property proven by Lemma 70, implies that every two different PopTop operations by the same process that returned ABORT have two different Pop operations that returned deque elements. Thus if a PopTop operation infinitely retries and keep returning ABORT, there must be an infinite number of Pop operations that returned a legitimate value. Therefore if a PopTop operation fails to complete, there must be an infinite number of a successful Pop operations. □

Theorem 75. *Our algorithm is a lock-free implementation of a linearizable deque, as defined by the sequential specification in Section 4.7.1.*

Proof. Theorem 71 states that our implementation is linearizable to the sequential specification given in Section 4.7.1. Theorem 74 showed that the implementation is lock-free. □

5 Conclusions

We have shown how to create a dynamic memory version of the ABP work-stealing algorithm, by implementing the work-stealing deque as a linked list of short arrays. In a more recent work, Chase and Lev [19] present another dynamic work-stealing algorithm, that uses array-based deque that can grow if overflowed. It may be interesting to see how these techniques can be applied to other schemes that improve on ABP-work-stealing, such as the locality-guided work-stealing of Blelloch et. al. [4] or the steal-half algorithm of Hendler and Shavit [9].

References

[1] Lev, Y.: A Dynamic-Sized Nonblocking Work Stealing Deque. MS thesis, Tel-Aviv University, Tel-Aviv, Israel (2004)

[2] Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems **34** (2001) 115–144

[3] Rudolph, L., Slivkin-Allalouf, M., Upfal, E.: A simple load balancing scheme for task allocation in parallel machines. In: In Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, ACM Press (1991) 237–245

[4] Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: ACM Symposium on Parallel Algorithms and Architectures. (2000) 1–12

[5] Flood, C., Detlefs, D., Shavit, N., Zhang, C.: Parallel garbage collection for shared memory multiprocessors. In: Usenix Java Virtual Machine Research and Technology Symposium (JVM '01), Monterey, CA (2001)

[6] Leiserson, Plaat: Programming parallel applications in cilk. SINEWS: SIAM News **31** (1998)

[7] Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. Journal of the ACM **46** (1999) 720–748

[8] Knuth, D.: The Art of Computer Programming: Fundamental Algorithms. 2nd edn. Addison-Wesley (1968)

[9] Hendler, D., Shavit, N.: Non-blocking steal-half work queues. In: Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing. (2002)

[10] Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. Technical report, Sun Microsystems – Sun Laboratories (2004) To appear.

[11] Agesen, O., Detlefs, D., Flood, C., Garthwaite, A., Martin, P., Moir, M., Shavit, N., Steele, G.: DCAS-based concurrent deques. Theory of Computing Systems **35** (2002) 349–386

[12] Martin, P., Moir, M., Steele, G.: DCAS-based concurrent deques supporting bulk allocation. Technical Report TR-2002-111, Sun Microsystems Laboratories (2002)

[13] Greenwald, M.B., Cheriton, D.R.: The synergy between non-blocking synchronization and operating system structure. In: 2nd Symposium on Operating Systems Design and Implementation. (1996) 123–136 Seattle, WA.

[14] Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments (extended abstract). In: Measurement and Modeling of Computer Systems. (1998) 266–267

[15] Arnold, J.M., Buell, D.A., Davis, E.G.: Splash 2. In: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, ACM Press (1992) 316–322

[16] Papadopoulos, D.: Hood: A user-level thread library for multiprogrammed multiprocessors. In: Master's thesis, Department of Computer Sciences, University of Texas at Austin. (1998)

[17] Prakash, S., Lee, Y., Johnson, T.: A non-blocking algorithm for shared queues using compare-and-swap. IEEE Transactions on Computers **43** (1994) 548–559

[18] Scott, M.L.: Personal communication: Code for a lock-free memory management pool (2003)

[19] Chase, D., Lev, Y.: Dynamic circular work-stealing deque. In: SPAA'05: Proceedings of the 17th annual ACM Symposium on Parallelism in Algorithms and Architectures, New York, NY, USA, ACM Press (2005) 21–28

About the Authors

Danny Hendler has returned to Academy in 2001 after working for 18 years in the Israeli High-tech industry. He has served in both technical and managerial positions in companies such as Elscint, Telrad, National Semiconductor and Sun Microsystems. He is currently a post-doctoral fellow in the theory group of the department of computer science in the university of Toronto. His current research focuses on lower bounds and algorithms for distributed systems and dynamic load balancing.

Yossi Lev received a B.Sc. degree in Computer Science and Math from Tel Aviv University, Israel in 1999. After a few years in the Israeli High-tech industry, he returned to the Academy and received a MSc. degree also in Computer Science from Tel Aviv University in 2004. Right now he is a PhD. student in Brown University, and an intern in the Scalable Synchronization Research Group in Sun Microsystems Laboratories. His current research interests include design and implementation of scalable concurrent data-structures and infrastructures, and various aspects of the transactional memory infrastructure in concurrent software design.

Mark Moir received the B.Sc. (Hons.) degree in Computer Science from Victoria University of Wellington, New Zealand in 1988, and the Ph.D. degree in Computer Science from the University of North Carolina at Chapel Hill, USA in 1996. From August 1996 until June 2000, he was an assistant professor in the Department of Computer Science at the University of Pittsburgh. In June 2000, he joined Sun Microsystems Laboratories, where he is now the Principal Investigator of the Scalable Synchronization Research Group.

Nir Shavit is a member of the Scalable Synchronization Research Group. He received an B.A. and M.Sc. from the Technion and a Ph.D. from the Hebrew University, all in Computer Science. He was a Postdoctoral Researcher at IBM Almaden Research Center, Stanford University, and MIT, and a Visiting Professor at MIT. He is currently on leave from Tel Aviv University. He is the recipient of the Israeli Industry Research Prize and the 1993 and the ACM/EATCS Gödel Prize in Theoretical Computer Science in 2004. His research interests include hardware and software aspects of Multiprocessor Synchronization, the design and implementation of Concurrent Data-Structures, and the Theoretical Foundations of Asynchronous Computability.