Replacing Competition with Cooperation to Achieve Scalable Lock-Free FIFO Queues

Thomas A. Henzinger and Hannes Payer and Ali Sezgin

Technical Report No. IST-2013-124-v1+1
Deposited at 13 Jun 2013 11:52
http://repository.ist.ac.at/124/1/main-spaa2013.pdf

Replacing Competition with Cooperation to Achieve Scalable Lock-Free FIFO Queues

(Regular Submission)

Thomas A. Henzinger
IST Austria, Klosterneuburg
tah@ist.ac.at

Hannes Payer
Google
hpayer@google.com

Ali Sezgin
IST Austria, Klosterneuburg
asezgin@ist.ac.at

Abstract

In order to guarantee that each method of a data structure updates the logical state exactly once, almost all non-blocking implementations employ Compare-And-Swap (CAS) based synchronization. For FIFO queue implementations this translates into concurrent enqueue or dequeue methods *competing* among themselves to update the same variable, the tail or the head, respectively, leading to high contention and poor scalability. Recent non-blocking queue implementations try to alleviate high contention by increasing the number of contention points, all the while using CAS-based synchronization. Furthermore, obtaining a wait-free implementation with competition is achieved by additional synchronization which leads to further degradation of performance.

In this paper we formalize the notion of competitiveness of a synchronizing statement which can be used as a measure for the scalability of concurrent implementations. We present a new queue implementation, the Speculative Pairing (SP) queue, which, as we show, decreases competitiveness by using Fetch-And-Increment (FAI) instead of CAS. We prove that the SP queue is linearizable and lock-free. We also show that replacing CAS with FAI leads to wait-freedom for dequeue methods without an adverse effect on performance. In fact, our experiments suggest that the SP queue can perform and scale better than the state-of-the-art queue implementations.

Keywords: Concurrent Queue, Scalability, Lock-freedom.

1 Introduction

Getting the right synchronization scheme in a concurrent data structure implementation so that it is correct (linearizable) and scalable is far from being systematic. For a typical data structure, such as FIFO queue (henceforth, queue), stack, priority queue, or set, one has to decide among other things what kind of low-level representation is to be used, what kind of synchronization primitives are to be employed, and how contention is to be handled.

The choice of the synchronization primitive depends on the data structure being implemented. If a shared counter is in question, then it is much better to use the Fetch-And-Increment primitive (FAI) instead of the Compare-And-Swap primitive (CAS), assuming both are provided by the underlying architecture. For many other data structure implementations with strong progress guarantees, like wait-freedom or lock-freedom, CAS, or slight variations thereof, seems to be the de-facto standard. We question whether the use of CAS in queue implementations is inevitable or other synchronization primitives can also yield comparable or better scalability.

In non-blocking implementations of data structures whose methods update a single field of the data structure, CAS seems to be the perfect tool. A queue, for instance, contains two fields: head and tail. Elements are enqueued to the tail and are dequeued from the head. Then, whenever a thread wants to dequeue an element, it has to atomically change the head from the current oldest element in the queue to the second oldest element which can be simply done with a CAS.

The existing CAS-based implementations of a queue have a well-known problem [1]. Among all the concurrent threads trying to perform the same operation there is always one winner. That is, if k dequeue operations are all trying to update the head concurrently, only the first thread to execute its CAS will succeed while all the other $k - 1$ will fail. Since failure in CAS takes a thread back to the beginning of trying to dequeue, losers are as good as never having executed at all which implies poor scalability. Furthermore, in order to guarantee wait-freedom instead of obstruction or lock-freedom, such implementations have to have an additional helping protocol to prevent threads from starving; that is, a thread should not be forced to fail for an unbounded number of times. Designing scalable CAS-based implementations with strong progress properties is thus a formidable task.

We observe the following about concurrent queues: It is correct to remove the k^{th}-oldest element from the queue if there are at least k concurrent dequeueing threads and $k - 1$ of those threads are guaranteed to dequeue the older $k - 1$ elements. In other words, a thread should not be forced to wait to dequeue the k^{th} element until $k - 1$ winners are determined in sequential order. With this weaker constraint, it becomes just a matter of counting the number of concurrent dequeueing threads and pairing up these threads to the contents of the queue.

In this paper we formalize the notion of competitiveness for synchronization. Intuitively, a synchronizing statement in a method is competitive if its outcome depends on the interference due to other concurrent threads and this outcome determines whether the same statement is going to be executed once more or not. The CAS statement to update the head or the tail in existing queue implementations (see related work below) is a prime example of a competitive synchronizing statement.

We then present our queue implementation, called the Speculative Pairing (SP) queue. The SP queue, to capitalize on the above observation, uses FAI in the common path of its dequeue methods. With this modification, the dequeue operations do not wait for the oldest element to be removed to proceed. Furthermore, replacing CAS with FAI provides wait-free dequeue methods. The dequeue methods of the SP queue are not competitive among each other (they are cooperative), unlike all the other existing queue implementations. This means that if during a run of the SP queue, all the running threads are executing dequeue methods, then each thread performs exactly one synchronizing statement, namely an FAI. Similarly, the dequeue methods are competitive with enqueue methods within a limit. This means that regardless of what the other threads do, a thread executing a dequeue method always terminates, and hence satisfies wait-freedom. Overall, the

SP queue is lock-free because the enqueue methods are lock-free. We also prove the correctness of the SP queue by showing that it is linearizable [6].

For various workloads, we show that the SP queue can scale and perform better (up to fifty percent more than the second best) than the other implementations we compare: a lock-based queue, the Michael-Scott queue and the flat combining queue of [3].

To summarize, our two main contributions are:

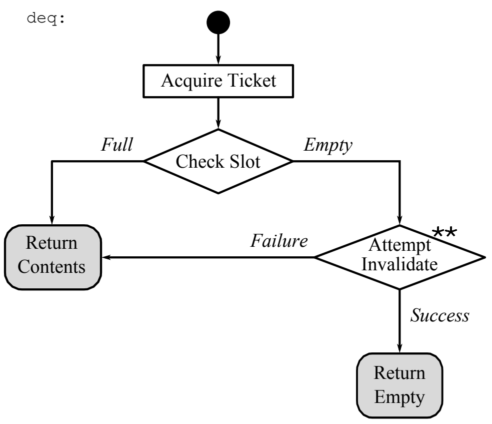
- the formalization of competitiveness which can serve as a conceptual tool to analyze the potential for scalability and progress properties of concurrent implementations, and
- the Speculative Pairing queue implementation which can scale better than the current state-of-the-art queue implementations under various workloads while guaranteeing a stronger progress property.

Related Work. Over the last decade, several alternative implementations to the Michael-Scott queue have been proposed, e.g. [3, 7, 8, 9, 13]. With the exception of [8] which improves the progress condition of the Michael-Scott queue from lock-free to wait-free [5], all the recent work aim at achieving scalable behavior. This can be by using a secondary container like the elimination array as in [13], relaxing the strict ordering of insertions as in [7], using a different low-level structure leading to less synchronization as in [9], or, at the expense of non-blocking, that is down-grading to a weaker progress condition, serializing updates to the queue as in [3]. It is worth noting that even though it was not mentioned explicitly, all these implementations try to remedy for the competitiveness of the CAS statements. The elimination queue of [13] pairs concurrent enqueue and dequeues provided that the queue is logically empty, which leads to less competitive enqueue and dequeue methods only when the queue is empty. Similarly, the baskets queue of [7] distributes the point of entry for enqueue methods by relaxing the constraint that each node has to be inserted at the tail. Finally, the flat-combining queue of [3] bounds the competitiveness of both enqueue and dequeue methods by limiting the number of possible failures the CAS statements can have. None of these implementations use FAI as a synchronization primitive nor attempt to relax the synchronization among dequeues only. We compare the performance of the SP queue with i) a lock-based queue implementation, representing locking algorithms, ii) the Michael-Scott queue since almost of all the subsequently published queues seem to derive from it and it is still one of the best performing queue implementations, and iii) the flat-combining queue because it is one of the most recent queue implementations representing the state-of-the-art in terms of performance. In fact, for similar workloads that we use in this paper, in [3], the flat-combining queue has been shown to out-perform the baskets queue which in turn has been shown in [7] to out-perform the optimistic queue of [9]. The elimination technique of [13] is an orthogonal approach and in principle can be applied to any queue implementation including ours.

Implementation overview. In the remainder of this section, we will give a high-level description of the Speculative Pairing queue. The main container for the contents of the queue is an array, called the *pairing array*. A pairing array can be in one of two states: *valid* or *invalid*. Intuitively, a pairing array is initially valid and remains valid until at least one dequeue instance observes an empty slot. Once a pairing array is invalidated, it is the responsibility of the enqueue instances to create a new pairing array and replace the invalid with the new (valid) one.

When the pairing array is in the valid state, each enqueue instance tries to insert its data item into the next available slot of this pairing array. Each dequeue tries either to remove a data item or to invalidate the current pairing array. In Fig. 1, we give the flowcharts for the dequeue and enqueue methods. Some branching boxes are tagged with one or two stars, the use of which will be explained below.

Let us begin with the explanation of the dequeue method, `deq` (left side of Fig. 1). A `deq` instance starts by obtaining the unique ticket that will point to the slot whose contents it is required to remove. This operation is synchronized with other concurrent `deq` instances to guarantee uniqueness of the ticket per



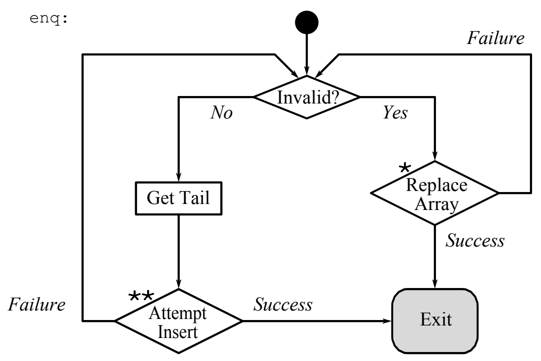


Figure 1: High-level description of the SP algorithm.

instance. The `deq` instance then checks whether the slot to which the ticket points to is empty or not. If the slot is not empty, it returns the data item contained in the slot and terminates. Otherwise, it tries to invalidate the current pairing array. If the attempt to invalidate is successful, it returns empty. If the attempt fails, it returns the contents of the slot which should now contain a data item. The slot definitely contains a data item because the only way an invalidation attempt fails is if a concurrent `enq` instance inserts an element into the same slot.

The operation of the `enq` instance (right side of Fig. 1) starts by checking whether the current pairing array is valid or not. If the pairing array is not valid, the `enq` instance tries to replace the current invalid pairing array with a local array it creates which only contains the element the instance is trying to insert. If the attempt to replace the invalid pairing array with the local copy is successful, it terminates. However, if the attempt fails, which can only be due to another `enq` instance successfully setting its local copy as the new pairing array, control goes back to the beginning stage.

If the pairing array is valid, the `enq` instance gets the index of the next available slot in the pairing array and tries to insert its item into this slot. If the insertion is successful, it terminates. If the insertion fails, this can be due to two possibilities. One possibility is that a concurrent `enq` instance inserted its item into the same slot. The other possibility is that a concurrent `deq` invalidated the pairing array by marking the current slot with a tag denoting empty return. In either case, it goes back to the beginning stage.

The marked branching boxes represent statements which need synchronization. In particular, the two branching boxes marked with two stars (attempt to invalidate of `deq`, and attempt to insert of `enq`) imply that these synchronizing statements are conflicting. That is, when there are concurrent `enq` and `deq` instances about to commit the changes they want to make, only one of them will succeed. An invalidation of a pairing array, which consists of inserting a special symbol into an empty slot, will conflict with the insertion of a valid data item into the same slot. Similarly, the replacement of the pairing array is conflicting for all `enq` instances trying to replace the array with their local copies.

Observe that while failed attempts in the `enq` method results in a new iteration of its main loop, the failure in the `deq` method does not lead to a similar loop. Unfortunately, low-level implementation details make this impossible. However, as shown in App. B, for each `deq` instance, the maximum number of failed attempts to invalidate a pairing array is bounded by the number of times an `enq` instance can insert a new item into the same linked list. In practice, it is very unlikely that a `deq` instance loses more than once to some `enq` instance.

Let us end this section by emphasizing that the novelty of our algorithm comes from the way we handle dequeues. As opposed to having `deq` instances compete for every slot, dual to `enq` instances, we instead assign tickets to each instance. If the common path is the one in which `deq` instance finds an element in the slot it was assigned, then the execution path is short and contains only a single FAI executed to acquire the ticket.

2 Competitive vs. Cooperative Synchronization

The underlying operational semantics for each data structure implementation is given by a *labeled transition system* (LTS). An LTS is a tuple $(Q, \rightarrow, L)$, where Q is the set of *states*, $\rightarrow$ is the *transition relation*, and L is the set of *transition labels*. Each label $l \in L$ is of the form (t, m, j), where $t \in \mathbb{N}$ is a *thread identifier*, m is a method instance (a method whose input parameters have values), and $j \in \mathbb{N}$ is the *statement identifier*. We assume that each method has a unique *entry* and *exit* statements. The entry statement has identifier 1, the exit statement of m has identifier $exit_m$. Entry and exit statements of any method instance are executed exactly once, the former marking the beginning and the latter marking the completion of the execution of the method instance. The transition $(t, m, 1)$ is the *invocation* of m by t, and $(t, m, exit_m)$ is the *response* of m by t.

Each state $q \in Q$ is assumed to hold all the necessary information about the concurrent execution, including the control flow information about each thread. At any state q, the local state of thread t will be denoted by $q(t)$, which contains thread local control flow information, and the valuation of all variables accessable by t; that is, the value of all thread-local variables visible to t and the value of all shared variables. Two states q and q' are *t-equivalent* if $q(t) = q'(t)$. We will write $q \xrightarrow{l} q'$ to denote $(q, l, q') \in \rightarrow$. A label l is *enabled* at state q, written $q \xrightarrow{l}$, if there is a state q' such that $q \xrightarrow{l} q'$.

The *run* of an implementation $I = (Q, \rightarrow, L)$ is an alternating sequence $q_0 l_1 q_1 \ldots l_n q_n$ of states and labels such that for all $0 < i \leqslant n$, we have $q_{i-1} \xrightarrow{l_i} q_i$. Note that, any segment $q_i l_{i+1} \ldots l_j q_j$, for $0 \leqslant i \leqslant j \leqslant n$ is a run of I. The sequence of labels generated by a run is a *behavior* of I.

Let $\mathbf{r} = q_0 l_1 q_1 \ldots l_n q_n$ be a run. A thread t is *idle* at q_i, if for all (t, m, j) enable at q_i, $j = 1$. The run $\mathbf{r}$ is *initial* if all threads are idle at q_0. A state q is *reachable* in I, if I has an initial run that ends at q.

A thread t is *invisible* during $\mathbf{r}$ if none of the labels l_i in $\mathbf{r}$ is of the form (t, m, j), for some m and j. A thread t is *executing* (m) at state q_i if a transition (t, m, j) for some $j > 0$ is enabled at q_i. The thread t is *pending* during $\mathbf{r}$ if t is invisible and executing at q_0. The *environment signature* for the run $\mathbf{r}$ is the tuple (T, M), where $T \subset \mathbb{N}$ is the set of all identifiers of threads that are not invisible during $\mathbf{r}$, and $M \subseteq \widehat{\mathcal{M}}$ is the set of all method names at least one instance of which is executed by some thread in T at some state q_j for $0 \leqslant j < n$.

An *execution* for (t, m) is the run $\mathbf{e}(t, m) = q_{i-1} l_i \ldots l_k q_k$ such that l_i is the invocation of m by t, l_k is the response m by t, and at any state between q_i and q_{k-1}, t is executing m. The execution segment $q_h \ldots q_j$, for $i - 1 \leqslant h \leqslant j \leqslant k$ is *isolated* for (t, m) if all threads $u \neq t$ are invisible during that segment.

Let s be some statement of m, and j_s be the identifier of s. Let q and q' be two t-equivalent reachable states of I such that the transition $l = (t, m, j_s)$ is enabled at both q and q'. Then, s at $q(t)$ is *competitive* if there is a run starting at q with the transition l and ending with the first response for (t, m) in which l occurs at least twice, and there is no run starting at q' with l and ending with the first response of (t, m) in which l occurs more than once. Informally, the statement s is competitive at the local state $q(t)$, if whether s is going to be executed more than once or not depends on the state of the other threads. The state q represents the possibility of repetition; we do not require inevitable repetition since depending on the program, there could be more than one statement having the same effect. The state q' represents the impossibility of repetition; no matter what the global state will be, the statement s will not be executed again by t until it completes the execution of m.

The number of times a state at which a statement s of method m is competitive occurs during an execution $\mathbf{e}(t, m)$ is called the *degree of competitiveness* of s in $\mathbf{e}(t, m)$. The statement s of m is *absolutely* *competitive* if there is an execution $\mathbf{e}(t, m)$ in which whenever a reachable state q with s competitive at $q(t)$ occurs in $\mathbf{e}(t, m)$, there exists another state q' coming after q in $\mathbf{e}(t, m)$ such that s is competitive at $q'(t)$. In other words, s is absolutely competitive if there is an (infinite) execution in which s has infinite degree of competitiveness.

A statement with 0 degree of competitiveness is called *cooperative*. A statement can be cooperative within a constrained environment signature. The statement s in m is M*-cooperative* if it is cooperative in every execution for (t, m) with environment signature (T, M') such that $M' \subseteq M \cup \{\widehat{m}\}$.

Finally, we would like to classify competitive statements according to the bound on the number of occurrences of the competitive statement as a function of the state at which they are first executed. Let $\mathbf{e}(t, m) = q_0 l_1 q_1 \ldots l_n q_n$, with q_0 reachable, be an execution for (t, m) and let l_i be the first occurrence of the competitive statement s executed by t. Then, s is *bounded-competitive* at $q_i(t)$ if there exists a bound k_s such that in *any* execution for (t, m) with the prefix $q_0 l_1 q_1 \ldots l_i q_i$, (t, m, s_j) occurs at most k_s times after q_i. The following is implied by the definitions.

Fact 2.1 *A method containing an absolutely competitive statement cannot be wait-free. An obstruction-free method containing only bounded-competitive statements is wait-free.*

Example. To illustrate these concepts, we consider the Michael-Scott queue [12] whose dequeue and enqueue method implementations are given below (complete listings are in Fig. 3 and Fig. 4).

```
1   int deq() {
2     while (true)

12      if (CAS(&Head, h, c))
13        break; // exit loop
14  }
```

```
1   enq(int x) {
2     while (true)

8       if (CAS(&t.next, n, c))
9         exit;

11          CAS(&Tail, t, c);
12    CAS(&Tail, t, c);
13  }
```

Consider the CAS statement $s_{\texttt{deq}}$ at line 12 of the deq procedure. As a convention, each statement has its line number as its identifier; e.g., the identifier of $s_{\texttt{deq}}$ is 12. First, let us consider all executions for $(t, \texttt{deq})$ with environment signature $(T, \{\texttt{deq}\})$. That is, we only consider those run segments during which no thread is trying to execute an enqueue method. In this case, there exist runs in which $s_{\texttt{deq}}$ is competitive. Let $T = \{t, u\}$. Let us choose a run which starts at a state which there are three nodes reachable from Head, Tail is pointing to the last node, and all threads are idle. Let q denote this state. Start executing $(t, \texttt{deq})$ in isolation until $(t, \texttt{deq}, s_{\texttt{deq}})$ is enabled. Call the reached state q'. Then, execute $(u, \texttt{deq})$ in isolation from q' until it becomes idle at some state q''. Since t has not modified any shared data, u has successfully updated Head. Now, resume the execution of $(t, \texttt{deq})$, again in isolation. Since the value of Head is different from t's local variable h, we will have the following execution segment for $(t, \texttt{deq})$:

$$\mathbf{e}(t, \texttt{deq}) = q''(t, \texttt{deq}, 12) q_1 (t, \texttt{deq}, 2) q_2 \ldots q_9 (t, \texttt{deq}, 12) q_{10} (t, \texttt{deq}, 13)$$

Thus, $s_{\texttt{deq}}$ occurs twice in $\mathbf{e}(t, \texttt{deq})$. Alternatively, if we let t execute its CAS statement at q', then no matter what other threads do afterwards, $s_{\texttt{deq}}$ will not occur again. This shows that $s_{\texttt{deq}}$ at $q'(t)$ is competitive. In fact, $s_{\texttt{deq}}$ has infinite degree of competitiveness and is absolutely competitive. However, $s_{\texttt{deq}}$ is bounded-competitive for $\{\texttt{deq}\}$. Going back to the previous example, we see that if we were to execute $(u, \texttt{deq})$ in isolation starting at q_9, after $(u, \texttt{deq})$ becomes idle, the next execution of $s_{\texttt{deq}}$ will again take the control back to the beginning of the loop. This failure can only happen as long as there is something in the queue; that is, there are nodes reachable from Head. This means that if $(t, \texttt{deq})$ starts executing at some state q, the number of occurrences of $s_{\texttt{deq}}$ is bounded by the nodes reachable from Head at q. Therefore, $s_{\texttt{deq}}$

is actually bounded-competitive for deq. Note that, $s_{\texttt{deq}}$ is $\{\texttt{enq}\}$-cooperative since the synchronization primitives of enq and deq methods update different shared variables.

Now let us consider the synchronization primitive $s_{\texttt{enq}}$ at line 8 of enq (see Fig. 4). With a similar analysis, we can show that $s_{\texttt{enq}}$ is $\{\texttt{enq}\}$-competitive. However, unlike the dequeue case, $s_{\texttt{enq}}$ is not bounded-competitive for enq. This is because, we can construct a run with environment $(\{t, u\}, \{\texttt{enq}\})$ where $(t, \texttt{enq}(x))$ does not terminate. In the constructed run, each isolated execution segment of $(t, \texttt{enq}(x))$ delimited by $(t, \texttt{enq}(x), 3)$ and $(t, \texttt{enq}(x), 8)$ is followed by the full execution of $(u, \texttt{enq}(y))$. This way, the queue will grow unboundedly, all elements are inserted by some execution of $(u, \texttt{enq}(y))$, and $(t, \texttt{enq}(x))$ will never perform a successful CAS statement. Finally, it is easy to show that the CAS statements of lines 11 and 12 are both $\{\texttt{enq}, \texttt{deq}\}$-cooperative. For the former, this is because it is impossible to be at a state from which the statement can never be executed again (q' of the definition does not exist). For the latter, this is because it is impossible to extend any run in which it is executed again (q of the definition does not exist). Intuitively, this is because the outcome of these statements do not affect the control flow at all.

3 Implementation

In this section, details on the low level representation, the enqueue and the dequeue method implementations will be given and the main correctness result will be stated. We will also describe how explicit memory management can be incorporated into the implementation of the SP queue.

Low-level representation. The low-level representation of the queue uses the three data types given in Fig. 5. The QueueType is the main structure holding the contents of the queue. The flag Invalid notifies whether the current pairing array is valid or not. The two array indices Cnt_deq and Tail are used by the dequeue and enqueue methods, respectively. The value of Cnt_deq is used to exclusively own a particular entry, whereas the entry denoted by the value of Tail is not exclusively assigned to an enqueue instance. Each element of the array pair is a linked-list. The size of pair is set to SIZE.

Each linked list of type SlotType has three pointers. The head pointer denotes the first element of the list. The last and removed are helpers for the enqueue and dequeue methods, respectively. In a sequential execution, last will always point to the last node of the linked list, whereas removed will point to the most recent removed entry among those located in the same slot. However, these pointers are used as best-effort indicators where a dequeue or enqueue should operate.

Finally, each node in the linked list has type NodeType. Nodes have a value field and a next pointer, val and next, respectively. Additionally, we will also use a counter ver. Each element of the queue acquires a unique entry value and nodes store this entry value. These values are used to pair each dequeue instance with a unique node in some slot of the pair array.

A schematic view of how the low level data structures are being used is given in Fig. 6. The top row of slanted boxes represent the pair array. The head pointer of each slot is denoted by the solid circle in each box. The nodes of each linked list are tagged with the values of their ver fields. The nodes in each linked list are strictly increasing in ver values (difference between each consecutive node is equal to SIZE).

Each node can be in one of three possible states. One is to have the value logically removed by a dequeue. These nodes are shaded, e.g. pair[1].head->next, the node with ver=SIZE+1. Another possibility is to have a dequeue assigned to that node, but the value has not been removed yet. These nodes have dashed contours, e.g. pair[2].head, the node with ver=2. A third option is to have a valid entry with no dequeue assigned to it. These have solid contours. In the figure, the only such node is the one with ver=2SIZE+1. Finally, all concurrent enqueues compete to insert the next entry into the queue. This would replace the fictitious place-holder node with jagged contour.

In Fig. 6, the nodes that are pointed to by the slot's removed and last pointers are tagged with r and l, respectively. Note that, if only sequential executions were allowed, only the linked list of slot 0 would be possible. The configuration in slot 1 would not be possible because both `removed` and `last` should point to the next of what they are currently pointing to. The configuration in slot 2 would not be possible because nodes in the linked list would have to be removed in order. The configuration in slot `SIZE-1` would not be possible because `last` should always point to the last element of the linked list. Both `removed` and `last` are used as heuristics to access the proper node for both enqueue and dequeue methods. This imprecision in the concurrent case does not affect the correctness (linearizability) of the implementation.

The code of the two routines used in the implementation are given in Fig. 7. `CreateNewQueue` allocates memory for and initializes the fields of a new `QueueType` variable. It also inserts a single node containing `x` into the queue. The dual routine `CloseQueue` invalidates the current queue pointed to by the parameter `q`. Invalidation is done by setting the `Invalid` to `true`, and making the `removed` pointer of the slot with index `idx` point to a special static node called `PICKET`. The `ver` field of the `PICKET` node is set to -1, a value which can never be given to any enqueue or dequeue instance.

```
 1  void enq(int x) {
 2    while (true) {
 3      queue = Queue;
 4      if (queue->Invalid) {
 5        new_queue = CreateNewQueue(x);
 6        if (CAS(&Queue, queue, new_queue))
 7          exit;
 8        continue;
 9      }
10      tail = queue->Tail;
11      idx = tail % SIZE;
12      node = queue->pair[idx].last;

    // 13-43 skipped

44      while (node->next != NULL &&
45             node->ver < tail)
46        node = node->next;
47      if (node->ver >= tail) {
48        CAS(&queue->Tail, tail, tail+1);
49        continue;
50      }
51      if (node != PICKET) {
52        new_node = Node(x, tail);
53        if (CAS(&node->next,
54                NULL, new_node)) {
55          queue->pair[idx].last = new_node;
56          break;
57        }
58      } else {
59        queue->Invalid = true;
60      }
61    }
62    CAS(&queue->Tail, tail, tail+1);
63  }
```

The enqueue method. The code of the enqueue method is given above. It has one main loop (lines 2-61) which ends only when `enq` logically inserts its parameter `x` into the queue. Due to space constraints, we omit some part of the code which handles the insertion of the first `SIZE` elements. The code in its entirety is given in App. A.1.

Each iteration starts by reading the current state of the queue, `Queue`. If the pairing array is invalid (line 4), `enq` tries to atomically replace the current queue with a new queue (line 6). If the replacement is successful, the method completes execution since as the new queue is created, the element `x` is already inserted into the first position (line 6 of `CreateNewQueue`, Fig. 7). If the replacement fails, then `enq` starts a new iteration.

If the pairing array was valid at the time line 3 was executed, `enq` proceeds to read the current enqueue counter, stored in `Tail` (line 10). Using this counter, the correct slot index `idx` is determined (line 11). As a first guess where the end of the linked list of `pair[idx]` can be, the helper pointer `last` is read and assigned to the local variable `node` (line 12).

If the node is not the last node of the linked list, a loop for reaching the end of the linked list is executed (lines 44-46). The loop goes on until either the end of the list is reached or the `ver` value of the current node is greater than or equal to the local enqueueing ticket, `tail`. If the end of the list is not reached because a node with a `ver` greater than or equal to `tail` exists, it means that a concurrent `enq` has succeeded in inserting a node with the same enqueue counter. This case is taken care of by making sure that the global enqueue counter `Tail` is at least one more than the local enqueue counter and aborting the current iteration (lines 47-50). Otherwise, `node` was pointing to the end of the list when it was last accessed (line 44). This last node is either the node with `ver` equal to `tail-SIZE` or is the `PICKET`. Note that no other possibility exists; ver cannot be less than tail-SIZE. So, if the node is not the PICKET, the enq method tries to append the new node new_node to the linked list (lines 53-54). If successful, the last pointer is set to point to this new node and the loop terminates. Otherwise, the next iteration starts. As usual, if the last node is the PICKET, the queue is marked invalid and the next iteration starts (lines 58-59).

Finally, when the loop terminates, the enq method ensures that the global enqueue counter Tail is at least one more than the local enqueue counter tail (line 62). Note that, the local enqueue counter tail is equal to the ver field of the node inserted by this enq method.

Observe that each enq instance competes with both other enq instances, much like in the Michael-Scott queue, and occasionally with deq instances if the number of deq instance invocations are greater than or equal to the number of completed enq instances after the most recent invalidation. Whether this occurs frequently or not depends on the workload and the temporal distribution of method calls.

```
1  int deq() {
2    queue = Queue;
3    if (queue->Invalid)
4      return EMPTY;
5    ticket = FAI(queue->Cnt_deq);
6    idx = ticket % SIZE;

   // 7-23 skipped
```

```
24     if (node->ver > ticket)
25       node = queue->pair[idx].head;
26     while (node->ver < ticket) {
27       if (node->next == NULL) {
28         if (CAS(&node->next,
29                 NULL,PICKET)) {
30           Close(queue,idx);
31           return EMPTY;
32         }
33       }
34       node = node->next;
35       if (node == PICKET) {
36         Close(queue,idx);
37         return EMPTY;
38       }
39     }
40     x = node->val;
41     queue->pair[idx].removed = node;
42     return x;
43   }
```

The dequeue method. The code of the dequeue method is given above.We had to omit again some part of the code which handles the removal of the first SIZE elements. The full deq code is given in App. A.2. Similar to enq, deq starts by copying the current state, Queue, into its local variable queue (line 2). If the queue has already been invalidated, deq ends by returning EMPTY.

If the queue was valid at the time line 2 was executed, deq receives its *unique* removal ticket. This is done by atomically reading the current value of the global dequeue counter Cnt_deq and incrementing it by one (line 5). The correct slot index is the ticket modulo the size of the array (line 6).

Once node is properly set, the node to remove is searched in such a way that if it is there, it is found; if not, the queue is invalidated (lines 24-37). At each iteration, first the enqueue counter of the current node is checked (line 24). If that value is not less than the local dequeue counter ticket, the only possibility (that they are equal) implies that the correct node has been found and the loop terminates. Otherwise, deq checks whether this is the last node in the linked list (line 25). If it is, it implies that the node that deq is supposed to removed has not been inserted into the queue yet. Thus, deq tries to invalidate the queue by appending the PICKET to the current linked list (lines 26-27). If successful, after completing invalidation deq terminates by returning EMPTY (lines 28-29). If the appending fails, there are two possibilities. Either a concurrent enq instance successfully inserted its element, or a concurrent deq instance invalidated this pairing array. If the latter, deq tries to help the other deq instance by completing invalidation and returns EMPTY (lines 33-36). If the former, deq tries again by advancing to the next (non-NULL) node (line 32).

Finally, if deq exits the loop, it means that it has found the node that it was required to remove. It reads the value stored in the node (line 38), sets the removed pointer point to the node it is about to logically remove (line 39), and completes by returning the desired value (line 40).

It is worth mentioning again that in the common case that the element that a deq instance is going to remove is in the queue, the only syncrhonization primitive that is going to be used will be a Fetch-And-Increment (`FAI` in the code). If this primitive is supported by the architecture, it is not only usually faster than Compare-And-Swap, but also leads to a cooperative behavior.

Correctness. We state the main properties of the SP queue implementation. The proof is given in the Appendix.

Theorem 3.1 *The SP queue is linearizable and lock-free. The* `deq` *method is wait-free.*

Managing memory. As is true with all the other implementations with which we compare the SP queue, our code does not explicitly manage memory. Instrumenting the code so that unused memory is recycled while guaranteeing wait-freedom is known to be a difficult and open research problem [14]. In order to put a bound on the size of unused memory by our implementation, we have instrumented the code with hazard pointers [11] which satisfy lock-freedom.

Due to space constraints, we will only highlight the main design ideas. Our hazard pointers can either be a queue pointer or a node pointer. Each method keeps track of all the pointers it needs for safe access in its hazard pointer list. There are at most 4 simultaneous nodes and at most 1 queue that are kept safe at any point during the execution of each method. A `deq` instance which invalidates a queue retires its pointer.

Each `deq` (resp. `enq`) instance, after a successful removal (resp. insertion), marks a node as removed (inserted). A node n is called *stale* if n is marked as inserted and removed, is either pointed to by `head` or is pointed to by a stale node and `last` and `removed` are reachable from n. With probability p_{stale}, each `deq` instance before starting its operation tries to update the `head` pointer of the slot such that `head` points to the last stale node. If a `deq` instance succeeds in updating the `head` pointer, it retires all the stale nodes that `head` does not point to anymore.

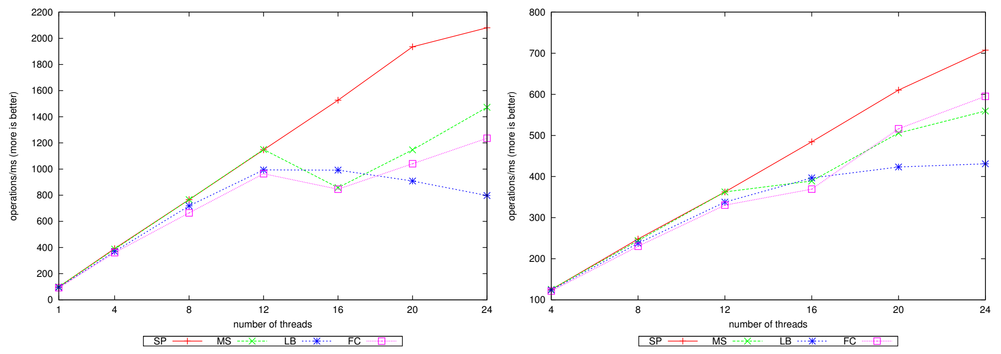
Let the bound on the number of retired nodes be k_{node}, the number of retired queues be k_{queue}, and the number of threads be N. Then, under the assumption that no thread crashes, the maximum amount of unusable memory during the execution of the SP queue with hazard pointers will have an expected value of $O(N \cdot (k_{node} + k_{queue} \cdot \texttt{SIZE}/p_{stale}))$. Intuitively, each thread can have at most k_{node} many retired nodes and each slot reachable from a retired queue pointer will have an expected maximum size of $1/p_{stale}$.

4 Experiments

We run experiments on an AMD-based server machine with four 6-core 2.1GHz AMD Opteron processors (24 cores), 6MB shared L3-cache, and 48GB of memory running Linux 2.6.32. All queues are implemented in C and compiled using gcc 4.3.3 with -O3 optimizations.

We compare the SP queue in several benchmarks with the following queue implementations. The lock-based queue implementation (LB) acquires a single global lock before every queueing operation. The Michael-Scott queue [12] (MS) is a lock-free queue implementation that reads the state of the queue and then tries to add or remove an element with a CAS operation. The Flat Combining queue [3] (FC) is also a lock-based queue where threads first announce their desired operation in a shared list and then try to commit all the announced operations sequentially by acquiring a global lock.

We experimented on various workloads. Two workloads had an (expected or strictly) equal number of 10^6 enqueue and dequeue calls per thread. Three other workloads were based on a producer-consumer partitioning with a variety of producer to consumer ratios: 1 producer to 1 consumer, 1 producer to 3 consumers, and 3 producers to 1 consumer. The common observation is that in all cases the SP queue scales up to 24 threads, which is also the number of cores in the machine we used in our experiments (figs. 8 to 9). The other implementations scale up to 8 threads, the MS queue scales up to 12 threads and perform worse than the SP queue beyond 12 threads. There are two exceptions for this general tendency. In the case where the number of enqueue and dequeue operations are on the average equal, but not enforced strictly, the MS



(a) Strict Enq-Deq (b) 3:1 Producer Consumer Ratio

Figure 2: Benchmarks on a 24-core server for various workloads.

queue scales slightly better than the SP queue up to 12 threads, but degrades beyond 12 threads performing worse than the SP queue. In the case where there is 1 producer to 3 consumers, the MS queue scales as well as the SP queue. These two exceptions are most likely due to the overhead of invalidating and creating a new queue in the SP implementation.

We also checked the effect of using FAI instead of using CAS in the implementation of a shared counter (Fig. 10(a)). We observed that when contention increases among threads competing for the counter, the CAS-based implementation performs worse than the FAI-based implementation. As the amount of contention decreases, the performance of the CAS-based implementation approaches that of the FAI-based implementation. Recall that a CAS-based implementation is expected to perform worse when the number of failed attempts is significant. The similarity in performance hints that the underlying cache coherence protocol minimizes the number of failed CAS attempts which would make FAI cheaper than CAS. Even though this implies that the scalability in the SP queue is not only due to the use of FAI, we still maintain the potential benefits of using FAI instead of CAS for scalability and for stronger progress guarantees.

Finally, the effect of using hazard pointers to explicitly manage memory seems to be in line with what is reported in [11] (Fig. 10(b)). We checked the performance of the instrumented code in which recycling was limited to visiting the nodes, i.e., without the actual system call, and the performance was essentially identical with the uninstrumented code. This suggests that the SP queue with hazard pointers will continue to perform better than the other implementations with hazard pointers. We leave a more detailed analysis of the effect of using memory management as future work.

5 Conclusion

In this paper, we presented a new concurrent queue implementation, the Speculative Pairing queue. The SP queue uses FAI in dequeue methods, as opposed to the more common CAS. We argued from a theoretical point of view, by introducing the concept of competitive synchronization, why such a switch in synchronization primitives, besides paving the way for stronger progress guarantees, can improve performance and supported this claim by empirical evidence.

We are not advocating the replacement of CAS with FAI in every implementation, if for nothing else, but for the simple reason that CAS with infinite consensus number is provably more expressive than FAI with consensus number equal to 2 [4]. However, following similar observations, e.g. [2, 10], we would like to highlight the potential for scalability by considering different (weaker) kinds of synchronization primitives.

Acknowledgment. This work was supported in part by the Austrian Science Fund NFN RISE (Rigorous Systems Engineering) and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

References

[1] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *J. ACM*, pages 779–805, 1997.

[2] Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional synchronization primitives. In *Proc. of the 23rd annual ACM symp. on Principles of distributed computing*, PODC '04, pages 80–87, 2004.

[3] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. of the 22nd ACM symp. on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, 2010.

[4] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, pages 124–149, 1991.

[5] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Proc. of the 15th int. conf. on Principles of Distributed Systems*, OPODIS'11, pages 313–328, 2011.

[6] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, pages 463–492, 1990.

[7] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In *Proc. of the 11th int. conf. on Principles of distributed systems*, OPODIS'07, pages 401–414, 2007.

[8] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proc. of the 16th ACM symp. on Principles and practice of parallel programming*, PPoPP '11, pages 223–234, 2011.

[9] Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free fifo queues. In *Proc. of the 18th Int. Conf. of Distributed Computing*, DISC 2004, pages 117–131, 2004.

[10] Victor Luchangco, Mark Moir, and Nir Shavit. On the uncontended complexity of consensus. In *Proc. of the 17th Int. Conf. of Distributed Computing*, DISC 2003, pages 45–59, 2003.

[11] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[12] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th annual ACM symp. on Principles of distributed computing*, PODC '96, pages 267–275, 1996.

[13] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proc. of the 17th annual ACM symp. on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, 2005.

[14] Erez Petrank. Can parallel data structures rely on automatic memory managers? In *MSPC*, page 1, 2012.

A The SP Queue Implementation in Detail

In this section, we explain the `enq` and `deq` methods, including the parts omitted in Sec. 3.

A.1 The `enq` Method

```
1  void enq(int x) {
2    while (true) {
3      queue = Queue;
4      if (queue->Invalid) {
5        new_queue = CreateNewQueue(x);
6        if (CAS(&Queue, queue, new_queue))
7          exit;
8        continue;
9      }
10     tail = queue->Tail;
11     idx = tail % SIZE;
12     node = queue->pair[idx].last;
13     if (tail == idx) {
14       if (node == NULL) {
15         new_node = Node(x, tail);
16         if (CAS(&queue->pair[idx].head,
17                 NULL, new_node)) {
18           queue->pair[idx].last = new_node;
19           break;
20         } else {
21           if (queue->pair[idx].head == PICKET)
22             queue->Invalid = true;
23           else
24             CAS(&queue->Tail, tail, tail+1);
25           continue;
26         }
27       } else {
28         if (node == PICKET)
29           queue->Invalid = true;
30         else
31           CAS(&queue->Tail, tail, tail+1);
32         continue;
33       }
34     }
35     if (node == NULL) {
36       node = queue->pair[idx].head;
37     }
38     if (node == PICKET) {
39       new_queue = CreateNewQueue(x);
40       if (CAS(&Queue, queue, new_queue))
41         exit;
42       continue;
43     }
44     while (node->next != NULL &&
45            node->ver < tail)
46       node = node->next;
47     if (node->ver >= tail) {
48       CAS(&queue->Tail, tail, tail+1);
49       continue;
50     }
51     if (node != PICKET) {
52       new_node = Node(x, tail);
53       if (CAS(&node->next,
54               NULL, new_node)) {
55         queue->pair[idx].last = new_node;
56         break;
57       }
58     } else {
59       queue->Invalid = true;
60     }
61   }
62   CAS(&queue->Tail, tail, tail+1);
63 }
```

The code of the enqueue method is given above.It has one main loop (lines 2-61) which ends only when `enq` logically inserts its parameter `x` into the queue.

Each iteration starts by reading the current state of the queue, `Queue`. If the pairing array is invalid (line 4), `enq` tries to atomically replace the current queue with a new queue (line 6). If the replacement is successful, the method completes execution since as the new queue is created, the element `x` is already inserted into the first position (line 6 of `CreateNewQueue`, Fig. 7). If the replacement fails, then `enq` starts a new iteration.

If the pairing array was valid at the time line 3 was executed, `enq` proceeds to read the current enqueue counter, stored in `Tail` (line 10). Using this counter, the correct slot index `idx` is determined (line 11). As a first guess where the end of the linked list of `pair[idx]` can be, the helper pointer `last` is read and assigned to the local variable `node` (line 12).

Lines 13-34 take care of the special case where the node to be inserted will be the first node in the linked list (`tail` and `idx` are equal). If `node` is `NULL`, then this can be because no node has been inserted into the list yet, or a node has already been inserted by a concurrent `enq` or `deq` which did not yet set the `last` pointer. Assuming the former case, `enq` method creates a new node (line 15) and attempts to insert it as the first node of the linked list by atomically changing the value of the `head` pointer from the expected value `NULL` to the new node `new_node` (lines 16-17). If the replacement is successful, `last` is set to point to `new_node` and the loop terminates. If the replacement fails, then this could be either because a concurrent `enq` method has successfully inserted its node, or because a concurrent `deq` has invalidated the pairing array. If the former, the current iteration is simply aborted after atomically incrementing the value of the enqueue counter, `Tail` (line 24). Otherwise, `head` must point to the special node `PICKET` (line 21), which means that the `enq` method should abort trying to insert its element to the current invalid copy. Before aborting the current iteration, it makes sure that the current copy is marked invalid by setting the `Invalid` flag. Note that the update of the flag without synchronization is safe, because at no point can an asserted flag be reset to false. If `node` is not `NULL` (lines 27-33), then again this is either because a node is inserted by a concurrent `enq` or because the queue has been invalidated by a concurrent `deq`. It is treated in a similar manner as the previous paragraph.

If this was not the special first round of insertions (lines 35-60), the first check is to make sure that `node` is not `NULL` (line 35). If so, then `node` is assigned to `head`, which cannot be `NULL` at this point. Then, it checks whether the node pointed to by `last` is `PICKET` (line 38). If it is, then an attempt to create a new queue is done (line 40). If successful, the method completes (line 41); otherwise, this iteration is aborted (line 42).

If the node is not the last node of the linked list, a loop for reaching the end of the linked list is executed (lines 44-46). The loop goes on until either the end of the list is reached or the `ver` value of the current node is greater than or equal to the local enqueueing ticket, `tail`. If the end of the list is not reached because a node with a `ver` greater than or equal to `tail` exists, it means that a concurrent `enq` has succeeded in inserting a node with the same enqueue counter. This case is taken care of by making sure that the global enqueue counter `Tail` is at least one more than the local enqueue counter and aborting the current iteration (lines 47-50). Otherwise, `node` was pointing to the end of the list when it was last accessed (line 44). This last node is either the node with `ver` equal to `tail-SIZE` or is the `PICKET`. Note that no other possibility exists; `ver` cannot be less than `tail-SIZE`. So, if the node is not the `PICKET`, the `enq` method tries to append the new node `new_node` to the linked list (lines 53-54). If successful, the `last` pointer is set to point to this new node and the loop terminates. Otherwise, the next iteration starts. As usual, if the last node is the `PICKET`, the queue is marked invalid and the next iteration starts (lines 58-59).

Finally, when the loop terminates, the `enq` method ensures that the global enqueue counter `Tail` is at least one more than the local enqueue counter `tail` (line 62). Note that, the local enqueue counter `tail` is equal to the `ver` field of the node inserted by this `enq` method.

A.2 The `deq` Method

```
1  int deq() {
2    queue = Queue;
3    if (queue->Invalid)
4      return EMPTY;
5    ticket = FAI(queue->Cnt_deq);
6    idx = ticket % SIZE;
7    if (ticket>queue->=Tail &&
8        ticket==idx) {
9      if (CAS(&queue->pair[idx].head,
10         NULL,PICKET)) {
11       Close(queue,idx);
12       return EMPTY;
13     }
14   }
15   node = queue->pair[idx].removed;
16   if (node == NULL)
17     node = queue->pair[idx].head;
18   if (node == PICKET) {
19     Close(queue,idx);
20     return EMPTY;
21   }
22   if (node->ver > ticket)
23     node = queue->pair[idx].head;
24   while (node->ver < ticket) {
25     if (node->next == NULL) {
26       if (CAS(&node->next,
27               NULL,PICKET)) {
28         Close(queue,idx);
29         return EMPTY;
30       }
31     }
32     node = node->next;
33     if (node == PICKET) {
34       Close(queue,idx);
35       return EMPTY;
36     }
37   }
38   x = node->val;
39   queue->pair[idx].removed = node;
40   return x;
41 }
```

The code of the dequeue method is given above.Similar to `enq`, `deq` starts by copying the current state, `Queue`, into its local variable `queue` (line 2). If the queue has already been invalidated, `deq` ends by returning `EMPTY`.

If the queue was valid at the time line 2 was executed, `deq` receives its ***unique*** removal ticket. This is done by atomically reading the current value of the global dequeue counter `Cnt_deq` and incrementing it by one (line 5). The correct slot index is the `ticket` modulo the size of the array (line 6). The `deq` method is paired up with the node having its `ver` equal to `ticket` found in the slot with index `idx`. If such a node already is in the linked list, then there will be no need for further (expensive) synchronization primitives.

Much like the `enq` method, the first round of entries in the array is treated as a special case (lines 7-14). If this is the first round, i.e., if `deq` is supposed to return the value of the first node of the linked list of some slot, it first checks whether the number of completed enqueues, value of `Tail`, is less than or equal to the local dequeue counter `ticket`. If this is the case, it tries to invalidate the current queue (lines 9-11). Note that, equality between `Tail` and ticket implies that there might be a concurrent enq which has already inserted a node waiting to update the value of Tail. That is why deq checks whether head of the current slot is NULL, signifying no insertion so far, or not. If the atomic insertion of the special node PICKET is successful, the current queue is invalidated by a call to Close, which simply sets Invalid to true and sets the removed pointer point to PICKET. The deq method terminates by returning EMPTY (line 12).

If control reaches line 15, it must mean that either this is not first round, or there is some node in the linked list of slot with index idx. In either case, it means that the head pointer of the linked list cannot be NULL. However, instead of starting from the very first element in the linked list, deq starts by finding the node pointed to by removed (line 15). Recall that the removed pointer does not necessarily point to the last removed node. In order to make sure that the node deq is pointing is not NULL, a final check is done to cover for the case removed is equal to NULL (line 16). If so, then deq starts its search from the beginning of the list (line 17). It then checks again whether head points to PICKET (line 18). If so, it completes the invalidation procedure and returns EMPTY (line 16-17). Another possibility is that removed points to a node that was removed by a deq instance with a greater dequeue counter. Note that, in such a case the other deq instance must have a dequeue counter equal to $\texttt{ticket} + k\texttt{SIZE}$ for some $k \in \mathbb{N}$. If this is the case, there are two consequences. Firstly, the node that this deq instance is going to remove has already been inserted into the linked list. Secondly, that node is located between head and removed. The check and the proper adjustment to start searching is done in lines 22-23.

Once node is properly set, the node to remove is searched in such a way that if it is there, it is found; if not, the queue is invalidated (lines 24-37). At each iteration, first the enqueue counter of the current node is checked (line 24). If that value is not less than the local dequeue counter ticket, the only possibility (that they are equal) implies that the correct node has been found and the loop terminates. Otherwise, deq checks whether this is the last node in the linked list (line 25). If it is, it implies that the node that deq is supposed to removed has not been inserted into the queue yet. Thus, deq tries to invalidate the queue by appending the PICKET to the current linked list (lines 26-27). If successful, after completing invalidation deq terminates by returning EMPTY (lines 28-29). If the appending fails, there are two possibilities. Either a concurrent enq instance successfully inserted its element, or a concurrent deq instance invalidated this pairing array. If the latter, deq tries to help the other deq instance by completing invalidation and returns EMPTY (lines 33-36). If the former, deq tries again by advancing to the next (non-NULL) node (line 32).

Finally, if deq exits the loop, it means that it has found the node that it was required to remove. It reads the value stored in the node (line 38), sets the removed pointer point to the node it is about to logically remove (line 39), and completes by returning the desired value (line 40).

B Proof of Correctness

In this section, we are going to prove certain properties about our implementation. We will show that the deq method contains cooperative statements for $\{\texttt{deq}\}$, and a bounded competitive statement for $\{\texttt{enq}\}$. Since we essentially use the same approach as the Michael-Scott queue for enqueueing elements, the enq method contains unbounded competitive statements. We finally prove that the SP implementation is linearizable.

Proposition B.1 *The FAI statement of line 5 of the* deq *method is cooperative.*

Proof. A control flow analysis trivially shows that regardless of the interference by concurrent threads, the control can never reach line 5 more than once. This means that in any execution of a deq instance, this statement will occur exactly once. □

Proposition B.2 *The CAS statement of line 9 of the* deq *method is cooperative.*

Proof. Again, control never reach line 9 more than once. Thus, in each execution of deq, line 9 occurs at most once. □

Proposition B.3 *The CAS statement of line 26 is cooperative for* $\{\texttt{deq}\}$.

Proof. Let us denote the CAS statement of line 26 with s_{26}. The only time a deq instance will try to modify the structure of the linked list is when it tries to invalidate the pairing array by inserting the PICKET node. Note that if s_{26} succeeds, the method completes at line 29. In this case s_{26} will only appear once in the execution. Otherwise, if

s_{26} fails, then since by assumption we only have deq instances, it must be due to another CAS statement. Regardless of which CAS that actually is (of line 9 or of line 26), the end result will be the same: the next node in the linked list will be the PICKET. Note that the next link of a node is updated only when it is pointing to NULL. In fact, this applies to all the CAS statements in the implementation, both those of deq and those of enq instances. Thus, once the next pointer is set to PICKET, it will remain unchanged. This implies that after executing line 32, node will be equal to PICKET and the method will complete. □

Before we state the bounded competitive result for the CAS statement of line 26, we have to state several invariants.

Lemma B.4 *At any state, $k <$ queue->Tail implies that there is a node with* ver *equal to k in the slot* Queue->pair[j]*, where $j = k\%$SIZE.*

Proof. As long as the pairing array remains valid, the value of Queue->Tail never decreases. A node with ver is inserted only if at some state prior to the insertion Queue->Tail was equal to ver. This is easy to see since ver always gets the value of tail whose value is only assigned to queue->Tail. Finally, Queue->Tail is incremented exactly once between two consecutive insertions. □

Lemma B.5 *A linked list of a slot can contain at most one* PICKET *and if it contains one, it is the last node of the list.*

Proof. As we have already discussed above, PICKET can only replace NULL. If the invalidation occurs during the first round (line 9 of deq), the linked list after the PICKET insertion contains only PICKET because head was NULL. Let us consider the case where the last node in the list is PICKET. The helper pointer removed either is NULL or points to some node that can reach PICKET. The pointer head cannot be equal to NULL because the list contains at least one node. Note that, the stale segment removal can never set head to NULL. Then, prior to the check at line 24, either node is PICKET or is not the last node. If it is PICKET, then we are done. Otherwise, we enter the loop. In the loop node can never be NULL and either it will be equal to the node with the right ticket or it will be PICKET. The version number of PICKET is -1 so it can never be equal to ticket. In either case, no update to the list is possible. Thus, if a linked list contains PICKET, there is exactly one PICKET and that is the last node. □

Lemma B.6 *A node n points to a node m only when n is not* PICKET*, and either m is* PICKET *or $n.\texttt{ver} + \texttt{SIZE} = m.\texttt{ver}$.*

Proof. The first part is immediate from the previous lemma. We only have to prove that $n.\texttt{ver} + \texttt{SIZE} = m.\texttt{ver}$. It is trivial to show that $m.\texttt{ver} - n.\texttt{ver}$ is an integer multiple of SIZE. Similarly, it is easy to show that the linked list is always in ascending order. Note that m cannot be inserted in the first round, because that requires an empty list, contradicting the assumption that n is already in the list. Then, m must have been inserted by CAS statement in line 53 of enq. That CAS statement is only reached if node->ver is less than tail, node is the last node in the list and is not PICKET. Furthermore, by Lemma B.4, node->ver is equal to tail-SIZE. This completes the proof. □

Now we are ready to state the bounded competitive property.

Proposition B.7 *The CAS statement of line 26 is bounded competitive for* $\{\texttt{enq}\}$.

Proof. Let j denote the value of the ver field of the last node of the list queue->pair[idx] immediately following the execution of line 6 by the deq instance. Let k denote the value of ticket. Then, s_{26} is $n = 1 + (k - j)/\texttt{SIZE}$-bounded competitive. We have to show that deq can fail in setting the last node to PICKET at most n times. First observe that, deq instance will not try to update the linked list if a node with ver equal to ticket is in the list. If the node is not in the list, in the worst case, the first time deq fails will be against the enq instance inserting the node with ver equal to $j + \texttt{SIZE}$, by Lemma B.6. The second time, failure will be due to the insertion of the node with ver equal to $j + 2\texttt{SIZE}$. Thus, PICKET insertion will fail at most n times after which the node will be in the list and deq will not attempt another CAS. □

As a consequence, we have the following result.

Theorem B.8 *SP queue is lock-free. In particular, the* `deq` *method is wait-free.*

The reason why SP queue is not wait-free is because an `enq` instance might starve by failing every time it tries to insert its item. It is lock-free, however, because there is always one CAS attempt that will be successful and thus there will always be one `enq` instance that completes its execution. This argument is similar to the lock-freedom argument for Michael-Scott queue from which the SP queue is derived.

Theorem B.9 *SP queue is linearizable.*

Proof. (Sketch) The commit points of methods are not fixed and depend on the ordering of certain events during their execution. We will now explain where each method, depending on its execution, commits. The rest of the proof is a tedious but straightforward analysis showing that a sequential execution constructed by replacing the concurrent executions of methods with their sequential versions according to the order of their commit points is a valid queue behavior.

Let $e(q, k)$ denote the `enq` instance which inserts the node with `ver` $= k$ for `Queue` $= q$. Let $\mathbf{r}_e$ denote the execution of an `enq` instance $e(q, k)$. The commit point of $e(q, k)$ is specified as follows:

- If in $\mathbf{r}_e$ there is no creation of a new pairing array (lines 6 or 40, executed by a concurrent `enq` instance), then the commit point of $e(q, k)$ is the successful execution of either line 16 (first round insertion) or line 53 (ordinary insertion).
- If $e(q, k)$ replaces the pairing array (successful execution of either line 6 or line 40), then the commit point of $e(q, k)$ is the execution of the successful replacement.
- If $k > 0$ and $e(q, k)$ inserts into a pairing array that is invalidated in $\mathbf{r}_e$ by a concurrent `enq` instance e', then the commit point of $e(q, k)$ comes after the commit point of $e(q, k-1)$ and the invocation of $e(q, k)$. If $k = 0$, then the commit point of $e(q, k)$ is the last occurrence of the execution of the statement at line 3 (reading `Queue`). Note that, in this case, the commit point of e has to precede the commit point of e', because otherwise $e(q, k)$ could not read the invalid pairing array.

Let $d(q, k, v)$ denote the `deq` instance which executes line 5, observes q as the value of `Queue`, k as the value of `q->Cnt_deq`, and v is the non-`EMPTY` value it returns. Let $d(q, k)$ denote a `deq` instance which executes line 5, observes q as the value of `Queue`, k as the value of `q->Cnt_deq`, and returns `EMPTY`. Similarly let $d(q)$ denote the `deq` instance which executes line 4, and observes q as the value of `Queue`.

Let $\mathbf{r}_{dv}$ denote the execution of a `deq` instance $d(q, k, v)$. The commit point of $d(q, k, v)$ is specified as follows:

- If the node with `ver` $= k$ is inserted before $\mathbf{r}_{dv}$ has started and there is no pairing array replacement in $\mathbf{r}_{dv}$, then the commit point of $d(q, k, v)$ is the execution of line 5.
- If the node with `ver` $= k$ is inserted in $\mathbf{r}_{dv}$ and there is no pairing array replacement in $\mathbf{r}_{dv}$, then the commit point of $d(q, k, v)$ is immediately after the commit point of $e(q, k)$.
- If there is a pairing array replacement in $\mathbf{r}_{dv}$ by an `enq` instance e, then the commit point of $d(q, k, v)$ is before the commit point of e and after the commit points of both $e(q, k)$ and $d(q, k-1, v')$. If $k = 0$, then the commit point of $d(q, k, v)$ is the execution of line 2. For $k > 0$, the commit point of $e(q, k)$ precedes the commit point of e (see the commit points of `enq` instances above). That the commit point of $d(q, k-1, v')$ comes also before that of e can be proved by induction on k.

Let $\mathbf{r}_{de}$ denote the execution of a `deq` instance $d(q, k)$. The commit point of $d(q, k)$ is specified as follows:

- If in $\mathbf{r}_{de}$ there is no pairing array replacement, then the commit point of $d(q, k)$ is the response of $d(q, k)$.
- If in $\mathbf{r}_{de}$ there is a pairing array replacement, then the commit point of $d(q, k)$ is immediately before the commit point of the first $e(q', 0)$, for $q' \neq q$.

Let $\mathbf{r}_{ds}$ denote the execution of a `deq` instance $d(q)$. The commit point of $d(q)$ is specified as follows:

- If in $\mathbf{r}_{ds}$ there is no pairing array replacement, then the commit point of $d(q)$ is the execution of line 3.
- If in $\mathbf{r}_{ds}$ there is a pairing array replacement, then the commit point of $d(q)$ is immediately before the commit point of the first $e(q', 0)$, for $q' \neq q$.

Observe that, for empty returning `deq` instances, the particular linear order is not specified; any total ordering of those instances subject to the constraints given above will give a valid linearization of a queue behavior. □

```
int deq()
  while (true)
    h = Head;
    t = Tail;
    c = h.next;
    if (h == Head)
      if (h == t)
        if (c == NULL)
          return E;
      else
        x = c.val;
        if (CAS(&Head, h, c))
          break;
```

Figure 3: The `deq` method of the Michael-Scott queue.

```
void enq(int x)
  n = new_node(x);
  while (true)
    t = Tail;
    c = t.next;
    if (t == Tail)
      if (c == NULL)
        if (CAS(&t.next,c,n))
          exit;
      else
        CAS(&Tail,t,c);
  CAS(&Tail,t,c);
```

Figure 4: The `enq` method of the Michael-Scott queue.

```
typedef struct QueueType {
  boolean Invalid;
  int Cnt_deq;
  int Tail;
  SlotType pair[SIZE];
} QueueType;

typedef struct SlotType {
  NodeType * head, last, removed;
} SlotType;

typedef struct NodeType {
  int val;
  int ver;
  NodeType * next;
} NodeType;
```

Figure 5: The main data types used in our implementation.

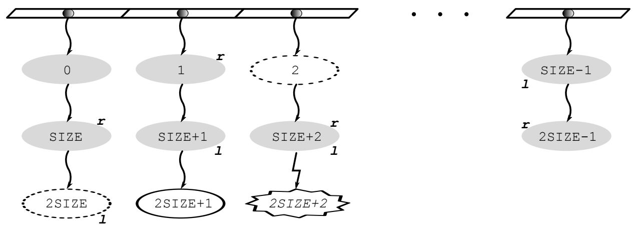
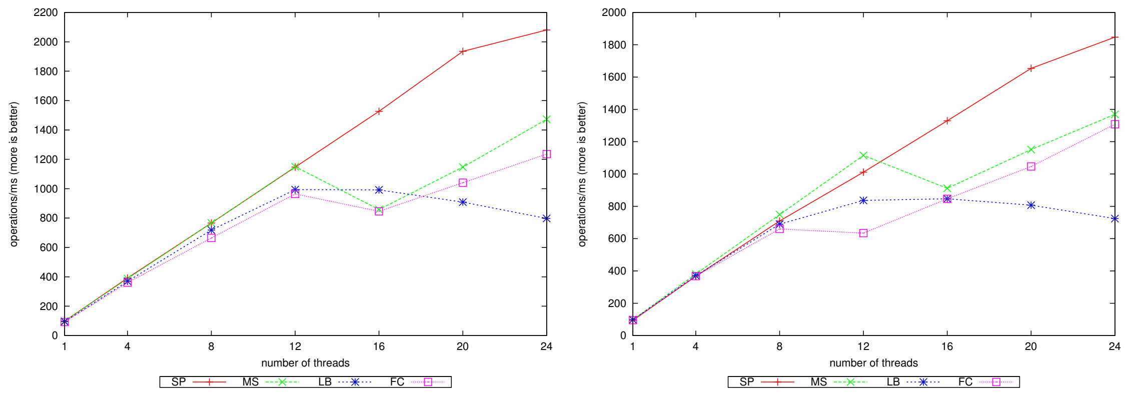


Figure 6: A typical state of the low-level representation of the implementation.

```
QueueType * CreateNewQueue(int x) {
  new_queue = new QueueType(SIZE);
  new_queue->Invalid = false;
  new_queue->Cnt_deq = 0;
  new_queue->Tail = 1;
  new_node = new NodeType(x,0);
  new_queue->pair[0].head = new_node;
  new_queue->pair[0].last = new_node;
  return new_queue;
}

void CloseQueue(QueueType * q, int idx) {
  q->Invalid = true;
  q->pair[idx].removed = PICKET;
}
```

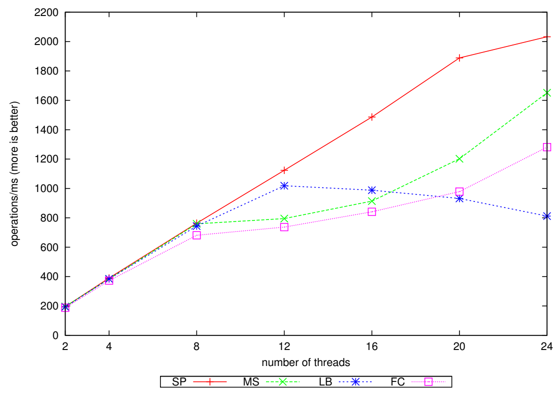
Figure 7: The auxiliary routines used in the implementation.



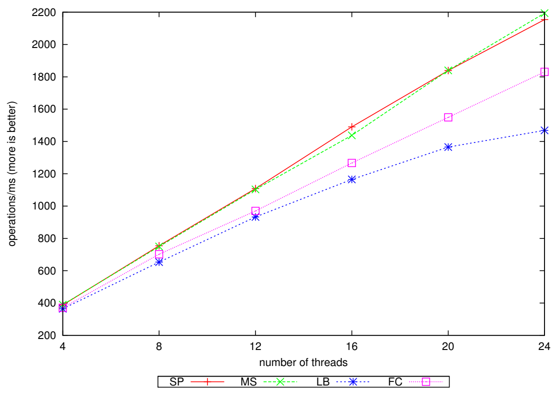
(a) Strict Enq-Deq

(b) Random Enq-Deq

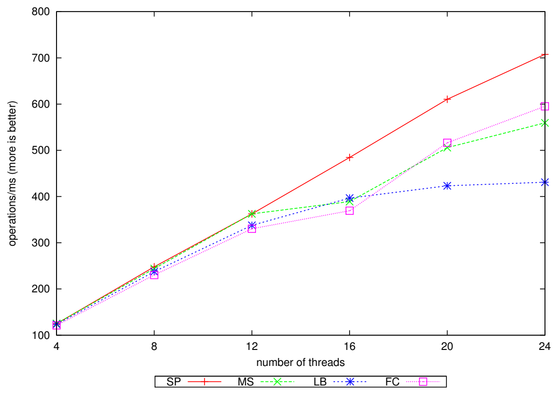
Figure 8: Benchmarks on a 24-core server for almost equal number of `enq` and `deq` instances.



(a) 1:1 Prod-Cons

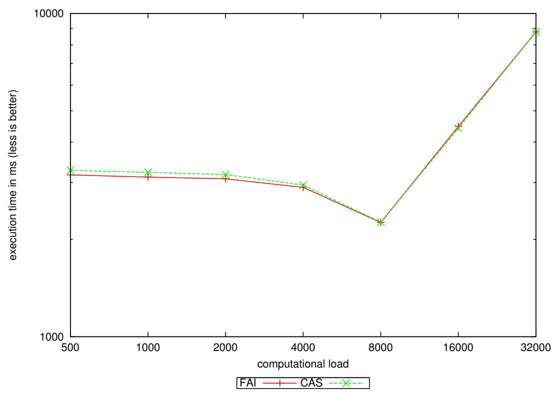


(b) 1:3 Prod-Cons

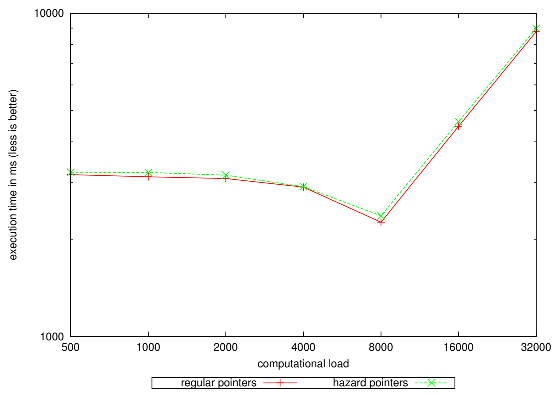


(c) 3:1 Prod-Cons

Figure 9: Benchmarks on a 24-core server for various Producer-Consumer ratios.



(a) FAI vs. CAS.



(b) Impact of hazard pointers on performance.

Figure 10: Additional performance figures for 24 threads. Contention decreases along x-axis.