How Free is Your Linearizable Concurrent Data Structure?

Thomas A. Henzinger[1] and Ali Sezgin[1]

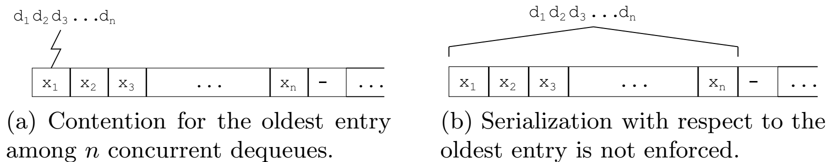
IST Austria
{tah,asezgin}@ist.ac.at

Abstract. Linearizability requires that the outcome of calls by competing threads to a concurrent data structure is the same as *some* sequential execution where each thread has exclusive access to the data structure. In an ordered data structure, such as a queue or a stack, linearizability is ensured by requiring threads commit in the order dictated by the sequential semantics of the data structure; e.g., in a concurrent queue implementation a dequeue can only remove the oldest element.
In this paper, we investigate the impact of this strict ordering, by comparing what linearizability allows to what existing implementations do. We first give an operational definition for linearizability which allows us to build the most general linearizable implementation as a transition system for any given sequential specification. We then use this operational definition to categorize linearizable implementations based on whether they are bound or free. In a bound implementation, whenever all threads observe the same logical state, the updates to the logical state and the temporal order of commits coincide. All existing queue implementations we know of are bound. We then proceed to present, to the best of our knowledge, the first ever free queue implementation. Our experiments show that free implementations have the potential for better performance by suffering less from contention.

1 Introduction

Concurrent data structures form the crux of every application software intended to run on a multi-core architecture which covers almost all computing domains. This is why correct and scalable implementations of concurrent data structures, especially at the library level, is crucial. The common perception, however, indicates that there is a trade-off between scalability and correctness for such implementations. Ideally, correctness is captured by *linearizability* [6]: A concurrent implementation of data structure D is correct if it is linearizable with respect to the sequential definition of D. In this paper, we pose the following question: Have the existing implementations reached the limits of linearizability? We argue that there exist promising optimizations that future linearizable implementations can use.

As a motivating example, think of the following scenario. We have a concurrent queue implementation and the execution has reached a point where there



(a) Contention for the oldest entry among n concurrent dequeues.

(b) Serialization with respect to the oldest entry is not enforced.

Fig. 1: Representative contention for concurrent dequeues.

are n elements in the queue, $\mathtt{x_1}$ to $\mathtt{x_n}$ with $\mathtt{x_1}$ being the oldest, and there are n dequeues, $\mathtt{d_1}$ to $\mathtt{d_n}$, executing concurrently. Existing concurrent queue implementations proceed by removing the elements in the queue in the correct order, illustrated in Fig. 1a. What is not known and scheduler dependent is which dequeue instance $\mathtt{d}_i$ is going to return the first element, $\mathtt{x_1}$. It is this non-determinism that leads to a better use of computational resources. However, the serialization implied by this approach results in high contention for the same element since all the dequeue instances initially try to remove the same element, $\mathtt{x_1}$. Once the oldest element is removed, then the remaining $n-1$ dequeue instances compete for the current oldest element $\mathtt{x_2}$, perpetuating high contention for the oldest element. Implementations following this serialization scheme are called *bound*.

We observe that this high contention scenario is not necessary to guarantee linearizability. In fact, at one extreme, with the scenario above, each dequeue instance can remove any element from the queue, as illustrated in Fig. 1b. In fact, as long as each element is removed exactly once, any permutation of removal of the elements in the queue will yield a linearizable execution.

It is desirable to have implementations that are not bound, which we call *free*. We show that neither the Herlihy-Wing queue [6] nor the Michael-Scott queue [10] is free. In fact, to the best of our knowledge, all existing implementations of concurrent queue implementations are bound.

As far as whether one can realize a free implementation, we provide two solutions. The first of these solutions, on a more abstract level, is based on an operational definition of linearizability. This definition allows us to effectively keep track of all possible linearizations of a given concurrent execution in an inductive manner. We use this definition to give a generic (keeping the sequential specification S as a parameter) coarse-grained reference model for all possible linearizable implementations of S.

The second solution is an instance of a concurrent queue implementation which is free. This implementation is derived from the Michael-Scott queue by adding a contention manager. We show that the manager indeed reduces contention, measured through the number of failed Compare-And-Swap calls, resulting in increased performance.

Related Work. To the best of our knowledge, this is the first work that investigates the gap between what linearizability allows and what existing implementations do. In the domain of concurrent queue implementations, with which we are mainly concerned in this paper, there have been extensive work on designing more efficient implementations (e.g., [4, 7, 8, 10, 12]) and on verify-

ing these implementations correct (e.g., [1, 2, 9, 13, 14]). However, none of these queue implementations is free. Thus, the free queue implementation presented in this paper is, to the best of our knowledge, the first of its kind. On the other hand, without being identified as such, a free linearizable implementation does exist [5]. We should note that the sequential behavior of the counting network of [5] can also be specified using a non prefix-closed set, in contrast to our definition of free which requires a prefix-closed sequential specification.

Recently, a different taxonomy of linearizable implementations has been suggested by [3], which introduces the notion of strong linearizability. Intuitively, an implementation is strongly linearizable if its linearization does not depend on future behavior. The class of bound implementations properly subsumes the class of strongly linearizable implementations; e.g., the Herlihy-Wing queue is not strongly linearizable, even though it is bound.

2 Preliminaries

In this section, we develop the formal framework, by introducing notation and necessary terminology, in order to present our main theoretical results.

Notation. Let A be a set. A^* and $\mathscr{P}(A)$ denote the set of all sequences over A and the power-set (set of all subsets) of A, respectively. Empty string and empty set are denoted by ε and $\varnothing$, respectively. For any sequence $\mathbf{s} \in A^*$ and any subset $B \subseteq A$, $\mathbf{s}\downarrow_B$ denotes the subword obtained by removing from $\mathbf{s}$ all symbols in $A \backslash B$. We call sequence $\mathbf{x}$ a prefix of $\mathbf{y}$, written $\mathbf{x} \preceq \mathbf{y}$, whenever there exists a sequence $\mathbf{z}$ such that $\mathbf{y}$ is the concatenation of $\mathbf{z}$ to $\mathbf{x}$; that is, $\mathbf{y} = \mathbf{xz}$. If $\mathbf{z} \neq \varepsilon$, $\mathbf{x}$ is a proper prefix of $\mathbf{y}$, written $\mathbf{x} \precneq \mathbf{y}$. A subset B of A^* is called *prefix-closed* if $\mathbf{x} \preceq \mathbf{y}$ and $\mathbf{y} \in B$ imply $\mathbf{x} \in B$. For subsets $A_1, \ldots, A_k \subseteq A$, the set $A_1 A_2 \ldots A_k$, denotes the set of all sequences $\mathbf{a}_1 \ldots \mathbf{a}_k$, where $\mathbf{a}_i \in A_i$.

A labelled transition system (LTS) is a tuple $M = (Q, q_{init}, L, \rightarrow)$, where Q is the set of *states*, q_{init} is the initial state, L is the set of *labels*, and $\rightarrow \subseteq Q \times L \times Q$ is the transition relation. A *run* of M is an alternating sequence $\mathbf{r} = q_0 l_1 q_1 \ldots l_n q_n$ of states and labels such that $q_0 = q_{init}$ and for all $1 \leqslant i \leqslant n$, $q_{i-1} \xrightarrow{l_i} q_i$. The sequence of labels $\mathbf{l} = l_1 l_2 \ldots l_n$ is the *trace* of $\mathbf{r}$. The language of M, written $L(M)$, contains all sequences $\mathbf{l} \in L^*$ such that there is a run of M with trace $\mathbf{l}$.

We represent the set of natural numbers with $\mathbb{N}$. $\widetilde{A}$ is a shorthand for $\mathbb{N} \times A = \{(n, a) \mid n \in \mathbb{N}, a \in A\}$. For a collection of sets $A_1 \ldots A_k$, $A_1 \times \ldots A_k$ denotes the set of all k-tuples $(a_1, \ldots, a_k)$ such that $a_i \in A_i$. For any k-tuple $a = (a_1, \ldots, a_k)$, $\pi_i(a)$ denotes the projection of a on to the ith component, a_i. We extend π_i pointwise to sequences over k-tuples. For any sequence $\widetilde{\mathbf{a}}$ over $\widetilde{A}$, $\widetilde{\mathbf{a}}\downarrow_i$ is a shorthand for $\widetilde{\mathbf{a}}\downarrow_{\{i\}\times A}$.

Sequential specification, refinement. A *sequential specification* S is a prefix-closed set of sequences, called *sequential behaviors*, over a set $\mathcal{M}$, called *sequential alphabet*. Let $\mathcal{M}$ be a sequential alphabet and $\mathcal{A}$ be a set. A mapping ρ from $\mathcal{M}$ to $\mathscr{P}(\mathcal{A}^*)$ is called an $\langle \mathcal{M}, \mathcal{A} \rangle$-refinement if

- for any $m \neq m'$, $\rho(m) \cap \rho(m') = \varnothing$, and

– for any $m \in \mathcal{M}$ and $\mathbf{a} \in \rho(m)$, there does not exist a sequence $\mathbf{a}' \neq \varepsilon$ and $m' \in \mathcal{M}$ such that $\mathbf{a}\mathbf{a}' \in \rho(m')$.

Intuitively, each sequence in $\rho(m)$ gives one possible execution path of a procedure implementing m in the programming language $\mathcal{A}$.

An $\langle \mathcal{M}, \mathcal{A} \rangle$-refinement ρ is called *canonical* if $\mathcal{A} = \{m_i | m \in \mathcal{M}\} \cup \{m_r | m \in \mathcal{M}\}$ and for all $m \in \mathcal{M}$, $\rho(m) = \{m_i m_r\}$. Canonical refinement simply splits each method into invocation and response events, m_i and m_r, respectively.

With an abuse of notation, we set $\rho^{-1}(\mathbf{a}) = m$ iff $\mathbf{a} \in \rho(m)$. The sequences for which ρ^{-1} is defined are called *complete* with respect to ρ.

The *fate set* of a sequence $\mathbf{a} = a_1 \dots a_j \in \mathcal{A}^*$, $\mathit{fate}_\rho(\mathbf{a})$, is the set of methods m for which there exists a ρ-refinement $\mathbf{a}'$ containing $\mathbf{a}$ as a prefix. Intuitively, method m is in the fate set of a sequence of instructions in $\mathcal{A}$ if there is an execution path of m that starts with the same sequence. Formally, $\mathit{fate}_\rho(\mathbf{a}) = \{m | \exists \mathbf{a}' \in \rho(m). \mathbf{a} \preceq \mathbf{a}'\}$. The sequence $\mathbf{a}$ is called unifate if $|\mathit{fate}_\rho(\mathbf{a})| = 1$.

Sequential execution, ρ-trace. For the following, fix $\mathbf{a} = a_1 \dots a_n$ as a sequence over $\mathcal{A}$. The sequence $\mathbf{a}$ is called *quiescent* if it is the concatenation of complete subsequences. Formally, $\mathbf{a}$ is quiescent if there exists a sequence $\mathbf{m} = m_1 \dots m_k$ over $\mathcal{M}$ such that $\mathbf{a} \in \rho(m_1)\rho(m_2)\dots\rho(m_k)$. Observe that if $\mathbf{m}$ exists, it is unique. The methods m_1 to m_n are said to *occur* in $\mathbf{a}$. For quiescent $\mathbf{a}$, let $\rho^{-1}(\mathbf{a})$ denote $\mathbf{m}$.

Let $\mathbf{a}^q$ and $\mathbf{a}^r$ denote the partition of $\mathbf{a} = \mathbf{a}^q\mathbf{a}^r$ such that $\mathbf{a}^q$ is the maximal quiescent prefix of $\mathbf{a}$. Then, a sequence $\mathbf{a}^q\mathbf{a}^t$ is called a *completion* of $\mathbf{a}$, if either $\mathbf{a}^t = \varepsilon$, or $\mathbf{a}^r \neq \varepsilon$ and $\mathbf{a}^t \in \mathit{fate}_\rho(\mathbf{a}^r)$. In other words, a completion of $\mathbf{a}$ either discards the incomplete suffix, or extends the incomplete suffix to some complete sequence. Let $CS_\rho(\mathbf{a})$ denote the set of all completions of $\mathbf{a}$. By convention, we set $CS_\rho(\varepsilon) = \varnothing$.

The sequence $\mathbf{a}$ is called a *ρ-trace* if it can be extended into a quiescent sequence. Formally, either $\mathbf{a} \in CS_\rho(\mathbf{a})$ or $|CS_\rho(\mathbf{a})| > 1$. Note that, any prefix of a ρ-trace is also a ρ-trace.

A method m_i has *terminated* in $\mathbf{a}$ if m_i occurs in $\mathbf{a}^q$. The method m_k is *executing* in $\mathbf{a}$ if m_k has not terminated in $\mathbf{a}$ and there is a (quiescent) completion of $\mathbf{a}$ in which m_k occurs.

Concurrent execution, linearizability. A sequence $\mathbf{c}$ over $\widetilde{\mathcal{A}}$ is called a *concurrent execution induced by ρ*, if for all $i \in \mathbb{N}$, $\pi_2(\mathbf{c} \downarrow i)$, called the *local history of thread i*, is a ρ-trace. We will also use $\mathbf{c}[i]$ as a shorthand for $\pi_2(\mathbf{c} \downarrow i)$. Observe that for the canonical $\langle \mathcal{M}, \mathcal{A} \rangle$-refinement, concurrent execution is equivalent to the well-known notion of *history* [6]. We call i *idle after* $\mathbf{c}$ if the local history of thread i in $\mathbf{c}$ is quiescent.

Let $\mathbf{c}$ be a concurrent execution induced by ρ and $m, n \in \mathcal{M}$ be two method instances such that there are $i, j \in \mathbb{N}$ for which m (resp., n) occurs in some completion sequence of the local history of thread i (resp., j). The method instance m *precedes* n in $\mathbf{c}$, written $m \rightarrowtail n$, if there exists a prefix $\mathbf{c}'$ of $\mathbf{c}$ such that i is idle after $\mathbf{c}'$, m appears in the local history of thread i in $\mathbf{c}'$, and n is neither executing nor has terminated in the local history of thread j in $\mathbf{c}'$.

Let $\mathbf{c}$ be a concurrent execution induced by ρ. The sequence $\mathbf{s_c} \in \mathcal{M}^*$ is a *linearization* of $\mathbf{c}$ if there is a collection of sequences $\mathbf{a}_i \in CS_\rho(\mathbf{c}[i])$ such that $\mathbf{s_c}$ is a permutation of methods appearing in all $\mathbf{a}_i$, and if $m \rightarrowtail n$ in $\mathbf{c}$, then m appears before n in $\mathbf{s_c}$.

A *concurrent implementation* I of a sequential specification S is a tuple $(\mathcal{A}, \rho, M_I)$, where $\mathcal{A}$ is the set of actions, ρ is an $\langle\mathcal{M}, \mathcal{A}\rangle$-refinement and $M_I = (Q, q_{init}, \widetilde{\mathcal{A}}, \rightarrow)$ is an LTS such that each trace in $L(M_I)$ is a concurrent execution induced by ρ. A concurrent implementation I of S is *linearizable* if for every trace $\mathbf{c} \in L(M_I)$, there exists a linearization $\mathbf{s_c} \in S$.

3 Operational Definition for Linearizability

In this Section, we will give an alternative and operational definition for linearizability. We will first describe the algorithm which computes on-the-fly a set of sequences as possible linearizations of the concurrent execution. We will then state the main result of the section which establishes the equivalence between the existence of possible linearizations and linearizability.

For the following, let us fix a sequential specification S over $\mathcal{M}$ and a $\langle\mathcal{M}, \mathcal{A}\rangle$-refinement ρ. Let Ω stand for $\widetilde{\mathcal{A}^*} = \{(i, \mathbf{a}) | i \in \mathbb{N}, \mathbf{a} \in \mathcal{A}^*\}$. An element $(i, \mathbf{a}) \in \Omega$ is *closed* if there is an $m \in \mathcal{M}$ with $\mathbf{a} \in \rho(m)$; otherwise, $(i, \mathbf{a})$ is *open*. Similarly, a sequence $\omega \in \Omega^*$ is closed (resp., open) if all symbols in ω are closed (resp., open).

Algorithm 1 Computing potential witnesses, $\mathcal{W}(\mathbf{c})$.

```
1:  function Apply(W,(i, a))
2:      W' ← ∅
3:      for all w ∈ W do
4:          if there is no open symbol (i, a') in w then
5:              for all x, y such that w = xy and y is open do
6:                  W' ← W' ∪ {x(i, a)y}
7:          else
8:              W' ← W' ∪ {x(i, a'a)y}, where w = x(i, a')y
9:      for all w ∈ W' do
10:         if w = x(i, a')y and x is closed then
11:             if a'' ∈ fate_ρ(a') implies ρ^{-1}(π_2(x · (i, a''))) ∉ S then
12:                 W' ← W'\{w}
13:     W(c) ← W'
```

Let $\mathbf{c} = (i_1, a_1) \ldots (i_n, a_n)$ be a concurrent execution induced by ρ. The set of *potential witnesses* for the concurrent execution $\mathbf{c}$, $\mathcal{W}(\mathbf{c}) \subseteq \Omega^*$, is defined inductively as follows:

- $\mathcal{W}(\varepsilon) = \{\varepsilon\}$

– $\mathcal{W}((i_1,a_1)\dots(i_j,a_j)) = \mathsf{Apply}(\mathcal{W}((i_1,a_1)\dots(i_{j-1},a_{j-1})),(i_j,a_j))$

where the function Apply constructs a new potential witness set, for a given potential witness set and a symbol in $\widetilde{\mathcal{A}}$. The definition of Apply is given in Alg. 1. Apply has two phases: the expansion phase (lines 3-9), and the pruning phase (lines 9-12). Given the current set of potential witnesses W and the current symbol (i,a), in the expansion phase, there are two possibilities. Either i was idle and (i,a) is the first action of a new method instance, or (i,a) is the continuation of an executing method. In the first case, all sequences $x(i,a)y$, where x and y is a partitioning of some potential witness in W such that y contains no terminated methods, are added to the potential witness set. In the second case, in each potential witness, (i,a) is concatenated to the unique symbol which corresponds to thread i executing some method.

In the pruning phase, all the sequences generated after the expansion phase are checked whether they can be extended to sequences in the specification. The main observation is that if all symbols in a prefix of w are closed, then no further extension of the current execution will have an effect on that prefix. Thus, if the current modified open symbol to which (i,a) is appended in w comes immediately after a closed prefix of w and no matter how thread i continues its execution, the closed symbol is not allowed to follow the closed symbols in that prefix, w is removed from the potential witness set. Observe that, if at any point we get $\mathcal{W}(\mathbf{c}) = \varnothing$, then for all extensions $\mathbf{c}'$ of $\mathbf{c}$, that is, $\mathbf{c} \preceq \mathbf{c}'$, we will have $\mathcal{W}(\mathbf{c}') = \varnothing$. The following equivalence result follows from these explanations.

Theorem 1. *A concurrent execution* $\mathbf{c}$ *has a linearization in* S *iff* $\mathcal{W}(\mathbf{c}) \neq \varnothing$.

3.1 Most General Linearizable Implementations

Before we move on to defining free linearizable implementations, we present a way to construct an implementation $I(S)$ for a given sequential specification S such that $I(S)$ generates all possible concurrent traces with linearizations in S.

Let S be a sequential specification over $\mathcal{M}$, and let ρ be the canonical $\langle\mathcal{M},\mathcal{A}\rangle$-refinement. The concurrent implementation $I(S)$ is given as $(\mathcal{A},\rho,M_{I(S)})$, where $M_{I(S)}$ is the LTS $(Q,q_{init},\widetilde{\mathcal{A}},\rightarrow)$. Each state $q \in Q$ is a potential witness set. The initial state $q_{init} = \{\varepsilon\}$. The transition $q \xrightarrow{(i,a)} q'$ is allowed iff $q' = \mathsf{Apply}(q,(i,a))$ and $q' \neq \varnothing$. Note that, if there is no transition out of q with label (i,m_r), it means that terminating m_r at q leads to a non-linearizable concurrent execution, by Thm. 1.

Corollary 1. *A concurrent history* $\mathbf{h}$ *has a linearization in* S *iff* $\mathbf{h}$ *is a trace in* $L(M_{I(S)})$.

4 Free Linearizable Implementations

In this section, we are going to formalize free linearizable implementations.

Let S be a sequential specification over $\mathcal{M}$. For any sequence $\mathbf{s} \in S$ and any symbol $m \in \mathcal{M}$, let $d(\mathbf{s}, m)$ be the length of the shortest sequence $\mathbf{t}$ such that $\mathbf{st}m \in S$. Formally, $d(\mathbf{s}, m) = \min\{|\mathbf{t}| \mid \mathbf{st}m \in S\}$. In particular, if $\mathbf{s}m \in S$, $d(\mathbf{s}, m) = 0$.

A specification S over $\mathcal{M}$ is *trivial* if $\forall m \in \mathcal{M}, \mathbf{s} \in S \,.\, d(\mathbf{s}, m) = 0$. Observe that $\varnothing$ and $\mathcal{M}^*$ are the only trivial (prefix-closed) specifications over $\mathcal{M}$. We are now ready to give the main definition of this section.

Definition 1 (Free). *Let S be a non-trivial specification over $\mathcal{M}$, and let $\mathbf{s} \in S$ and $m \in \mathcal{M}$ be such that $d(\mathbf{s}, m) = k > 0$. Let I be a linearizable implementation of S. Let $\mathbf{r} = q_0 \ldots l_s q_s l_{s+1} \ldots l_n q_n$ be a run of M_I such that i) all threads after $\mathbf{c}_1 = l_1 \ldots l_s$ are idle and the only potential witness w in $\mathcal{W}(\mathbf{c}_1)$ is such that $\mathbf{s} = \rho^{-1}(\pi_2(w))$, and ii) the last transition $q_{n-1} \xrightarrow{l_n} q_n$ of $\mathbf{r}$ terminates m. The implementation I is* free *if there are at most $|\mathbf{s}| + k$ unifate method instances in $l_1 \ldots l_n$.*

We call an implementation *bound* if it is not free.

Imagine a concurrent queue which contains $[1; 2]$, 1 being the oldest entry. Assume that there are three concurrent dequeue instances, d_1, d_2, d_3, and that d_1 completes its execution by returning 2. If the implementation is bound, then for all continuations of this execution, the dequeue instance removing 1 will be the same. That is, if there is an extension in which 1 is removed by, say, d_2, then in all extensions where 1 is removed, it will have been removed by d_2. On the other hand, if the implementation is free, there might be two different extensions such that 1 is removed by d_2 in one extension and removed by d_3 in the other.

We will end this section by justifying our claim that $I(S)$ is the most general linearizable implementation.

Proposition 1. *$I(S)$ is free.*

5 Concurrent Queue Implementations

In this section, we will analyze two of the best-known concurrent queue implementations. The first is the Herlihy-Wing queue, abbreviated as HW-queue, given as an example for linearizable data structures in [6]. The second implementation we consider is the Michael-Scott queue[10], abbreviated as MS-queue. Almost all of the state-of-the-art concurrent queue implementations are derived from the MS-queue and the argument presented for the MS-queue applies to all the existing implementations we know of. We will show that both the HW-queue and the MS-queue are bound linearizable implementations.

The sequential alphabet for the queue implementations we consider in this paper, $\mathcal{M}_Q$ is given as the union of $Enq = \{\texttt{enq}(i) | i \in \mathbb{N}\}$ and $Deq = \{\texttt{deq}(i) | i \in \mathbb{N} \cup \{\texttt{NULL}\}\}$. The sequential specification for queues, S_Q, has the usual semantics in that `enq` instances can be appended to any sequence in S_Q whereas which `deq` instance can be appended depends on the contents of the queue. A queue is called *partial* if the queue is empty after a sequence of operations, no `deq` can be

appended. The queue is called *total* if whenever the queue is *empty*, deq(NULL) can be appended. Note that for any given $\mathbf{s} \in S_Q$, there is at most one $d \in Deq$ such that $\mathbf{s}d \in S_Q$.

We note that if $d(\mathbf{s}, m) > 0$, then $m \in Deq$. In other words, for any $m \in Enq$, we have $d(\mathbf{s}, m) = 0$ for all $s \in S_Q$.

```
1: procedure enq(x)
2:     atomic
3:         i ← q.back
4:         q.back ← q.back + 1
5:     end atomic
6:     atomic
7:         q.items[i] ← x
8:     end atomic
```

```
 1: procedure deq()
 2:     while true do
 3:         atomic
 4:             range ← q.back − 1
 5:         end atomic
 6:         for i = 0 to range do
 7:             atomic
 8:                 x ← SWAP(q.items[i], NULL)
 9:             end atomic
10:             if x! = NULL then return x
11:         end for
```

Fig. 2: HW-queue methods.

The Herlihy-Wing Queue. The pseudo-code for the HW-queue is given in Fig. 2. We are going to use the notation enq$[l_i]$ (resp., deq$[l_i]$) to denote the statement at line i of method enq (resp., deq). The atomic regions, delimited by the keywords, **atomic** and **end atomic**, represent code blocks that execute without interference. For atomic regions, the corresponding action is the list of instructions appearing within; e.g., the first atomic region of enq(x) is represented by enq$(x)[l_{3,4}]$. We assume that $\mathcal{A}$ is the set of all possible syntactically correct statements of some programming language, to which all codes presented in this paper belong.

The $\langle \mathcal{M}_Q, \mathcal{A} \rangle$-refinement ρ implied by the given code is all possible unfoldings of the programs following standard control flow. For instance, an enq instance, enq(x) has the unique refinement enq$(x)[l_{3,4}]$enq$(x)[l_7]$. A deq instance, deq(x) has infinitely many refinement sequences, the shortest of which is given by

$$\rho(\texttt{deq}(x)) = \texttt{deq}(x)[l_2]\texttt{deq}(x)[l_4]\texttt{deq}(x)[l_6]\texttt{deq}(x)[l_8]\texttt{deq}(x)[l_{10}]$$

corresponding to finding an enqueued element in $q.items[0]$ and dequeueing it successfullly. We now show that the HW-queue is not free.

Lemma 1. *The HW-queue is bound.*

Proof (Sketch). Observe that $d(\mathbf{s}, m) = k > 0$ implies that m is a deq instance, the state represented by $\mathbf{s}$ has two possibilities: 1) it has greater than k elements, m is removing the $k + 1^{st}$, 2) it has exactly $k - 1$ elements and m is returning an element currently not in the queue. Note that, since the HW-queue is partial, an empty returning deq instance is not possible. These cases are similar, we will

```
1: procedure enq(x)
2:     n ← new(x)
3:     while true do
4:         t ← Tail
5:         c ← t.next
6:         if t == Tail then
7:             if c == NULL then
8:                 if CAS(t.next, c, n) then
9:                     return
10:                else
11:                    CAS(Tail, t, c)
12:    end while
```

```
1: procedure deq()
2:     while true do
3:         h ← Head, t ← Tail, c ← h.next
4:         if h == Head then
5:             if h == t then
6:                 if c == NULL then
7:                     return NULL
8:                 CAS(Tail, t, c)
9:             else
10:                x ← c.val
11:                if CAS(Head, h, c) then
12:                    break
13:    end while
14:    return x
```

Fig. 3: MS-queue methods.

only consider the first case. If $\mathbf{s}\mathbf{t}m \in S_Q$ and $|\mathbf{t}| = k$, then $\mathbf{t}$ is a sequence of k `deq` instances, removing all of the k oldest elements in the queue represented by $\mathbf{s}$. The `deq` instances not concurrent with previously terminated `enq` instances, observe these instances according to the temporal ordering of the first blocks of `enq` instances; i.e. in ascending order of slot indices. The main loop of `deq`(x) starts looking for the element to dequeue from the beginning of the array. A `deq` instance is unifate after a non-`NULL` swap (l_8). Thus, the `deq`(x) can complete only if all of the oldest k elements have been successfully removed, implying that there must be at least k `deq` instances that are unifate. □

The Michael-Scott Queue. The pseudo-code for the MS-queue is given in Fig. 3. We will omit a detailed explanation of this well-known implementation details of which can be found in [10].

Lemma 2. *The MS-queue is bound.*

Proof (Sketch). $d(\mathbf{s}, m) > 0$ only when m is a `deq` instance. The state of the queue is given by the order of the nodes in the linked list, whose first element is always pointed to by the *next* field of the sentinel pointed to by `Head`, and whose last element is either pointed to by `Tail` or the *next* pointer of the node pointed to by `Tail`. The commit point of an `enq` instance is the execution of a successful CAS updating the next pointer of the last node (line 8). Similarly, the commit point of a `deq` instance not returning `NULL` is the execution of a successful CAS updating the `Head` pointer (line 11). Then, if $d(\mathbf{s}, m) = k > 0$, it means that m removes the $k + 1^{st}$ node counting from the sentinel node of `Head`. The commit points mentioned above imply that for m to complete, the `Head` pointer must have been updated at least k many times for m to observe the node, which m is going to remove, pointed to by `Head`. On the other hand, executing l_{11} successfully means that the `deq` instance becomes unifate. These imply that when m terminates there are at least $|\mathbf{s}| + k + 1$ unifate method instances: $|\mathbf{s}|$

comes from the fact that all the method instances in `s` are complete, k is the minimum number of unifate `deq` instances that have managed to update `Head` and 1 is for m, which is unifate when it terminates. □

6 Free Concurrent Queue Implementation

In this section, we will present a free concurrent queue implementation, I_f. Our implementation is derived from the MS-queue.

The low level representation of the queue uses new structures. A *segment* structure has an array of *nodes*, called *seg*, of size `SEGSIZE`. Each node in the *seg* array corresponds to a slot of the queue. The *val* field holds the element, denoting an empty slot if it is equal to -1 (we assume that the queue contains only non-negative values). The *marked* field denotes whether the value in the slot, if valid, has already been removed by a dequeue (*marked*=1) or not (*marked*=0). A segment structure has also an array *deqvec*, again of size `SEGSIZE`. This dequeue vector *deqvec* is used as an under-approximation for the number of concurrent dequeue instances.

The code for the `deq` method is given in Fig. 4. The code for the `enq` method is given in App. A, which is essentially the same as the one given in Fig. 3; each enqueue instance competes for the same available slot in a segment, and when the segment is full a new segment is inserted.

The `deq` method stays in a loop until either an element is removed by this instance (line 18) or the queue is found in an empty state (lines 10-11 and 24-25). The main loop starts by copying the necessary synchronization information into local variables. The variable h is assigned to the current value of $Head$. The pointer f gets the current dequeue vector for the segment pointed to by h. The variable *index* ranging over the slots of the segment is initialized to 0.

The first loop (lines 4-7) skips over all the busy slot of the segment pointed to by h, adjusting *index* accordingly. At the end of this loop, the variable *skipcnt* will hold the number of busy slots that were skipped following the logically removed slot with the greatest index.

Then we enter the main inner loop (lines 8-28). We first check whether the current slot is busy and the current number of skipped slots (line 9). If the slot is busy and we are allowed to skip a busy slot, we increment *index* to point to the next slot (line 21) and start a new iteration. If the slot is not busy or we are not allowed to skip any more slots, we check the contents of the slot. If the slot is empty, we simply return `EMPTY`, without updating any shared variable (lines 10-11). If the slot is not empty, we update the dequeue vector to notify other dequeue instances by marking this slot as busy (line 12). Since marking the bit is not synchronized, it is possible that other dequeue instances are also trying to remove the same element. In order to guarantee exactly one removal by all competing dequeue instances, we attempt to mark the slot as deleted (setting the *marked* field of the slot to 1, line 14). If successful (line 15) and the slot that was marked is the last slot of the segment (line 16), then we also ensure that the $Head$ pointer does not stay pointing to a stale segment (line 17) and complete

```
1: procedure deq()
2:     while true do
3:         h ← Head; f ← &(h->deqvec); index ← 0; skipcnt ← 0
4:         while (*f)[index] ∧ index < SEGSIZE do
5:             index ← index + 1; skipcnt ← skipcnt + 1
6:             if h->seg[index].marked then
7:                 skipcnt ← 0
8:         while index < SEGSIZE do
9:             if !(*f)[index] ∨ skipcnt > LOOKAHEAD then
10:                if h->seg[index].val = −1 then
11:                    return EMPTY
12:                (*f)[index] ← 1
13:                curr ← (h->seg[index]); val ← (curr->val)
14:                if CAS(curr->marked, 0, 1) then
15:                    if index = SEGSIZE − 1 then
16:                        if h->next ≠ NULL then
17:                            CAS(Head, h, h->next)
18:                    return val
19:                else
20:                    skipcnt ← 0
21:            index ← index + 1
22:        t ← Tail
23:        if h = t then
24:            if t->next = NULL then
25:                return EMPTY
26:            else
27:                CAS(Tail, t, t->next)
28:            CAS(Head, h, h->next)
29: end procedure
```

Fig. 4: The dequeue method of the free queue implementation, I_f.

the removal by returning the value held in the slot (line 18). If the CAS fails, we reset the skipped node count (line 20), and move on to the next slot by adjusting *index* (line 21).

If the loop terminates by reaching the end of the segment (line 22), we then check whether this is indeed the last segment (line 23), in which case *Head* and *Tail* should be pointing to the same segment and there would be no other segment (line 24), in which case we return `EMPTY`. Otherwise, that is, if the queue is not empty, we ensure that *Head* does not point to the current stale segment (line 28) and go back to the beginning of the main loop.

Before we proceed to state the properties of I_f, we will illustrate the behavior of the modified `deq` method. A sample configuration is given in Fig. 5. The list contains at least two segments, represented by gray rectangles, and `Tail` ≠ `Head`. The slots of a segment are denoted by rectangles; a cross ('`x`') means that the slot is marked and a plus ('`+`') means that it contains a valid entry ($val \geqslant 0$) and

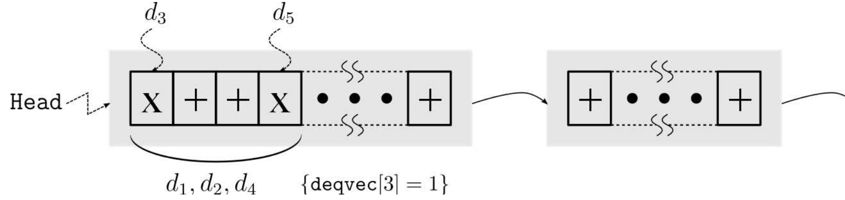


Fig. 5: A representative state of the implementation I_f.

is unmarked. There are currently five concurrent `deq` instances, d_1 to d_5. The current dequeue vector, *deqvec*, by having its first 4 slots set to 1, conveys the information that there are at least four `deq` instances running concurrently. Two of these instances, d_3 and d_5, have successfully marked two slots, the first and the fourth, respectively. The remaining three instances will compete for the second and third slots, but it is not known at this state which instance will remove which slot, which informally shows that the implementation is free. Observe also that any `deq` instance starting after this point will not compete for any of the first four slots, because of the dequeue vector, which demonstrates the potential for low contention.

We first prove that the relaxation in removing nodes does not lead to non-linearizable concurrent executions.

Theorem 2. *I_f is linearizable.*

Proof (Sketch). It is clear that each `deq` method either does not modify the shared state or removes exactly one element from the queue. Let us denote by d a fixed `deq` instance. If d returns `EMPTY` by executing line 11, then its linearization point is immediately after the linearization point of the `deq` instance d' which successfully removes the element at slot *index*-1. Observe that d' exists and cannot start after d has completed because at least one `deq` sets the the bit corresponding to *index*-1 in the dequeue vector, and marks the slot by a successful CAS. Then, d either sees this bit set in the dequeue vector or also sets the same bit. If latter, d must have failed its own CAS by the `EMPTY` return assumption. In both cases, d' cannot start after d has completed. For returning `EMPTY` via executing line 25, the linearization point is the latest of the times when t->*next* is observed to be `NULL` and when the dequeue instance removing the last unmarked slot has committed. Note that at that instant, the queue is logically empty, because all the slots in the last segment are found to be either busy or marked. If d removes a value by executing line 18, then its linearization point is immediately after the commit point of the dequeue instance removing the element at *index*-1. If *index*=0, then the commit point is the time when the slot is marked as busy. The linearization points of the `enq` method follow from the correctness proof of the ms-queue. □

Finally, we show that our implementation unlike the queue implementations we discuss in this paper and, to the best of our knowledge, all the existing queue implementations is not bound.

Theorem 3. *I_f is free.*

Proof. Assume that `LOOKAHEAD` is greater than 1. Let $\mathtt{s} = \mathtt{enq}(1)\mathtt{enq}(2)$ and $x = \mathtt{deq}(2)$, which results in $d(\mathbf{s}, m) = k = 1$. Now consider the run which is
1) an isolated (sequential) execution of $\mathtt{enq}(1)$ by thread 1, followed by
2) an isolated (sequential) execution of $\mathtt{enq}(2)$ by thread 1, followed by
3) $(1, \mathtt{deq}[l_2 - l_{12}])$, followed by
4) $(2, \mathtt{deq}[l_2 - l_{13}])$, followed by
5) $(1, \mathtt{deq}[l_{12}, l_{14}])$, followed by
6) an isolated (sequential) execution of $\mathtt{deq}()$ by thread 3 which completes by removing 2.
Here, we use the notation $l_i - l_j$ to denote the isolated execution of all the statements from l_i to l_j, the latter being exclusive, i.e. not executed. Note that all six parts are possible to execute. In particular, the last part where `deq` of thread 3 terminates with value 2 found in the second slot of the current segment is possible because the first slot was marked busy by both of the two other dequeue instances run by threads 1 and 2.

Now, at this point, neither the method instance of thread 1 nor that of thread 2 is unifate. If we extend by having thread 1 execute in isolation, followed by thread 2, the former will return 1 and the latter will return `NULL`. If on the other hand, we swap the order of execution of the two instances, then the dequeue of thread 2 will return 1 and that of thread 1 will return `NULL`. Thus, the number of unifate method instances is given by 3, the two `enq` instances plus the $\mathtt{deq}(2)$ instance, which is strictly less than $|\mathbf{s}| + 1 + 1 = 4$. This concludes the proof that I_f is free. □

7 Experimental Evaluation

In this section, we empirically show that it is possible to improve performance of a bound implementation by turning it into a free implementation. As our reference model, we use the ms-queue and compare its performance with that of our free implementation I_f described in the previous section.

We ran our experiments on an 2.3GHz AMD Opteron Processor with 8 cores. Each experiment starts with a queue containing 160,000 elements and eight threads run concurrently, each making 20,000 calls to dequeue. Our experiments are meant to measure the speed of the dequeue instances, which is where our implementation I_f differs from the MS-queue. The variable `LOOKAHEAD` was used to measure the impact of allowing more elements to be removed by concurrent dequeue instances. Intuitively, `LOOKAHEAD` gives a bound on the number of concurrent dequeue instances that can remove elements from the queue simultaneously. In other words, it gives a bound on the length of consecutive successful CAS calls (line 14, Fig. 4). The results for varying values of `LOOKAHEAD` are reported in Table. 1.

In addition to measuring the time of execution, we also report the number of failed CAS calls for marking a node (line 14, Fig. 4), which is the main

Table 1: Experimental Results

	MS-queue	$(I_f, 0)$	$(I_f, 1)$	$(I_f, 2)$	$(I_f, 4)$	$(I_f, 8)$	$(I_f, *)$
Failed CAS	373219	208270	150492	141970	139253	138567	121141
Execution time	890000	882000	842000	844000	852000	842000	816000

source of contention for the dequeue implementation. The numbers reported are the average of five experiments per each entry. The pair (I_f, j) denotes I_f instantiated with `LOOKAHEAD` set to j. The last column, $(I_f, *)$, denotes the free implementation which does not check *skipcnt* at all.

As expected, the number of failed CAS calls decreases with increasing the value `LOOKAHEAD` and saturates around the number of cores on which the experiments ran. The execution time of the instance of I_f with `LOOKAHEAD` taken to be 0, which essentially is the same as the original MS-queue, is observed to be comparable to that of the MS-queue. Note however that the number of failed CAS attempts is much lower in $(I_f, 0)$ compared to the MS-queue. This is expected: if the computation load of methods is increased, the contention decreases (it becomes less likely that two CAS calls overlap). The computation load of $(I_f, 0)$ is more than that of the MS-queue because of the extra cache traffic for checking and updating busy slots.

As long as the mechanism for monitoring the number of skipped busy slots, by comparing *skipcnt* with `LOOKAHEAD`, is present, the execution times do not improve much, after picking a `LOOKAHEAD` strictly greater than 0. This suggests that, even though the number of failed CAS attempts decreases with increasing values of `LOOKAHEAD`, the communication overhead prevents further speed-up. Finally, when `LOOKAHEAD` is completely ignored (last column), we obtain the best performance, both in terms of number of failed CAS attempts and execution time, improvements of approximately sixty and ten percent over the performance figures of the MS-queue, respectively.

8 Conclusion

In this paper, we defined a new property for linearizable implementations. A linearizable implementation is called free if a thread t can commit without observing the abstract state at which t linearizes as long as t has the knowledge of the existence of other threads which will reach some consistent state with which what t does agrees. For instance, t can remove the third oldest element in a queue without observing the two oldest elements being removed, as long as t knows that there are at least two more concurrent dequeue instances which have not committed yet. This, as we argue, can reduce contention, which in turn has the potential to improve performance. The data structure of interest in this paper

is queue, and to the best of our knowledge, no existing queue implementation is free. We show that by turning an existing queue implementation, the Michael-Scott queue, into a free implementation, we are able to improve performance by cutting down the number of failed synchronization attempts, made via calls to Compare-And-Swap.

We believe that similar improvements can be done to any other concurrent data structure, which requires a total order over its operations in its sequential specification. Theoretically interesting is the question whether it is provable that any bound linearizable implementation can be turned into one that is free and performs better, a question which we leave as future work.

It is worth noting that free implementations will constitute a challenge to the verification community as the existing approaches rely on the property of being bound; that is, the existence of a correspondence between the temporal order of linearization points in the implementation and the logical order of updates to the abstract data structure.

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. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. pp. 323–337. FM'11, Springer-Verlag (2011)
2. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. pp. 296–311. TACAS'10, Springer-Verlag (2010)
3. Golab, W., Higham, L., Woelfel, P.: Linearizable implementations do not suffice for randomized distributed computation. In: Proceedings of the 43rd annual ACM symposium on Theory of computing. pp. 373–382. STOC '11, ACM (2011)
4. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. pp. 355–364. SPAA '10, ACM (2010)
5. Herlihy, M., Shavit, N., Waarts, O.: Linearizable counting networks. Distrib. Comput. 9(4), 193–203 (Feb 1996)
6. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
7. Hoffman, M., Shalev, O., Shavit, N.: The baskets queue. pp. 401–414. OPODIS'07, Springer-Verlag (2007)
8. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. In: DISC '04. pp. 117–131. Springer-Berlin (2004)
9. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. pp. 321–337. FM '09, Springer-Verlag (2009)
10. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. pp. 267–275. PODC '96, ACM (1996)
11. Moir, M., Anderson, J.H.: Fast, long-lived renaming (extended abstract). In: Proceedings of the 8th International Workshop on Distributed Algorithms. pp. 141–155. WDAG '94, Springer-Verlag (1994)

12. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA '05. pp. 253–262. ACM (2005)
13. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. pp. 85–94. PODC '10, ACM (2010)
14. Vafeiadis, V.: Automatically proving linearizability. pp. 450–464. CAV'10, Springer-Verlag (2010)

A The Enqueue Method

```
1: procedure enq(x)
2:     nseg ← NewSeg(x)
3:     index ← 0; tail ← Q.tail
4:     while true do
5:         if tail->seg[index].val = −1 then                ▻ Optimization
6:             if CAS(tail->seg[index].val, −1, 0) then
7:                 CAS(Q.tail, tail, tail->next)
8:                 break
9:         if index = SEGSIZE − 1 then
10:            if CAS(tail->next, NULL, nseg) then
11:                CAS(Q.tail, tail, tail->next)
12:                break
13:            tail ← Q.tail; index ← 0
14:        else
15:            index ← index + 1
16:    end while
17: end procedure
```

Fig. 6: The enqueue method of a free queue implementation.