CQS: A Formally-Verified Framework for Fair and Abortable Synchronization

NIKITA KOVAL, JetBrains, The Netherlands
DMITRY KHALANSKIY, JetBrains, Germany
DAN ALISTARH, IST Austria, Austria

Writing concurrent code that is both correct and efficient is notoriously difficult. Thus, programmers often prefer to use synchronization abstractions, which render code simpler and easier to reason about. Despite a wealth of work on this topic, there is still a gap between the rich semantics provided by synchronization abstractions in modern programming languages—specifically, *fair* FIFO ordering of synchronization requests and support for *abortable* operations—and frameworks for implementing it correctly and efficiently. Supporting such semantics is critical given the rising popularity of constructs for asynchronous programming, such as coroutines, which abort frequently and are cheaper to suspend and resume compared to native threads.

This paper introduces a new framework called `CancellableQueueSynchronizer` (CQS), which enables simple yet efficient implementations of a wide range of fair and abortable synchronization primitives: mutexes, semaphores, barriers, count-down latches, and blocking pools. Our main contribution is algorithmic, as implementing both fairness and abortability efficiently at this level of generality is non-trivial. Importantly, all our algorithms, including the CQS framework and the primitives built on top of it, come with *formal proofs* in the Iris framework for Coq for many of their properties. These proofs are modular, so it is easy to show correctness for new primitives implemented on top of CQS. From a practical perspective, implementation of CQS for native threads on the JVM improves throughput by up to two orders of magnitude over Java's `AbstractQueuedSynchronizer`, the only practical abstraction offering similar semantics. Further, we successfully integrated CQS as a core component of the popular Kotlin Coroutines library, validating the framework's practical impact and expressiveness in a real-world environment. In sum, `CancellableQueueSynchronizer` is the first framework to combine expressiveness with formal guarantees and solid practical performance. Our approach should be extensible to other languages and families of synchronization primitives.

CCS Concepts: • **Theory of computation → Data structures design and analysis**; **Program reasoning**.

Additional Key Words and Phrases: Concurrent Data Structures, Iris, Synchronization Primitives, Abortability, Kotlin Coroutines

ACM Reference Format:
Nikita Koval, Dmitry Khalanskiy, and Dan Alistarh. 2023. CQS: A Formally-Verified Framework for Fair and Abortable Synchronization . *Proc. ACM Program. Lang.* 7, PLDI, Article 116 (June 2023), 23 pages. https://doi.org/10.1145/3591230

1 INTRODUCTION

Providing the "right" set of programming abstractions to enable efficient and correct concurrent code is a question as old as the field of concurrency [Dijkstra 2001; Knuth 1966]. One of the most basic primitives is the *mutex*, which allows access to the critical section to at most one thread, via `lock()` and `unlock()` invocations. Standard libraries, e.g., the Java concurrency library [Lea 2005],

Authors' addresses: Nikita Koval, JetBrains, The Netherlands, nikita.koval@jetbrains.com; Dmitry Khalanskiy, JetBrains, Germany, dmitry.khalanskiy@jetbrains.com; Dan Alistarh, IST Austria, Austria, dan.alistarh@ist.ac.at.

2475-1421/2023/6-ART116
https://doi.org/10.1145/3591230

provide more general primitives, such as the *semaphore*, which allows at most a fixed number of threads to be in the critical section simultaneously, the *barrier*, which allows a set of threads to wait for each other at a common program point, and the *count-down latch*, which allows threads to wait until a given set of operations is completed.

Although basic versions of the above primitives exist in most specialized libraries, programmers often require *stronger semantics* from synchronization abstractions, which are supported by modern programming languages such as Java, C#, Go, Kotlin and Scala. One particularly desirable property is *fairness* [Izraelevitz and Scott 2017], by which the order of critical section traversals should respect the FIFO order of arrivals, to avoid starvation. A second key property is *abortability*, which enables a thread to *cancel* its request, due to a time-out or user-specified behavior. Abortability and scalability are especially important in the context of *coroutines* [kot 2022; Kahn and MacQueen 1976], the number of which can be in the millions simultaneously, and which can be frequently cancelled. Coroutines are also significantly cheaper to suspend and resume compared to native threads: internally, they are scheduled on a thread pool, so when a coroutine is suspended, the corresponding thread immediately picks up another one and executes it instead, so the native thread never blocks. In such a setting, efficient cancellation is vital, while fairness becomes less expensive and more natural. Coroutines are now a key component of modern programming languages such as Java, C++, Go, Scala, and Kotlin.

Implementing fairness and cancellation efficiently in a concurrent context is known to be very challenging, and there is a long line of work proposing highly non-trivial designs [Alon and Morrison 2018; Danek and Golab 2010; Giakkoupis and Woelfel 2017; Jayanti 2003; Lee 2010; Pareek and Woelfel 2012]. Modern languages and libraries typically either restrict the generality of the semantics, providing more efficient *unfair* synchronization, or implement complex constructs, which may lead to correctness and performance issues. This paper addresses the question of implementing *fair* and *abortable* synchronization primitives in a way that is general, efficient, and easy to reason.

For intuition, we begin with the observation that most of the synchronization operations we focus on are inherently *blocking*: threads attempt to acquire a shared resource or synchronize, and may have to wait for the resource to become available. For example, the mutex `lock()` operation either acquires the lock instantly or adds the currently running thread to a queue of waiting operations and suspends. The `unlock()` invocation resumes the first waiting `lock()` request, handing the lock over. To our knowledge, the only practical abstraction to implement such synchronization primitives in full generality is the `AbstractQueuedSynchronizer` in Java [Lea 2005], which maintains a FIFO queue of suspended requests in a way reminiscent of the state-of-the-art CLH mutex [Magnusson et al. 1994]. While the `AbstractQueuedSynchronizer` has been extremely influential, its design does not scale to high contention. Our goal is to provide a design that is general enough to support a wide range of abstractions, but also efficient enough to support modern usage scenarios.

Our Contribution. We introduce a new framework called `CancellableQueueSynchronizer` (CQS), which enables simple and efficient implementations of a wide range of *fair* and *abortable* synchronization and communication primitives, such as mutexes, semaphores, barriers, count-down latches, and blocking pools. We show that CQS can implement this wide range of synchronization primitives, for which we provide both formal proofs and extensive experimental validation, showing significant practical improvements over the state of the art.

Conceptually, the goal of the `CancellableQueueSynchronizer` framework is to efficiently maintain a FIFO queue of waiting threads, corresponding to operations to be completed. To this end, CQS provides two main operations: (1) `suspend()`, which adds the current thread as a waiter into the queue and suspends, and (2) `resume(result)`, which tries to retrieve and resume the first waiter with the specified result. One key advantage of the CQS semantics is that it allows operations to invoke `resume(..)` before `suspend()`: we actively use this property for implementation simplicity and better performance. Despite the relative simplicity of the `CancellableQueueSynchronizer` API, it allows us to support a rich set of synchronization and communication primitives.

The data structure behind `CancellableQueueSynchronizer` uses techniques from modern concurrent queue implementations [Morrison and Afek 2013; Yang and Mellor-Crummey 2016], leveraging the `Fetch-and-Add` instruction for better scalability. In brief, our solution is based on a logically-infinite array, equipped with two position counters, indexing `suspend()` and `resume(..)` operations, respectively. Each operation starts by incrementing its counter via `Fetch-and-Add`, thus reserving the cell in the first-come-first-served order. The rest of the synchronization is performed in the cell: `suspend()` stores the current thread, while `resume(..)` wakes up the suspended thread.

The main novelty behind `CancellableQueueSynchronizer` is the efficient built-in support for *aborting/cancelling* operations. We emulate the infinite array with a linked list of fixed-sized *cell segments*, so each cell stores a waiting operation. On thread cancellation, the cell state should be reclaimed to avoid memory leaks, but also segments full of cancelled requests should be physically removed from the linked list. One naive way to implement such functionality is by linearly searching the waiting queue for the corresponding segment, which is then unlinked [Lea 2005]. However, this would have a worst-case linear time in the queue size, making frequent cancellations lead to significant overheads. While for threads this approach is sufficient (since their number is small and they rarely abort), for coroutines, of which millions can exist at the same time and which cancel frequently, significant complexity improvements are required. We propose a more efficient design, where segments form a concurrent doubly-linked list, enabling constant time removals via careful pointer manipulations. This also allows us to support different cancellation modes: in case of *simple* cancellation, `resume(..)` is allowed to fail if the waiter located in the corresponding cell was cancelled, whereas *smart* cancellation provides a mechanism to efficiently skip a sequence of aborted requests but requires complex mechanisms to ensure that some thread is always resumed.

Formal Proofs in Iris/Coq. The complexity of the resulting CQS implementation renders manual correctness proofs quite challenging and error-prone. We provide *modular formal proofs* for all the presented primitives in Coq [Krebbers et al. 2017] using the Iris separation logic [Jung et al. 2018]. We formally specify the CQS operations and then demonstrate that they obey the semantics corresponding to the primitives we consider. One property we do not show formally is the FIFO order, which is notoriously difficult to approach in Iris but can be demonstrated through classical proofs. We emphasize the complexity of our formalization task, as only a few similar real-world implementations are formally verified [Chajed et al. 2021; Jung et al. 2017; Krishna et al. 2020; Krogh-Jespersen et al. 2016; Vindum and Birkedal 2021]. Our proofs for CQS span approximately 8000 lines of Coq code, often requiring non-trivial reasoning. Yet, the proofs are *modular*, so they can be employed as the basis for proving new synchronization primitives implemented on top of CQS, reducing formalization effort. Specifically, proving each higher-order CQS-based primitive presented in the paper on this basis takes only around 500 lines.

Evaluation. We integrated the CQS framework as part of the standard Kotlin Coroutines library and used it to implement several fundamental synchronization primitives. To validate performance, we implemented `CancellableQueueSynchronizer` on the JVM for native threads and compared it against the state-of-the-art `AbstractQueuedSynchronizer` framework in Java [Lea 2005], which aims to solve the same problem, and, to our knowledge, is the only practical abstraction that provides similarly general semantics. We present different versions of mutex and semaphore, barrier and count-down-latch primitives, and two versions of blocking pools. Our algorithms outperform existing implementations in almost all scenarios and are sometimes faster by orders of magnitude.

In particular, our semaphore implementation outperforms the standard Java solution, which is implemented via `AbstractQueuedSynchronizer` [Lea 2005], up to `4x` in the uncontended case where the number of threads does not exceed the number of permits, and up to `90x` in a highly-contended scenario. For the count-down-latch implementation, our solution shows up to `7x` speedup compared to the Java library, while the barrier synchronization is faster by up to `4x`. For blocking pools, which share a limited set of resources, our approach is faster than the Java library implementation by up to `150x`. In some cases, the *fair* synchronization primitives we present even outperform the *unfair* variants in the Java standard library. Finally, results show that the cancellation support of CQS is more efficient than the one of the `AbstractQueuedSynchronizer` framework. Our analysis shows that these improvements come mainly because from the superior scalability of our design.

2 BASIC CQS ALGORITHM

In this section, we describe the key ideas behind the `CancellableQueueSynchronizer` algorithm in an iterative fashion, using a simple non-abortable mutex construct as an example. We then focus on the complexities of supporting cancellation/abortability in the next section.

Thread Management. We manipulate threads to suspend and resume operations. While our main application is coroutines, we will use threads for illustration, as they may be more familiar to the readers. Listing 1 presents the API we use in the paper. We emphasize that our approach can be directly adapted to any concurrency model, such as coroutines, futures, or continuations. [1] Our implementations for Java native threads and Kotlin coroutines (Section 6) support this claim.

Our API assumes that the currently-running thread can be obtained by calling `currentThread()`, and suspended by invoking `park(..)`. While suspended, the thread can be aborted via `cancel()` call, becoming unable to resume. In that case, the `onCancel` lambda provided in `park(..)` is executed. If a thread is cancelled in an active state, the cancellation takes effect with the following `park(..)` invocation.

```
interface Thread {
 fun park(onCancel: lambda () -> Unit[2]): Any
 fun unpark(result: Any): Bool
 fun cancel() // invoked by user
}
fun currentThread(): Thread
```

Listing 1. Thread management API.

To resume a thread, the `unpark(result)` function should be called. It returns `true` if the resumption succeeds, so the corresponding `park(..)` invocation completes with the specified result. Otherwise, if the thread is already cancelled, `unpark(result)` returns `false`. Notably, `unpark(result)` can be called before `park(..)` – in this case, the following `park(..)` invocation immediately completes without suspension, returning the provided result.

Environment. For simplicity, we assume the sequentially-consistent memory model, which matches our implementation, as all real-world weak memory models provide sequential consistency for data-race-free programs. In addition to plain reads and writes, we use atomic `Compare-and-Swap (CAS)`, `Get-and-Set`, and `Fetch-and-Add (FAA)` instructions, which are available in all modern programming languages. We also assume that the runtime environment supports garbage collection (GC). Reclamation techniques such as hazard pointers [M. Michael 2004] or hazard eras [Ramalhete and Correia 2017] can be used in environments without GC.

[1] Many languages support asynchronous programming either explicitly via `Future`-s, or implicitly via the `async`/`await` construct that internally manipulates continuation objects.

[2] `Unit` is a type with only one value: the `Unit` object. This type corresponds to the `void` type in Java.

High-Level Algorithm Overview. At the logical level, the CancellableQueueSynchronizer maintains a first-in-first-out (FIFO) queue of waiting requests and provides two main functions:

- suspend(): T, which adds the current thread as a waiter into the queue and suspends, and
- resume(result: T): Bool, which tries to retrieve and resume the next waiter, passing the specified value of type T.

A key advantage is that the framework allows to invoke resume(..) before suspend() as long as it is known that suspend() will happen eventually, so synchronization primitive implementations can allow such races. In Section 4, we present several algorithms that leverage this property for better performance and simplicity.

```
 1 val cells = InfiniteArray()
 2 var suspendIdx: Int64 = 0
 3 var resumeIdx:  Int64 = 0
 4
 5 fun suspend(): T {
 6   i := FAA(&suspendIdx, +1)
 7   // Try to suspend in cells[i].
 8   t := currentThread()
 9   if CAS(&cells[i], null, t):
10     return park() // enqueued, suspend
11   // Read the result and finish.
12   result := cells[i]; cells[i] = TAKEN
13   return result
14 }
```

```
15 fun resume(result: T) {
16   i := FAA(&resumeIdx, +1)
17   t := cells[i]
18   if t == null: // is the cell empty?
19     // `suspend()` is coming, try to
20     // install the result and finish.
21     if CAS(&cells[i], null, result):
22       return
23     // The cell stores a thread.
24     t = cells[i]
25   // Resume the waiting request.
26   cells[i] = RESUMED
27   t.unpark(result) // t is Thread
28 }
```

Listing 2. High-level CQS implementation on top of an infinite array without abortability support.

A useful mental image of CQS is that of an infinite array supplied with two counters: one that references the cell in which the new waiter should be enqueued as part of the next suspend() call, and one that references the next cell for resume(..). The intuition is that suspend() atomically increments its counter via Fetch-and-Add, stores the currently running thread in the corresponding cell, and suspends. Likewise, resume(..) increments its counter, visits the corresponding cell, and resumes the stored thread with the specified value. However, if resume(..) comes before suspend(), it simply places the value in the cell and finishes — suspend() grabs the value later and completes without an actual suspension. [3]

Listing 2 provides a high-level pseudocode for this simplified CancellableQueueSynchronizer, without abortability support. An infinite array cells (line 1) stores waiting threads and values inserted by racing resumptions. Counters suspendIdx and resumeIdx (lines 2–3) reference cells for the next suspend() and resume(..) operations.

When suspend() starts, it first gets its index and increments the counter atomically via Fetch-And-Add (FAA), which returns the value right before the increment (line 6). Next, it obtains the currently running thread to be inserted into the cell (line 8) and tries to do so via Compare-And-Swap (CAS) (line 9). If this CAS succeeds, the operation parks the thread, finishing when resumed (line 10). Otherwise, a concurrent resume(..) has already visited the cell — thus, suspend() extracts the placed value, cleans the cell by placing a special TAKEN token (line 12), and returns the extracted value (line 13). Note that in the mutex implementation, we always pass Unit through CQS; other data structures, such as blocking pools discussed in Section 4.4, may pass different values.

[3] The suspend() and resume(..) race behavior is similar to the thread parking mechanism in both our API and Java, where unpark(..) followed by park() results in the latter operation returning immediately.

```
1  val cqs = CQS<Unit>()
2  var state: Int = 1 // "unlocked" intially
3  fun lock() {
4    s := FAA(&state, -1)
5    if s > 0: return // was the lock acquired?
6    cqs.suspend() // suspend otherwise
7  }
8  fun unlock() {
9    s := FAA(&state, +1)
10   // Resume the first waiting
11   // request if there is one.
12   if s < 0: cqs.resume(Unit)
13 }
```

Listing 3. Basic mutex algorithm without abortability support using the CQS framework.

Symmetrically, `resume(..)` increments `resumeIdx` first (line 16). It then checks whether the cell is empty (line 18), in which case it tries to place the resumption value directly into the cell (line 21). If the attempt fails, a waiter is already stored in the cell, so the algorithm re-reads it (line 24). After the waiter is extracted, the operation stores a special `RESUMED` token in the cell to avoid memory leaks and resumes the extracted thread (lines 26–27).

Mutex on Top of CQS. To illustrate how primitives should use CQS, consider the simple mutex implementation from Listing 3. The rough idea is to maintain a `state` field (line 2) that stores 1 if the mutex is unlocked, and $w \leq 0$ if the mutex is locked. In the latter case, the negated value of w is the number of waiters on this mutex.

Initially, the mutex is unlocked and its `state` equals 1. When a `lock()` operation arrives, it atomically decrements the state, setting it to 0 (line 4), so the logical state becomes "locked". Since the previous logical state was "unlocked", the operation completes immediately (line 5). However, if another `lock()` arrives after that, it changes `state` to -1, keeping the logical state as "locked" and incrementing the number of waiters. Since the mutex was already locked, this invocation suspends via CQS (line 6). Likewise, `unlock()` increments `state`, either making the mutex "unlocked" if the counter was 0, or decrementing the number of waiters (line 9). In the latter case, `unlock()` resumes the first waiter via CQS (line 12). It is worth emphasizing that `lock()` and `unlock()` contain only five lines of easy-to-follow code in total.

Non-Blocking Operations. Synchronization primitives typically provide non-blocking variants of operations, such as the `tryLock()` sibling of `Mutex.lock()`, which succeed only when the operation does not require suspension. However, supporting them becomes non-trivial when `resume(..)` comes before `suspend()`, so the data (e.g., the lock permit) is stored in CQS and cannot be extracted without suspension; thus, the non-blocking sibling cannot access it.

To solve the problem, we introduce a special *synchronous* resumption mode, so that `resume(..)` always makes a rendezvous with `suspend()` and does not leave the value in CQS, failing when this rendezvous cannot happen in bounded time. We view this as an extension to `CancellableQueueSynchronizer` and present it in the full version of the paper [Koval et al. 2023b].

Infinite Array Implementation. The last building block of the basic CQS implementation is the emulation of an infinite array. Since all cells are processed in sequential order, the algorithm only requires having access to the cells between `resumeIdx` and `suspendIdx` and does not need to store an infinite number of cells. We follow the approach behind the channels implementation in Kotlin [Koval et al. 2023a], maintaining a linked list of cell segments, each containing a fixed number of cells, as illustrated in Figure 1.

Each segment has a unique id and can be seen as a node in a Michael-Scott queue [Michael and Scott 1996]. Following this structure, we maintain only those cells that are in the current active range (between

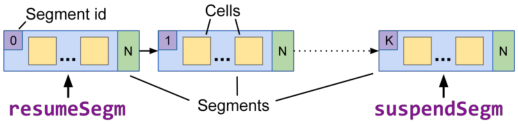

Fig. 1. An infinite array as a linked list of cell segments.

`resumeIdx` and `suspendIdx`) and access them similarly to an array. Specifically, we change the current working segment after completing operations equal to the number of cells in each segment.

Despite conceptual simplicity, the implementation of this structure is non-trivial, as shown in [Koval et al. 2023a]. We discuss the implementation and the required changes to the CQS algorithm in the full version of the paper [Koval et al. 2023b].

3 CANCELLATION SUPPORT

In this section, we extend the basic construct above with cancellation support. We assume that threads can be aborted via `Thread.cancel()` call, which bounds the following `unpark(..)` to fail. Additionally, the `onCancel` *cancellation handler* provided in the `park(..)` call[4] is invoked when the thread aborts. We will use this functionality later in this section.

We support two cancellation modes: *simple* and *smart*. Intuitively, the difference is that in the *simple* cancellation mode, `resume(..)` fails if the thread in the corresponding cell has been cancelled, whereas the *smart* cancellation enables efficient skipping a sequence of aborted requests.

3.1 Simple Cancellation

The *simple* cancellation mode is relatively straightforward — when a waiter becomes cancelled, the `resume(..)` operation that processes this cell is bound to fail. Thus, the code for `resume(..)` in Listing 2 should return `true` if `t.unpark(result)` at line 27 succeeds, and `false` on failure, indicating that the thread has already been aborted. Figure 2 shows the corresponding cell life-cycle.

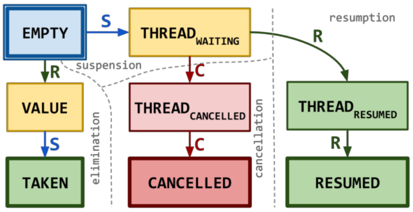

Fig. 2. Cell life-cycle with *simple* cancellation. When a thread becomes cancelled by a successful `Thread.cancel()` invocation (tagged with "C"), its content changes to `CANCELLED` to avoid memory leaks, and the corresponding `resume(..)` fails. The edges marked with S and R correspond to the transitions by `suspend()` and `resume(..)`.

An important technical detail is that aborted threads should be physically removed from the waiting queue to allow the garbage collector to reclaim the related memory. Thus, we specify a *cancellation handler* that replaces the aborted `Thread` with a special `CANCELLED` marker, according to the diagram in Figure 2. In addition, we must remove segments full of cancelled cells from the linked list to avoid memory leaks; we discuss the corresponding part of the algorithm in the full version of the paper [Koval et al. 2023b].

Mutex with Simple Cancellation. Please recall the mutex algorithm presented in Listing 3. With simple cancellation, `resume(..)` fails when the resuming thread is already aborted (THREAD$_{\text{CANCELLED}}$ or CANCELLED state). In this case, the corresponding `lock()` request is no longer valid, and the `unlock()` invocation, which performs `resume(..)` on this cell, incremented the counter (which must have been decremented by a `lock()` operation earlier). Thus, the balance is met, and `unlock()` should restart.

Limitations. One issue with the cancellation logic above is that it requires the `resume(..)` operation to process all the cancelled cells. Consider N `lock()` operations which execute and then immediately abort — the following call to `unlock()` increments `state` and unsuccessfully invokes `resume(..)` exactly N times. This leads to $\Theta(N)$ complexity, which is, nevertheless, amortized by `Thread.cancel()` invocations. Ideally, however, `unlock()` should require $O(1)$ time under no contention and should not "pay" for the cancelled requests.

[4] Since in practice we manipulate threads or coroutines, cancellation should be handled via an existing mechanism. In Java, for example, aborted threads throw `InterruptedException`, which can be caught and processed by the user. Moreover, some coroutines libraries, such as Kotlin Coroutines [kot 2022], already support an API similar to the one we use.

Another problem is that it is sometimes infeasible to wait until a *resume(..)* operation observes that the waiter is cancelled. We often wish to immediately learn about a waiter being cancelled and change the state correspondingly. As an example, consider a readers-writer lock and the following execution: (1) a reader comes and takes a lock, (2) a writer arrives and suspends, (3) then, another reader arrives and also suspends, because it should take a lock after the suspended writer. After that, (4) the suspended writer becomes cancelled, so the second reader should be resumed and take the lock. However, with simple cancellation, the effect of cancellation is postponed until another operation tries to resume the cancelled waiter, so the reader does not wake up. Making cancellations take effect immediately is critical in this context.

3.2 Smart Cancellation

A better option would be to skip cancelled waiters in `resume(..)` and install a cancellation handler that de-registers the operation when it aborts. For mutex, this could be incrementing the `state` field. However, a naive approach where `resume(..)` simply skips aborted threads would be incorrect.

The Problem. Figure 3 illustrates a potential problematic execution with such a mutex. Assume it is initially locked and two threads start. The first thread invokes `lock()`, placing itself in the CQS, and immediately aborts; however, the `state` is not incremented back yet. After that, the second thread calls `unlock()`, which increments the `state` counter (so it becomes 0) and intends to wake up a waiting `lock()` operation. The corresponding `resume(..)` sees the first cell in CANCELLED state and places its value in the next empty cell. The execution switches back to the first thread, and the cancellation handler of the aborted `lock()` increases the counter to 1. The resulting state is shown in the figure. Finally, two `lock()` calls by both threads are performed (they are under the dashed red line). One of them decrements `state` to 0 and enters the critical section; the other suspends via CQS and, observing the value in the cell, also proceeds to enter the critical section, thus breaking the mutex semantics.

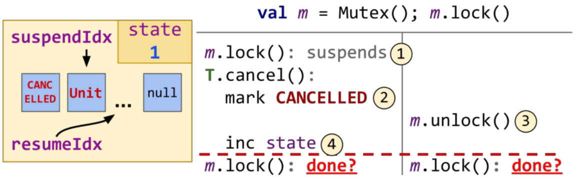

Fig. 3. An incorrect execution of mutex with naive cancellation strategy, where the cancellation handler increments `state` back, while `resume(..)` simply skips cancelled cells.

The `REFUSE` State. Notice that the naive version above would work fine in cases where the cancellation handler does not change the mutex state from "locked" to "unlocked" (thus, `state` stays non-positive). The problem occurs when the "last" waiter becomes cancelled, and a concurrent `resume(..)` tries to complete it. In this case, `resume(..)` must be informed that there is no longer any waiter in the `CancellableQueueSynchronizer` that could receive the value.

To signal this, a new REFUSE state is added to the cell life-cycle; see Figure 4 on the right for the updated cancellation part. This state signals that an operation attempted to abort, but determined that there is an upcoming `resume(..)` and the aborted waiter was the last one in the CQS. Thus, the `resume(..)` that inevitably visits the cell should be refused by CQS and will no longer attempt to pass the value to any waiter.

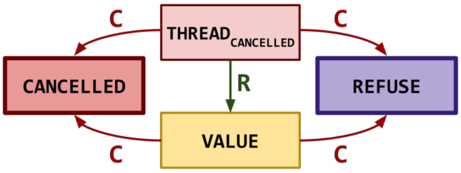

Fig. 4. Cell life-cycle for *smart* cancellation, with a special REFUSE state. The suspension, elimination, and resumption parts stay unchanged (see Figure 2).

Smart Cancellation API. Users who develop primitives on top of CQS with smart cancellation should implement onCancellation() and completeRefusedResume(value) functions, whose semantics are described in Listing 4. Specifically, when a waiter is cancelled (the cell state changes to $\mathtt{THREAD}_{\mathtt{CANCELLED}}$), the cancellation handler invokes onCancellation(), which tries to logically remove the waiter from the data structure. If the resume(..) operation that sees this cell can safely skip it and still match with another non-cancelled suspend(), the operation returns true, and the cell state becomes CANCELLED. Otherwise, when the cell state becomes REFUSE, and the corresponding resume(..) should be refused, onCancellation() should return false. This way, the refused resume(..) invokes completeRefusedResume(...) to complete the operation.

```
// Invoked on cancellation and returns
// `true` if the cell should enter to
// CANCELLED state and `false` when it
// should  transition to REFUSE.
fun onCancellation(): Bool
// Defines how to process a refused
// resume(..) with the specified value
fun completeRefusedResume(value: T)
```

Listing 4. Smart cancellation API.

Note that the behavior of resume(..) depends on whether the aborted thread moves the cell to CANCELLED or REFUSE state. However, if resume(..) observes $\mathtt{THREAD}_{\mathtt{CANCELLED}}$ state (the cell stores a Thread instance while unpark(..) fails), the expected behavior can not yet be predicted. We resolve this race by *delegating* the rest of the current resume(..) to the cancellation handler, replacing the thread instance with the resumption value — see the corresponding transition from $\mathtt{THREAD}_{\mathtt{CANCELLED}}$ to VALUE in Figure 4. After that, when the cancellation handler changes the cell's state to CANCELLED or REFUSE, it receives the value and completes the resumption correspondingly. In this case, the value passed to resume(..) can be out of the data structure for a while but is guaranteed to be processed eventually. Note that resume(..) never fails when using the smart cancellation mode.

Mutex with Smart Cancellation. Consider the mutex example again. Listing 5 presents the onCancellation() and completeRefusedResume(..) implementations for the basic algorithm from Listing 3; the rest stays the same.

When a lock() request aborts, the onCancellation() operation increments state (thus, decrementing the number of waiters). However, when the increment changes state to 1 (“unlocked”), the operation must return false to refuse the upcoming resume(..). After that, the resume(..) that comes to the cell sees it in REFUSE state and invokes completeRefusedResume(..). For the mutex, the lock is already successfully returned at the moment of incrementing state in onCancellation(), so this function does nothing. However, when CQS is used to transfer elements (see blocking pools in Subsection 4.4 as an example), the refused element should be returned back to the data structure via completeRefusedResume(..).

```
fun onCancellation(): Bool {
  // Increment the counter back.
  s := FAA(&state, +1)
  // s < 0: still in the "locked" state;
  // s = 0: the mutex has become "unlocked",
  //        refuse the upcoming resume(..).
  return s < 0
}
fun completeRefusedResume(permit: Unit) {
  // Do nothing, the mutex has already
  // been moved to the "unlocked" state.
}
```

Listing 5. Cancellation handling in the *smart* mode for the basic mutex algorithm from Listing 3.

The resume(..) Operation. Listing 6 presents a pseudocode for resume(..) that supports all cancellation modes and for the cancellation handler — the function that is invoked when Thread becomes cancelled; it is set in the park(..) invocation (see Listing 1). For simplicity, we assume

```
1  fun resume(value: T): Bool {
2    i := FAA(&resumeIdx, +1)
3    while (true): // modify the cell state
4      cur := cells[i]
5      when {
6        cur == null:
7          if CAS(&cells[i], null, value):
8            return true
9        cur is Thread:
10         if cur.unpark(value):
11           cells[i] = RESUMED
12           return true
13         // The thread is cancelled.
14         if cancellationMode == SIMPLE:
15           return false
16         // In smart cancellation, delegate
17         // this resume(..) completion
18         // to the cancellation handler.
19         if CAS(&cells[i], cur, value):
20           return true
21       cur == CANCELLED:
22         // Fail with simple cancellation.
23         if cancellationMode == SIMPLE:
24           return false
25         // Skip the cell in the smart mode
26         return resume(value)
27       cur == REFUSE:
28         completeRefusedResume(value)
29         return true
30     }
31 }

32 fun cancellationHandler(s: Segment,
33                         i: Int) {
34   // Which cancellation mode do we use?
35   if cancellationMode == SIMPLE:
36     // Mark the cell state to
37     // CANCELLED and finish.
38     s[i] = CANCELLED
39     s.onCancelledCell()
40     return
41   // Smart cancellation mode is used.
42   markCancelled := onCancellation()
43   if markCancelled:
44     // Mark the cell as CANCELLED.
45     old := GetAndSet(&s[i], CANCELLED)
46     // Did it store an aborted thread?
47     if old is Thread:
48       s.onCancelledCell()
49     else: // old is a value of type T
50       // A concurrent resume(..) has
51       // delegated its completion.
52       resume(old)
53   else:
54     // Move the cell state to REFUSE.
55     old := GetAndSet(&s[i], REFUSE)
56     // Did it store an aborted thread?
57     if old is Thread: return
58     // A concurrent resume(..) has
59     // delegated its completion;
60     // old is a value of type T.
61     completeRefusedResume(old)
62 }
```

Listing 6. Pseudocode for resume(..) that supports all cancellation modes and the corresponding cancellation handler. The suspend() implementation stays the same. The user-specified operations are highlighted in yellow. The onCancelledCell() operation, highlighted with green, informs the segment about a new cancelled cell — we have to remove segments full of cancelled cells to avoid memory leaks; the details are discussed in the full version of the paper [Koval et al. 2023b].

that CQS uses an infinite array in resume(..); the changes required for support of cancellation in its emulation are discussed in the full version of the paper [Koval et al. 2023b].

Like before, resume(..) increments resumeIdx first (line 2). After that, the corresponding cell should be modified — this logic is wrapped with a while(true) loop (lines 3–29); the current cell state is obtained in the beginning of it (line 4). When the cell is empty (line 6), resume(..) tries to set the resumption value to the cell (lines 7–8). If the corresponding CAS succeeds, this resume(..) finishes immediately. If the CAS fails, the cell modification procedure restarts.

When the cell stores a suspended thread (line 9), resume(..) tries to complete it (line 10). If successful, the cell value is cleared for garbage collection, and the operation finishes (lines 11–12). Otherwise, the thread has been cancelled. In the simple cancellation mode, resume(..) simply fails (lines 14–15). With the smart cancellation, resume(..) tries to replace the cancelled waiter with the resumption value, thus, delegating its completion to the cancellation handler, and finishes on success (line 19–20). On failure, one of the branches below will be entered.

When the cell is in `CANCELLED` state (line 21), `resume(..)` either fails in the simple cancellation mode (lines 23–24), or skips this cell in the smart one, invoking `resume(..)` one more time (line 26). In the full version of the paper [Koval et al. 2023b], we describe how to skip a sequence of `CANCELLED` cells in $O(1)$ under no contention, with the infinite array implemented as a linked list of segments.

In the remaining case, when the cell is in the `REFUSE` state (line 27), this `resume(..)` should be refused, and `completeRefusedResume(..)` is called (line 28). After that, the operation successfully finishes (line 29).

The Cancellation Handler. The cancellation handler can be specified as a parameter of the `park(..)` call (see Listing 1) and is invoked when the thread becomes aborted. Here, the `cancelationHandler(..)` function accepts the segment and the location index of the cell inside it — we know them at the point of invoking `park(..)`, so the handler has access to the cell and can update its state to `CANCELLED` or `REFUSE`.

In the first case, when the simple cancellation mode is used (lines 35–40), the cell state is always updated to `CANCELLED` and a special `onCancelledCell()` function is invoked on the segment (lines 38–39). This `onCancelledCell()` function signals that one more cell in this segment was cancelled and removes the segment if all the cells become cancelled (see the full version of the paper [Koval et al. 2023b] for details).

With smart cancellation, `onCancellation()` is invoked first (line 42). If it succeeds (returns `true`), then the cell state can be moved to `CANCELLED`. However, a concurrent `resume(..)` may come and replace the aborted thread with its resumption value, see the cell state diagram in Figure 4. Therefore, we put the `CANCELLED` token via an atomic `GetAndSet` operation, which returns the previous cell state (line 45). If a thread instance was stored in the cell (line 47), `resume(..)` has not come there: the handler signals about a new cancelled cell, removing the segment if needed (line 48), and finishes. Otherwise, if a resumption value was stored in the cell, the cancellation handler completes the corresponding resumption by invoking `resume(..)` with this value (line 52).

In case `onCancellation()` returns `false` (line 53), the matching `resume(..)` should be refused. Thus, the cell state moves to `REFUSE` via an atomic `GetAndSet` (line 55). If the cell stored the cancelled thread, the handler finishes (line 57). Otherwise, a concurrent `resume(..)` has replaced it with the resumption value — we complete it with `completeRefusedResume(..)` (line 61).

4 SYNCHRONIZATION PRIMITIVES ON TOP OF CQS

To show the expressiveness of the `CancellableQueueSynchronizer` framework, we present several algorithms developed on top of it. Starting with the barrier, we present a new count-down-latch algorithm, then several semaphore algorithms, and finish with blocking pools.

4.1 Barrier

A simple but popular synchronization abstraction is the *barrier*, which allows a set of parallel threads wait for each other at a common program point, via a provided `arrive()` operation.

Algorithm. Listing 7 on the right presents the algorithm on top of CQS. The implementation is straightforward: it maintains a counter of the parties who arrived (line 2) and increments it in the beginning of the `arrive()` operation (line 5). All but the last `arrive()` invocations suspend (line 6), while the latter one resumes all those who previously arrived (line 7).

```
1 val cqs = CQS<Unit>()
2 var remaining: Int = parties
3
4 fun arrive() {
5   r := FAA(&remaining, -1)
6   if r > 1: return cqs.suspend()
7   repeat(parties - 1) { cqs.resume(Unit) }
8 }
```

Listing 7. Barrier algorithm via CQS.

Once the last thread arrives, all the waiters should be resumed. However, if any of these waiters becomes cancelled, the barrier contract is violated — fewer waiters will be successfully resumed and overcome the barrier. Unfortunately, solving this problem would require an ability to *atomically* resume a set of waiters (so either all the waiters are resumed or none), but no real system provides such a primitive. Thus, similarly to the implementation in Java, we do not support cancellation. However, instead of breaking the barrier when a thread is cancelled, we ignore cancellation. The intuition behind this design is that even if a waiter has been cancelled, it has successfully reached the barrier point and should not block the other parties from continuing.

4.2 Count-Down-Latch

The next synchronization primitive we consider is the *count-down-latch*, which allows waiting until the specified number of operations are completed. It is initialized with a given count, and each `countDown()` invocation decrements the number of operations yet to be completed. Meanwhile, the `await()` operation suspends until the count reaches zero.

Basic Algorithm. The pseudocode of our count-down-latch implementation is presented in Listing 8. Essentially, the latch maintains two counters: `count`, representing the number of remaining operations (line 5), and `waiters`, which stores the number of pending `await()`-s (line 7).

```
1  val cqs = CQS<Unit>(
2    cancellationMode = SMART
3  )
4  // initialized by user
5  var count: Int = initCount
6  // the number of waiters
7  var waiters: Int = 0
8
9  fun countDown() {
10   r := FAA(&count, -1)
11   // Has the counter reached zero?
12   if r <= 1: resumeWaiters()
13 }
14
15 fun await() {
16   if count <= 0: return
17   w := FAA(&waiters,+1)
18   // Is DONE_BIT set?
19   if w & DONE_BIT != 0: return
20   // Suspend until count reaches zero
21   cqs.suspend()
22 }
23 fun resumeWaiters() = while(true) {
24   w := waiters
25   // Is DONE_BIT set?
26   if w & DONE_BIT != 0: return
27   // Set DONE_BIT and resume waiters.
28   if CAS(&waiters, w, w | DONE_BIT):
29     repeat(w) { cqs.resume(Unit) }
30     return
31 }
32
33 fun onCancellation(): Bool {
34   w := FAA(&waiters, -1)
35   // Move the cell to CANCELLED if the
36   // bit is unset; otherwise, to REFUSE.
37   return w & DONE_BIT == 0
38 }
39
40 fun completeRefusedResume(token: Unit) {
41   // Ignore cancelled await()-s.
42 }
43
44 const DONE_BIT = 1 << 31
```

Listing 8. Count-down-latch implementation on top of CQS with smart cancellation. When manipulating with DONE_BIT, we use bitwise "and", "or", and "left shift" operators, denoted as **&**, **|**, and **<<**, respectively.

The `countDown()` function is straightforward: it decrements the number of remaining operations (line 10), resuming the waiting `await()`-s if the count reached zero (line 12).[5] Meanwhile, `await()` checks whether the counter of remaining operations has already reached zero, immediately completing in this case (line 16). If `await()` observes that count is positive, it increments the number of waiters (line 17) and suspends (line 21).

[5] We allow the number of `countDown()` calls to be greater than the initially specified one. However, we could check in `countDown()` that the counter is still non-negative after the decrement, throwing an exception otherwise.

Given that `resumeWaiters()`, which is invoked by the last `countDown()`, can be executed concurrently with `await()`, they should synchronize. For this purpose, `resumeWaiters()` sets the `DONE_BIT` in the `waiters` counter (line 28), forbidding further suspensions and showing that this count-down-latch has reached zero. Thereby, `await()` checks for this `DONE_BIT` before suspension and completes immediately if the bit is set (line 19).

Cancellation. The simplest way to support cancellation is to do nothing: the algorithm already works with the simple cancellation mode, where `resume(..)`-s silently fail on cancelled `await()` requests (line 29). This strategy results in resuming cancelled waiters, which makes `resumeWaiters()` work in a linear time on the total number of `await()` invocations, including the aborted ones.

Smart cancellation, on the other hand, makes it possible to optimize `resumeWaiters()` so that the number of steps is bounded by the number of non-cancelled `await()`-s. The `onCancellation()` function is invoked when a waiter becomes cancelled. It attempts to decrement the number of waiters (line 34), making `resume(..)` skip the corresponding cell in the CQS. However, if the `DONE_BIT` is already set at the moment of the decrement, a concurrent `resumeWaiters()` is going to resume this cancelled waiter. The corresponding `resume(..)` call should be ignored, so `onCancellation()` returns `false`, while `completeRefusedResume(..)` does nothing (lines 40–42).

4.3 Semaphores

The barrier and count-down latch algorithms described above do not actually require waiting requests to be resumed in FIFO order. However, this property is critical for some primitives such as the mutex or the semaphore. While the mutex allows at most one thread to be in the critical section protected by `lock()` and `unlock()` invocations, the semaphore is a generalization of mutex that allows the specified number of threads to be the critical section simultaneously by taking a permit via `acquire()` and returning it back via `release()`.

In fact, the semaphore algorithm is almost the same as the one for the mutex, presented under the CQS framework presentation in Listing 3 (the basic version) and Listing 5 (the cancellation part). The only difference is that the `state` field is initialized with K instead of 1, when K is the number of threads allowed to be in the critical section concurrently. We present the implementation details in the full version of the paper [Koval et al. 2023b].

4.4 Blocking Pools

While the previous algorithms use `CancellableQueueSynchronizer` only for synchronization, it is also possible to develop *communication* primitives on top of it. Here, we discuss two blocking pool implementations. When using expensive resources such as database connections or sockets, it is common to reuse them — this usually requires an efficient and accessible mechanism. The *blocking pool* abstraction maintains a set of elements that can be retrieved in order to process some operation, after which the element is placed back in the pool:

- `take()` either retrieves one of the elements (in an unspecified order), suspending until an element appears if the pool is empty;
- `put(element)` either resumes the first waiting `take()` operation and passes the element to it, or puts the element into the pool.

Intuitively, the blocking pool contract reminds the semaphore one. Similarly to the semaphore, it transfers resources, with the only difference being that the semaphore shares logical non-distinguishable permits while blocking pool works with real elements. The rest is almost the same. In the full version of the paper [Koval et al. 2023b], we present two pool implementations: queue-based and stack-based. Intuitively, the queue-based implementation is faster since queues can be built on segments similar to CQS and leverage `Fetch-And-Add` on the contended path. In contrast, the stack-based pool retrieves the last inserted, thus the “hottest” element. Please note that both algorithms we discuss are *not* linearizable and can retrieve elements out-of-order under some races. However, since pools do not guarantee that the stored elements are ordered, these queue- and stack-based versions should be considered bags with specific heuristics; these semantics matches practical applications.

5 CORRECTNESS AND PROGRESS GUARANTEES

In this section, we discuss correctness and progress guarantees for both CQS operations and the primitives we built on top of the framework.

5.1 Formal Proofs of Correctness in Iris/Coq

Correctness is formally proven in the state-of-the-art concurrent higher-order separation logic Iris [Jung et al. 2018] using its Coq formalization [Krebbers et al. 2017]. Here, we highlight the key ideas behind the proofs and discuss their limitations. **The source code of the proofs is available in [pro 2023]. We complement this with a detailed outline in the full version of the paper [Koval et al. 2023b].**

Operation Specifications. Iris is a framework designed for reasoning about the safety of concurrent programs, and several non-trivial algorithms have already been formally proved using it [Carbonneaux et al. 2022; Chajed et al. 2021; Jung et al. 2017; Krishna et al. 2020; Krogh-Jespersen et al. 2016; Vindum and Birkedal 2021; Vindum et al. 2022]. When constructing formal proofs, one should provide a specification for each of the data structure operations. In the Iris logic, operations manipulate *resources*, which are pieces of knowledge about the system-wide state and can be held by threads or the data structure itself. These resources are *logical* and do not affect the program execution. A specification describes which resources are required for the operation to start and how they change when it finishes.

Consider again the mutex as an example. Intuitively, we parameterize it with a resource R, which serves as an exclusive right to be in the critical section and, thus, to invoke `unlock()`. Initially, this resource R is held by the mutex object. The specification ensures that:

(1) when the `lock()` operation finishes, the resource R is transferred to the caller thread, and
(2) when `unlock()` starts, the corresponding thread must provide R.

If the resource R is unique, the specification still holds; however, as it cannot be held by multiple threads by construction, the mutual exclusion contract is satisfied.

All our specifications are defined in a similar manner. For example, to specify the semaphore contract, we simply need to maintain K non-distinguishable copies of R; thus, allowing at most K threads to enter the critical section. However, the actual specifications in Coq contain many additional details, mainly due to support for cancellation semantics. Please refer to the proofs outline in the full version of the paper [Koval et al. 2023b] and the source code [pro 2023] for details.

Modularity. Our Iris proofs are *modular*: specifications treat each operation separately and do not concern the state of the system as a whole, locally manipulating logical resources instead. As a result, the proof of CQS itself spans 8000 lines of Coq; by comparison, the proof of the barrier, including its definition, takes only 400 lines, the semaphore proof requires less than 300 lines, and the proofs for the count-down-latch and blocking pools take up to 700 lines each. Modularity dramatically reduces the effort for someone wishing to formally verify their CQS-based primitive.

Limitations. One main limitation is that the existing formal specifications do not highlight the FIFO semantics, allowing the waiting operations to complete in any order. Instead, these specifications verify high-level properties, such as "at most one thread can be in the critical section" for the mutex. This limitation stems from the modularity of proofs and the fact that the user code parameterizes the cancellation handler in CQS. The fairness of end-to-end structures on top of the CQS is easy to see by the analogy with the state-of-the-art linearizable queues [Morrison and Afek 2013; Yang and Mellor-Crummey 2016], but proofs of such form are not modular. While the modular Iris proofs are powerful enough to show fairness, this requires significant effort even for simple data structures such as the classic Michael-Scott queue [Vindum and Birkedal 2021], and constructing them for non-trivial and, especially, higher-order structures like CQS is currently impractical. Most importantly, a modular proof of fairness of structures on top of the CQS would require placing highly involved contracts on the cancellation handler as well as the uses of `suspend` operations that may interact with it, making it significantly more difficult to prove the correctness of primitives on top of the CQS for the end user.

Another limitation of the provided Iris specifications is that they do not assert the lack of memory leaks. In particular, they do not prevent us from always storing the whole infinite array. Nevertheless, the lack of memory leaks follows by construction, as we always physically remove segments full of canceled cells. Beyond that, we have thoroughly tested our implementation for the absence of memory leaks via the Lincheck framework [Koval et al. 2020], which enables model checking of concurrent algorithms on the JVM.

Finally, we assume a strong sequentially-consistent memory model. We find this assumption reasonable as almost all the operations that manipulate shared data are atomic in the presented algorithms, while considering relaxed memory may significantly increase the proofs complexity [Dang et al. 2019; Kaiser et al. 2017]. We also rely on the SC-DRF (sequential consistency for data-race-free programs) property of all real-world weak memory models, such as C++11 and JMM, which makes reasoning in the strong memory model sufficient. However, we plan to extend our proofs to support the release-acquire semantics [Kaiser et al. 2017] and, thus, match the LLVM memory model for languages such as C/C++ and Rust.

5.2 Progress Guarantees

Similarly to the dual data structures formalism [Scherer III et al. 2006], we reason about progress independently of whether the operation was suspended. When we say that some blocking operation is lock- or wait-free, we mean that it performs all the synchronization with this progress guarantee, either completing immediately or adding itself to the queue of waiters followed by suspension.

Unfortunately, the progress guarantees cannot be mechanized in our Iris proofs. The reason for this is that, at the time of writing, there are two forms of specifying program behavior in Iris. The first way is to use *(partial) weakest preconditions*, which do not ensure that an operation terminates. In fact, an infinite loop satisfies any such specification. The second less popular form uses the *total weakest precondition* [Jung et al. 2018], which requires that every operation must terminate in a bounded number of steps. This type of specification can be used to show wait-freedom of algorithms, but is not applicable to our case, as some of the operations guarantee only lock-freedom.

We do not consider the lack of formal proof of progress guarantees a major issue. Although it is possible to write such proofs (see [Jia et al. 2015] for a comprehensive analysis), we find it much easier to discuss this question separately. In essence, most of the presented primitives including the CQS framework itself guarantee wait-freedom under no cancellation and at least lock-freedom when requests may abort. We provide the detailed analysis in the full version of the paper [Koval et al. 2023b].

6 EVALUATION

Our main practical contribution is integrating CQS, along with the mutex and semaphore implementations, into the standard Kotlin Coroutines library [kot 2022]. Other presented synchronization and communication primitives are implemented in tests, enabling their fast development when needed.

To validate performance, we implemented `CancellableQueueSynchronizer` on the JVM and compared it against the state-of-the-art `AbstractQueuedSynchronizer` framework for implementing synchronization primitives in Java [Lea 2005]. The latter provides similar semantics to CQS, and is the only practical framework that addresses the same general problem. Notably, CQS-based algorithms are significantly more straightforward to reason. For fair performance evaluation, we use threads as waiters in CQS; it should benefit the Java implementation, which is well-optimized for this case.

Our implementations for coroutines in Kotlin and native threads in Java confirm the flexibility of our design, let alone matching the real-world semantics.

Experimental Setup. Experiments were run on a server with 4 Intel Xeon Gold 6150 (Skylake) sockets; each socket has 18 2.70 GHz cores, each of which multiplexes 2 hardware threads, for a total of 144 hardware threads. We used `OpenJDK 15` in all the experiments and the `Java Microbenchmark Harness (JMH)` library [jmh 2021] for running benchmarks. When measuring operations, we also add some uncontended work after each operation invocation — the work size is geometrically distributed with a fixed mean, which we vary in benchmarks. In our CQS implementation, we have chosen the segment size of `64` based on minimal tuning.

6.1 Barrier

We compare the CQS-based barrier implementation with the standard one in Java. In addition, we add a baseline counter-based solution, which is organized in the same way as ours, but performs active waiting instead of suspension, spinning in a loop until the `remaining` counter becomes zero.

Benchmark. Each of the threads performs barrier point synchronizations followed by some uncontended work. This process is repeated a fixed number of times. We measure a single synchronization phase, a set of `arrive()`-s with additional work for each thread. Without any synchronization, the execution time is expected to stay the same independently of the number of threads.

Results. The experimental results are presented in Figure 5. We evaluated all three algorithms on various numbers of threads and with three average work sizes — `100`, `1000`, and `10000` uncontended loop iterations on average. The graphs show an average time per operation, so lower is better.

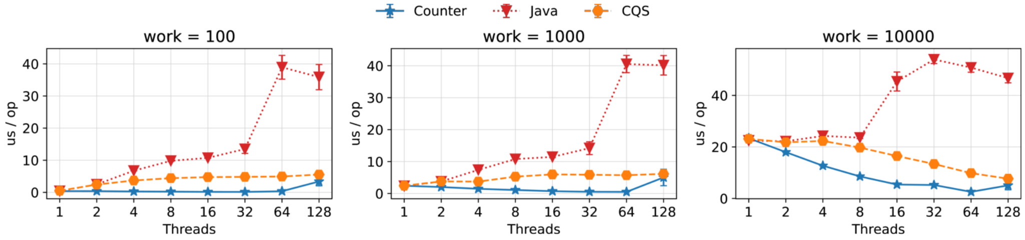

Fig. 5. Evaluation of `CancellableQueueSynchronizer`-based barrier implementation against the standard one available in Java and a simple counter-based solution with active waiting. The plots show average time per synchronization phase, **lower is better**.

Since one of the main synchronization costs is thread suspensions and resumptions, the counter-based solution with active waiting is predictably the fastest. Nevertheless, our CQS-based algorithm shows similar performance trends, due to all the operations being based on `Fetch-And-Add`. In contrast, the solution from the standard concurrency library in Java shows significantly less scalability—we find the reason for such performance degradation in using a mutex under the hood; surprisingly, it does not use `AbstractQueuedSynchronizer` directly. As a result, we find our simple CQS-based algorithm to provide superior performance.

6.2 Count-Down-Latch

Next, we evaluate our count-down latch implementation against the one in Java's concurrency package, which is built on top of the `AbstractQueuedSynchronizer` framework.

Benchmark. We consider a workload with a fixed number of `countDown()` invocations distributed among threads, each followed by additional uncontended work. Besides, we add a baseline that does not invoke `countDown()` and only performs the work. Thus, comparing with this baseline we can measure the overhead caused by the count-down-latch synchronization.

Results. Figure 6 shows the evaluation results with different additional work sizes (50 uncontended loop iterations on average on the left, 100 in the middle, and 200 on the right). It is apparent that the CQS implementation significantly outperforms the standard one from Java, by up to 4×. Compared to the baseline, it follows the same trend, providing an extremely small overhead on the right graph, where the work is 200 uncontended loop cycles.

Similar to our CQS-based algorithm, the implementation in Java maintains a counter of remaining `countDown()` invocations. However, they update this counter in a CAS loop: the algorithm reads the current counter value and tries to replace it with the reduced by one via CAS, restarting the process on failure. We find this difference the main reason for the superior scalability of our solution.

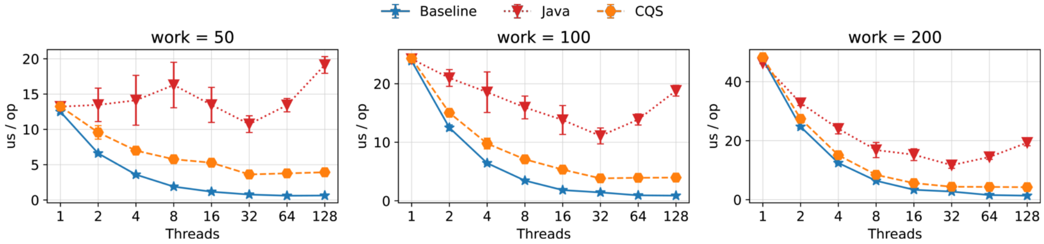

Fig. 6. Evaluation of the CQS-based count-down-latch implementation against the standard one in Java. The baseline illustrates how the operation time would change if `coundDown()` took no time. **Lower is better.**

6.3 Mutex and Semaphores

Since the semaphore is a generalization of the mutex, we equate its implementation with $K = 1$ permits as mutual exclusion. We compare our algorithm against alternatives from the standard Java library, unfair implementations of mutex and semaphore in Java, and the state-of-the-art fair CLH and MCS lock algorithms. In Section 2 we also mention that implementing non-blocking `Mutex.tryLock()` and `Semaphore.tryAcquire()` operations would require extending CQS with a special *synchronous* resumption mode, leaving the details to the full version of the paper [Koval et al. 2023b]. We included both semaphore implementations in the experiment to show that the complexity introduced by this `synchronous` resumption mode does not affect performance.

Benchmark. Consider the workload of many operations to be executed by the specified number of threads with the parallelism level restricted via semaphore. Thus, each operation invocation is wrapped with the `acquire()`-`release()` pair. When the parallelism level equals 1, the semaphore is de facto a mutex, so we can compare our semaphore against other mutex algorithms. As before, the operations are simulated with uncontended geometrically distributed work. In addition, we perform some work before acquiring a permit, thus, simulating a preparation phase for the operation guarded by the semaphore. We used `100` uncontended loop iterations on average for both pieces of work; the results for other work sizes do not differ significantly and, therefore, are omitted.

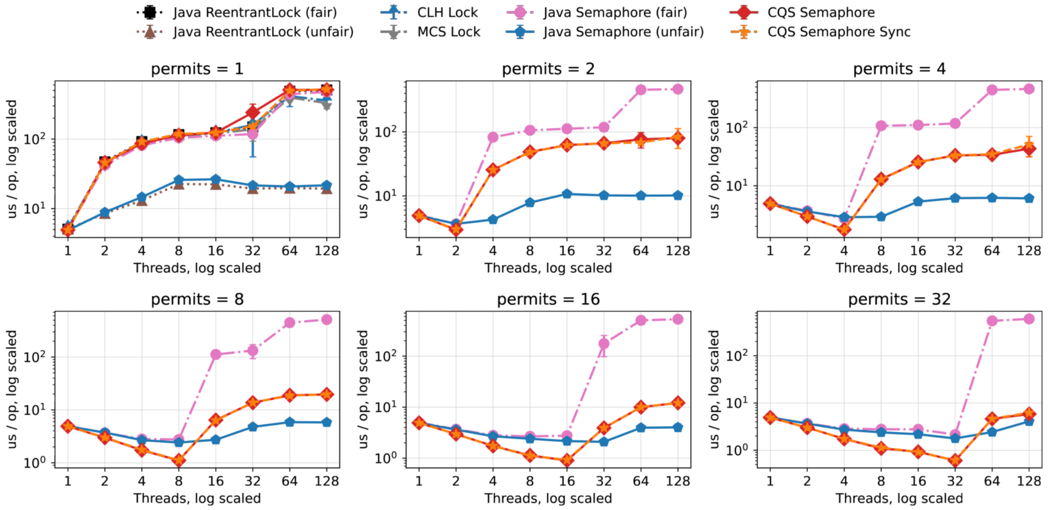

Fig. 7. Evaluation of CQS-based semaphore implementations compared to the standard ones in Java, including the unfair variants. In addition, we compare our semaphores against the standard fair and unfair lock implementations in Java and the classic CLH and MCS fair mutexes. **Lower is better.**

Results. The results against both fair and unfair versions of the standard ReentrantLock and Semaphore primitives in Java, as well as against the classic CLH [Magnusson et al. 1994] and MCS [Mellor-Crummey and Scott 1991] fair locks, are shown in Figure 7. Our semaphore implementation with the *synchronous* resumption mode in CQS, which enables tryAcquire() implementation, is denoted with the suffix «Sync».

In the mutex scenario, all the fair algorithms show the same performance, while Java's unfair mutex and semaphore are predictably faster, as unfairness significantly reduces context switches under high contention.

In the semaphore scenario, our solution outperforms both fair and unfair Java implementations by up to 4x when the number of threads does not exceed the number of permits (so suspensions do not happen). The main reason is that our solution leverages Fetch-and-Add to update the number of available permits, which can be negative, indicating the number of waiters. In contrast, the implementation in Java must ensure that this number stays non-negative, so it has to perform this update in a CAS loop, reading the current number of available permits, trying to decrement it via CAS if there is a permit to acquire, and restarting if the CAS fails.

With the increase in the number of threads, our algorithm is almost on par with the unfair version when the number of permits ≥16, and significantly outperforms the fair one in all scenarios. In particular, our semaphore implementation outperforms the standard fair algorithm in Java by up to 90x in a highly-contended scenario. More scalable queue design behind CQS is the key to achieving such an outstanding performance. Notably, the complexity introduced by the synchronous resumption is negligible and does not affect results.

6.4 Blocking Pools

We implemented both queue- and stack-based pools and compared them against the existing ArrayBlockingQueue (both fair and unfair) and LinkedBlockingQueue collections from the standard Java library. Notably, they do not leverage the AbstractQueuedSynchronizer framework, as it serves only for synchronization, while CQS enables communication out-of-the-box. Relatively, their solutions provide linearizability, while our pools may be non-linearizable when threads abort. This experiment considers all data structures as solutions for pools of shared resources.

Benchmark. We use the same benchmark as for semaphores. In essence, we run many operations on the specified number of threads with a shared pool of elements. Each operation performs some work (`100` uncontended loop iterations on average in our experiment) first, then takes an element, performs some other work with this element (`100` more loop iterations on average in our experiment), and returns it to the pool at the end. The results with other work amounts are omitted but were examined and do not differ significantly.

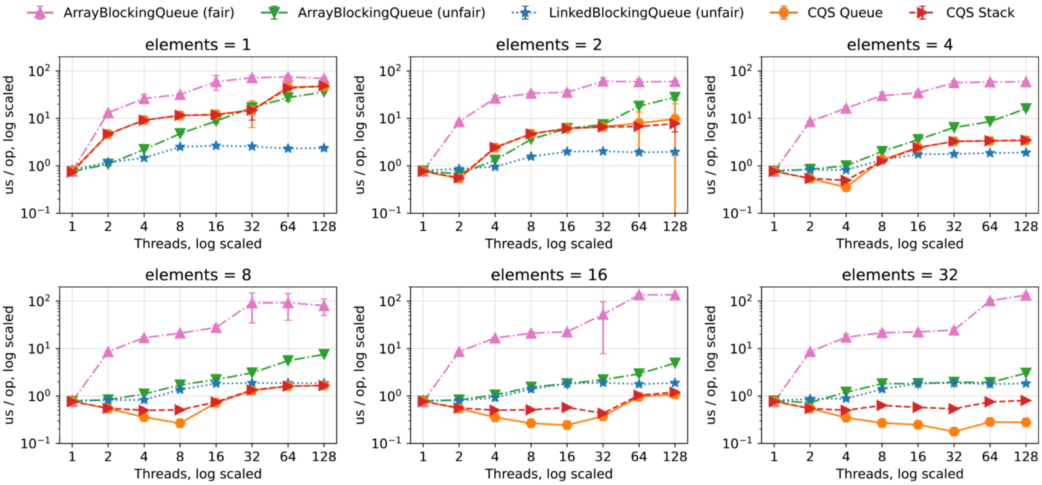

Fig. 8. Evaluation of the presented queue- and stack-based pool algorithms with various numbers of shared elements against the existing `ArrayBlockingQueue` (both fair and unfair versions) and `LinkedBlockingQueue` collections from the standard Java library. **Lower is better.**

Results. Figure 8 shows results with different numbers of elements shared in the pool. First, our queue-based version shows better results on larger numbers of elements, which is expected as the queue perform a `FAA` on the contended path instead of `CAS` in the stack-based solution; the latter often fails under high contention, resulting in the operation restart. Compared to the fair `ArrayBlockingQueue`, both of our implementations are more performant by up to `100`× times. The synchronization behind `ArrayBlockingQueue` uses coarse-grained locking, while our solution is non-blocking for storing elements and managing the queue of waiting requests.

The *unfair* `LinkedBlockingQueue` is more scalable than the *unfair* version of `ArrayBlockingQueue`, and they slightly outperform our *fair* implementations on a large number of threads with a small number of shared elements, which is when our solutions suspend a lot. However, both our solutions consistently outperform these unfair primitives by up to `10`× times when at least `8` elements are shared, showing the same or better performance when the number of threads does not exceed the number of elements.

6.5 Abortability Support

Up to this point, we have primarily focused on situations where suspended requests do not get aborted. While cancellation performance might not always be crucial, as it typically occurs due to a more resource-intensive coroutine or thread interruption, removing aborted waiters from the queue in constant time remains essential. This is particularly true for coroutines, where thousands may be waiting on a mutex or semaphore. The CQS framework fulfills this need by physically removing aborted threads in $O(1)$ under no contention. In contrast, Java's `AbstractQueuedSynchronizer` takes linear time in the queue size to remove an interrupted thread.

To assess the practical impact of constant-time cancellation, we conducted a benchmark comparison between CQS (with both `SIMPLE` and `SMART` cancellation modes) and `AbstractQueuedSynchronizer`. Initially, we populate the synchronization framework with a specified number of suspended threads. After that, we measure the time required to suspend and instantly abort, without parking the native thread. Based on the initial queue size, we anticipate that the performance of Java's solution will degrade, while CQS should consistently deliver the same level of performance. Figure 9 displays the results, which confirm our hypothesis. In particular, CQS outperforms Java's solution by `1.9x` on an empty queue, and by `65x` when the queue contains 1000 waiters.

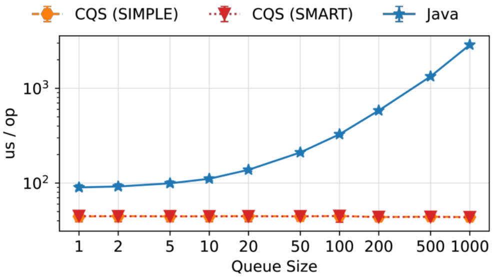

Fig. 9. Comparison of CQS' and Java's `AbstractQueuedSynchronizer` cancellation mechanisms. **Lower is better.**

Regrettably, it is not feasible to compare the cancellation mechanisms of CQS and `AbstractQueuedSynchronizer` under concurrent conditions at the time of writing the paper. The reason is that the latter gets into a deadlock in its internal `cleanQueue()` function.

7 RELATED WORK

Our work is part of a wider effort of formalizing and implementing expressive, safe, and efficient support for asynchronous operations in modern programming languages [Bierman et al. 2012; Cutner and Yoshida 2021; Haller and Miller 2019; Okur et al. 2014; Prokopec and Liu 2018]. In this context, we provide contributions at the level of algorithms, semantics, and formal proofs, with Kotlin/JVM as a practical application. Specifically, we perform one of the first thorough explorations of how fairness and abortability semantics can be efficiently supported at the data structure level, and present one of the first formally-verified such designs for this type of data structure. Importantly, our approach enables high-performance implementations in a range of practical applications, and could serve as a basis for standard library implementations in modern languages.

We emphasize that few real-world implementations of similar complexity are formally verified [Chajed et al. 2021; Jung et al. 2017; Krishna et al. 2020; Krogh-Jespersen et al. 2016; Vindum and Birkedal 2021]. In line with prior work, we do not formally prove full linearizability, which is notoriously difficult to approach in Iris but can be demonstrated through classical proofs. There are successful linearizability proofs of data structures of comparable complexity using the approach of *contextual refinement* [Vindum and Birkedal 2021; Vindum et al. 2022] using the ReLoC proof framework. Iris itself permits making specifications *logically atomic*, which can also represent linearizability [authors 2022]. We find this approach much less applicable to ensuring linearizability of a *framework*, which depends heavily on the behavior of the code passed to it.

At the algorithmic level, our CQS implementation builds on ideas from both the classic Michael-Scott queue [Michael and Scott 1996] and the highly-efficient LCRQ queue design of Afek and Morrison [Morrison and Afek 2013]. The latter was also used by Izraelevitz and Scott [Izraelevitz and Scott 2017] to build blocking synchronous queues, and by Koval et al. [Koval et al. 2019, 2023a] to build channels. Relative to these latter modern works, CQS supports much more general semantics, requiring significant changes to the design, in particular, to support cancellations. Specifically, CQS is general enough to provide full support for coroutines, while staying flexible and efficient, whereas these prior design focus on narrower applications, such as blocking queues.

To our knowledge, the only abstraction that provides similarly-general semantics is the `AbstractQueuedSynchronizer` in Java [Lea 2005], which CQS outperforms by a wide margin due to superior algorithmic design, complemented by formal proofs. More precisely, the `AbstractQueuedSynchronizer` framework combines the classic CLH [Magnusson et al. 1994] lock algorithm to maintain the queue of suspended requests with an integer counter, which represents the synchronization primitive state and is updated by `CAS` operations. In contrast, the CQS enables more efficient state updates via `Fetch-And-Add`-s, also maintaining the queue of waiters with `FAA`-s on the contended path; thus, providing a more scalable solution.

8 DISCUSSION

We have presented a new `CancellableQueueSynchronizer` framework enabling efficient implementations for a whole range of fundamental synchronization primitives in a fair and abortable manner. We observed that the interplay between fairness and cancellation semantics can raise subtle semantic and correctness questions. We found formalization extremely useful when identifying correctness issues in our implementation, notably w.r.t. cancellation semantics. A practical consequence of our work is efficient support for such primitives in the context of Kotlin Coroutines, which we show to generally outperform existing designs offering similar semantics in a wide range of scenarios. Specifically, our algorithms on top of CQS outperform existing Java implementations in almost all scenarios and can be faster by orders of magnitude. Surprisingly, the CQS-based primitives frequently surpass even the *unfair* versions of primitives from the standard Java library in our experiments, thanks to the superior scalability of our design.

We believe that CQS could serve as a basis for more complex semantics, designs, and primitives (e.g., fair readers-writer locks and synchronous queues), enabling efficient synchronization not only for Kotlin Coroutines but for other languages and platforms as well, such as C++, Rust, and Go. We plan to investigate this in future work, along with proof extensions to the release-acquire memory model semantics [Kaiser et al. 2017].

REFERENCES

2021. JMH - Java Microbenchmark Harness. https://openjdk.java.net/projects/code-tools/jmh/.

2022. Kotlin Coroutines. https://github.com/Kotlin/kotlin-coroutines.

2023. CQS Formal Proofs. https://github.com/Kotlin/kotlinx.coroutines/tree/cqs-proofs.

Adam Alon and Adam Morrison. 2018. Deterministic Abortable Mutual Exclusion with Sublogarithmic Adaptive RMR Complexity. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. 27–36.

Various authors. 2022. https://gitlab.mpi-sws.org/iris/examples/-/tree/0260d3d08e2f56bbccd44c3d56436baea30da4c9/theories/logatom.

Gavin Bierman, Claudio Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. 2012. Pause'n'Play: Formalizing Asynchronous C#. In *European Conference on Object-Oriented Programming*. Springer, 233–257.

Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W O'Hearn, and Francesco Zappa Nardelli. 2022. Applying Formal Verification to Microkernel IPC at Meta. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 116–129.

Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI). Virtual, 423–439.*

Zak Cutner and Nobuko Yoshida. 2021. Safe Session-Based Asynchronous Coordination in Rust. In *International Conference on Coordination Languages and Models*. Springer, 80–89.

Robert Danek and Wojciech Golab. 2010. Closing the Complexity Gap between FCFS Mutual Exclusion and Mutual Exclusion. *Distributed Computing* 23, 2 (2010), 87–111.

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. 2019. RustBelt Meets Relaxed Memory. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

Edsger W Dijkstra. 2001. Solution of a Problem in Concurrent Programming Control. In *Pioneers and Their Contributions to Software Engineering*. Springer, 289–294.

George Giakkoupis and Philipp Woelfel. 2017. Randomized Abortable Mutual Exclusion with Constant Amortized RMR Complexity on the CC Model. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. Association for Computing Machinery, New York, NY, USA, 221–229. https://doi.org/10.1145/3087801.3087837

Philipp Haller and Heather Miller. 2019. A reduction semantics for direct-style asynchronous observables. *Journal of Logical and Algebraic Methods in Programming* 105 (2019), 75–111.

Joseph Izraelevitz and Michael L Scott. 2017. Generality and Speed in Nonblocking Dual Containers. *ACM Transactions on Parallel Computing (TOPC)* 3, 4 (2017), 1–37.

Prasad Jayanti. 2003. Adaptive and efficient abortable mutual exclusion. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. 295–304.

Xiao Jia, Wei Li, and Viktor Vafeiadis. 2015. Proving Lock-Freedom Easily and Automatically. In *Proceedings of the 2015 Conference on Certified Programs and Proofs* (Mumbai, India) *(CPP '15)*. Association for Computing Machinery, New York, NY, USA, 119–127. https://doi.org/10.1145/2676724.2693179

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–34.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018).

Gilles Kahn and David MacQueen. 1976. Coroutines and Networks of Parallel Processes. (1976).

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Donald E Knuth. 1966. Additional comments on a problem in concurrent programming control. *Commun. ACM* 9, 5 (1966), 321–322.

Nikita Koval, Dan Alistarh, and Roman Elizarov. 2019. Scalable FIFO Channels for Programming via Communicating Sequential Processes. In *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11725)*, Ramin Yahyapour (Ed.). Springer, 317–333. https://doi.org/10.1007/978-3-030-29400-7_23

Nikita Koval, Dan Alistarh, and Roman Elizarov. 2023a. Fast and Scalable Channels in Kotlin Coroutines. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming* (Montreal, QC, Canada) *(PPoPP '23)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3572848.3577481

Nikita Koval, Dmitry Khalanskiy, and Dan Alistarh. 2023b. A Formally-Verified Framework for Fair Synchronization in Kotlin Coroutines. arXiv:2111.12682 [cs.PL]

Nikita Koval, Maria Sokolova, Alexander Fedorov, Dan Alistarh, and Dmitry Tsitelov. 2020. Testing Concurrency on the JVM with Lincheck. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 423–424.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 205–217.

Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 181–196.

Morten Krogh-Jespersen, Thomas Dinsdale-Young, and Lars Birkedal. 2016. Verifying a concurrent data-structure from the Dartino Framework in Iris. (2016).

Doug Lea. 2005. The `java.util.concurrent` synchronizer framework. *Science of Computer Programming* 58, 3 (2005), 293–309.

Hyonho Lee. 2010. Fast Local-Spin Abortable Mutual Exclusion with Bounded Space. In *International Conference On Principles Of Distributed Systems*. Springer, 364–379.

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *Parallel and Distributed Systems, IEEE Transactions on* 15 (07 2004), 491 – 504. https://doi.org/10.1109/TPDS.2004.8

Peter Magnusson, Anders Landin, and Erik Hagersten. 1994. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of 8th International Parallel Processing Symposium*. IEEE, 165–171.

John M Mellor-Crummey and Michael L Scott. 1991. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)* 9, 1 (1991), 21–65.

Maged M Michael and Michael L Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 267–275.

Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. 2014. A Study and Toolkit for Asynchronous Programming in C#. In *Proceedings of the 36th International Conference on Software Engineering*. 1117–1127.

Abhijeet Pareek and Philipp Woelfel. 2012. RMR-Efficient Randomized Abortable Mutual Exclusion. In *International Symposium on Distributed Computing*. Springer, 267–281.

Aleksandar Prokopec and Fengyun Liu. 2018. Theory and Practice of Coroutines with Snapshots. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs, Vol. 109)*, Todd D. Millstein (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:32. https://doi.org/10.4230/LIPIcs.ECOOP.2018.3

Pedro Ramalhete and Andreia Correia. 2017. Brief announcement: Hazard Eras — Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 367–369.

William N Scherer III, Doug Lea, and Michael L Scott. 2006. Scalable Synchronous Queues. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 147–156.

Simon Friis Vindum and Lars Birkedal. 2021. Contextual Refinement of the Michael-Scott Queue (Proof Pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 76–90.

Simon Friis Vindum, Dan Frumin, and Lars Birkedal. 2022. Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 100–115.

Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. *SIGPLAN Not.* 51, 8, Article 16 (Feb. 2016), 13 pages. https://doi.org/10.1145/3016078.2851168

Received 2022-11-10; accepted 2023-03-31