Quantitative Relaxation of Concurrent Data Structures

Thomas A. Henzinger* Christoph M. Kirsch+ Hannes Payer+ Ali Sezgin* Ana Sokolova+

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

+University of Salzburg
firstname.lastname@cs.uni-salzburg.at

Abstract

There is a trade-off between performance and correctness in implementing concurrent data structures. Better performance may be achieved at the expense of relaxing correctness, by redefining the semantics of data structures. We address such a redefinition of data structure semantics and present a systematic and formal framework for obtaining new data structures by quantitatively relaxing existing ones. We view a data structure as a sequential specification containing all "legal" sequences over an alphabet of method calls. Relaxing the data structure corresponds to defining a distance from *any* sequence over the alphabet to the sequential specification: the k-relaxed sequential specification contains all sequences over the alphabet within distance k from the original specification. In contrast to other existing work, our relaxations are semantic (distance in terms of data structure states). As an instantiation of our framework, we present two simple yet generic relaxation schemes, called out-of-order and stuttering relaxation, along with several ways of computing distances. We show that the out-of-order relaxation, when further instantiated to stacks, queues, and priority queues, amounts to tolerating bounded out-of-order behavior, which cannot be captured by a purely syntactic relaxation (distance in terms of sequence manipulation, e.g. edit distance). We give concurrent implementations of relaxed data structures and demonstrate that bounded relaxations provide the means for trading correctness for performance in a controlled way. The relaxations are monotonic, which further highlights the trade-off: increasing k increases the number of permitted sequences, which as we demonstrate can lead to better performance. Finally, since a relaxed stack or queue also implements a pool, we obtain new concurrent pool implementations that outperform the state-of-the-art ones.

Categories and Subject Descriptors D.3.1 [*Programming languages*]: Formal definitions and theory—semantics; E.1 [*Data Structures*]: Lists, stacks, and queues; D.1.3 [*Programming languages*]: Programming techniques—concurrent programing

General Terms Theory, Algorithms, Design, Performance

Keywords (concurrent) data structures, relaxed semantics, quantitative models, costs

1. Introduction

Concurrent data structures may be a performance and scalability bottleneck and thus prevent effective use of increasingly parallel hardware [18]. There is a trade-off between scalability (performance) and correctness in implementing concurrent data structures. A remedy to the scalability problem is to relax the semantics of concurrent data structures. The semantics is given by some notion of equivalence with sequential behavior. The equivalence is determined by a consistency condition, most commonly linearizability [7], and the sequential behavior is inherited from the sequential version of the data structure (e.g., the sequential behavior of a concurrent stack is a regular stack). Therefore, relaxing the semantics of a concurrent data structure amounts to either weakening the consistency condition (linearizability being replaced with sequential consistency or quiescent consistency) or redefining (relaxing) its sequential specification. In this paper, we present a framework for relaxing sequential specifications in a quantitative manner.

For an example of a relaxation, imagine a k-stack in which each pop removes one of the most recent k elements and an operation size which returns a value that is at most k away from the correct size. It is intuitively clear that such a k-stack relaxes a regular stack, but current theory does not provide means to quantify the relaxation. Our framework does, it provides a way to formally describe and quantitatively assess such relaxations.

We view a data structure as a sequential specification S consisting of all semantically correct sequences of method calls. We identify the sequential specification with a particular labeled transition system (LTS) whose states are sets of sequences in S with indistinguishable future behavior and transitions are labeled by method calls. A sequence is in the sequential specification if and only if it is a finite trace of this LTS.

Our framework for quantitative relaxation of concurrent data structures amounts to specifying costs of transitions and paths. In the LTS, only *correct* transitions are allowed, e.g., a transition labeled by pop(a) is only possible in a state of a stack with a as top element. In a relaxation, we are exactly interested in allowing the *wrong* transitions, but they will have to incur cost. Our framework makes this possible in a controlled quantitative way.

The framework is instantiated through specifying two cost functions: A local function, *transition cost*, that assigns a penalty to each wrong transition, and a global function, *path cost*, that accumulates the local costs (using, e.g., maximum, sum, or average) to obtain the overall distance of a sequence. Via this local-global dichotomy, we are able to achieve a separation of concerns, modularity and flexibility: Different transition costs can be used with the same path cost, or vice versa, leading to different relaxations. Once the distance of a sequence from the original sequential specification S is defined in this way, a k-relaxation of the data structure becomes the set of all sequences within distance k from S.

Returning to the stack example above, we can set the transition cost of a pop transition at a state to be the number of elements that

POPL'13, January 23–25, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1832-7/13/01. . . $15.00

are between the popped element and the top of the stack. We can define the path cost to be the maximum transition cost that occurs along a sequence. Then, the corresponding k-relaxation precisely captures what we intuitively described.

We instantiate the framework on two levels. On the abstract level, we present two generic relaxations called out-of-order and stuttering relaxation, which provide a way to assign transition costs, together with several different path cost functions for *any* data structure. On the concrete level, we instantiate the out-of-order relaxation to stacks, queues, and priority queues. We spell out the effects of the relaxation in these concrete cases and prove that they indeed correspond to the intuitive idea of bounded relaxed out-of-order behavior. We also instantiate the stuttering relaxation to a Compare-And-Swap (CAS) object and a shared counter and prove correspondence results as well.

We show that the relaxation framework is indeed of practical value: we give an efficient new implementation of an out-of-order stack and fit an existing efficient implementation of an out-of-order queue [11] in our framework as well. The experimental results demonstrate ideal behavior: linear scalability and performance. In particular, the relaxed stack implementation we present outperforms and outscales state-of-the-art stack, queue, and pool algorithms on various workloads. We also present implementations for a stuttering CAS, a stuttering shared counter using this stuttering CAS and a different stuttering shared counter, all of which further demonstrate increased scalability and performance.

The main contributions of this paper are: (1) the framework for quantitative relaxation of data structures, and (2) efficient concurrent implementations. The way to the framework is paved by formally capturing the semantics of a data structure. Other contributions made possible by the framework are: the generic out-of-order and stuttering relaxations of data structures; characterizations of the out-of-order relaxation in concrete terms for stacks, queues, and priority queues; characterization of the stuttering relaxation in concrete terms for CAS and shared counters.

The structure of the paper is as follows. In the remainder of this section, we provide motivation for the main features of our work. We present the formal view on data structures in Section 2, followed by the framework for quantitative relaxation in Section 3. Throughout the formal part we use a stack as running example. We present the two generic instances, out-of-order and stuttering relaxations, in Section 4 and instantiate them further to concrete data structures in Section 5 and Section 6, respectively. We discuss related work in Section 7. In Section 8 we present implementation details and in Section 9 experimental results confirming our original scalability and performance goal. We wrap up with concluding remarks in Section 10.

In the related work survey, we put special emphasis on quasi-linearizability [2], the only other work we are aware of that also tackled the problem of quantitatively relaxing sequential data structures for better performance in the concurrent setting. As opposed to our semantic (state-based) approach in assigning distances to sequences, the relaxation of [2] is syntactic (permutation-based). We argue that (1) the semantic approach is more expressive than the syntactic one, and (2) it allows the designer of a data structure to formally capture the intent of a specific relaxation more easily and naturally.

Highlights

Relaxation improves performance. A relaxation of the sequential specification of a data structure can lead to a distribution of contention points, diminishing the need for, and thus, the cost of synchronization. For instance, instead of requiring that each pop operation updates the top pointer of a concurrent stack, allowing a relaxation which sets the size of the window from which a removal is deemed acceptable to some $k > 1$ (most recent elements) effectively reduces contention for the top pointer. In Section 9, we show that even such a simple relaxation for stacks with $k = 80$ on a 40-core (2 hyperthreads per core) server machine can lead to an eight-fold increase in performance compared to the existing state-of-the-art implementations of strict stacks.

Note that a larger sequential specification increases the potential for better performance. Since our relaxations are monotonic, increasing k increases the performance potential. However, the extent to which this potential can be utilized in practice depends on many factors among which is the choice of hardware.

Generality. Consider three different sequences belonging to three different data structures, stack, queue, and priority queue, respectively:

```
push(a)push(b)push(c)push(d)
enq(a)enq(c)enq(b)enq(d)
ins(a)ins(b)ins(d)ins(c)
```

where for the priority queue `b` has top priority, followed by `a` and `c` that have the same medium priotity, and `d` has low priority.

If these sequences are extended with a removal operation (`pop`, `deq`, `rem`, respectively), the expected return values are `d` (element at the top of the stack), `a` (element at the head of the queue), and `b` (element with the highest priority).

Imagine instead, that the removal operation returns `c` for all of these three sequences. At first sight, that `c` is returned seems to be arbitrary. However, a careful examination reveals a common pattern: In each sequence, `c` is not the current, but *next* (possible) value to be removed. That is, in the stack it is the element immediately below the top element; in the queue it is the element immediately after the head element; in the priority queue it is an element with the second highest priority. It then seems natural to view all these relaxations as an instantiation of a common relaxation scheme.

Our framework allows one to precisely express this and other common types of relaxations. For instance, our generic out-of-order relaxation provides exactly this for data structures in which information is retrieved according to some order, temporal in the case of queue and stack, logical in the case of a priority queue. The generic relaxation removes the need of relaxing each data structure separately.

Modularity. Let us now consider the situation immediately following the removal of `c` from the stack as was depicted above. We have the following sequence:

```
push(a)push(b)push(c)push(d)pop(c)
```

The stack now contains the elements `a`, `b`, and `d`, the last of which is on top. One might desire a particular relaxation where two consecutive out-of-order removals are not allowed, and hence, the next removal has to return `d`. Yet another might find it acceptable that at all times one of the top two elements are removed; it does not matter how long the top element remains on the stack. Our framework allows one to express both. Each transition incurs a transition cost. Observe that in both relaxations the same cost (out-of-order removal cost) is assigned to each transition. Each sequence of transition costs incurs a path cost, and this is what distinguishes the two relaxations. The first will require that there are no two consecutive transitions with non-zero cost; the second will require that the maximum of any transition cost is at most 1. We thus obtain a modular framework in which existing relaxations can be tailored by modifying transition costs, path costs, or both.

Measurability. The code given in Figure 1(a) represents a CAS-based strict shared counter. The shared variable `c` is a counter, and each thread tries to increment the value of the counter. Representative of many concurrent implementations, this code leads to poor scalability as all threads trying to increment the counter will compete for access to `c`.

```
while(true):
 x = c;
 if (CAS(&c,x,x+1)):
  return x+1;
```

(a) Single counter.

```
while(true):
 x,v = getMaxAndValueAt(c, f(t));
 if (CAS(&c[f(t)],v,x+1)):
  return x+1;
```

(b) Distributed counter.

Figure 1. Shared counters

Next, consider a modified version of this shared counter, given in Figure 1(b). Unlike the strict implementation, here we use an array `c` of k counters and the logical value of the shared counter is taken to be the maximum value among all the counters in `c`. Each thread `t` can write only to the slot with index `f(t)`. Each attempt of `t` incrementing the counter starts by reading the value contained in `f(t)` (stored in `v`) and the maximum value of all the counters in `c` (stored in `x`). Then, it tries to update its counter to $x+1$ provided that `f(t)` is not updated by a concurrent thread.

The behavior of this code depends crucially on the value chosen for the size k of the array. For instance, if $k = 1$, then this implementation will be behaviorally equivalent to the single counter code. For other values of k, it is evident that there will be a discrepancy between the behaviors of the two codes.

We go beyond this qualitative notion (existence vs. absence of relaxation) and provide a measure for any relaxation defined in our framework. The distributed counter given in Figure 1(b) is in fact a k-stuttering relaxation of the shared counter of Figure 1(a). This way an application developer using a quantitatively relaxed data structure can evaluate the gain in performance for k-relaxation vs. the effort of modifying an application that uses it, and try to optimize k. For instance, if the relaxed shared counter is used as a performance counter counting the occurrence of a given event, e.g. context switches in a multi-processor scheduler, not all occurrences of events will be registered. Knowing that the number of unregistered event occurrences within one counter increment can not exceed k (the size of `c`) is a crucial information for the application designer.

Transparency. Now consider the code given in Figure 2. This code is very similar to the strict shared counter code of Figure 1(a) except for the call to the method `kCAS` instead of `CAS`. The `kCAS` is a relaxed version of `CAS` such that up to at most k many concurrent threads trying to update the value of the `CAS` object can complete with false positive (see Section 6 and Section 8 for details).

```
while(true):
 x = c;
 if (kCAS(&c,x,x+1)):
  return x+1;
```

Figure 2. `kCAS` counter.

Although the `kCAS` and the distributed counter of Figure 1(b) take fundamentally different approaches in relaxing the strict semantics of a counter, they both implement a k-stuttering relaxation. This illustrates another use of our framework: It can be used as a simpler way to establish abstract equivalence, thus providing transparency for a higher-level application. If one shows that two implementations implement the same relaxation, then a client application using either implementation will observe the same behavior, regardless of the differences in actual implementation details.

2. Data structures, specifications, states

Let Σ be a set of methods including input and output values. We will refer to Σ as the sequential alphabet. A *sequential history* $\mathbf{s}$ is an element of Σ^*, i.e., a sequence over Σ. As usual, by ε we denote the empty sequence in Σ^*. A *data structure* is a *sequential specification* S which is a prefix-closed set of sequential histories, $S \subseteq \Sigma^*$.

EXAMPLE 2.1. The set of methods of a stack, with data in a set D, is

$$\Sigma_\mathbb{S} = \{\mathtt{push(d)} \mid \mathtt{d} \in D\} \cup \{\mathtt{pop(d)} \mid \mathtt{d} \in D \cup \{\mathtt{null}\}\}.$$

The sequential specification $S_\mathbb{S}$ consists of all stack-valid sequences, i.e., sequences in which each `pop` pops the top of the stack and each `push` pushes an element at the top. For instance, the sequence $\mathbf{s}_\mathbb{S} = \mathtt{push(a)pop(a)push(b)}$ is in the sequential specification $S_\mathbb{S}$, whereas the sequence $\mathbf{t}_\mathbb{S} = \mathtt{push(a)push(b)pop(a)}$ is not.

The following definition is the core of our way of capturing semantics. Let S be a sequential specification.

DEFINITION 2.2. *Two sequential histories $\mathbf{s}, \mathbf{t} \in S$ are S-*equivalent*, written $\mathbf{s} \equiv_S \mathbf{t}$, if for any sequence $\mathbf{u} \in \Sigma^*$, $\mathbf{su} \in S$ if and only if $\mathbf{tu} \in S$.*

It is clear that $\equiv_S$ is an equivalence relation. By $[\mathbf{s}]_S$ we denote the S-equivalence class of $\mathbf{s}$. Intuitively, two sequences in the sequential specification are S-equivalent if they lead to the same "state". The following simple property follows directly from the definition of S-equivalence.

LEMMA 2.3. *If $\mathbf{s} \equiv_S \mathbf{t}$ and $\mathbf{su} \in S$, then $\mathbf{su} \equiv_S \mathbf{tu}$.*

The intuition about states is made explicit in the next definition. In addition, we point out particular "minimal" representatives of a state.

DEFINITION 2.4. *A state of a data structure with sequential specification S is an equivalence class $[\mathbf{s}]_S$ with respect to $\equiv_S$. For a state $q = [\mathbf{s}]_S$, the kernel of q is the set*

$$\ker(q) = \{\mathbf{t} \in [\mathbf{s}]_S \mid \mathbf{t} \textit{ has minimal length }\}.$$

A sequence $\mathbf{s} \in S$ is a kernel sequence if $\mathbf{s} \in \ker([\mathbf{s}]_S)$.

EXAMPLE 2.5. One can easily show that kernel sequences of a stack are all sequences in $\{\mathtt{push(d)} \mid d \in D\}^*$. Moreover, for any state $q = [\mathbf{s}]_{S_\mathbb{S}}$ of a stack, there is a unique sequence in $\ker(q)$, i.e., $|\ker(q)| = 1$. This implies that different sequences in $\mathbf{s} \in \{\mathtt{push(d)} \mid \mathtt{d} \in D\}^*$ represent different states.

Having identified states, a data structure corresponds to a labeled transition system (LTS) that we define next.

DEFINITION 2.6. *Let S be a (sequential specification of a) data structure. Its corresponding LTS is $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$ with*

- *set of states $Q = S/\equiv_S = \{[\mathbf{s}]_S \mid \mathbf{s} \in S\}$,*
- *set of labels Σ,*
- *transition relation $\rightarrow \subseteq Q \times \Sigma \times Q$ given by*

$$[\mathbf{s}]_S \xrightarrow{m} [\mathbf{s}m]_S \textit{ if and only if } \mathbf{s}m \in S, \textit{ and}$$

- *initial state $q_0 = [\varepsilon]_S$.*

Note that the transition relation is well defined (independent of the choice of a representative) due to Lemma 2.3. Also q_0 is well defined since S is prefix closed. We write $q \xrightarrow{m}$ if there is an m-labeled transition from q to some state; $q \overset{m}{\nrightarrow}$ if there is no m-labeled transition from q. We also write $q \xrightarrow{\mathbf{u}}$ if there is a $\mathbf{u}$-labeled path of transitions starting from q, and $q \overset{\mathbf{u}}{\nrightarrow}$ if it is not the case that $q \xrightarrow{\mathbf{u}}$. The following immediate observation provides the exact correspondence between the sequential specification of a data structure and its LTS: S is the set of finite traces of the initial state of $LTS(S)$.

LEMMA 2.7. *Let S be a sequential specification with $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$. Then for any $\mathbf{u} \in \Sigma^*$ we have $\mathbf{u} \in S$ if and only if $q_0 \xrightarrow{\mathbf{u}}$.*

EXAMPLE 2.8. Since different stack-kernel sequences represent different states, cf. Example 2.5, the transitions of $LTS(S_\mathbb{S})$ are fully described by

$$\begin{aligned}
&[\mathbf{s}]_{S_\mathbb{S}} \xrightarrow{\texttt{push(a)}} [\mathbf{s}\cdot\texttt{push(a)}]_{S_\mathbb{S}}\\
&[\mathbf{s}]_{S_\mathbb{S}} \xrightarrow{\texttt{pop(a)}} [\mathbf{s}']_{S_\mathbb{S}} \quad \text{if } \mathbf{s} = \mathbf{s}'\cdot\texttt{push(a)}, \text{ and}\\
&[\mathbf{s}]_{S_\mathbb{S}} \xrightarrow{\texttt{pop(null)}} [\varepsilon]_{S_\mathbb{S}} \text{ if } \mathbf{s} = \varepsilon
\end{aligned}$$

where $\mathbf{s}$ is a kernel sequence in $\{\texttt{push(d)} \mid d \in D\}^*$. Note that if $\mathbf{s} = \mathbf{s}'\cdot\texttt{push(a)}$, then $[\mathbf{s}\cdot\texttt{pop(a)}]_{S_\mathbb{S}} = [\mathbf{s}']_{S_\mathbb{S}}$.

3. Framework for quantitative relaxations

We are now ready to present the framework for quantitatively relaxing data structures. Let $S \subseteq \Sigma^*$ be a data structure with $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$. Our goal is to relax S to a so-called k-relaxed specification $S_k \subseteq \Sigma^*$ in a bounded way, with k providing the bound.

Giving a relaxation for a data structure S amounts to the following three steps:

1. **Completion.** From $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$ we construct the completed labeled transition system
$$\mathrm{LTS}_c(S) = (Q, \Sigma, Q \times \Sigma \times Q, q_0)$$
with transitions from any state to any other state by any method.
2. **Transition costs.** From $\mathrm{LTS}_c(S)$ a quantitative labeled transition system $\mathrm{QLTS}(S) = (Q, \Sigma, Q \times \Sigma \times Q, q_0, C, \mathrm{cost})$ is constructed. Here C is a well-ordered cost domain, hence it has a minimum that we denote by 0, and $\mathrm{cost}\colon Q \times \Sigma \times Q \rightarrow C$ is the transition cost function satisfying
$$\mathrm{cost}(q, m, q') = 0 \quad \text{if and only if} \quad q \xrightarrow{m} q' \text{ in } \mathrm{LTS}(S).$$
We write $q \xrightarrow{m,k} q'$ for the quantitative transition with $\mathrm{cost}(q, m, q') = k$. A quantitative path of $\mathrm{QLTS}(S)$ is a sequence
$$\kappa = q_1 \xrightarrow{m_1,k_1} q_2 \xrightarrow{m_2,k_2} q_3 \ldots q_n \xrightarrow{m_n,k_n} q_{n+1}.$$
The sequence $\tau = (m_1, k_1)(m_2, k_2)\ldots(m_n, k_n) \in (\Sigma \times C)^*$ is the quantitative trace of κ, notation $\mathrm{qtr}(\kappa)$, and the sequence $\mathbf{u} = m_1 \ldots m_n$ is the trace of the quantitative path κ and of the quantitative trace $\mathrm{qtr}(\kappa)$, notation $\mathrm{tr}(\kappa) = \mathrm{tr}(\mathrm{qtr}(\kappa)) = \mathbf{u}$. By $\mathrm{qtr}(\mathbf{u})$ we denote the set of all quantitative traces of quantitative paths starting in the initial state with trace $\mathbf{u}$ and by $\mathrm{qtr}(S)$ the set of all quantitative traces of quantitative paths starting in the initial state.
3. **Path cost function.** We choose a monotone path cost function $\mathrm{pcost}\colon\ \mathrm{qtr}(S) \rightarrow C$. Monotonicity here is with respect to prefix order: if a quantitative trace τ is a prefix of a quantitative trace τ', then $\mathrm{pcost}(\tau) \leqslant \mathrm{pcost}(\tau')$.

Having performed these three steps, we can define the k-relaxed specification.

DEFINITION 3.1. *The k-relaxed specification S_k for $k \in C$ contains all sequences that have a distance at most k from S,*
$$S_k = \{\mathbf{u} \in \Sigma^* \mid d_S(\mathbf{u}) \leqslant k\}$$
where $d_S(\mathbf{u})$ is the distance of $\mathbf{u}$ *to the sequential specification S given by*
$$d_S(\mathbf{u}) = \min\{\mathrm{pcost}(\tau) | \tau \in \mathrm{qtr}(\mathbf{u})\}.$$

REMARK 3.2. Both the distance d_S and the relaxed specification S_k are actually parametric in the transition cost function as well as in the path cost function. For simplicity, we prefer a light, overloaded notation that does not explicitly mention these parameters. Also, for some applications one may wish for two different cost domains, one for the transition, one for the path cost, of which only the second one needs to be well ordered. Again for simplicity, we restrict the presentation to a single cost domain.

Some obvious properties of the quantified relaxations resulting from our framework are:

- $S_0 = S$, ensured by the condition on the transition cost function.
- Every relaxation S_k is prefix closed, ensured by the monotonicity of the path cost function.
- The relaxations are monotone, i.e., if $k \leqslant m$, then $S_k \subseteq S_m$.

To conclude, in order to relax a data structure all that one needs is a cost domain C, a transition cost for each transition in the completed LTS (item 2. above), and a path cost function (item 3. above).

REMARK 3.3. The current framework does not allow for relaxations that leave the original state space of $LTS(S)$. An example of such is a prophetic relaxation of e.g. stack, where sequences with $\texttt{pop(a)}$ preceding $\texttt{push(a)}$ need to be assigned finite distance. This can be done by slightly changing the definition of $LTS_c(S)$: Instead of keeping the original states $S/\equiv_S$, one can take as set of states the quotient $\Sigma^*/\sim$ where $\sim$ is an equivalence that coincides with $\equiv_S$ when restricted to S. For simplicity and since we do not use such relaxations in this paper, our current definition of $LTS_c(S)$ keeps the states unchanged.

4. Generic relaxations

In this section we illustrate the relaxation framework on two generic examples. The value and generality of these particular examples become evident in Section 5 and Section 6 when we instantiate them to concrete data structures. Let $S \subseteq \Sigma^*$ be a data structure with $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$. We first fix the cost domain to $C = \mathbb{N} \cup \{\infty\}$.

4.1 Out-of-order relaxation

For the out-of-order generic relaxation we define a transition cost function $scost\colon Q \times \Sigma \times Q \rightarrow C$, called *segment cost*, and mention two other related transition cost functions.

DEFINITION 4.1. *Let $t = (q, m, q')$ be a transition in* $\mathrm{LTS}_c(S)$. *Let* $\mathbf{v}$ *be a sequence with minimal length satisfying one of the following two conditions:*

(1) There exist sequences $\mathbf{u}$, $\mathbf{w}$ *such that* $\mathbf{uvw} \in \ker(q)$ *and* $\mathbf{uw}$ *is a kernel sequence and either*
  - *(i)* $[\mathbf{uw}]_S \xrightarrow{m} [\mathbf{u'w}]_S$ *and* $q' = [\mathbf{u'vw}]_S$*, or*
  - *(ii)* $[\mathbf{uw}]_S \xrightarrow{m} [\mathbf{uw'}]_S$ *and* $q' = [\mathbf{uvw'}]_S$.

(2) There exist sequences $\mathbf{u}$, $\mathbf{w}$ *such that* $\mathbf{uw} \in \ker(q)$ *and* $\mathbf{uvw}$ *is a kernel sequence and either*
  - *(i)* $[\mathbf{uvw}]_S \xrightarrow{m} [\mathbf{u'vw}]_S$ *and* $q' = [\mathbf{u'w}]_S$*, or*
  - *(ii)* $[\mathbf{uvw}]_S \xrightarrow{m} [\mathbf{uvw'}]_S$ *and* $q' = [\mathbf{uw'}]_S$.

Then the segment cost is given by the length of $\mathbf{v}$, *$scost(t) = |\mathbf{v}|$. If such a sequence* $\mathbf{v}$ *does not exist for t, then $scost(t) = \infty$.*

Intuitively, segment cost of a relaxed transition is the length of the shortest subword ($\mathbf{v}$) whose removal (1) or insertion (2) into the kernel sequence enables a transition. Observe that the transition can be taken in $\mathrm{LTS}(S)$ if and only if its segment cost is 0, obtained by setting $\mathbf{v} = \varepsilon$. We will see in the next section that this cost quantifies *out-of-order* updates or observations, such as returning an element other than the top element in a stack or removing an element other than the head of a queue. We note that segment cost just as any transition cost can also be used per method, i.e., some methods may be relaxed, some not.

4.2 Stuttering relaxation

For the stuttering generic relaxation, we define the so-called *stuttering cost*.

DEFINITION 4.2. *Let $t = (q,m,q')$ be a transition in* $\mathrm{LTS}_c(S)$. *Then, the* stuttering cost, *stcost is defined as*

$$stcost(q,m,q') = \begin{cases} 0 & \textit{if } q \xrightarrow{m} q' \\ 1 & \textit{if } q = q' \wedge q \overset{m}{\not\rightarrow} q' \wedge q \xrightarrow{m} \\ \infty & \textit{otherwise} \end{cases}$$

where $\rightarrow$ is the transition relation of $LTS(S)$.

Intuitively, the stuttering relaxation allows for (already enabled) transitions to have no effect on the state. If, in the specification S, q goes to $\hat{q}$ with method m and $q \neq \hat{q}$, then the stuttering cost of applying m at q *and* staying at q after the transition is 1. All other transitions which are not part of the original specification are set to have infinite cost.

An example of an unbounded (except in the maximal size of the queue) stuttering relaxation is presented in [15], where workers are allowed to work on the same task by a relaxed queue semantics and an element in the queue can be dequeued a number of times (up to the maximal size of the queue). In favor of bounded stuttering we note that, typically, implementations can benefit from retiring (completing rather than retrying) mutator method calls when there is too much contention and have the client handle false positives.

4.3 Path cost functions

Let $S \subseteq \Sigma^*$ be a data structure and $\tau = (m_1,k_1)(m_2,k_2)\dots(m_n,k_n)$ a quantitative trace in $\mathrm{qtr}(S)$. We define the following generic path cost functions (to be used with any transition cost):

- The *maximal cost*, $\mathrm{pcost}_{max}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, maps τ to the maximal transition cost along it. Formally,

  $$\mathrm{pcost}_{max}(\tau) = \max\{k_i \mid 1 \leqslant i \leqslant n\}.$$

- The φ-*interval cost*, $\mathrm{pcost}_{[\varphi]}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, for φ a binary predicate (first order formula with two free variables making statements about positions in the quantitative trace), maps τ to the length of a maximal consecutive quantitative subtrace that satisfies φ. Hence, we have

  $$\mathrm{pcost}_{[\varphi]}(\tau) = \max\{j - i + 1 \mid \varphi(i,j) \text{ and } 1 \leqslant i \leqslant j \leqslant n\}.$$

- The φ-*interval restricted maximal cost*, $\mathrm{pcost}_{max|[\varphi]}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, for φ as in the φ-interval cost, is given by

  $$\mathrm{pcost}_{max|[\varphi]}(\tau) = \max\{l_{i,j} \mid \varphi(i,j) \text{ and } 1 \leqslant i \leqslant j \leqslant n\},$$

  where

  $$l_{i,j} = \max\{k_r + (r - i + 1) \mid i \leqslant r \leqslant j\}.$$

We instantiate the out-of-order relaxation along with the maximal, φ-interval, and φ-interval restricted maximal cost on stacks, queues, and priority queues in Section 5. The φ-interval restricted maximal cost is more complex and less intuitive than the other path cost functions, but when instantiated it provides valuable relaxation examples that are efficiently implementable. In Section 6 we apply the stuttering relaxation along with the φ-interval cost on a CAS object and on a shared counter. Note that the other two cost functions do not make much sense together with the stuttering cost (the maximal cost is two-valued and the φ-interval restricted maximal cost amounts to the φ-interval cost plus one).

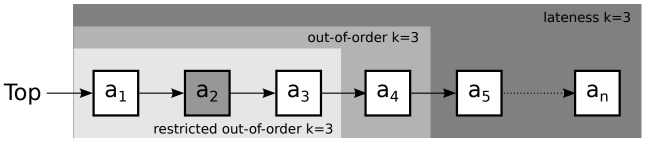


Figure 3. The ranges of elements which may be returned by a pop operation of a k-stack with restricted out-of-order, out-of-order, and lateness relaxation with $k = 3$. The element a_2 is already removed.

5. Out-of-order stacks, queues, and priority queues

In this section we apply the relaxation of Section 4.1 to stacks, FIFO queues, and priority queues. Due to lack of space, here we leave out some common methods, e.g., `top` (for stack), `head` (for queue), `size` (for all). Inclusion of these methods does not change the results, in particular Propositions 5.1-5.3, presented in this section.

Stack. We have already given the set of methods of a stack, its states, and its LTS in Example 2.1, Example 2.5, and Example 2.8. Let us recall that the sequential specification $S_\mathbb{S}$ consists of all stack-valid sequences, i.e., sequences in which each `pop` pops the top of the stack, and each `push` pushes an element at the top.
Let $\mathbf{s}$ be a kernel sequence. A kernel sequence $\mathbf{s}'$ is

- `push(a)`-out-of-order-k from $\mathbf{s}$ if $\mathbf{s}' = \mathbf{u} \cdot \texttt{push(a)} \cdot \mathbf{v}$ where $\mathbf{s} = \mathbf{uv}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- `pop(a)`-out-of-order-k from $\mathbf{s}$ if $\mathbf{s}' = \mathbf{uv}$ where $\mathbf{s} = \mathbf{u} \cdot \texttt{push(a)} \cdot \mathbf{v}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- `pop(null)`-out-of-order-k from $\mathbf{s}$ if $\mathbf{s} = \mathbf{s}'$ and $|\mathbf{s}| = k$;

By inspecting all cases, we can show the following proposition.

PROPOSITION 5.1. *Let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be two kernel sequences of a stack. Then* $[\mathbf{s}]_{S_\mathbb{S}} \xrightarrow{m,k} [\mathbf{s}']_{S_\mathbb{S}}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s}'$ *is m-out-of-order-k from* $\mathbf{s}$.

As mentioned in Section 4.1, the relaxations can be applied method-wise. We implemented k-relaxed stacks with only `push` and `pop` methods, of which only `pop` is relaxed according to the segment cost.The interpretation of the path cost functions from Section 4.3 and the corresponding relaxations are as follows:

- The maximal cost represents the maximal distance from the top of a popped element, leading to an *out-of-order k-stack*. Hence, in an out-of-order k-stack, each `pop` pops an element that is at most k away from the top.

Let $\varphi(i,j)$ be the following first order formula with free variables i and j:

$$\forall r \in [i,j].\, k_r \neq 0$$

- The φ-interval cost represents lateness, i.e., the maximal number of consecutive pops needed to pop the top, leading to a *lateness k-stack*. Hence, in a lateness k-stack at most the k-th consecutive pop pops the top.
- The φ-interval restricted maximal cost represents the maximal size of a "shrinking window" starting from the top from which elements can be popped, leading to a *restricted out-of-order k-stack*. In a restricted out-of-order k-stack, each pop removes an element at most $k - l$ away from the top, where l is the current lateness of the top.

Figure 3 presents a snapshot of a relaxed stack in each of the three out-of-order relaxations. It shows a state of a stack in which the element a_2, marked in grey, has been removed after the last removal of the top or the last push had happened. The ranges show which elements may be returned by a pop operation applied to this state in each out-of-order relaxed version for $k = 3$.

FIFO queue. We now briefly describe the out-of-order relaxation of a queue.The set of methods for a FIFO queue, with data set D, is

$$\Sigma_{\mathrm{Q}} = \{\mathtt{enq(d)} \mid \mathtt{d} \in D\} \cup \{\mathtt{deq(d)} \mid \mathtt{d} \in D \cup \{\mathtt{null}\}\}.$$

The sequential specification S_{Q} consists of all queue-valid sequences, i.e., sequences in which each deq deques the head of the queue and each enq enqueues at the tail of the queue. For instance, the following sequence $\mathbf{s}_{\mathrm{Q}} = \mathtt{enq(a)enq(b)deq(a)}$ is in the sequential specification S_{Q}, whereas the sequence $\mathbf{t}_{\mathrm{Q}} = \mathtt{enq(a)enq(b)deq(b)}$ is not.

One can easily show that kernel sequences of a FIFO queue are all sequences in $\{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$. Moreover, also here, for any state $q = [\mathbf{s}]_{S_{\mathrm{Q}}}$ of the FIFO queue, there is a unique sequence in $\ker(q)$, i.e., $|\ker(q)| = 1$. Hence different sequences in $\mathbf{s} \in \{\mathtt{enq(d)} \mid \mathtt{d} \in D\}^*$ represent different states. As a consequence, the transition relation of $\mathrm{LTS}(S_{\mathrm{Q}})$ can be described in a concise way. Let $\mathbf{s}$ be a kernel sequence of a queue. We have,

$$\begin{aligned}
&[\mathbf{s}]_{S_{\mathrm{Q}}} \overset{\mathtt{enq(a)}}{\to} [\mathbf{s} \cdot \mathtt{enq(a)}]_{S_{\mathrm{Q}}},\\
&[\mathbf{s}]_{S_{\mathrm{Q}}} \overset{\mathtt{deq(a)}}{\to} [\mathbf{s}']_{S_{\mathrm{Q}}} \quad \text{if } \mathbf{s} = \mathtt{enq(a)} \cdot \mathbf{s}', \text{ and}\\
&[\mathbf{s}]_{S_{\mathrm{Q}}} \overset{\mathtt{deq(null)}}{\to} [\varepsilon]_{S_{\mathrm{Q}}} \text{ if } \mathbf{s} = \varepsilon.
\end{aligned}$$

In a similar way as for stack, we can define when a queue kernel sequence is m-out-of-order-k from another kernel sequence, for m being a queue method. Furthermore, the analogue of Proposition 5.1 (obtained by replacing "stack" by "FIFO queue") holds for queues as well which we state below.

PROPOSITION 5.2. *Let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be two kernel sequences of a queue. Then* $[\mathbf{s}]_{S_{\mathrm{Q}}} \overset{m,k}{\to} [\mathbf{s}']_{S_{\mathrm{Q}}}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s}'$ *is m-out-of-order-k from* $\mathbf{s}$.

The maximal path cost function leads to analogous out-of-order k-queue. For lateness and restricted out-of-order k-queues we need to employ slightly different path cost functions.

Priority queue. The data set of a priority queue needs to be well-ordered, since data items carry priority as well. We take the data set to be $\mathbb{N}$. The smaller the number, the higher the priority. The set of methods is

$$\Sigma_{\mathcal{P}} = \{\mathtt{ins(n)} \mid \mathtt{n} \in \mathbb{N}\} \cup \{\mathtt{rem(n)} \mid \mathtt{n} \in \mathbb{N} \cup \{\mathtt{null}\}\}.$$

The sequential specification $S_{\mathcal{P}}$ consists of all priority-queue-valid sequences, i.e., sequences in which each rem removes an element with highest available priority.

Kernel sequences of a priority queue are all sequences in $\{\mathtt{ins(n)} \mid \mathtt{n} \in \mathbb{N}\}^*$. Unlike for stack and queue, there may be more than one sequence representing a state of a priority queue. For a state q, if $\mathbf{s} \in \ker(q)$, then also any permutation of $\mathbf{s}$ is in $\ker(q)$. Nevertheless, the order provides a canonical representative of a state: the unique kernel sequence ordered in non-increasing priority[1]. Let $\mathbf{s}$ be a canonical kernel sequence. The transitions of $LTS(S_{\mathcal{P}})$ are fully described by

$$\begin{aligned}
&[\mathbf{s}]_{S_{\mathcal{P}}} \overset{\mathtt{ins(n)}}{\to} [\mathbf{s} \cdot \mathtt{ins(n)}]_{S_{\mathcal{P}}},\\
&[\mathbf{s}]_{S_{\mathcal{P}}} \overset{\mathtt{rem(n)}}{\to} [\mathbf{s}']_{S_{\mathcal{P}}} \quad \text{if } \mathbf{s} = \mathtt{ins(n)} \cdot \mathbf{s}', \text{ and}\\
&[\mathbf{s}]_{S_{\mathcal{P}}} \overset{\mathtt{rem(null)}}{\to} [\varepsilon]_{S_{\mathcal{P}}} \text{ if } \mathbf{s} = \varepsilon.
\end{aligned}$$

Again, we define when a canonical kernel sequence is m-out-of-order-k from another canonical kernel sequence, where m is a priority queue method. We have the following result.

PROPOSITION 5.3. *Let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be two kernel sequences of a priority queue. Then* $[\mathbf{s}]_{S_{\mathcal{P}}} \overset{m,k}{\to} [\mathbf{s}']_{S_{\mathcal{P}}}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s}'$ *is m-out-of-order-k from* $\mathbf{s}$.

Analogous relaxations are again possible, only the path cost functions are more complex since they need to capture when an element with higher priority than all existing elements in the priority queue is inserted.

6. Stuttering relaxed CAS and shared counter

In this section, we instantiate the relaxation from Section 4.2 to two concrete examples.

CAS. The set of methods for a Compare-And-Swap (CAS) object with a data set D and an initial data value $\mathtt{init} \in D$ can, for our purposes, be modeled as

$$\Sigma_{CAS} = \{\mathtt{cas(d,d',b)} \mid \mathtt{d},\mathtt{d}' \in D, \mathtt{b} \in \{\mathtt{T},\mathtt{F}\}\}.$$

The sequential specification S_{CAS} is defined inductively as follows: The empty sequence ε is in S_{CAS} and any sequence of length one and shape $\mathtt{cas(init,d,T)}$ is in S_{CAS} for $\mathtt{d} \in D$. If $\mathbf{s} \in S_{CAS}$, let $\mathbf{t}$ be the maximal prefix of $\mathbf{s}$ such that $\mathbf{s} = \mathbf{t} \cdot m \cdot \mathbf{u}$ for $m = \mathtt{cas(d,d',T)}$. Then $\mathbf{s} \cdot m' \in S_{CAS}$ if either

(1) $m' = \mathtt{cas(d',d'',T)}$, or

(2) $m' = \mathtt{cas(d'',d''',F)}$ and $\mathtt{d}'' \neq \mathtt{d}'$.

Let $\mathbf{s} \in S_{CAS}$ and let as before $\mathbf{t}$ be the maximal prefix of $\mathbf{s}$ such that $\mathbf{s} = \mathbf{t} \cdot m \cdot \mathbf{u}$ for $m = \mathtt{cas(d,d',T)}$. Then, it is not difficult to show that, $s \equiv \mathtt{cas(init,d',T)}$. Hence, there is a unique kernel sequence in each equivalence class and it has length one. The transitions of $LTS(S_{CAS})$ are given by

$$[\varepsilon]_{S_{CAS}} \overset{\mathtt{cas(init,d,T)}}{\to} [\mathtt{cas(init,d,T)}]_{S_{CAS}},$$

$$[\mathtt{cas(init,d,T)}]_{S_{CAS}} \overset{\mathtt{cas(d,d',T)}}{\to} [\mathtt{cas(init,d',T)}]_{S_{CAS}}, \text{ and}$$

$$[\mathtt{cas(init,d,T)}]_{S_{CAS}} \overset{\mathtt{cas(d',d'',F)}}{\to} [\mathtt{cas(init,d,T)}]_{S_{CAS}} \text{ if } \mathtt{d} \neq \mathtt{d}'.$$

Intuitively, the state of the CAS object is given by one data value d, initially set to init. In such a state, a transition by method $\mathtt{cas(d,d',T)}$ is enabled since the comparison of the first argument and the current value succeeds (returns T, true) leading to the new state value $\mathtt{d}'$, that is, a successful comparison results in a swapped value. A transition by method $\mathtt{cas(d',d'',F)}$ in which the comparison fails (returns F, false) is enabled if indeed $\mathtt{d} \neq \mathtt{d}'$ after which no swap happens and the state value remains d. The state with data value d is formally represented by the equivalence class of a sequence with a single method $\mathtt{cas(init,d,T)}$.

Now let us formalize the notion of allowing invisible failures for CAS object updates. For this purpose we define another object called failCAS over the same set of methods, with a somewhat different set of legal sequences. We call a sequence

$$\mathtt{cas(d,d',T)} \cdot \mathbf{y} \cdot \mathtt{cas(d,d'',T)}$$

over Σ_{CAS} a *false positive sequence* if $\mathtt{cas(d,d',T)} \cdot \mathbf{y} \in S_{CAS}$, $\mathbf{y}$ is a sequence of symbols of the form $\mathtt{cas(d''',-,F)}$ with $\mathtt{d}''' \neq \mathtt{d}$, and $\mathtt{d} \neq \mathtt{d}'$. Then, $\mathbf{s} = s_0 \dots s_n \in S_{failCAS}$ if there exists a set of

[1] The canonical representative is a matter of choice. Equally justified is using the unique kernel sequence ordered in non-decreasing priority, in which case the transitions of a priority queue resemble more the transitions of a stack, highlighting the duality between FIFO queues and stacks.

positions $0 = i_0 < i_1 < \ldots < i_r = n$ for $\mathbf{s}$ such that each sequence $s_{i_{j-1}} \ldots s_{i_j}$, for $1 \leqslant j \leqslant r$, is either a false positive sequence or is in S_{CAS}. Let the *failure count* of $\mathbf{x} \in S_{failCAS}$ be the maximum number of consecutive false positive sequences $\mathbf{x}$ contains.

The corresponding relaxation in our framework is obtained by using stuttering cost on the CAS object and φ-interval cost for the predicate $\varphi(i,j)$ given by

$$\forall r \in [i,j].(k_r \neq 0 \vee \exists \mathtt{d},\mathtt{d}' \in D.(m_r = \mathtt{cas}(\mathtt{d},\mathtt{d}',\mathtt{F}))),$$

leads to a k-CAS in which up to k methods may stutter at the same state (fail to perform a swap even though the data values match). It is then easy to show the following correspondence result.

PROPOSITION 6.1. *A sequence $\mathbf{x} \in S_{failCAS}$ has failure count k if and only if $\mathbf{x}$ is in the specification of k-CAS.*

Shared counter. The set of methods of a shared counter is

$$\Sigma_{SC} = \{\mathtt{get\&Inc(n)} \mid \mathtt{n} \in \mathbb{N}\}.$$

The sequential specification S_{SC} of a shared counter contains the empty sequence ε and a sequence $\mathbf{s}$ of length $n > 0$ is in S_{SC} if and only if $\mathbf{s}(\mathtt{i}) = \mathtt{get\&Inc(i)}$, for all $1 \leqslant \mathtt{i} \leqslant n$.

One can easily show that each state of a shared counter is a singleton, i.e., for $\mathbf{s},\mathbf{t} \in S_{SC}$ we have $\mathbf{s} \equiv \mathbf{t}$ if and only if $\mathbf{s} = \mathbf{t}$. The unique sequence representing a state is automatically a kernel sequence. The transitions of $LTS(S_{SC})$ are obviously given by

$$[\varepsilon]_{S_{SC}} \overset{\mathtt{get\&Inc(1)}}{\rightarrow} [\mathtt{get\&Inc(1)}]_{S_{SC}}, \text{ and}$$

$$[\mathbf{s}]_{S_{SC}} \overset{\mathtt{get\&Inc(n+1)}}{\rightarrow} [\mathbf{s} \cdot \mathtt{get\&Inc(n+1)}]_{S_{SC}},$$

if $\mathbf{s}(\mathtt{i}) = \mathtt{get\&Inc(i)}$, for all $1 \leqslant \mathtt{i} \leqslant n$.

We define a failing shared counter analogously to a failing CAS object. A sequence $\mathbf{s}$ is a behavior of failSC if either $\mathbf{s} \in \{\varepsilon, \mathtt{get\&Inc(1)}\}$, or $\mathbf{t} = \mathbf{u} \cdot \mathtt{get\&Inc(n)}$ is a behavior of failSC and $\mathbf{s} = \mathbf{t} \cdot \mathtt{get\&Inc(n}+a\mathtt{)}$, for $a \in \{0,1\}$. The *failure count* of $\mathbf{s} \in S_{failSC}$ is one less than the length of a maximal subsequence of identical symbols in $\mathbf{s}$.

The corresponding relaxation of the shared counter is obtained by the stuttering cost and the φ-interval cost, for $\varphi(i,j) = \forall r \in [i,j].k_r \neq 0$. Thus, we get the k-stuttering shared counter, k-SC, in which a method can stutter (fail to produce a new, incremented by one, value) at most k times. For instance, the sequence

$$\mathtt{get\&Inc(1)get\&Inc(2)get\&Inc(2)get\&Inc(2)get\&Inc(3)}$$

is in the sequential specification of the 2-stuttering shared counter, 2-SC. We again have the following correspondence result.

PROPOSITION 6.2. *A sequence $\mathbf{x} \in S_{failSC}$ has failure count k if and only if $\mathbf{x}$ is in the specification of k-SC.*

7. Related work

The general topic of this paper is part of a recent trend towards scalable but semantically weaker concurrent data structures [18]. We first discuss work related to our framework and then focus on work related to our implementations.

Framework. The relaxation framework generalizes previous work on so-called semantical deviation and k-FIFO queues [10, 12] which correspond to restricted out-of-order k-FIFO queues here.

Our work is closely related to relaxing the semantics of concurrent data structures through quasi-linearizability [2]. Just like quasi-linearizability, we provide quantitative relaxations of concurrent data structures. Unlike quasi-linearizability which uses syntactic distances, our relaxations are based on semantical distances from a sequence to the sequential specification. We briefly present the quasi-linearizability approach, identify two main issues, and how our method overcomes these.

We call two sequences $\mathbf{x}$, $\mathbf{x}'$, both of length n, permutation equivalent, written $\mathbf{x} \sim \mathbf{x}'$, if there exists a permutation p on $\{1,\ldots,n\}$ such that for all $1 \leqslant i \leqslant n$, $\mathbf{x}(i) = \mathbf{x}'(p(i))$. We write $\mathbf{x} \sim_p \mathbf{x}'$ to emphasize the permutation witnessing $\mathbf{x} \sim \mathbf{x}'$. In such a case, the permutation distance between $\mathbf{x}$ and $\mathbf{x}'$ is given as $\max\{|i - p(i)| \mid 1 \leqslant i \leqslant n\}$.

Let S be a sequential specification over Σ. In [2], the distance of a sequence $\mathbf{x} \in \Sigma^*$ to S is defined via a collection D of subsets of Σ. Let $\mathbf{y} \in \Sigma^*$ be a sequence such that $\mathbf{z} = \mathbf{xy}$ has a permutation equivalent $\mathbf{z}' \in S$. Then, for $A \in D$, the A-cost of obtaining $\mathbf{z}'$ from $\mathbf{z}$ is the permutation distance between $\mathbf{z}|A$ and $\mathbf{z}'|A$, where $|$ denotes restriction. Let k_A denote the minimal A-cost over all $\mathbf{y}$. Then, $\mathbf{x}$ is quasi-linearizable with quasi-linearization factor $Q\colon D \to \mathbb{N}$, if for all $A \in D$, $k_A \leqslant Q(A)$. Observe that the distance for $\mathbf{x}$ is obtained by quantifying over all possible extensions of $\mathbf{x}$ whose permutations are in S. We now show that this definition fails to capture desired relaxation distances.

1. *Not precise.* Consider the following sequence:

$$\mathtt{enq(1)enq(2)enq(3)deq(1)deq(2)enq(4)deq(4)}$$

In order to assign a relaxation cost of 1 to this sequence belonging to an out-of-order queue, quasi-linearizability employs a scheme where only enqueue operations are allowed to commute. Formally, quasi-linearizability uses $D = \{Enq, Deq\}$ with $Q(Enq) = k$ and $Q(Deq) = 0$, where Enq (resp., Deq) contains all enq (resp., deq) symbols. However, with this scheme the sequence $\mathtt{deq(i)}^n$ which removes elements from an empty queue will always be in any k-relaxation of the queue because setting

$$\mathbf{z} = \mathtt{deq(i)}^n\mathtt{enq(i)}^n, \quad \mathbf{z}' = \mathtt{enq(i)}^n\mathtt{deq(i)}^n$$

will give $k_{Enq} = k_{Deq} = 0$, independent of the value n. This means that the following implementation is a 0-relaxation of queue.

```
enq(x): { while(true);}  deq(): { return random();}
```

This implementation is clearly not implementing a queue, nor any intended bounded relaxation of a queue, but all the sequences it generates will have zero distance relative to D as given above. Thus, quasi-linearizability cannot exactly capture intended relaxations and might allow wrong behaviors. Observe that we have already shown in Proposition 5.2 that out-of-order k-queues can never generate such erroneous behavior.

2. *Not general.* For a stack, consider the sequence

$$\begin{aligned}\mathbf{x} \quad = \quad & \mathtt{push(a)[push(i)pop(i)]}^n \\ & \mathtt{push(b)[push(j)pop(j)]}^m \\ & \mathtt{pop(a)[push(l)pop(l)]}^r\end{aligned}$$

where all symbols, $\mathtt{a},\mathtt{b},\ldots$ have distinct values. Prior to the removal of a, the stack contains a and b, with the latter at the top position. The distance in the out-of-order relaxation induced by maximal path cost and segment cost in this case is 1 since the element popped is immediately after the top entry. However, with quasi-linearization factor Q, it is impossible to precisely capture out-of-order penalty for data structures like stacks. First, consider the case where we pick $\mathbf{z} = \mathbf{x}$, which we can do since $\mathbf{x}$ has a permutation equivalent valid stack sequence. In order to get a permutation $\mathbf{x}'$ of $\mathbf{x}$ such that $\mathbf{x}'$ is a valid sequence of a stack, either one of pop(a) or push(b) has to move over m copies of push and pop operations, or one of push(a) or push(b) has to move over n copies of push and pop operations. So either D is empty which allows for any sequence to be in the relaxation or it is always possible to pick the values for n and m such that the penalty is arbitrarily large. Second, consider the case where we extend $\mathbf{x}$ with $\mathbf{y}$ such that $\mathbf{xy}$ has a permutation equivalent valid stack behavior. But because of the $2r$-long suffix of $\mathbf{x}$, a similar reasoning as in the previous case applies to this case as well.

Similarly, a stuttering relaxation will not have a finite quasi-linearizability distance, since no permutation of (an extension of) a stuttering sequence is in the original sequential specification.

Consistency conditions. As opposed to relaxing the sequential specification of a concurrent data structure, one may also relax the consistency condition, e.g., quiescent consistency [4] instead of linearizability. We note that linearizable out-of-order relaxation of a stack is incomparable to a quiescently consistent stack. To see this, first, consider a concurrent history $\mathbf{c}$ with two threads t_1 and t_2. The history $\mathbf{c}$ starts with the invocation of push(a) by t_1, followed by a sequence $\texttt{pop(i)}^n\texttt{push(i)}^n$ all executed by t_2. This history is quiescently consistent for stack because the reordering of methods (even those that do not overlap in time) is allowed as long as they are not separated by a quiescent state. On the other hand, any linearization of $\mathbf{c}$ will have to observe out-of-order pop operations since the operations of t_2 do not overlap. So, for each k, there exists a history which is quiescently consistent for stack but is not in the specification of an out-of-order k-stack. Second, consider the sequential history push(a)push(b)pop(a) which has an out-of-order relaxation distance of 1. Since the history is sequential, quiescent consistency will not allow any reordering. Thus, for any k, there exists a history which is in the specification of an out-of-order k-stack but not quiescently consistent. A comprehensive overview of variants of weaker and stronger consistency conditions than linearizability can be found in [6].

Implementations. Work related to our implementations and experiments is discussed in more detail in Section 8 and Section 9. Here we briefly refer to all the work considered. Our relaxed stack implementation is closely related to the very recent efficient lock-free implementation of a relaxed k-FIFO queue [11] (by some of us and a third coauthor), but the change from queue to stack semantics imposes a significant difference as well. The k-FIFO queue [11] is in turn related to implementations of relaxed FIFO queues such as the Random Dequeue and Segment Queue [2] as well as Scal queues [10, 12]. Both the Random Dequeue Queue and the Segment Queue implement the restricted out-of-order relaxation. The Segment Queue [2] and the k-FIFO queue [11] implement a queue of segments. However, the implementations are quite different with significant impact on performance, see Section 9. Our relaxed stack implementation implements a stack of segments. Scal queues are relaxed queues with, in general, unbounded relaxation. Since any relaxed stack or queue implementation also implements a pool, we compare our work also to state-of-the-art pool implementations [1, 3, 19].

In [5], the authors show that implementing deterministic data structure semantics requires expensive synchronization mechanisms which may prohibit scalability in high contention scenarios. We agree with that and show in our implementations and experiments that the non-determinism introduced in the sequential specification provides scalability and performance benefits. In [15], the authors present a work-stealing queue with relaxed semantics where queue elements may be returned any number of times instead of just once. In comparison to other state-of-the-art work-stealing queues with non-relaxed semantics this may provide better performance and scalability. Again the introduced non-determinism pays off. Overview of different relaxations on hardware and software level is presented in [9, 18].

8. Implementations of relaxed data structures

In this section we present the new implementation of a restricted out-of-order stack (k-stack, for short) and present the two new implementations of a stuttering shared counter. It is interesting to note that the "restricted" relaxation seems to be crucial for obtaining performance[2], which is why we focus on it.

8.1 k-Stack

The top pointer of a concurrent stack may become a scalability bottleneck under high contention [20]. The main idea behind our k-stack implementation is to reduce contention on the top pointer by maintaining a stack of so-called k-segments[3]. We implemented the stack that holds the k-segments similarly to the lock-free stack of [20] with the difference that there is always at least one k-segment, even if it is empty, on the stack. This avoids unnecessary removal and adding of a k-segment, e.g. in the empty state. A k-segment (or just segment, when no confusion arises) contains k "atomic values" (see next paragraph) which may either point to `null` indicating an empty slot or may hold a so-called item. Both push and pop operations are served by the top segment. Hence, up to k stack operations may be performed in parallel. A push operation tries to insert an element in the top segment. It adds a new segment to the stack if the top segment is full. A pop operation tries to remove an element from the top segment. It removes the top segment from the stack if it is empty and is not the only segment on the stack. Additionally, each segment contains an atomic counter `remove` that counts how many threads are trying to remove it from the stack. The counter is initially set to zero.

The pseudo code of the lock-free k-stack algorithm with $k > 0$ is depicted in Figure 4. The occurrence of the ABA problem is made unlikely through version numbers. We refer to values enhanced with version numbers as atomic values. Hence an atomic value has two fields, the actual value `val` and its version number `ver`.

The methods `init`, `try_add_new_ksegment`, and `try_remove_ksegment` implement the stack of segments. In the latter, the atomic counter `remove` is updated and the method `empty` that performs an empty check is called. We discuss the method `empty` within the pop method, as it is also called there.

Let `item` represent an element to be pushed on the k-stack. The `push` method first tries to find an empty slot for the `item` using the `find_empty_slot` method (line 45). The `find_empty_slot` method randomly selects an index in the top k-segment and then linearly searches for an empty slot starting with the selected index and wrapping around at index k. Then the `push` method checks if the k-stack state has been consistently observed by testing whether `top` changed in the meantime (line 46) which would trigger a retry. If an empty slot is found (line 47) the method tries to insert the `item` at the location of the empty slot using a compare-and-swap (`CAS`) operation (line 49). If the insertion is successful the method verifies whether the insertion is also valid by calling the `committed` method (line 50), as discussed below. If any of these steps fails, a retry is performed. If no empty slot is found in the current top segment, the `push` method tries to add a new segment to the stack of segments (line 53) and then retries.

The `committed` method (line 24) validates an insertion, it ensures that the inserted element is really inserted *on the stack*. This method is the core and the main novelty of the algorithm. It returns `true` when the insertion is valid and `false` when it is not valid. An insertion is invalidated if a concurrent thread removes the segment to which the element was inserted before the effect of the

[2] For an efficient implementation one needs a sequential specification that fits the properties of the hardware that it will run on.

[3] The same high-level idea is used in the Segment Queue [2] and the k-FIFO queue [11] discussed in the next subsection.

```
 1 global top;
 2
 3 void init():
 4   new_ksegment = calloc(sizeof(ksegment));
 5   top = atomic_value(new_ksegment, 0);
 6
 7 void try_add_new_ksegment(top_old):
 8   if top_old == top:
 9     new_ksegment = calloc(sizeof(ksegment));
10     new_ksegment->next = top_old;
11     top_new = atomic_value(new_ksegment, top_old.ver+1);
12     CAS(&top, top_old, top_new);
13
14 void try_remove_ksegment(top_old):
15   if top_old == top:
16     if top_old->next != null:
17       atomic_increment(&top_old->remove);
18       if empty(top_old):
19         top_new = atomic_value(top_old->next,
                top_old.ver+1);
20         if CAS(&top, top_old, top_new):
21           return;
22       atomic_decrement(&top_old->remove);
23
24 bool committed(top_old, item_new, index):
25   if top_old->s[index] != item_new:
26     return true;
27   else if top_old->remove == 0:
28     return true;
29   else:  //top_old->remove >= 1
30     item_empty = atomic_value(EMPTY, item_new.ver+1);
31     if top_old != top:
32       if !CAS(&top_old->s[index], item_new, item_empty):
33         return true;
34     else:
35       top_new = atomic_value(top_old.val, top_old.ver+1);
36       if CAS(&top, top_old, top_new):
37         return true;
38       if !CAS(&top_old->s[index], item_new, item_empty):
39         return true;
40   return false;
41
42 void push(item):
43   while true:
44     top_old = top;
45     item_old, index = find_empty_slot(top_old);
46     if top_old == top:
47       if item_old.val == EMPTY:
48         item_new = atomic_value(item, item_old.ver+1);
49         if CAS(&top_old->s[index], item_old, item_new):
50           if committed(top_old, item_new, index):
51             return true;
52       else:
53         try_add_new_ksegment(top_old);
54
55 item pop():
56   while true:
57     top_old = top;
58     item_old, index = find_item(top_old);
59     if top_old == top:
60       if item_old.val != EMPTY:
61         item_empty = atomic_value(EMPTY, item_old.ver+1);
62         if CAS(&top_old->s[index], item_old, item_empty):
63           return item_old.val;
64       else:
65         if only_ksegment(top_old):
66           if empty(top_old):
67             if top_old == top:
68               return null;
69         else:
70           try_remove_ksegment(top_old);
```

Figure 4. Lock-free *k*-stack algorithm

insertion took place. Therefore, an insertion is valid if the inserted item already got popped at validation time by a concurrent thread (line 25, 32, 38) or the segment where the item was inserted was not removed by a concurrent thread (line 27). A `remove` counter larger than zero indicates that the segment has been removed or concurrent threads are trying to remove the segment from the stack (line 29). If the current top segment is not equal to the segment where the item was inserted we have to conservatively assume that the segment was removed from the stack (line 31) and undo the insertion (line 32). If the current top segment is equal to the segment where the item was inserted, a race with concurrent popping threads may occur which may not have observed the insertion of the item and may try to remove the *k*-segment from the stack in the meantime. This would result in loss of the inserted item. To prevent that, the method tries to increment the version number in the `top` atomic value using `CAS` (line 36) forcing threads that concurrently try to remove that *k*-segment to retry. If this fails, a concurrent pop operation may have changed `top` (line 20) which would make the insertion potentially invalid. Hence, in case of losing the race, the method tries to undo the insertion using `CAS` (line 38).

The `pop` method returns an item if the *k*-stack is not empty. Otherwise it returns `null`. Similar to the `push` method, the `pop` method first tries to find an item in the top segment using the `find_item` method (line 58). The `find_item` method randomly selects an index in the top *k*-segment and then linearly searches for an item starting with the selected index and wrapping around at index *k*. Then, the `pop` method checks if the *k*-stack state has been consistently observed by checking whether `top` changed in the meantime (line 59) which would trigger a retry. If an item was found (line 60) the method tries to remove it using `CAS` (line 62) and returns it if the removal was successful (line 63). Otherwise a retry is performed. If no item is found and the current segment is the only segment on the stack (line 65) an empty check is performed using the method `empty` (line 66). This method stores the values of the *k* slots of the segment in a local array (if they are empty) and subsequently checks in another pass over the segment slots whether the values in the slots changed in the meantime. If a non-empty slot was found, the empty method immediately returns `false`. If the empty check succeeded and the top did not change in the meantime (line 67), `null` is returned (line 68). Otherwise, if no item is found in the current segment and there is more than one segment in the stack, the method tries to remove the segment (line 70) and performs a retry.

Correctness: *k*-Stack

We now prove that the *k*-stack implementation is correct for the relaxed stack semantics.

PROPOSITION 8.1. *The k-stack algorithm is linearizable with respect to restricted out-of-order k-stack.*

Proof. Without loss of generality, we assume that each item pushed on the stack is unique. A segment `s'` is *reachable* from a segment `s` if either `s'=s` or `s'` is reachable from `s->next`. An item `i` is *on the stack*, if `push(i)` has already committed and there exists a segment reachable from the top segment containing a slot whose value is `i`. Note that reachability is important, i.e., only having a slot containing the item is not enough to guarantee that the item is logically on the stack, because the slot could be in a segment (to be) removed by a concurrent pop operation.

We begin by identifying a linearization point of each method call. The goal is to show that the sequential history obtained from a concurrent history by ordering methods according to their linearization points is in the specification of a restricted out-of-order *k*-stack. The linearization point of `push` is the reading of the empty slot (line 45) in the last iteration (successful insertion) of the main loop. The linearization point of `pop` that does not return `null` is the reading of a non-empty slot (line 58) in the last iteration (successful removal) of the main loop. The linearization point of a `null`-returning `pop` is the point after the first pass of the segment in the call to `empty` method (line 66) which returns `true`.

The correctness argument is based on the following facts.

1. An item is pushed on the stack exactly once. This is a consequence of our unique-items assumption and the control flow of `push(i)`, the only method that can modify a slot to contain `i`.

2. An item is popped at most once. If an item `i` is on the stack, then it can only be removed once, because of 1. and the existence of a unique statement which replaces `i` with empty. If `i` is in some slot but not on the stack, then `push(i)` will erase `i` and retry insertion before committing. We have to show that while `i` is in some slot but not on the stack, no pop operation can return `i`. Clearly, the call to method `committed` by `push(i)` must return `false`. This implies that until `committed` completes, the slot where `i` resides is not modified by any other thread. Otherwise, either after the first `if` statement (line 25) or following failed `CAS` attempts (lines 32 and 38) of replacing `i` with empty will lead to returning true. Furthermore, when control reaches the only exit point for returning `false`, it is guaranteed that there is no slot containing `i`. Thus, if `i` is not on the stack, no pop operation could have replaced it with empty.

3. If a pop operation returns `null`*, then during its execution, there must be a state at which there are no items on the stack.* Since returning `null` is without any side-effect, it suffices to prove the existence of a state which corresponds to a logically empty stack. The call to `empty` is only done when the top segment is the only segment in the stack. In the `empty` method, the value of `top` is checked at the beginning and after the first pass to ensure that the pointer is not updated by concurrent operations. Hence, the stack is indeed logically empty at the linearization point since the second pass succeeds.

4. An item `j` *cannot be popped before an item* `i`*, if they are both on the stack, and* `i`, `j` *are in segments* `s`, `s'`*, respectively, with* `s'` *reachable from* `s` *and* `s'` $\neq$ `s`. The segment `s'` can become a top segment only after the segment `s` has been removed by some pop operation. Moreover, if a segment becomes unreachable from the top segment, it remains unreachable. These two observations imply that the linearization point of `pop(i)` must precede the linearization point of `pop(j)` which can only happen when `s'` is a top segment.

5. An item `i` *on the stack is popped only if it is one of the* $k-l$ *youngest items on the stack, where* l *is the current lateness of the youngest item.* By youngest we mean most recently pushed. Recall that lateness is the number of pops that were performed after the pop of the previous youngest item or the push of the current youngest one. Assume `i` is popped from the stack at the current moment in time and at that point the youngest item is `t` with current lateness l. This means that ever since `t` is the youngest item on the stack, no push operation was performed and there have been l pop operations performed none of which removed `t`. Let `j` be any of these l popped items. Since `t` is the last item pushed and it is still on the stack, `t` is in the top segment. Since `j` is removed before `t`, by 4. it must have also resided in the top segment. For the same reason, also `i` is in the top segment prior to its removal. Hence, at the moment in which `pop(i)` happens, there are at most $k-l$ items in the top segment.

Now, 5. shows that the sequential behavior obtained by ordering methods according to their linearization points satisfies the restricted-out-of-order k-stack. Moreover, 3. shows an even stricter behavior, a linearizable empty check.[4] Thus, any concurrent execution generated by the given algorithm is linearizable for restricted out-of-order k-stack. Observe that already 1. and 2. show that the k-stack has pool semantics. □

```
struct blk_original {
  pidtype X;
  bool Y;
  val_t V;
  bool C;
  val_t D;
};
```

(a) CAS.

```
struct blk_modified {
  pidtype X[k];
  bool Y[k];
  val_t V;
  bool C[k];
  val_t D;
};
```

(b) k-CAS.

Figure 5. Wait-free CAS and k-CAS state structures.

Without any particular difficulty, but with a somewhat lengthy argument, one can show that the k-stack algorithm is lock-free by showing that whenever a thread retries an operation, another thread completes its operation ensuring progress of at least one thread.

8.2 k-Stuttering shared counters

We implemented the two versions of a stuttering k-shared counter, as discussed in the introduction. The first version is based on a k-relaxed stuttering version of a wait-free software CAS operation [13] (k-CAS for short). It uses a structure `blk_original`, shown in Figure 5(a), to keep track of the state of concurrent CAS operations. The atomic value is located in field `V` and the CAS operation uses the decision fields `X`, `Y`, and `C` to determine which thread gets permission to change `V`. We modified the `blk_original` structure so that the fields `X`, `Y`, and `C` are arrays of size k depicted in structure `blk_modified` in Figure 5(b). We keep the main CAS operation unmodified but use a balancing function that maps threads to array indices i smaller than k (the thread ID modulo k). A thread determines the state of its CAS operation by just accessing position i in the arrays `X`, `Y`, and `C`. On success, a thread writes the new value into `V`. Hence, up to k concurrent threads may perform the k-CAS operation in parallel and change `V` resulting in a loss of at most $k-1$ state changes, which further results in at most $k-1$ lost shared counter updates.

The second version of the k-shared counter is the k-distributed counter depicted in Figure 1(b).

It is not difficult to show that both implementations are linearizable with respect to the k-stuttering shared counter and they are lock-free.

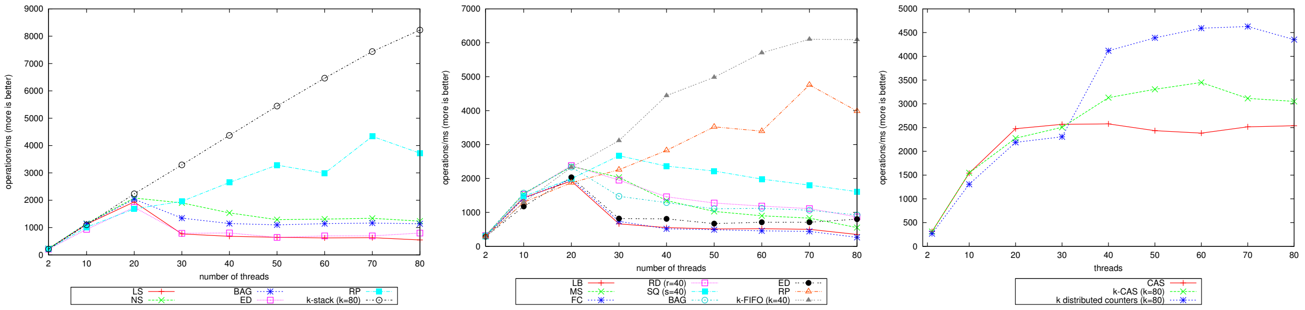
9. Experiments

We evaluate the performance and scalability of the k-stack, several existing quantitatively relaxed FIFO queues, and the k-stuttering shared counter implementations. All experiments ran on an Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (40 cores, 2 hyperthreads per core), 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39. We implemented a benchmarking framework to analyze our k-stack and k-shared counter implementations, as well as the implementations of relaxed queues. The benchmarking framework can be configured for a different number of threads (n), number of operations each thread performs (o), and the computational load performed between each operation (c). The computational load between two consecutive operations is created by iteratively calculating π and c is the number of iterations performed. We use $c = 2000$ which takes a total of 4600ns on average in our experiments. The framework uses static preallocation for memory used at runtime with touching each page before running the benchmark to avoid paging issues.

9.1 k-Stack

We compare our k-stack implementation with a standard lock-based stack (LS), which acquires a global lock for each stack operation, and a non-blocking stack (NS) [20], which uses a CAS operation to manipulate the top pointer of the stack. Moreover, we compare our k-stack with different pools. The lock-free (BAG) [19]

[4] We could easily relax the linearizable empty check to fit the restricted out-of-order specification, by removing line 66 in the code. However, a linearizable empty check is a valuable feature of a concurrent implementation.

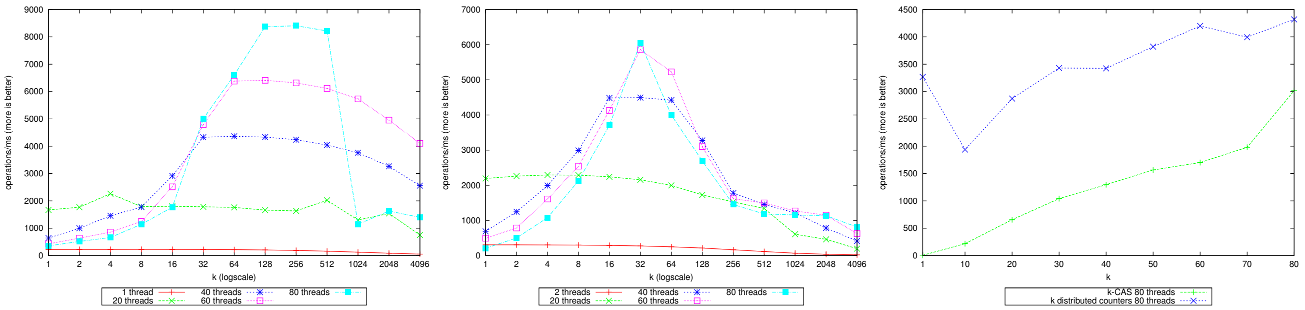


(a) Stack

(b) FIFO queue

(c) Shared counter

Figure 6. Benchmarks on a 40-core (2 hyperthreads per core) server with an increasing number of threads



(a) Stack

(b) FIFO queue

(c) Shared counter

Figure 7. Benchmarks on a 40-core (2 hyperthreads per core) server with increasing k

pool is based on thread-local lists of elements. Threads put elements on their local list and take elements from their local list if it is not empty. If it is empty they take elements from the lists of other threads. The lock-free elimination-diffraction pool (ED) [1] uses a set of FIFO queues to store elements. Access to these queues is balanced using elimination arrays and a diffraction tree. The synchronous rendezvousing pool (RP) [3] implements a single elimination array using a ring buffer. A get operation marks a slot identified by its thread id and waits for a take operation to insert an element. Take operations scan the array for pending get operations.

Figure 6(a) depicts the performance analysis of our k-stack. We configure $k = 80$, which is a good k (on average) for a broad range of thread combinations and workloads on our server machine. This is no surprise since the server machine has (logically) 80 cores. The analysis is done on a producer-consumer workload where half of the threads are producers and half are consumers. The k-stack outscales and outperforms all considered stack and pool implementations. Figure 7(a) shows the effect of k on performance and scalability. There exists an optimal k with respect to performance, which is also robust in the sense that there exists only a single range of close-to-optimal k. For large k performance decreases due to higher sequential overhead, e.g. scanning for elements in almost empty k-segments. Note that an increase in performance above $k = 80$ is not to be expected on the given architecture.

9.2 Quantitatively relaxed FIFO queues

We also evaluate the existing implementations of a k-FIFO queue [11] and different FIFO queues, quasi-linearizable FIFO queues, and the pools introduced in the previous section. The k-FIFO queue [11] implements a restricted out-of-order k-queue as a lock-free linked list of k-segments. An enqueue operation is served by the tail k-segment and a dequeue operation is served by the head k-segment. Hence, up to k enqueue and k dequeue operations may be performed in parallel. The k-FIFO queue is empty if head and tail point to the same k-segment which does not contain any elements. The lock-based (strict) FIFO queue (LB) locks a global lock for each queue operation. The lock-free Michael-Scott (strict) FIFO queue (MS) [14] uses CAS operations to change head, tail, and next pointer in a linked list of elements. The flat-combining (strict) FIFO queue (FC) [8] is based on the approach that a single thread performs the queue operations of multiple threads by locking the whole queue, collecting pending queue operations, and applying them to the queue. The Random Dequeue Queue (RD) [2] implements a quasi-linearizable FIFO with quasi-factor r where r defines the range $[0, r-1]$ of a random number. It actually implements a restricted out-of-order r-FIFO queue. RD is based on MS where the dequeue operation was modified in a way that the random number determines which element is returned starting from the oldest element. The Segment Queue (SQ) [2] is a quasi-linearizable FIFO queue with quasi-factor s. It is logically similar (both implement a queue of segments and hence a restricted out-of-order queue) to the k-FIFO queue but does not provide a linearizable empty check, i.e., it may return null in the not-empty state. Also, SQ comes with a different segment management strategy than the k-FIFO queue, which results on average in significantly more CAS operations.

Figure 6(b) depicts the performance analysis of the queues. We configure $k = r = s = 40$, which turns out be a good k (on average) for a broad range of thread combinations and workloads on our server machine. This is no surprise since then the level of possible parallelism is $2k = 80$, the number of (logical) cores. We use a producer-consumer workload where half of the threads are producers and half consumers. The k-FIFO queue outscales and outperforms all considered FIFO queue, quasi-linearizable FIFO queue, and pool implementations. Figure 7(b) shows the effect of k on performance and scalability. Again, there exists a robust and optimal k with respect to performance. Also here, for large k performance decreases due to larger sequential overhead.

9.3 k-Shared counter

We compare our k-CAS-based shared counter and distributed shared counter implementations with a shared counter implementation based on a regular CAS operation depicted in Figure 6(c). The threads perform in total one million counter increment operations in each benchmark run. The CAS version performs best until 30 threads. After that the k-CAS-based shared counter and the distributed shared counter version outperform it. Figure 7(c) shows the effect of k on performance and scalability. In the k-CAS version performance monotonically increases with larger k, whereas in the distributed shared counter version performances decreases until $k = 10$, but monotonically increases after that. Our educated guess is that this is caused by the trade-off between two possible sources of contention: (1) CAS on the same memory location, and (2) bad caching, i.e., accessing many different locations in memory. The distributed shared counter decreases (1) but increases (2). However, except for small values of k, we observe that the gain is larger than the loss.

10. Final remarks

We have presented a framework for quantitative relaxation of concurrent data structures together with generic as well as further concrete instances of it. Our main motivation is the belief that relaxed data structures may decrease contention and thus provide the potential for scalable and well-performing implementations. Indeed, the potential advantage which we demonstrate utilizable is striking. The lessons learned can be summarized as follows: The way from a sequential implementation to efficient concurrent implementation is always hard. Just because a sequential specification is relaxed, it does not necessarily mean that an efficient implementation immediately follows. However, efficient implementations that benefit from quantitative relaxations are possible, as we demonstrate in this paper. In our opinion, the framework provides a firm formal ground for quantitative relaxation of concurrent data structures and paves the road to designing efficient concurrent implementations.

Our current results open up several directions for future work. One important issue is the applicability of relaxed data structures. Demonstrating applicability can either be achieved by exploring applications that tolerate a relaxation, e.g. provide less accurate but nevertheless acceptable results, or showing that end-to-end quality may remain the same despite the actual relaxation of semantics. In the latter case, relaxations do not influence correctness in the sense of [16, 17]. Another evident but difficult goal would be to synthesize well-performing implementations from relaxations. As a first step we believe it is important to study the main principles that lead to good performance. This is another line of future work that we plan to undertake in small steps.

Acknowledgements

This work has been supported by the European Research Council advanced grant QUAREM, the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23), and an Elise Richter Fellowship (Austrian Science Fund V00125). We thank the anonymous referees for their constructive and inspiring comments and suggestions. Ana Sokolova wishes to thank Dexter Kozen and in particular Joel Ouaknine: had they not saved her life, she would have missed a lot of the fun involved in working on this paper and seeing it finished.

References

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proc. European Conference on Parallel Processing (Euro-Par)*, pages 151–162. Springer, 2010.

[2] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proc. Conference on Principles of Distributed Systems (OPODIS)*, pages 395–410. Springer, 2010.

[3] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. In *Proc. International Conference on Distributed Computing (DISC)*, pages 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41:1020–1048, 1994.

[5] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of Principles of Programming Languages (POPL)*, pages 487–498. ACM, 2011.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[7] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[8] D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364. ACM, 2010.

[9] C. Kirsch and H. Payer. Incorrect systems: It's not the problem, it's the solution. In *Proc. Design Automation Conference (DAC)*. ACM, 2012.

[10] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Brief announcement: Scalability versus semantics of concurrent FIFO queues. In *Proc. Symposium on Principles of Distributed Computing (PODC)*. ACM, 2011.

[11] C. Kirsch, M. Lippautz, and H. Payer. Fast and scalable k-fifo queues. Technical Report 2012-04, Department of Computer Sciences, University of Salzburg, June 2012.

[12] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. In *Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 273–287. LNCS 7439, 2012.

[13] V. Luchangco, M. M., and N. Shavit. On the uncontended complexity of consensus. In *Proc. International Symposium on Distributed Computing (DISC)*, pages 45–59. Springer-Verlag, 2003.

[14] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.

[15] M. Michael, M. Vechev, and V. Saraswat. Idempotent work stealing. In *Proc. Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54. ACM, 2009.

[16] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard. Quality of service profiling. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE) - Volume 1*, pages 25–34. ACM, 2010.

[17] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: approximate data types for safe and general low-power computation. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174. ACM, 2011.

[18] N. Shavit. Data structures in the multicore age. *Communications ACM*, 54:76–84, March 2011.

[19] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 335–344, New York, NY, USA, 2011. ACM.

[20] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.