Expander: Lock-free Cache for a Concurrent Data Structure

Pooja Aggarwal
IBM Research Labs, India
aggarwal.pooja@in.ibm.com

Smruti R. Sarangi
Indian Institute of Technology, Delhi
srsarangi@cse.iitd.ac.in

***Abstract*—Parallel programming models and paradigms are increasingly becoming more expressive with a steady increase in the number of cores that can be placed on a single chip. Concurrent data structures for shared memory parallel programs are now being used in operating systems, middle-ware, and device drivers. In such a shared memory model, processes communicate and synchronize by applying primitive operations on memory words. To implement concurrent data structures that are linearizable and possibly lock-free or wait-free, it is often necessary to add additional information to memory words in a data structure. This additional information can range from a single bit to multiple bits that typically represent thread ids, request ids, timestamps, and other application dependent fields. Since most processors can perform compare-And-Set (CAS) or load-link/store-conditional (LL/SC) operations on only 64 bits at a time, current approaches either use some bits in a memory word to pack additional information (*packing*), or use the bits to store a pointer to an object that contains additional information (*redirection*), and the original data item.**

The former approach restricts the number of bits for each additional field and this reduces the range of the field, and the latter approach is wasteful in terms of space. We propose a novel and universal method called a memory word expander in this paper. It caches information for a set of memory locations that need to be augmented with additional information. It supports traditional atomic get, set, and CAS operations, and tries to maintain state for a minimum number of entries. We experimentally demonstrate that it is possible to reduce the runtime memory footprint by 20-35% for algorithms that use redirection. For algorithms that use packing, the use of the EXPANDER can make them feasible. The performance overhead is within 2-13% for 32 threads. When we compare the performance of the EXPANDER based non-blocking algorithms with the version that uses locks, we have a performance gain of at least 10-100X.

I. Introduction

Concurrent data structures for shared memory parallel programs are now being used in operating systems, middleware, and device drivers. In such a shared memory model, processes communicate and synchronize by applying primitive operations on memory words. Each process is assumed to run at an arbitrary speed, and might be subject to arbitrarily long delays. This significantly complicates the task of designing and verifying correct concurrent data structures.

To implement concurrent non-blocking data structures, it is often necessary to add additional information to memory words in a data structure. This additional information can range from a single bit to multiple bits that represent the id of the owner thread, time-stamp, and other application dependent fields. There are many implementations [1]–[5], which store the ownerId within the memory word itself. Occasionally, a timestamp field is stored to avoid ABA issues [6] (see [7]–[11]). This additional information is temporary. Once the operation is over, typically only the final value of the memory word is required.

Since most processors can perform CAS or LL/SC operations on only 64 bits at a time, current approaches either assign a few bits to *pack* additional information in a single memory word, or use a pointer to an object. We refer to the former approach as *packing*. The second method uses the bits to store a pointer to an object that contains the original data item along with additional information. We refer to this approach as *redirection*. The former approach, *packing*, has two major drawbacks. First, it restricts the number of bits for each additional field and thus reduces the range of the field (reduces scalability to larger systems). It also restricts the space available for the original data item such as an entry in a queue or a stack. It might not always be possible to reduce the size of the data. In this case, the second approach is used, which is wasteful in terms of space because we allot space for temporary fields even when they might not be used. The memory overhead in case of lock-free algorithms is a well know problem [12]–[14]. Sagonas et al. [15] observe that reduction in the memory footprint of concurrent programs is key to efficient memory management. We try to solve this problem by reducing the space wastage associated with temporary fields (20-35%).

Our aim in this paper is to provide a generic(universal) method to implement concurrent data structures without performing packing or redirection. The main insight that we use is that after an operation on a concurrent data structure finishes, the memory words contain the final values, and typically do not require the additional information that is packed into them. If we can treat the memory words that are currently being used in a special way, then we can avoid both packing and redirection. We propose a novel data structure called an EXPANDER that caches the set of memory locations that are currently being used. All the accesses to these memory locations go through the EXPANDER. It can pack an arbitrary number of fields, and can simulate atomic operations (examples: get, set, and compare-and-set(CAS)). It provides an illusion to the programmer that a memory word consists of a large number of "packed" fields without complicating the programming model. Once the operation on a data structure is over, we can typically discard the temporary information. At this point the contents of the memory location can be removed from the EXPANDER. The EXPANDER is a linearizable and lock-free structure, and works in user space. It can be either used as a library or can be implemented by the compiler/JVM (in this paper, we implement the EXPANDER as a library).

Most changes to the code are very simple: simply replace an atomic operation by a call to the EXPANDER (see Section VI for a reference implementation of a wait-free queue). The percentage of atomic instructions per se as compared to non-atomic instructions is typically less than 1% in most parallel programs [16]. Hence, we can afford to slow them down quite a bit, if the gains are commensurate. We show that with such simple transformations (i.e modifying $< 1\%$ instructions), the EXPANDER can reduce the memory footprint (by 20-35%) (Section VIII), and eliminate the need for packing fields without a significant drop in performance. For algorithms that use packing, we make them feasible for large systems with 100s of cores (e.g:Intel MIC type processors), and for algorithms that use redirection we significantly alleviate the pressure on the caches by reducing the memory footprint by a fifth to a third. All of this is achieved with a 2-13% drop in performance for 32 threads. This is **miniscule**, when we consider the fact that non-blocking implementations are several orders of magnitude faster than blocking implementations.

II. Related Work

Let us now outline the advantages and disadvantages of a memory word expander with respect to packing and redirection. Packing places very strict limitations on the size of the temporary fields such as the thread id, request id, and the original data item. With an EXPANDER we avoid these limitations. An approach that uses redirection wastes space in storing the temporary variables in a data structure that are seldom used. We avoid this.

Harris et al. [17] proposed a method that is a combination of packing and redirection. A memory word has 2 bits to indicate the status of the rest of the bits. If the first bit is 1, the last $n-2$ bits are a pointer to an object that contains the data item along with temporary fields. On the other hand, the last $n-2$ bits contain the data item if the first bit is 0. In languages such as Java, it is not possible to implement this scheme because Java does not allow users to modify pointers. The EXPANDER does not have this limitation.

There is a vast amount of literature on concurrent data structures and algorithms with a variety of progress guarantees and correctness conditions. Table I lists the methods and additional fields used by some of the highly cited papers on non-blocking algorithms and data structures. However, we are not aware of any work that is similar to the scheme that we propose in this paper. The notion of caching a part of a data structure has been used in universal constructions [10]. However, they construct a per-thread private cache and later update the global data structure. In comparison our approach proposes a cache that is global.

Algorithm	Type	Temporary Fields
Wait-free multi-word CAS [4]	P	index, thread id, descriptor
Universal construction [1]	P	thread id, valid bit, count
CAS and LL/SC [2]	P	thread id
Wait-free multi-object operations [18]	R	parent id, operation id, lock, value
Universal construction [19]	P	unique id(timestamp)
Multiword CAS [17]	P	2 bits to indicate the state of the rest of the bits
LLX/SCX primitives [20]	R	pointer to record and a marked bit
Lock free hashtable [8],linked list, queue [10]	P	counter/tag to avoid ABA issues
Wait-free queue [21]	R	enqueue Id, dequeue Id
Wait-free priority queue [22]	R	value, type, freeze
Wait-free linked list [23]	R	mark bit and a success bit
Wait-free slot scheduler [24]	P	request id, thread id, round, timestamp, slot number, state

Table I: Packing (P) and Redirection (R) in concurrent data structures

A. Example

Let us consider the multi-word compare-and-set (MCAS) operation. In its quintessential form, it requires an array, $addr$, of n addresses, and two other n element arrays: old and new. If $\forall i, *addr[i] == old[i]$, then we set $\forall j, *addr[j] = new[j]$. Let us consider a non-blocking implementation. Assume that $n = 5$, thread t_i has scanned all the locations, and it has found the values in memory to be equal to the old values. It then starts writing the new values (using write or CAS operations). Assume that after writing the first 4 values, another thread t_j comes and modifies the 5^{th} value. $t_i's$ operation cannot complete. It will need to roll back the first 4 writes. However, it is possible that another thread t_k might have read some of the values that t_i wrote and later rolled back. We cannot allow thread t_k to alter its behavior on seeing the status of an un-linearized request. To avoid such complexities, most implementations of MCAS typically set a temporary lock on a word once it has been read, and before it has been modified. This stops other concurrent accesses from modifying the word. We thus require to pack at least 1 bit. However, other concurrent accesses cannot wait indefinitely if we need to provide lock-free or wait-free guarantees. To support such progress conditions, it is often necessary to pack a thread id with each word to identify the owner of the memory word. In our example, thread t_j will typically have two options: help thread t_i in completing its request, or cancel thread t_i's request and move forward with its own request. Both the schemes have been used in prior work [25]. Once an operation is over, we do not need the lock bit (assumed to be 0), or the owner id (irrelevant). Hence, the EXPANDER stops allocating space for them; instead it assumes that they have default values. Aggarwal et al. [24] considered a more advanced version of the MCAS operation where the problem is to reserve a set of M slots in consecutive columns of a large slot matrix. For their wait-free implementation they required to pack 59 bits: $state$ (2 bits), tid(thread id) (10 bits), $slotNum$ (6 bits), $round$ (5 bits), $requestId$ (15 bits) and $timestamp$ (21 bits). Packing a large number of fields into a memory word that contains data values is typical of most sophisticated lock-free and wait-free algorithms.

III. Overview

There are two kinds of accesses made to the EXPANDER. The first kind comprises of get, set, and CAS (compare-And-Set) operations on memory words. These methods internally allocate an entry in the EXPANDER using the $lookupAlloc$ method. After the high level operation is done (eg: queue enqueue/dequeue), the memory entry in the EXPANDER needs to be explicitly freed (second type). Note that it is easily possible to support many more types of operations such as swap, LL/SC, and fetch&inc. Also note that we use Java for our implementation in this paper (C/C++ can also be used).

Internally, the EXPANDER consists of a lock-free list-based hash table as shown in Figure 1. We resolve collisions by chaining (each entry has a linked list). Based on the hash function, a memory word in the baseline data structure (such as a concurrent list or queue) is mapped to one of the buckets of the hash table. Operations on different hash buckets are inherently disjoint and can proceed without interference. Whenever there is a write request for a memory location in the baseline data structure and additional fields are associated with that memory word, a node is added to the EXPANDER in one of the buckets. Subsequent read or write requests for that location are serviced through the EXPANDER. A single memory word can be expanded to contain an arbitrary number of temporary fields. This value is specified by the user depending on the application. The EXPANDER is dynamic, allowing the hashtable and additional memory used to grow and shrink arbitrarily.

A. Data Structures

Figure 1 shows the high level design of the EXPANDER. We have an array $listHead$ that contains $numSets$ entries. Each entry of the array $listHead$ points to a linked list. Each node of this linked list is of type $MemCell$, and is uniquely identified by the field $memIndex$(also acts as the hash key).

In the class $MemCell$, the $next$ field maintains an atomic reference to the next element in the list along with a timestamp. The timestamp field can be updated atomically and is a combination of $stamp$ and $mark$ (MSB of timestamp). The stamp field is used to implement the notion of a version for the $next$ pointer. This strategy avoids ABA problems. The $mark$ field is used to indicate whether a node is marked for deletion or not. The $dataState$ field is a pointer to an instance of the $DataState$ class, which contains three fields: contents of the memory word ($DataType\ data$, also referred to as *data item*), temporary fields ($TmpType[]\ tmpFields$), and an atomic integer, $versionState$. The $versionState$ field is a combination of the data $version$ and $state$. Here, $version$ acts as a time stamp. There are different ways of packing the version and the state in the $MemCell$ data structure. We chose the fastest implementation (determined experimentally). Additionally, the $MemCell$ class supports two static methods ($getState$ and $getVersion$) to retrieve the state and version from $versionState$.

The $state$ field (of $versionState$) represents one of the four states of a node in the linked list. The states are **CLEAN**, **DIRTY**, **WRITEBACK** and **FLUSH**. By default each instance of $MemCell$ is created in the **CLEAN** state. It means that the contents in the EXPANDER are the same as that in the baseline data structure. **DIRTY** means that some write operation has taken place on this node. **WRITEBACK** means that a thread is trying to copy the contents of a node to the baseline data structure. **FLUSH** means that write back is complete and now the node can be deleted from the EXPANDER. Figure 2 shows how each entry of the EXPANDER moves from one state to the next. The input to all these functions is an implementation of the interface, $ExpNode$ (see Figure 3). This interface provides two methods ($getData$ and $setData$) for reading and writing a memory word irrespective of its type. Additionally, the $ExpNode$ interface makes it mandatory to provide an implementation of the function, $hash()$, which uniquely identifies the encapsulated memory word. It can either be the encapsulating object's default hashcode (natively supported) or a custom value such as an array index in the case of the lock-free multi-word compare-And-Set algorithm (see Section II-A). The result of this $hash()$ function is saved in the $memIndex$ field of $MemCell$ to uniquely identify the node in the EXPANDER. The operations provided by EXPANDER are shown in Figure 4.

IV. Algorithm

Here, we describe the implementation of the EXPANDER. All our algorithms are linearizable and lock free.

A. lookUpAlloc()

We use this method to search for a node ($MemCell$) with a given key, $memIndex$, in the EXPANDER. If the node is not present, then this method creates a node corresponding to $memIndex$ with default fields: $state$ as **CLEAN** and stamp as 0. Now, a node is added to the EXPANDER only when we have a write operation (kSet or kCAS). We ensure this by calling the $lookUpAlloc$ method only upon receiving a write request. The input to this method is an argument of type $ExpNode$ corresponding to a memory word. We first calculate the hash key ($memIndex$ = $expNode.hash()$). Then, we search for a node in the EXPANDER with its key as $memIndex$. If we do not find a matching node, then it is necessary to create a new node for the memory word

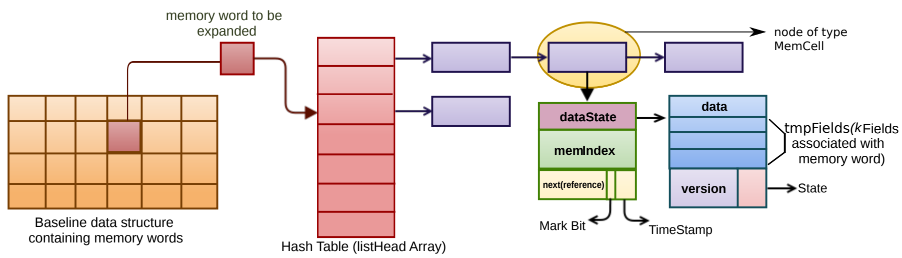

Figure 1: High level design of the EXPANDER

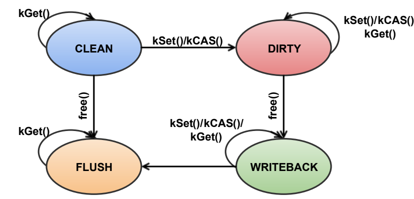

Figure 2: FSM of an EXPANDER's node

```
class Expander<DataType, TmpType> {
  AtomicReferenceArray<MemCell> listHead; int numSets;

  class MemCell {
  /* stores the result of the hash() method */
    final int memIndex;
    AtomicReference<DataState> dataState;
    AtomicStampedReference<MemCell> next;
  }

  public class DataState { /* data + state */
    DataType data; TmpType[] tmpFields;
    AtomicInteger versionState;
  }
}
/* DataType same as that used in the Expander */
interface ExpNode<DataType> {
    DataType getData(); void setData(DataType);
int hash();
}
```

Figure 3: Types and structures

- kGet :- Returns the values (including temporary fields) stored in the EXPANDER for a particular memory word.
- kCAS :- The expanded values corresponding to a single memory word are read from the EXPANDER and are compared against a set of values, and if all of them match, the expanded values are updated atomically.
- kSet :- The values stored in a node inside the EXPANDER are updated atomically.
- free :- The node corresponding to a given memory word in the EXPANDER is deleted.

Figure 4: The basic operations provided by the EXPANDER

and insert it in to the EXPANDER. The implementation is this method is similar to the add method of the lock-free linked list described in [10].

B. kGet()

Similar to $lookUpAlloc$, this method also takes an $expNode$ as the single argument. It returns the data item and the values of temporary fields (return type: $DataState$). In the function, if a node corresponding to the key $memIndex$ is found (Line 3) in the EXPANDER, then the contents of the data item and temporary fields are returned (Line 5). Otherwise, we read the value from the baseline data structure, using $expNode$'s $getData$ method. In both the scenarios, the value associated with the key, $memIndex$, is returned (Line 10). To avoid unnecessary memory allocations, a node is not added to the EXPANDER in the case of a read operation ($kGet()$). Threads can directly read the contents from the baseline data structure itself.

C. kCAS()

The $kCAS()$ method has the same functionality as the atomic instruction $compareAndSet$ (CAS) and is similar to the $kSet$ operation. It compares a set of values corresponding to a particular memory word (with key $memIndex$) with a specified set of values. If the stored values match, it updates all the values atomically.

We first call $lookUpAlloc$ in Line 15 to find/allocate a node ($MemCell$). Then, we read the version number of the data and the state in Lines 16–17. Both of these are packed in the same word, $versionState$. If the state of the node is **FLUSH** then we call the $helpDel()$ method to delete the node from the EXPANDER (Line 36). Otherwise, we compare the stored and old data items and temporary fields in Line 19 and if all of them match, we proceed to update the contents of the node. However, if any of the values fail to match, then we return *false* in Line 20. For updating the contents of a node, we set the state to **DIRTY** if the state is **CLEAN** (Line 22), and then proceed to perform a CAS on the $node.dataState$ field in Line 29. Note that in this case, we change the version of the word. We assume a function, $newVersion$, that returns a unique version number. It can be implemented with a $fetchAndIncrement$ call, or by using a thread specific counter. In the latter case, the version is a combination of the thread id, and the local count of the thread. In case the state of the node is **WRITEBACK**, the state remains the same and only the version is updated. This call can return false only in the case of concurrent writes, CAS, or remove operations. In this case a thread tries again (Line 32).

D. kSet()

The $kSet()$ method has the same functionality as the atomic instruction Set and the implementation is similar to the $kCAS$ operation. Due to the lack of space we have not shown the algorithm in the main paper.

```
1  kGet(expNode)
2  memIndex ← expNode.hash()
3  (node, prev) ← lookUp(expNode)
4  if node.memIndex = memIndex then
5      dataState ←
       node.dataState.get()
6      return dataState
7  end
8  else
9      /* The value is not in the
       EXPANDER so return the value
       saved in the baseline data
       structure */
10     data ← expNode.getData()
11     return new DataState(data, null)
       /* null means no temporary
       field is associated with this
       memory word */
12 end
13 kCAS(expNode, oldData,
   oldValues[], newData, newValues[])
14 while true do
15     node ← lookUpAlloc(expNode)
16     dataState ← node.dataState.get()
17     nodeState ←
       MemCell.getState(dataState.
       versionState.get())
18     if nodeState ≠ FLUSH then
19         /* Check if (oldData,
           oldValues) =
           (dataState.data,
           dataState.tmpFields) */
20         /* if any one of the values
           differ then return false */
21         if nodeState = CLEAN
           then
22             newState ← (DIRTY,
               newVersion())
23         end
24         else
25             newState ←
               (nodeState,
               newVersion())
26         end
27         /* Try to atomically update
           the EXPANDER with a
           dataState containing the
           updated values */
28         newDataState ← new
           DataState(newData,
           newValues, newState)
29         if (node.dataState.CAS
           (dataState, newDataState)
           then
30             return true
31         end
32         else
33             continue
34         end
35     end
36     helpDel(node)
37 end
```

```
38 free(expNode)
39 memIndex ← expNode.hash()
40 (node, pred) ← lookUp(memIndex)
41 if node.memIndex ≠ memIndex then
42     return
43 end
44 while true do
45     dataState ← node.dataState.get()
46     oldValue ←
       dataState.versionState.get()
47     version ←
       MemCell.getVersion(oldValue)
48     state ←
       MemCell.getState(oldValue)
49     if state == WRITEBACK ||
       state == FLUSH then
50         return false /* Some other
           thread is deleting the node
           */
51     end
52     newState ← packCell
       (WRITEBACK, version)
53     newDataState ← new
       DataState(dataState.data,
       dataState.tmpFields, newState)
54     res ←
       node.dataState.compareAndSet
55     (dataState, newDataState)
56     if res == true then
57         break
58     end
59 end
60 /* Only one thread succeeds in
   updating the state to WRITEBACK.*/
61 while true do
62     expNode.setData(dataState.data)
63     dataState ← node.dataState.get()
64     oldState ←
       dataState.versionState.get()
65     version ←
       MemCell.getVersion(oldState)
66     newState ← packCell (FLUSH,
       version)
67     newDataState← new
       DataState(dataState.data,
       dataState.tmpFields, newState)
68     res ←
       node.dataState.compareAndSet
69     (dataState, newDataState)
70     if res = false then
71         continue/* There is a
           concurrent
           kSet()/kCAS() going on
           at memIndex */
72     end
73     else
74         break
75     end
76 end
77 helpDel(node)
```

E. *free()*

Once an entry is ready to be removed from the EXPANDER at the end of the high level operation, we need to call the $free$ method. This method can either be user initiated or compiler initiated. Note that it is possible for concurrent $free$ calls to remove the same word. This method is the most complex and intricate in our set of algorithms. However, this method will be called far more infrequently than get, set, and CAS methods, and thus the additional complexity is not expected to affect performance significantly.

The $free$ method has two phases: (1) write the value stored in the EXPANDER to the baseline data structure, and (2) remove the entry from the EXPANDER. The first phase needs to be performed by only one thread. We were not able to accommodate helpers in this phase, because there is no way to ensure that only one thread updates the value of a memory word in the baseline data structure. Multiple helpers can suffer variable delays and thus can possibly corrupt the state of the memory word. Let us now assume that there are two threads that want to remove the same memory word. Both of them cannot enter phase 1. If one of them enters phase 1, then the other thread needs to return **false** if it tries to enter phase 1.

To ensure this exclusivity, we first atomically update the state of the node, mapped to the key $memIndex$, to **WRITEBACK** (Line 55). For only one thread res is $true$ and for the rest of the threads, we return **false** (Line 49). This indicates that some other request is performing $free$ for the node with the key $memIndex$. Now, that a thread has entered phase 1, it proceeds to write the value to the baseline data structure and atomically updates the state of the node to **FLUSH** (Line 66). This is done as a part of a loop (Lines 61 - 75) since it is possible that other writes are in progress due to which the version of the node can get updated and result in a failed $compareAndSet$ call (Line 69). The important point to note is that it is only one thread's responsibility to atomically set the state to **FLUSH** and update the value of the baseline data structure corresponding to the key $memIndex$. After the state has been set to **FLUSH** no other thread can do any updates ($kSet$ or $kCAS$ operations). Threads then invoke the $helpDel$ method to remove the entry from the EXPANDER (Lines 78–91). This method admits helpers. We first logically delete the node from the EXPANDER by setting the mark bit in the stamp field (Line 82 and Line 83). Next, we try to physically remove the node from the EXPANDER (Line 89).

We prove in Section VII that as long as the $free$ method is called a bounded number of times per high level operation (such as enqueue/dequeue), there are no changes to our claims about correctness, linearizability, and progress guarantees. A programmer simply needs to call it at least once when she feels that the memory word will not be actively used any more. This can also be done automatically by a sophisticated compiler or garbage collector.

V. Usage with Wait-Free Algorithms

The EXPANDER is a lock-free data structure. If we use it in a wait-free algorithm it will render the latter lock-free, which is not desirable. Let us propose a simple modification inspired by the fast path-slow path methodology (Kogan and Petrank [26]) for an important subclass (S) of wait-free algorithms.

We assume that for each method (such as enqueue, dequeue) in the wait-free algorithm, we have a method $opDone(reqId)$, which returns true if the wait-free method

```
helpDel (node)
while true do
    /* Logically delete the node */
    succ ← node.next.getReference() ; stamp ← node.getStamp()
    newStamp ← setMark(stamp)
    status = node.next.attemptStamp(succ, newStamp)
    if !status then
        continue
    end
    /* Physically delete the node */
    pred ← getPredecessor (node) ; predStamp ← pred.getStamp()
    pred.next.CAS(node,succ,predStamp, predStamp+1)
    break
end
```

with id $reqId$ has completed (entire method, not $kSet$). Second, we also assume that after a method begins, if the rest of the threads complete a cumulative total of λ operations, then it is guaranteed that some thread (including the current thread) must have completed the current method. A lot of wait-free algorithms (class $\mathcal{S}$) that we consider such as queues, multi-CAS operations, and lists have this property because they use a high level request array, where older requests are helped by younger requests. In contrast a simple wait-free operation of atomically updating a memory word does not follow this property.

Let us now modify each EXPANDER operation to fail at most K times. For example, in the $lookUp$ operation, we can run the outermost $while$ loop a maximum of K times. Let it throw an exception (can return $false$ also, the method does not matter) after failing K times. When we call an EXPANDER's operation from the wait-free algorithm let us invoke it as follows (example with $kSet$).

```
do {
  flag = 0;
  try{ expander.kSet(...)}
  catch (Exception e){flag = 1;}
  if(!flag) break;
}while (!opDone(reqId));
```

Since the EXPANDER as a whole is lock-free, if an operation fails for K times, then it means that at least all other operations have been successful for K' times where K/K' is a constant factor (some operations have multiple atomic instructions). Since each EXPANDER operation (e.g: $kSet$) is one step for the high level wait-free algorithm, we can say that if an operation fails for l times, then other operations have been successful $l \times K/K'$ times. When $lK/K' = \lambda$, we are guaranteed that the wait-free method will be completed (either by the current thread or by some other thread). Thus for wait-free algorithms in $\mathcal{S}$, we can say that they remain wait-free even with an EXPANDER.

The aforementioned snippet of code can be inside the EXPANDER's API for the sake of elegance. It will be hidden from the user.

VI. Example: Wait-free Queue

Let us make a point about the universal nature of the expander. To use it in any setting, we just need to instantiate the EXPANDER class with the right values of the class names: $DataType$ and $TmpType$, and provide an implementation of the $ExpNode$ interface. We simply need to replace set, get, and CAS functions with their counterparts that use the EXPANDER. To use EXPANDER with wait-free algorithms we need to to minor modification as explained in Section V. In this section, we discuss the implementation details of the wait-free queue using the EXPANDER. The basic algorithm of our implementation is the same as that of Kogan et al. [21] (using linked list). Our approach is very simple: **mostly single line changes to make atomic operations use the EXPANDER**. The full code of the queue is shown in Appendix A. We only describe the enqueue method here.

```
1  public class Queue
2    AtomicReference<Node> head,tail
3    final int enqTid = 0, deqTid = 1
4    Expander <Node,Integer> exp
5    public Queue ()
6      /* The number of temporary fields associated
7      with a node in this case is 2: enqTid and deqTid*/
8      /* The number of threads are 64. */
9      exp ← new Expander
10     <Node, Integer> (2, 64)
11     /* Node contains the value and the reference field */
12     Node sentinel ← new Node(-1)
13     head ← new AtomicReference<Node>(sentinel)
14     tail ← new AtomicReference<Node>(sentinel)
15     public void enq(tid, value)
16       n ← new Node (value)
17       help() /* help the pending requests */
18       /* Create a node in the EXPANDER */
19       values ← new Integer[2]
20       values[enqTid] ← tid /* tid represents the enqTid */
21       exp.kSet(n, n.next, values, false) /* A new node is added to the
         EXPANDER */
22       help_enq() /* Tries to link n to the tail node same as in [21] */
23       help_finish_enq()
24       void help_finish_enq()
25       /* read the last node of the queue */
26       last ← tail.get() ; next ← last.next.get()
27       if next = null then
28           return
29       end
30       /* Find out which thread has added the last node */
31       tid ← exp.kGet(next).tmpFields[enqTid]
32       /* Node next is added by the thread tid */
33       /* Update the status of the thread tid and tail pointer */
34       tail.compareAndSet(last,next)
35       /* Remove the node from the EXPANDER */
36       exp.free(next)
```

With every node that is added in the queue, two temporary fields– $enqTid$ and $deqTid$ – are saved in addition to the value and reference fields. These fields correspond to the thread id of the thread, which has enqueued or dequeued the node respectively. We avoid saving these temporary fields as it consumes extra space.

The class $Queue$ shows the implementation of the wait-free queue with multiple enqueuers and dequeuers Lines 5-14. The nodes are hashed (added) in the EXPANDER on the basis of the hashcode of the $Node$. The $DataType$ in this case is of type $Node$ [10]. The temporary fields that need to be associated with a node of a queue are the two thread ids, so $TmpType$ is of type: $Integer$.

The code for the enqueue operation is shown in Lines 15-36. Note that some lines have been removed (or commented out) to enhance readability. The lines that use the EXPANDER have been encased in a rectangle. A thread (t_i) places a request r to enqueue a new node n in the linked list. Along with this, we add the node n in the EXPANDER using the function $kSet()$ (Line 21). This is done to inform the concurrent threads that an enqueue operation for thread t_i is in progress. Lets assume that the next pointer of the last node $tail$ points to the node n, indicating that some enqueue operation is in progress. The thread then searches for the node corresponding to the node n in the EXPANDER using the function $kGet()$ (Line 31). The function $kGet()$ returns an object of type $Expander :: DataState$. We subsequently access the $tmpFields$ array, which in this case is an array of integers. We read the 0^{th} entry corresponding to $enqTid$, which refers to the thread id of the thread that has added the node n to the queue. Once we know the thread id, all the helpers try to help thread t_i in completing its request r. The tail pointer is updated and points to the node n. Once the enqueue operation is completed the node n is removed from the EXPANDER using the $free$ method (Line 36).

VII. CORRECTNESS

We prove that our $kGET()$ and $free()$ algorithms are linearizable (appear to execute instantaneously) in this section. Please refer to Appendix B for the proof of lock-freedom and linearizability of $kSET()$ and $kCAS$.

Theorem 1: **kGet() is linearizable.**

Proof: We need to find a point of time between the start and end of $kGet$ at which it seems to instantaneously execute. Let us consider the first case when a node is mapped in the EXPANDER. In this scenario, the $kGet()$ method linearizes (has a point of linearizability) when a valid node having the same $memIndex$ is found by the $lookUp$ method. We then return its content. Before executing this statement, the $kGet$ request cannot affect any other request because it does not have its value, and after it has read the contents of the node, no other request can change its state. Note that no thread can change the value of the $DataState$ object after it is created. Thus, $kGet$ seems to execute at this point.

Next, let us assume that the memory index is not mapped in the EXPANDER and let the point in time at which the $lookUp$ method finds this fact be t_i, and let the value stored in the baseline structure at this point of time be x. Now, if the $kGet()$ method returns x, then it seems to execute at time t_i. We can thus make it linearize at time t_i, when the $lookUp$ method does not find a node in the linked list.

Let us now consider the case when $kGet()$ reads y ($x \neq y$) from the baseline data structure at time, t_j. This means that between t_i and t_j some other thread has updated the baseline data structure. This can only be done by the $free$ method. We claim that a $kSet$ or $kCAS$ method linearized after time t_i, and then a $free$ method wrote its value before t_j. Assume that this is not the case. It cannot be the case that the $free$ method started after t_i. It would not find the node corresponding to the memory word and would thus exit. It must have started before t_i, and continued till some point of time after t_i. This means at t_i it was alive. Since the value in the baseline structure was x at t_i, it must have written y after t_i. This is only possible if it set the state of the node to **WRITEBACK** before t_i (at time t_w) and then wrote the value after t_i. It could not have set the state of the node to **WRITEBACK** after t_i because the node itself was not there. This also means that this $free$ method did not delete the node. Then another $free$ method might have deleted it between t_w and t_i. However, to delete a node a $free$ method should have successfully updated the state of the node to **WRITEBACK** and at any point of time, only one such operation can be alive. This is not possible and there must have been a $kSet$ or $kCAS$ after t_i that wrote y to the memory word. Let us linearize the $kGet$ method after this point. In the sequential history, we will read y for that memory word and thus the execution is legal. We considered all three cases, and were able to find the points of linearizability for all three cases. Thus proved. ■

Theorem 2: **The $free$ method is linearizable.**

Proof: We need to prove that an execution of the $free$ operation to clean up a node appears to take place instantaneously. We define the point at which we read the node in the $lookUp$ function as the point of linearizability for the case where we find the node's state to be **WRITEBACK** or **FLUSH** (another thread is removing). Otherwise, we define Line 83 as a point of linearizability for the $free$ method. At this point, a node is logically deleted from the EXPANDER. Once a node for a particular memory word $memIndex$ is logically deleted, it is equivalent to saying that no mapping for $memIndex$ exists in the EXPANDER. All the read operations access data from the baseline data structure and writes allocate a new node for $memIndex$ in the EXPANDER. Before logically deleting a node, the state of the node is set as **FLUSH**. This does not alter the behavior of the reads ($kGet()$) as the value saved in the node is still visible to the reads and can be directly returned. In the case of write operations $kSet()$ and $kCAS()$, a thread first helps the $free$ method to complete its operation (remove the node from the EXPANDER) and then proceeds with allocating a new node and performing the write (conditional write) operation. Before the $free$ method has reached the point of linearizability, it does not alter the results of other write operations because either they can proceed with the write (before state is set to **FLUSH**), or they help $free$ to complete the delete, and then do the write. After the point of linearizability, the entry is deleted, and this is visible to all concurrent operations instantaneously. ■

Theorem 3: **For algorithms in set $\mathcal{S}$ (see Section V), if we have a bounded number of $free$ calls in each high level method, then the correctness of the program**

is not affected. Furthermore, it continues to maintain its original progress guarantees (lock-freedom or wait-freedom).

Proof: Let us distinguish between the terms *high level method* and *low level method*. A high level method is a method in the wait-free/lock-free algorithm that is using the EXPANDER. In comparison, a low level method is an EXPANDER operation such as $kGet$ or $kSet$.

To prove the premise of the theorem, let us proceed as follows. We know that $kGet$, $kSet$, $kCAS$ and $free$ are linearizable. Given a high level program with a bounded number of $free$ calls per high level method, we can write all of them in a serial schedule. The only effect that the extra $free$ calls will have is on the subsequent $kGet$, $kSet$, and $kCAS$ calls to the same node.

There will be no effect on $kGet$ because it is bound to get the most up to date value from either the EXPANDER or the baseline data structure. The $kSet$ and $kCAS$ operations call $lookUpAlloc$ first. Even if the node is not there in the EXPANDER, it will be brought in first. Hence, an additional $free$ call, will at best entail more work, but will not change the semantics of the program.

Secondly regarding progress conditions, we can use the same reasoning as in Section V. Lock-free algorithms will remain lock-free mainly because the additional $free$ calls are bounded in number per high level operation, and our proofs do not presume any particular order between calls to $free$ and calls to other methods. High level wait-free algorithms will remain wait-free as per the reasoning given in Section V. ■

VIII. EVALUATION

We performed all our experiments on a Dell PowerEdge R820 server running the Ubuntu Linux 12.10 operating system with the generic 3.5.0-17 kernel. It is a hyper-threaded four socket, 64 bit machine. Each socket has eight 2.20GHz Intel Xeon CPUs with a 16 MB L2 cache, and 64 GB main memory. The total number of cores visible to software is 64. We use the $totalMemory()$ and $freeMemory()$ functions of Java's built in Runtime class to estimate the memory usage of each program.

Let us now describe our experimental methodology. Let there be S threads in the system and let each thread complete N requests. Let the total number of requests completed by all the threads be N_{tot} ($S \times N$), and let the time taken from the start of the experiment be T. We measure the time per operation, t_{req}, as the average time taken to complete an operation (T/N_{tot}). In our experiments, we set N equal to 1 million and we set the number of buckets in the hash table equal to the number of threads.

We evaluated the EXPANDER with a wide variety of algorithms as listed in Table II and Table III which use redirection and packing respectively. Appendix C describes the temporary fields in each of the benchmarks, and the methods we use to store them in the EXPANDER.

A. Performance

Figure 6 shows the impact of using the EXPANDER on the time per operation (t_{req}) with 64 threads. The stars indicate the t_{req} with 32 threads. All our algorithms implemented using the EXPANDER are 10-100X faster than the algorithm using locks. Hence, we only report the slowdown with reference to the non-blocking version that does not use an EXPANDER.

The RADIR algorithm is a specialized slot scheduling algorithm for reserving bandwidth for solid state storage devices (SSDs). It uses an 1D array of slots. The problem is to reserve a set of k (varying from 2 to 256) contiguous slots in this array while respecting some constraints imposed by the physics of solid state drives. The wait-free resource allocation for SSD bandwidth reservation [30] is termed, $WFRadir$, and the EXPANDER version is $WFRadirExpander$. The performance of both the algorithms is nearly the same up to 40 threads. Beyond 40 threads, $WFRadirExpander$ is 18% slower.

The EXPANDER helps in efficiently implementing a wait-free multi word compare-And-Set ($MWCAS$) operation [4] ($WFMCAS$). The time taken per operation for $WFMCAS$ and $WFMCASExpander$ ($MWCAS$ implemented using the EXPANDER) is within 12% for 32 threads. Next, we consider slot schedulers ($SlotScheduler$) that reserve a set of slots (varying from 3 to 64) in contiguous columns in a 2D matrix of slots. They are used to implement scheduling in storage systems, networks, and video servers. We compare the results of slot scheduling using the EXPANDER ($WFSlotExpander$) with a wait-free slot scheduler proposed in [24] ($WFSlot$). The performance of $WFSlotExpander$ is 11% less than $WFSlot$ (for > 56 threads).

Now, we discuss a set of benchmarks which use redirection. We evaluated the performance of our expanded version of the wait-free queue ($WFQueueExpander$) by comparing it to the wait-free queue proposed by Kogan et al. [21] ($WFQueue$). The results presented in Figure 6 (for 70% push and 30% pop operations) show that the loss in performance (time per operation) is limited to 10-20%. The $WFQueue$ algorithm stores two temporary fields, $enqTid$ and $deqTid$, in each of the nodes in addition to the value and reference fields. With our EXPANDER we need not save these temporary fields in each node. Thus, the queue with the EXPANDER uses 20-30% less memory for experiments with more than 16 threads (see Figure 7).

We compared the performance of a lock-free linked list by Michael et al. [8] ($LFList$) with our expanded version of the lock-free linked (performing 60% add and 40% remove operations). Both the lists perform nearly the same (within 1.5%) (see Figure 6). Each node in the linked list ($LFList$) contains three fields: value, mark and the

Workload	Temporary fields
Wait-free Queue [21]	enqueueId(4bytes), dequeueId(4bytes)
Lock-free Linked List [27]	mark bit (1 bit)
Lock-free binary search tree [28]	tag (4bytes) , flag(1byte)
Lock-free Skiplist. [29]	mark bit(1 bit)

Table II: List of benchmarks (redirection)

Workload	Temporary fields (in bits)
Wait-free Multi word CAS [4]	index(30), thread id(30), pointer(2)
Generalized wait-free slot scheduling [24]	request id(15), thread id(10), round(5), timestamp(21), slot number(6), state(2)
RADIR(slot scheduling for SSD based storage devices) [30]	request id(15), thread id(10), state(2)

Table III: List of benchmarks (packing)

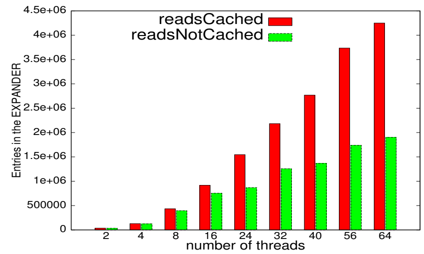

Figure 5: Space overhead for different schemes – $WFRadirExpander$

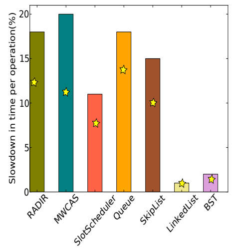

Figure 6: Slowdown in the time per operation (t_{req}) with the EXPANDER

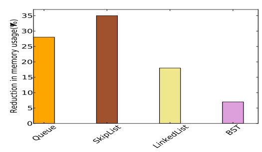

Figure 7: Reduction in memory usage for benchmarks (using redirection)

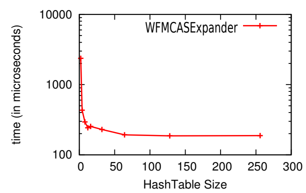

Figure 8: Impact of hashtable size on t_{req} for $WFMCASExpander$

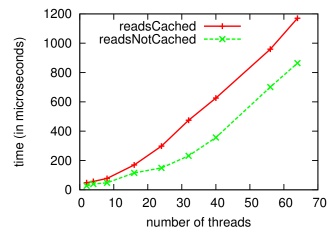

Figure 9: t_{req} for $WFRadir$-$Expander$

atomic reference to the next element. The temporary field, $mark$, is typically packed with the $next$ pointer using Java's AtomicMarkableReference type (internally uses the redirection method). It is used to indicate the fact that a node has been temporarily deleted. We need not associate this field with all the nodes. As and when a node is selected for deletion we need this information. Figure 7 shows that the linked list implemented using the EXPANDER uses 18% less memory with nearly no performance overheads. Similarly we compare the performance of our expanded version of a lock-free skiplist with a lock-free skiplist implementation proposed by Herlihy et al. [29] (performing 60% add and 40% remove operations). In this case, the memory usage is reduced by 35% since at each level the nodes need not save the mark field. The consequent slowdown in the time per operation is 15% (for $>$ 40 threads). Lastly, we show the comparison of a lock-free binary search tree (BST) proposed by Natarajan et al. [28] with a binary search tree implemented using our EXPANDER (performing 60% add and 40% remove operations). Both the algorithms perform nearly the same ($<$ 2%) and we save up to 7% memory when an EXPANDER is used.

B. Sensitivity

Let us now do some sensitivity studies for various implementations of the EXPANDER. We start by looking at when we should add memory words to the EXPANDER. We study this with the help of the RADIR benchmark (considered as a representative). Assume that we add memory words to the EXPANDER whenever it is accessed (read or write), even if no temporary fields are associated with a word. We refer to this implementation as ($readsCached$). In the second scheme, memory words are added to the EXPANDER only when writes take place using kSet() or kCAS() ($readsNotCached$). We observe that adding memory words on receiving a read request increases the size of the EXPANDER by nearly 75% for up to 32 threads (see Figure 5). Beyond 32 threads, the size of the EXPANDER nearly doubles. When reads are not cached, there is a roughly 75% improvement in performance for up to 40 threads and beyond 40 threads we get roughly 36% improvement in the time per operation (see Figure 9). This justifies our choice of not adding the memory words to the EXPANDER on receiving a read request.

Next, we experimented with two policies to recycle the nodes in the EXPANDER for the multi word compare-And-Set ($MWCAS$) benchmark. In the first scheme, nodes are deleted from the EXPANDER as soon as the corresponding memory words are no longer required by the threads ($WFMCASExpander$). In the second scheme, nodes are deleted from the EXPANDER only when a thread completes its operation ($MCASExpFull$). The results show that $MCASExpFull$ is approximately 10x slower than $WFMCAS-Expander$.

Lastly, we study the impact of the hash table size since it plays an important role in the performance of the EXPANDER. We used the multi word compare-And-Set ($MWCAS$) benchmark for this study. Figure 8 shows the time per operation for 64 threads with various hash table sizes. We observe that lower the hash table size, more is the time taken per operation. The reason for the increased time per operation is that the size of each hash bucket is large in the case of a small hash table and this increases the search overhead in each bucket. Once the table size is 64 (equivalent to the number of threads), the time per operation is 192μs, which is nearly the same as for the version of the code without the EXPANDER. If the table size is increased beyond 64, the effect on the time per operation is negligible. The last two experiments justify our choice of the hash table size, and the strategy of eager deletion.

IX. Conclusion

We designed a novel universal data structure called a memory word EXPANDER. It eliminated the need for both redirection and packing. We showed that it is possible to reduce the runtime memory footprint by 20-35% for algorithms that use redirection. We studied three algorithms that use packing and the performance overhead for all three algorithms was 2-13% for 32 threads ($<$ 20% for 56+ threads) and we further showed that we preserve the wait-free property of algorithms for a large class of wait-free algorithms.

References

[1] J. H. Anderson, S. Ramamurthy, and R. Jain, "Implementing wait-free objects on priority-based systems," in *PODC*, 1997.

[2] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *PODC*, 1994.

[3] S. Feldman, P. LaBorde, and D. Dechev, "A wait-free multi-word compare-and-swap operation," *IJPP*, 2014.

[4] H. Sundell, "Wait-free multi-word compare-and-swap using greedy helping and grabbing," *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 694–716, 2011.

[5] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Distributed Computing*, 2002.

[6] M. M. Michael, "Aba prevention using single-word instructions," *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

[7] ——, "Scalable lock-free dynamic memory allocation," *ACM Sigplan Notices*, vol. 39, no. 6, pp. 35–46, 2004.

[8] ——, "High performance dynamic lock-free hash tables and list-based sets," in *SPAA*, 2002.

[9] J. D. Valois, "Implementing lock-free queues," in *PDCS*, 1994.

[10] M. Herlihy and N. Shavit, *Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[11] M. M. Michael and M. L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, 1998.

[12] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[13] M. Herlihy, V. Luchangco, P. Martin, and M. Moir, "Non-blocking memory management support for dynamic-sized data structures," *ACM Transactions on Computer Systems (TOCS)*, 2005.

[14] M. M. Michael, "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes," in *PODC*. ACM, 2002.

[15] K. Sagonas and J. Wilhelmsson, "Efficient memory management for concurrent programs that use message passing," *Sci. Comput. Program.*, vol. 62, no. 2, pp. 98–121, 2006.

[16] S. Patel, R. Kalayappan, I. Mahajan, and S. R. Sarangi, "A hardware implementation of the mcas synchronization primitive," in *DATE*, 2017.

[17] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Distributed Computing*, 2002.

[18] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou, "Disentangling multi-object operations," in *PODC*, 1997.

[19] G. Barnes, "A method for implementing lock-free shared-data structures," in *SPAA*, 1993.

[20] T. Brown, F. Ellen, and E. Ruppert, "Pragmatic primitives for non-blocking data structures," in *PODC*, 2013.

[21] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," *ACM SIGPLAN Notices*, 2011.

[22] A. Israeli and L. Rappoport, "Efficient wait-free implementation of a concurrent priority queue," in *Distributed Algorithms*. Springer, 1993, pp. 1–17.

[23] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," in *Proceedings of the 17th ACM SIGPLAN PPoPP*, 2012.

[24] P. Aggarwal and S. R. Sarangi, "Lock-free and wait-free slot scheduling algorithms," in *IPDPS*, 2013.

[25] H. Attiya and E. Hillel, "Highly concurrent multi-word synchronization," *Theoretical Computer Science*, 2011.

[26] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures," in *ACM SIGPLAN Notices*, 2012, pp. 141–150.

[27] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *PODC*, 1995.

[28] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *Proceedings of the 19th ACM SIGPLAN PPoPP*, 2014.

[29] M. P. Herlihy, Y. Lev, and N. N. Shavit, "Concurrent lock-free skiplist with wait-free contains operator," May 3 2011, uS Patent 7,937,378.

[30] P. Aggarwal, G. Yasa, and S. R. Sarangi, "Radir: Lock-free and wait-free resource allocation model for flash drive bandwidth reservation," in *HiPC*, 2014.

APPENDIX A. EXAMPLE: WAIT-FREE QUEUE

In this section, we discuss the implementation details of the wait-free queue using the EXPANDER. Kogan et al. [21] describe an implementation of a wait-free queue, which supports multiple concurrent dequeuers and enqueuers. The queue is implemented in the form of a linked list and additionally holds two references to the head and tail of the list. To ensure that the implementation is wait-free, a thread helps another thread (which is waiting for long) in completing its operation.

```
1  public class Node implements ExpNode<Node>
2    Integer value
3    AtomicReference<Node> next
4    public Node(Integer val)
5      value=val
6    public int hash ()
7      return this.hashCode()
8    public Node getData ()
9      return next.get()
10   public void setData (Node b)
11     next.set(b)
12 end class
13 public class Queue
14   AtomicReference<Node> head,tail
15   final int enqTid = 0, deqTid = 1
16   [ Expander <Node,Integer> exp ]
17   public Queue ()
18     /* The number of temporary fields associated with a node in this case is
       2: enqTid and deqTid*/
19     /* The number of threads are 64. It is used to called the hashtable size. */
20     [ exp ← new Expander<Node, Integer> (2, 64) ]
21     /* Node contains the value and the reference field */
22     Node sentinel ← new Node(-1)
23     head ← new AtomicReference<Node>(sentinel)
24     tail ← new AtomicReference<Node>(sentinel)
25     public void enq(tid, value)
26       n ← new Node (value)
27       help() /* help the pending requests */
28       /* Create a node in the EXPANDER */
29       values ← new Integer[2]
30       values[enqTid] ← tid /* tid represents the enqTid */
31       /* A new node is added to the EXPANDER by hashing it on the basis
         of its hash code*/
32       [ exp.kSet(n, n.next, values, false) ]
33       help_enq() /* Tries to link the node n to the tail node same as in
         [21] */
34       help_finish_enq()
```

With every node that is added in the queue, two temporary fields– $enqTid$ and $deqTid$ – are saved in addition to the value and reference fields. These fields correspond to the thread id of the thread, which has enqueued or dequeued the node respectively. In case of an enqueue operation, a thread first finds which thread has attached the last node to the queue. This is done by reading the $enqTid$ field of the node. A thread then helps the thread with id as $enqTid$ to complete its operation. Similarly, for the dequeue operation a thread tries to help the request placed by the thread whose thread id is saved in the $deqTid$ field of the first node (head node) of the queue. We notice that once the enqueue/dequeue operation is over, we no longer require the $enqTid$ and $deqTid$ fields. Therefore, we avoid saving the temporary fields $enqTid$ and $deqTid$ in each node of the queue as these fields consume extra space.

Let us discuss the design of the queue implemented using the EXPANDER. The basic algorithm of our implementation is the same as that of Kogan et al. [21]. In the EXPANDER based design, each node of the queue requires only two fields: $value$ and the atomic reference to the $next$ element. The $Node$ class implements the interface $ExpNode$ and provides the implementation for the functions $hash$, $getData$ and $setData$ as shown in the algorithm (Lines 1-11).

The class $Queue$ shows the implementation of the wait-free queue with multiple enqueuers and dequeuers. The nodes are hashed (added) in the EXPANDER on the basis of the hashcode of the $Node$ (returned by Java's native $hashCode$ function). The $DataType$ in this case is of type $Node$. The temporary fields that need to be associated with a node of a queue are the two thread ids, so $TmpType$ is of type: $Integer$.

The code for the enqueue operation is shown in Lines 25-47. Note that some lines have been removed (or commented out) to enhance readability. The lines that use the EXPANDER have been encased in a rectangle. We assume that the code to ensure wait freedom is implemented as described in Section V. The additional details of implementing $opDone$ (part of the original algorithm also) and passing the request id are not shown for the sake of readability.

```
35 void help_finish_enq()
36 /* read the last node of the queue */
37 last ← tail.get()
38 next ← last.next.get()
39 if next ≠ null then
40   | /* Find out which thread has added the last node */
41   | [ tid ← exp.kGet(next).tmpFields[enqTid] ]
42   | /* Node next is added by the thread tid */
43   | /* Update the status of the thread tid and tail pointer */
44   | tail.compareAndSet(last,next)
45   | /* Remove the node from the EXPANDER */
46   | [ exp.free(next) ]
47 end
```

A thread (t_i) places a request r to enqueue a new node n in the linked list. Along with this, we add the node n in the EXPANDER using the function $kSet()$ (Line 32). This is done to inform the concurrent threads that an enqueue operation for thread t_i is in progress. After adding the node in the queue, a thread updates the tail pointer of the queue to n. Lets assume that the next pointer of the last node $tail$ points to the node n, indicating that some enqueue operation is in progress. The thread then searches for the node corresponding to the node n in the EXPANDER using the function $kGet()$ (Line 41). The function $kGet()$ returns an object of type $Expander :: DataState$. We subsequently access the $tmpFields$ array, which in this case is an array of integers. We read the 0^{th} entry corresponding to $enqTid$, which refers to the thread id of the thread that has added the node n to the queue. Once we know the thread id, all the helpers try to help thread t_i in completing its request r. The tail pointer is updated and points to the node n. Once the enqueue operation is completed the node n is removed from the EXPANDER using the $free$ method (Line 46).

```
48 public int deq(tid)
49     /* Place a request r to dequeue a node */
50     help() /* Help the pending requests */
51     help_deq(tid)
52     help_finish_deq()
53 void help_deq(tid)
54     /* If the queue is not empty and the operation of thread tid is pending */
55     /* Add the first node of the list in the EXPANDER */
56     first ← head.get()
57     oldValues[deqTid] ← -1
58     newValues[deqTid] ← tid /* tid represents the deqTid */
59     exp.kCAS(first, oldValues, newValues)
60     help_finish_deq()
61 help_finish_deq()
62     /* read the first node of the queue */
63     first ← head.get()
64     next ← first.next.get()
65     tid ← exp.kGet(first).tmpFields[deqTid]
66     if tid ≠ -1 then
67         /* update the status of the thread tid */
68         /* update the head pointer */
69         head.compareAndSet(first,next)
70         /* remove the node from the EXPANDER */
71         exp.free(first)
72     end
```

Similarly, in the dequeue operation a thread adds the node $first$ (head) to the EXPANDER, corresponding to the head node in the list (Line 59). We can subsequently perform atomic operations on the fields of this node. The thread (dequeuer) then tries to write its thread id in the $first$ node using the EXPANDER's $kCAS()$ method. This is done to indicate which thread is trying to dequeue a node. Now, if there are multiple dequeuers, the thread for which $kCAS$ returns true (Line 59) will be able to dequeue the head (*first*) node successfully. All the threads read the thread id (tid) associated with the $first$ node using the $kGet()$ method (Line 65) and then help the thread with id tid in completing its operation. The code is shown in Lines 48-72.

Note that, only one node is added to the EXPANDER per enqueue operation even though there are multiple helpers trying to help a request in completing its operation. At any point in time, the number of nodes added to the EXPANDER is equal to (the number of enqueuers + min (1, no of dequeuers)). As soon as an operation is completed, the corresponding node is deleted from the EXPANDER. Other than the lines for accessing the EXPANDER's functions, the rest of the Java code is the same for our version and the original wait-free version proposed by Kogan et al. [21].

APPENDIX B. IMPLEMENTATION OF $lookUp()$, $lookUpAlloc()$ AND $kSet()$

A. lookUp()

This function locates the entry (of type $MemCell$) corresponding to a memory word identified by $expNode.hash()$ in the EXPANDER. We search for the nodes in one of the linked list (buckets) of the hash table where the nodes are arranged in ascending order of their $memIndex$ fields. If we find a match, then we return the node and its predecessor. While traversing the list, if we reach a node of the linked list that has a key greater than $memIndex$, it indicates that the corresponding node of the specified index is not mapped in the EXPANDER. In this case we return the node with the least larger index and its predecessor. We can use this pair of nodes to insert a new node with index, $memIndex$, between them. Lastly, while traversing the list if a node is found to be logically deleted, then that node is physically deleted from the EXPANDER. We increment the timestamp associated with the $next$ field indicating that the linked list has been updated. The implementation of our $lookUp$ method is similar to the search method of the lock-free linked list described in [10].

B. lookUpAlloc()

We use this method to search for a node ($MemCell$) with a given key, $memIndex$, in the EXPANDER. If the node is not present, then this method creates a node corresponding to $memIndex$ with default fields: $state$ as **CLEAN** and stamp as 0. Now, a node is added to the EXPANDER only when we have a write operation (kSet or kCAS). We ensure this by calling the $lookUpAlloc$ method only upon receiving a write request. The input to this method is an argument of type $ExpNode$ corresponding to a memory word. We first calculate the hash key ($memIndex$ = $expNode.hash()$). Then, we search for a node in the EXPANDER with its key as $memIndex$. If we do not find a matching node, then it is necessary to create a new node for the memory word and insert it in to the EXPANDER. The implementation is this method is similar to the add method of the lock-free linked list described in [10].

C. kSet()

This method is used to update the value of a memory word (along with temporary fields) atomically. The parameters are: the $expNode$, data item's value, and list of temporary fields ($tmpValues$). We first invoke the $lookUpAlloc$ method to return a node (of type $MemCell$) corresponding to the $expNode$ in Line 103. Next, we check the state of the $node$. If the state is not equal to **FLUSH** then it means that no write back is currently in progress and the write request can proceed. The value stored in the $node$ is simply updated and its state is set to **DIRTY** (if it is in the **CLEAN** state). Note that in this case, we change the version of the word. We assume a function, $newVersion$, that returns a unique version number. It can be implemented with a $fetchAndIncrement$ call, or by using a thread specific counter. In the latter case, the version is a combination of the thread id, and the local count of the thread. In case the state of the node is **WRITEBACK**, the state remains the same and only the version is updated.

The $compareAndSet()$ call on $node$ tests the $versionState$ field (Line 115). The call fails in case the state changes or the version is updated (i.e., some write or remove operation is in progress). In both the cases a thread retries (Line 116). Lastly, if the node's state is **FLUSH**, then it means that some other thread is trying to remove the entry from the EXPANDER. In this case, the current thread helps in removing the entry (*node*) corresponding to the memory word (Line 121) and again looks for a valid entry in the EXPANDER.

D. Proofs

In this appendix we present the proof of lock-freedom for the algorithms described in Section IV.

Lemma 1: **Every memory word has at the most one valid entry in the EXPANDER at any point of time.**

Proof: Whenever there is a write operation on a memory word (mem), a node corresponding to it is added in the EXPANDER using the $lookUpAlloc$ method. The nodes are inserted in the EXPANDER in a sorted order based on the value of the field $memIndex$. The sorted property is ensured by Line 138. The $memIndex$ of the current node is always between the $memIndex$ fields of its neighbors in the linked list. We can easily prove by induction that this property is never violated.

Assume that at any point, we have two nodes in the linked list belonging to the same memory word (same $memIndex$). Let us consider the first such case, and let the requests that added them be: R_i and R_j. Let R_i add its node between nodes, A and B, and R_j between nodes C and D. Let us define the relation $<$ between requests A and B as follows: $A < B$, if $A.memIndex$ is less than $B.memIndex$. We have: $A < R_i < B$ and $C < R_j < D$. With no loss of generality assume that $B < C$. We thus have $R_i < R_j$, which is not true (they have the same memIndex), and the sorted property of the linked list is ensured by Line 138. Thus, every memory word will have at most one valid entry in the EXPANDER (proof by contradiction). ■

Lemma 2: **The $lookUp$ method is lock-free.**

Proof: Each memory word in the baseline data structure is accessed on the basis of its index. This index acts as a key in the EXPANDER based on which the nodes are sorted in the hash buckets. Whenever there is a search request for a node n with particular index ind ($lookUp$:Line 75), we first find a bucket b corresponding to that index ind. Next, we traverse the linked list in the bucket b to find a node with index ind. It is possible that new nodes are added/deleted in the bucket b between the indices 0 and ind. As the nodes are sorted on the basis of their index, at the most ind nodes can be added between 0 to ind. Therefore, the number of nodes that need to be traversed in the linked list is bounded by the value of the index of a node. Thus, the $lookUp$ method is *lock-free* (as well as *wait-free*) in the case of concurrent insert operations. In case the nodes are deleted, a thread restarts its search ($lookUp$:Line 88). This indicates that some other thread has made progress by deleting a node from the EXPANDER. Hence, our implementation is lock-free. ■

Theorem 4: **The $kGet()$ method is lock-free.**

```
73  lookUp(expNode)
74  memIndex ← expNode.hash()
75  setId ← hashExp(memIndex)
76  head ← listHead[setId]
77  while true do
78      retry: pred ← head
79      while true do
80          predStamp ← pred.getStamp()
81          curr ← pred.next.getReference()
82          succ ← curr.next.get(currStamp)
83          marked ← getMarked(currStamp)
84          /* Delete nodes that are marked */
85          while marked do
86              /* loop terminates when pred's next pointer points to an
                unmarked node */
87              status ← pred.next.compareAndSet(curr, succ, predStamp,
                predStamp+1)
88              if !status then
89                  continue retry
90              end
91              curr ← pred.next.getReference()
92              succ ← curr.next.get(currStamp)
93              marked ← getMarked(currStamp)
94          end
95          if curr.memIndex ≥ memIndex then
96              return (curr, pred)
97          end
98          pred ← curr
99      end
100 end
101 kSet(expNode, data, tmpValues[], flag)
102 while true do
103     node ← lookUpAlloc(expNode)
104     dataState ← node.dataState.get()
105     nodeState ← MemCell.getState(dataState.versionState.get())
106     if nodeState≠ FLUSH then
107         if nodeState = CLEAN then
108             newState ← (DIRTY, newVersion())
109         end
110         else
111             /* if the state is DIRTY or WRITEBACK then the state
                remains the same. Only the version is updated */
112             newState ← (nodeState, newVersion())
113         end
114         newDataState ← new DataState(data, tmpValues, newState)
115         res ← node.dataState.compareAndSet(dataState, newDataState)
116         if res = false then
117             continue
118         end
119         break
120     end
121     helpDel(node)
122 end
123 lookUpAlloc(expNode)
124 memIndex ← expNode.hash()
125 while true do
126     (curr, pred) ← lookUp(expNode)
127     if curr.memIndex = memIndex then
128         return curr
129     end
130     else
131         predStamp ← pred.getStamp()
132         marked ← getMarked(predStamp)
133         if marked then
134             continue
135         end
136         /* create a node in the EXPANDER */
137         node ← new MemCell(memIndex, expNode.getData(), curr)
138         if pred.next.CAS(curr, node, predStamp,predStamp+1) then
139             return node
140         end
141     end
142 end
```

Proof: In the $kGet()$ method a value is either returned from the EXPANDER ($kGet()$:Line 5) or from the baseline data structure ($kGet()$:Line 10). Reading a value of a memory word from the baseline data structure is a single step operation, hence it is *lock-free*. To read a value from the EXPANDER, we first search for a node in the EXPANDER using the $lookUp$ function ($kGet()$:Line 3) and then its contents are returned. The $lookUp$ method is lock-free as proved in Lemma 2. This implies that the $kGet()$ function is also lock-free. Concurrent writes or remove operations do not alter the behavior of the $kGet$ method since the process of reading the contents of a node is independent of the state of the node. ■

Lemma 3: **The $lookUpAlloc$ method is lock-free.**

Proof: A node with key k is inserted in a sorted order in the linked list corresponding to its hash bucket using the $lookUpAlloc$ method. A thread first finds the $pred$ and $curr$ nodes using the $lookUp$ method (Line 126). $pred$ is a node with the largest key less than k and $curr$ is the node with the least key greater than or equal to k. If a node with key k is not present, then a new node n is inserted between $pred$ and $curr$ using the $compareAndSet()$ primitive. The $compareAndSet()$ operation tests both the mark and the reference; it succeeds only if $pred$ is unmarked and refers to $curr$. If the $compareAndSet()$ call is successful, the method returns true; otherwise, we start from the beginning of the list. $compareAndSet()$ fails when some other thread has inserted a node between $pred$ and $curr$, $pred$ is marked, or $curr$ is deleted. In all the cases it means that some other thread is able to make progress by either adding/deleting a node to/from the EXPANDER. Thus, we have a lock-free implementation since the system as a whole has made progress. ■

Lemma 4: **The $helpDel$ method is lock-free.**

Proof: The main purpose of the $helpDel$ method is to delete the nodes whose status has been set to **FLUSH**. First, these nodes are logically deleted by setting their mark bits. Next, these logically deleted nodes are physically removed from the EXPANDER. For logically deleting a node n, its mark field is set (Line 83). If the $attemptStamp()$ call, to logically delete the node fails, it means some thread either modified the next pointer of the node n using the $lookUpAlloc$ method (and updated the stamp) or has set its mark bit. This indicates that some other thread has made progress. Once the node is marked for deletion, a single attempt is made to physically remove the node by updating the next pointer of its predecessor $pred$ using $compareAndSet()$. All the operations are performed using atomic primitives. These primitives guarantee that at least one of the threads succeeds in performing its operation. Thus, we have a lock-free implementation. ■

Theorem 5: **The $kSet()$ and $kCAS()$ methods are lock-free.**

Proof: The write operation at a memory word takes place using the $kSet()$ and $kCAS()$ methods. The write request is accomplished by first searching for a node n in the EXPANDER and then updating its value atomically if the state of the node is not **FLUSH**. In case, the node n is not present in the EXPANDER then a new node n is added in the EXPANDER using the $lookUpAlloc()$ method ($kSet()$:Lines 103, $kCAS()$:Line 15). All the threads that wish to write to a node n (in the **FLUSH** state), first help in deleting the node from the EXPANDER. Since the $lookUpAlloc()$ and $helpDel$ methods are lock-free (see Lemma 3, 4), it is ensured that at least one thread will eventually get a node with its state not equal to **FLUSH** (referred to as a valid node). Once a valid node is returned, a thread can proceed with its write operation. The write operation takes place using the $compareAndSet$ instruction. The $compareAndSet$ call ($kSet()$: Line 115, $kCAS()$:Line 29) can fail if some other thread is making progress by either performing a write operation on the node or removing the node from the EXPANDER. All the operations are performed using atomic primitives. These primitives guarantee that at least one of the threads succeeds in performing its operation. Thus, we have a lock-free implementation. ■

Theorem 6: **The $free()$ method is lock-free.**

Proof: A node is deleted from the EXPANDER by first updating the state of the node to **WRITEBACK** ($free()$:Line 55). Next, the state of the node is set to **FLUSH** so that the node can be physically removed from the EXPANDER. (Line 66). Both the steps are done in a loop.

In the write back phase, the loop ($free()$:Line 44-59) terminates when one thread succeeds in updating the state to **WRITEBACK**. A thread fails in updating the state of the node n when write operations are in progress. It means some thread is doing its operation. It is possible that multiple threads concurrently try to set the state of a node n as **WRITEBACK**. The $compareAndSet$ call ($free()$:Line 55) will succeed for only one thread t and the rest of the threads exit immediately. A thread that succeeds in updating the state, writes the contents to the baseline data structure. This step is done to ensure that only one thread proceeds with the write back operation (constraint imposed by the Java memory model). Next, the state of the node n is set to **FLUSH**. In this phase, the $compareAndSet$ call ($free()$:Line 69) will fail only in case of concurrent write operations. Once the state of the node n is set to **FLUSH**, a single attempt is made to remove the node n from the EXPANDER using the lock-free method $helpDel$. Thus, the implementation is lock-free since at any point in time at least one thread makes progress in completing its operation. ■

Theorem 7: **The $kSet()$ and $kCAS$ methods are linearizable.**

Proof: Next, for the $kSet()$ method we say that it appears to execute instantaneously at Line 115. It uses the atomic compareAndSet() instruction to update the value of a node. Before this point no changes are made to the node in the EXPANDER; after the compareAndSet() instruction is executed successfully, all the threads can see the new value written by $kSet()$.

Lastly, the point of linearizability of $kCAS()$ is Line 29. In this line it atomically updates the value of a memory word (along with the associated values) mapped in the EXPANDER. If the $compareAndSet()$ call is successful then the new values are visible to all the threads, otherwise this function does not have any affect. ■

APPENDIX C. BENCHMARK DETAILS

In this appendix we describe the temporary fields used in each of the benchmarks, and the methods we use to store them in the EXPANDER.

A. Wait-free Queues

Kogan et al. [21] describe an implementation of a practical wait-free queue, which supports multiple concurrent dequeuers and enqueuers. It stores two temporary fields, $enqTid$ and $deqTid$, in each of the nodes in addition to the value and reference fields. In the EXPANDER based design, each node of the queue requires only two fields: value and the atomic reference to the next element. When a thread t_i places a request for an enqueue operation, a node n that t_i wishes to insert is brought in to the EXPANDER and a temporary field, $enqTid$, is associated with it. The entries in the EXPANDER are hashed on the basis of the hashcodes (returned by Java's built in hashCode method) of the queue nodes.

All the other threads with concurrent enqueue operations try to help t_i in completing its operation. Once, the enqueue operation is complete, the node n is removed from the EXPANDER.

Similarly when a dequeue request comes in, concurrent threads try to add the first node (head node) of the queue in the EXPANDER. Along with the value field, a temporary field, $deqTid$ is added. The thread which is successful in adding the node in the EXPANDER is able to dequeue the first node. Rest of the threads try again. Once the dequeue operation is completed the node is removed from the EXPANDER.

B. Lock-free Linked List and SkipList

Each node in the linked list and skiplist (see [10]) contains three fields: $value$, $mark$ and an atomic reference to the $next$ element. The temporary field, $mark$, is typically packed with the $next$ pointer using Java's AtomicMarkableReference type (internally uses the redirection method). It is used to indicate the fact that a node has been temporarily deleted. In our EXPANDER whenever the next pointer (i.e., the atomic reference field is updated), the node is brought in to the EXPANDER and the $kCAS()$ method is used to perform the update of either the mark bit or the next pointer. If the mark bit is set, the node remains in the EXPANDER. Otherwise the node is removed from the EXPANDER. When the node is physically removed from the linked list or the skiplist, the node is also removed from the EXPANDER. At any time only those nodes that are marked (logically deleted) are in the EXPANDER. In the case of the skiplist only one entry corresponding to a node in a skiplist is maintained in the EXPANDER at the time of deletion, irrespective of the number of times the node occurs in the lanes.

C. Lock-free Binary Search Tree

In the lock-free implementation of a binary search tree proposed by Natarajan et al. [28], a node contains three fields: key, atomic reference to the $leftchild$ and atomic reference to the $rightchild$. Each reference field further contains a tag, $flag$ and $addressfield$. The tag and flag fields are used at the time of deletion. The implementation of a lock-free binary search tree using the EXPANDER is similar to the way we implemented a linked list. In this case the nodes for which either the flag bit or tag bit are to be set are brought in to the EXPANDER. Once the addition/deletion operation is over, we no longer require these temporary fields. Subsequently, the corresponding nodes are deleted from the EXPANDER.

D. Wait-free multi-word compareAndSet

In [4] a wait-free implementation of multi-word compare-AndSet is proposed. The algorithm is implemented in three stages. First, a temporary lock is acquired on the memory words. Next, we check the contents of the memory words and perform a conditional update. Finally, we unlock all the memory words. The information of a word's lock-status is stored within the memory word itself. The three temporary fields: $threadid$, $index$ and $descriptor$ are stored in the same memory word to indicate that the word is locked. The EXPANDER helps in efficiently implementing a wait-free multi word compare-And-Set algorithm. Whenever a memory word is locked, it is added to the EXPANDER. Then the temporary fields are set using the $kCAS$ method. During the unlock phase, all the memory words are removed from the EXPANDER (using the $free$ method) since we need not associate any more temporary fields with them.

E. Wait-free resource reservation model for SSDs: RADIR

Next, we consider the RADIR algorithm that is a specialized slot scheduling algorithm for solid state storage devices (SSDs) [30]. It uses an 1D array of slots. The problem is to reserve a set of k contiguous slots. The implementation is similar to wait-free multi-word compare-And-Set. In the lock phase, we save the $threadid$, $round$ and $state$ in each slot that a thread wants to reserve. In the EXPANDER based design, whenever a thread tries to reserve a slot (lock a slot) a write operation is issued for that particular slot (memory word). The entry is mapped in the EXPANDER and the temporary fields are associated with it using the $kCAS$ method. Subsequent read/write operations for that memory word/slot takes place from within the EXPANDER itself. In the reservation phase, the value written in the EXPANDER is written back to the baseline data structure using the $free$ method and the words are deleted from the EXPANDER.

F. Wait-free Slot Scheduling

We consider slot schedulers that reserve a set of slots in contiguous columns, in a 2D matrix of slots. They are used to implement scheduling in storage systems, networks, and video servers. Aggarwal et al. present wait-free implementations for slot scheduling in reference [24]. The slots are reserved in two passes. In the first pass, the slots are reserved temporarily. In this phase, the temporary fields that are associated with a slot are: state (2 bits), tid(thread id) (10 bits), slotNum (6 bits), round (5 bits), requestId (15 bits) and a timestamp (21 bits). Once the required number of slots are reserved, the reservation is made permanent. In the EXPANDER based design, the slots that a thread wishes to reserve are added to the EXPANDER one by one. All the temporary/book-keeping information is stored in the EXPANDER in the $tmpFields$ array. Once the $schedule$ operation is over (i.e., the reservation is made permanent), the final values are written in the actual slots of the baseline data structure and the corresponding entries are deleted from the EXPANDER.