Scaling Concurrent Queues by Using HTM to Profit from Failed Atomic Operations

Or Ostrovsky
Tel Aviv University, Israel

Adam Morrison
Tel Aviv University, Israel

Abstract

Queues are fundamental concurrent data structures, but despite years of research, even the state-of-the-art queues scale poorly. This poor scalability occurs because of contended atomic read-modify-write (RMW) operations.

This paper makes a first step towards designing a *scalable* linearizable queue. We leverage hardware transactional memory (HTM) to design TxCAS, a scalable `compare-and-set` (CAS) primitive—despite HTM being targeted mainly at uncontended scenarios.

Leveraging TxCAS's scalability requires a queue design that does not blindly retry failed CASs. We thus apply TxCAS to the *baskets queue*, which steers enqueuers whose CAS fails into dedicated *basket* data structures. Coupled with a new, scalable basket algorithm, we obtain SBQ, the scalable baskets queue. At high concurrency levels, SBQ outperforms the fastest queue today by 1.6× on a producer-only workload.

CCS Concepts • **Theory of computation → Concurrent algorithms**.

1 Introduction

Multi-producer/multi-consumer (MPMC) queues are fundamental, widely-studied concurrent data structures [7, 17, 22, 27, 28, 31, 41]. These queues are linearizable [16] shared-memory data structures that provide enqueue and dequeue operations with the usual first-in-first-out (FIFO) semantics.

Despite decades of research, even state-of-the-art concurrent queues scale poorly. In an ideal linearly scalable data structure, the latency of individual operations remains constant as the number of cores grows. In contrast, the latency of queue operations *grows at least linearly* with the core count, so that overall queue throughput remains constant or even degrades as concurrency grows.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374511

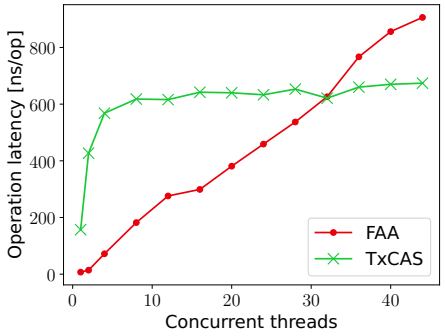


Figure 1. TxCAS vs. standard atomic operation latency.

Queues suffer from poor scalability because of *contended* atomic read-modify-write (RMW) operations, such as those updating the queue head or tail [7, 27, 41]. Even the fastest queues perform one contended `fetch-and-add` (FAA) per queue operation [31, 41]. Other queues [22, 24, 27, 28] use the `compare-and-set` (CAS) primitive, which can fail under contention. A failed CAS needs to be retried, and so these queues perform multiple contended CASs per operation.

One interesting exception is the *baskets queue* [17]. Its enqueue operations use the fact that a CAS failure indicates the presence of concurrent enqueuers to avoid retrying the failed CAS. Instead of retrying a CAS that fails to link a new node to the queue, contending enqueuers place their item in a *basket* data structure associated with the current tail node.

Even so, the baskets queue fails to scale better than FAA-based queues. It still performs one contended CAS, and the latency of any contended atomic operation—whether a failed CAS or a successful FAA—is linear in the number of contending cores, since every atomic operation acquires exclusive ownership of its location's cache line, and these acquisitions are serialized by the cache coherence protocol. This bottleneck seems inherent on current multi-core architectures.

This paper We make a first step towards designing a scalable linearizable queue, by leveraging hardware transactional memory (HTM) [15]—despite HTM being targeted mainly at *uncontended* scenarios. Our core insight is that a CAS properly implemented with HTM, which we call TxCAS, is *fundamentally more scalable* than a standard atomic operation. Figure 1 compares the latency of our TxCAS to a standard FAA on a 22-core (44-hyperthreaded) Intel Broadwell processor, as contention grows. TxCAS's latency remains roughly constant beyond 10 hardware threads, whereas the FAA latency grows linearly.

TxCAS obtains its scalability by not serializing failures. In implementing CAS as a hardware transaction, we break its cache coherence footprint into a read, which acquires shared ownership of the cache line, followed by a write. The write acquires exclusive ownership of the line and thereby aborts any TxCASs who have only read. As we explain in § 3, these aborts occur *concurrently*, resulting in scalable failures.

Leveraging TxCAS's scalability requires an algorithm that can profit from a failed CAS (and not merely retry it), which we obtain by improving upon the baskets queue. We introduce SBQ—the scalable baskets queue—which makes the baskets queue scale by using TxCAS and by further improving the queue with a new, scalable basket design. SBQ inherits the baskets queue's lock-free [14] progress guarantee.

Limitations We posit SBQ as showcasing a novel synchronization technique that may pave the way towards scalable queues, and not as strictly superior to prior queues on contemporary hardware, for the following reasons:

- TxCAS's scalability incurs a latency cost at low concurrency (Figure 1). TxCAS will thus be more effective on future processors with higher core counts.
- SBQ's dequeue operations are less scalable than its enqueue operations. On an enqueue-dominated workload, SBQ outperforms the fastest queue we are aware of—the FAA-based queue of Yang and Mellor-Crummey [41]—by 1.6× on a dual-processor machine with 88 hardware threads. However, its improvement in a mixed enqueue/dequeue workload is a more modest 1.16×.
- The HTM implementation in current hardware limits TxCAS effectiveness in cross-processor (NUMA) usage (§ 4.3). This limitation *does not* rule out NUMA execution; it only means that TxCASs of a location should be run on the same processor (NUMA node). Consequently, the scope of our evaluation is limited to such intra-processor TxCAS use. We propose a microarchitectural solution for future processors to address this problem (§ 3.4.1).

Contributions To summarize, our contributions are:

1. Showing that the cache coherence behavior of an HTM-based CAS results in scalable CAS failures (§ 3).
2. Designing TxCAS, an HTM-based CAS that realizes the above benefit on current Intel processors (§ 4).
3. Identifying and proposing a microarchitectural solution to the HTM implementation issue that limits TxCAS's effectiveness across NUMA domains (§ 3.4).
4. Designing SBQ, a queue with better scaling properties than prior work (§ 5).
5. Empirically evaluating SBQ on a dual-processor x86 machine with 44 cores (88 hyperthreads) in total (§ 6).

2 Preliminaries

Model We use a standard shared-memory system model [16] in which a program is executed by threads that communicate via atomic operations on a shared memory. For simplicity of presentation, we assume a sequentially consistent system [25] in which the execution is an interleaving of the thread operations. In practice, processors and programming languages provide weaker gaurantees [2, 34]. Our evaluated implementations use C11 atomic accesses and fences to prevent undesired reorderings by the compiler or hardware.

Atomic primitives We model the memory as an array, m, of 64-bit words. Let $m[a]$ be the word found at address a in the memory. The system's atomic primitives, which are supported by the 64-bit x86 processors we use, are as follows:

read/write Read/write the value of $m[a]$.
FAA(a, v) Returns $m[a]$ and stores $m[a] + v$ into $m[a]$.
SWAP(a, v) Returns $m[a]$ and stores v into $m[a]$.
CAS(a, t, v) If $m[a] = t$, then v is stored into $m[a]$ and the operation returns true; otherwise, it returns false.

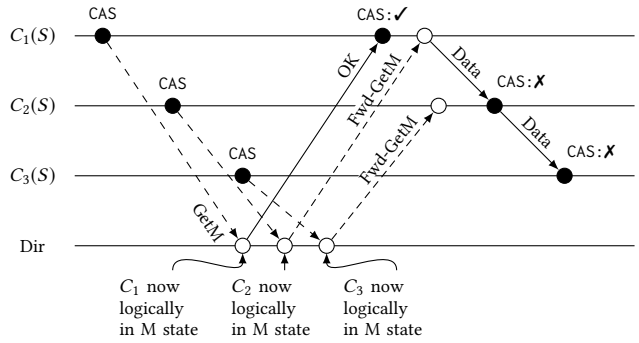
HTM Transactional memory (TM) allows grouping memory accesses into a transaction, such that they appear to either execute atomically or not to execute at all. Several modern architectures offer hardware-supported TM (HTM) [3, 42]. Here, we describe the HTM interface, based on Intel's HTM [42] (other architectures are similar). We discuss relevant implementation details in § 3.

Calling `_xbegin()` starts a transaction and checkpoints the processor's state. Calling `_xend()` attempts to *commit* a running transaction. The memory operations performed by a transaction take effect only if it successfully commits; otherwise, it *aborts* and the processor's state is restored from the checkpoint. The HTM provides no guarantee that a transaction will successfully commit. Transactions may be aborted by the system, either due to a *conflict*—when two concurrent transactions access the same memory location and at least one access is a write—or due to implementation-specific reasons (e.g., receipt of a hardware interrupt). A transaction can also abort itself by calling `_xabort()`. The system supports flat nesting of transactions: if a nested transaction aborts, the top-level transaction aborts as well.

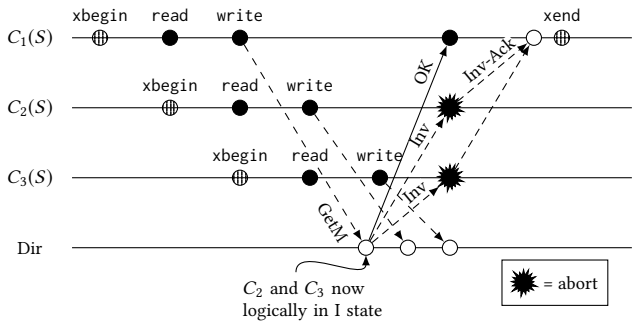
Linearizable queues A *FIFO queue* is an object whose state Q is a (possibly empty) sequence of elements. It supports an enqueue operation, which appends an element to Q, and a dequeue operation, which removes and returns the first element of Q, or returns `NULL` if Q is the empty sequence. Our correctness condition is *linearizability* [16], which states (informally) that every operation appears to "take effect" instantaneously at some point during its execution interval, and there is a total order on all such linearized operations.

3 Scalability of HTM-based CAS

We analyze the cache coherence dynamics of contended atomic operations, and show that a CAS implemented with an HTM transaction has inherently better scalability than a standard atomic CAS operation.



(a) Standard CAS: All CAS operations, even failures, are serialized.



(b) HTM-based CAS: Failures are not serialized.

Figure 2. Cache coherence dynamics of contended CAS. (Solid, empty, and dashed circles represent memory operations, receipt of coherence requests or invalidations, and transaction begin/end, respectively. CAS success/failure is denoted by ✓/✗.)

3.1 Multi-core cache coherence

We consider a multi-core processor with a private cache for each core and a last-level cache (LLC) shared by all cores, which models the architecture of modern processors [37].[1] The processor uses a *cache coherence protocol* to guarantee that at any point in time, each memory location has a single well-defined value across all caches. For simplicity, we consider a basic MSI protocol [37], but our analysis applies to the MOESI [38] and MESIF [9] protocols used commercially.

The MSI protocol maintains a *single-writer/multiple-reader* invariant for each cache line: at any point in time, either one core is allowed to write to the line, or multiple cores are allowed to read it. Each line in a core's private cache can be in one of the following states:

Modify The line may be read or written by the core. Any other private cache must refer to the line as Invalid.

Shared The line may only be read by the core. Multiple caches may hold the line in this state.

Invalid The line is either not present or has been invalidated by the protocol. It may not be accessed.

To change the state of a cache line, a cache controller initiates a *coherence transaction* that involves the exchange of coherence messages. We will walk through the relevant coherence transactions in the following sections.

Modern systems use a scalable *directory-based* protocol implementation, in which a shared *directory* structure keeps track of each line's state and the caches that contain it. Coherence transactions contact the directory, and it either responds directly or forwards the request to another cache controller. The directory and caches communicate via point-to-point communication over a shared interconnect [37]. We assume the interconnect supports multiple in-flight messages (i.e., is not a broadcast bus), which is the case for modern commercial multi-core processors [37].

3.2 Non-scalability of standard CAS

A core executes an atomic RMW operation by acquiring write ownership of the target location's cache line[2] and performing the read-modify-write sequence. To guarantee the RMW's atomicity, the core *stalls* any incoming coherence messages that will cause it to lose ownership of the line, and handles them only after the RMW completes [37].

Standard atomic RMWs do not scale because the coherence protocol serializes write ownership acquisitions. This serialization makes the average cost of an RMW contended by C cores to be about $C/2$ uncontended cache misses—regardless of whether the RMW is a failed or successful CAS, or another RMW type. Figure 2a illustrates the dynamics for CAS.

Initially, all cores hold the cache line in Shared state, having read the same "old" value and poised to execute CASs of different values. Executing the CAS initiates a coherence transaction to upgrade the line to Modify state, which issues a GetM request to the directory. On receipt of a GetM request, the directory makes the requester the *owner* of the line. If the line was in Shared state, the directory sends *invalidation* messages instructing the cores sharing the line to move it to Invalid state (for readability, invalidations are omitted from the figure). If the line was in Modify state, the directory sends a Fwd-GetM request to the previous owner, which invalidates the line and sends it to the new owner. If a core receives a Fwd-GetM before completing its own GetM (like C_2), it stalls the Fwd-GetM until it gets the line and attempts its CAS.

Because of the latency of this owner-to-owner line handoff, when C GetM requests arrive at the directory back-to-back (and result in back-to-back Fwd-GetMs) the i-th requester will receive ownership of the line and attempt its CAS only i message delays later. This results in an average CAS latency of $(C+1)/2$ message delays, where a message delay is about 15–30 cycles on a modern multi-core processor [19].

[1] In practice, cores can have multiple levels of private caches, but this does not alter our analysis.

[2] With respect to cache coherence, we refer to a core and its private cache controller interchangeably.

CAS vs other RMWs The above discussion applies to other RMWs, such as FAA. Therefore, under equal contention, the per-operation latency of CAS and FAA is identical. With FAA, however, all operations succeed, whereas with CAS, all operations but the first fail. Therefore, under high contention, performing N successful CASs requires $\approx N^2$ CAS attempts, which can increase contention and thus CAS latency.

3.3 Scalability properties of HTM-based CAS

We consider the scalability implications of implementing `CAS(a,t,v)` with an HTM transaction that reads the location m[a] and writes v to it if its value is t. Our insight is that such transactions are *not* serialized when they read, which allows CAS failure latency to scale. To explain this, we first describe how commercial HTMs implement concurrency control by "piggybacking" on the cache coherence protocol [15, 20].

The memory operations of a transaction execute as usual, but the accessed lines are marked as *transactional* in the core's private cache. The system implements a *requester-wins* conflict resolution policy: if a core receives a coherence message that will cause it to lose its permissions for a transactional line, it aborts the transaction. Committing a transaction clears the "transactional" marks, making the transaction's writes visible to the rest of the cores.

Figure 2b illustrates the dynamics of a contended HTM-based CAS. Again, all cores initially hold the cache line in Shared state. The HTM transactions first read the line—marking it as transactional—and then execute the write, which issues a GetM coherence request. The first GetM that reaches the directory triggers an invalidation of the Shared state in the other *sharer* cores. Crucially, *these invalidations are sent back-to-back—making their way to the sharers concurrently—and their receipt aborts the transactions.* (Figure 2a did not depict these invalidations because they did not affect the behavior of the cores, which had already started their CAS and could not "abort" it.)

Each invalidated sharer sends an Inv-ACK message to the new owner. Once all sharers have acknowledged their invalidation, the new owner's GetM coherence request completes, at which point its transaction commits. Overall, each HTM transaction commits or aborts *within a constant number of message delays*, implying scalable constant CAS latency.

Importantly, the HTM-based CAS does not avoid serialization altogether. It only avoids serializing failed CASs; successful CAS transactions remain serialized. In fact, if a failed CAS transaction issues a GetM request before aborting—as in Figure 2b—that request will be handled by the directory and the core's cache will eventually receive ownership of the line (we omit these messages from the figure). This process does not delay the core. Since its transaction has aborted, the core does not block on the GetM, and the related coherence messages are handled asynchronously by its cache controller.

On the other hand, the protocol's handling of such pending GetM requests can delay a future (ultimately successful) transaction. This transaction's coherence requests will be serialized after the pending GetM requests and it will obtain ownership of the line only after each previous requester obtains and hands off the line (similar to the effect described in § 3.2). Our TxCAS (§ 4) is designed to minimize this effect.

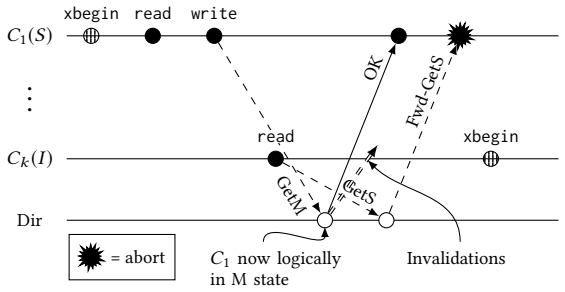


Figure 3. Tripped writer

Can standard failed CAS avoid serialization? It may seem that standard CAS failures could be made scalable by leveraging the above insights, i.e., if a standard CAS would first acquire Shared ownership of the target location and subsequently upgrade to Modify ownership only if the location has the expected "old" value. However, such an implementation would not fundamentally differ from current CAS implementations, as the check of the "old" value would have to be redone after the upgrade (since the directory may have handled some GetM in the mean time). The effect depicted in Figure 2a can therefore still occur, namely, when contending CASs all successfully check the "old" value and issue a GetM.

3.4 HTM-based CAS "tripped writer" problem

We identify the *tripped writer* problem, which occurs when an HTM-based CAS that has reached the CAS write step gets aborted due to a conflict. The conflict thus *trips* the transaction just before it reaches the "finish line" and commits.

In practice, these conflicts are overwhelmingly caused by a read by another core and not by a write. Figure 3 shows such a scenario. C_1 starts a CAS transaction and reaches the write step, which issues a GetM coherence request. Concurrently, another core C_k—which is about to start a CAS transaction and does not have a copy of the line—reads from the line to obtain the "old" argument. This read issues a GetS coherence request, which reaches the directory after C_1's GetM.

From the directory's perspective, C_1 *is the owner of the line* and only it holds the latest data. Therefore, the directory sends a Fwd-GetS to C_1, instructing it to downgrade its state to Shared and send a copy of the line to C_k [37]. The Fwd-GetS can arrive at C_1 while C_1 is still in the transaction, because it has not yet received all the invalidation acknowledgements required to complete its GetM request. Receiving the Fwd-GetS, which indicates a remote read of a transactionally written line, aborts C_1's transaction.[3]

[3]We are not able to determine whether the transaction aborts immediately upon receiving the Fwd-GetS or only after the GetM completes. In any case, the effect is the same.

Fortunately, tripped writers are not a fundamental problem. In § 3.4.1, we describe a minor microarchitectural modification that prevents them. As this fix only applies to future systems, however, our TxCAS design must currently address the problem in software, which leads to limited effectiveness on certain workloads on current hardware (§ 4).

3.4.1 Microarchitectural solution

Tripped writers can be eliminated with a minor microarchitectural modification. Importantly, our proposed change is generic and does not require HTM-based CAS transactions to be treated by the hardware as "special" in any way.

Our insight is as follows. Modern microarchitectures do not wait for a write instruction's GetM request to complete before moving on to execute subsequent instructions; the data is stored in a store buffer [19] and gets written to the cache asynchronously, once the GetM request completes. In an HTM-based CAS, the instruction following the write is the `_xend()`, which *does* block until the GetM completes. Specifically, `_xend()` blocks until all writes performed by the transaction have been propagated to the cache. This means that when a tripped writer condition occurs, *the core "knows" that the transaction is ready to commit.*

We propose to leverage this knowledge as follows. Rather than unconditionally aborting a transaction upon receiving a conflicting coherence message, check if (1) the core is blocked on an `_xend()`, and (2) the core has a single GetM request pending, and (3) the conflicting coherence request is by a read. If so, stall the incoming request until the transaction commits; otherwise, abort the transaction as usual. These stalls cannot deadlock the system, because GetM requests never get stalled.

With our proposed change, the execution depicted in Figure 3 will not cause C_1 to abort. Instead, the Fwd-GetS will be stalled at C_1 until all invalidation acknowledgments arrive and C_1's transaction commits. Then, C_1 will send C_k the data just written by the committed transaction.

4 TxCAS design

Designing a transactional CAS is conceptually simple, requiring only wrapping the read-compare-write sequence in a transaction. However, the design must overcome several practical challenges caused by limitations of the commercial HTM interface, its semantics, and its hardware implementation. Here, we walk through the design and its rationale.

Algorithm 1 presents our TxCAS design. We use Intel's HTM (called RTM [20]) in which an `_xbegin()` starting a transaction checkpoints the core's state and returns a special `successful` value, and an abort restores the checkpointed state and returns a bit mask that encodes the abort reason.

The algorithm performs the CAS read (Line 5) in a nested transaction and the CAS write (Line 10) in the main transaction. (We explain the use of nested transactions in § 4.2.) If

Algorithm 1 Transactional compare-and-set (TxCAS)

```
 1: function txn_cas(int* ptr, int old, int new)
 2:   loop
 3:     if successful(ret := _xbegin()) then
 4:       if successful(_xbegin()) then
 5:         value := *ptr
 6:         if value ≠ old then _xabort(1)
 7:         delay()
 8:         _xend()
 9:       end if
10:       *ptr := new
11:       _xend()
12:       return true            ▷ Code following successful commit
13:     end if
14:     ▷ On abort, execution resumes here
15:     if self-abort(ret) then return false
16:     if not (conflict(ret) and nested(ret)) then
17:       continue
18:     end if
19:     delay()
20:     if *ptr ≠ old then return false
21:   end loop
22: end function
```

the value read does not match the CAS' `old` argument, the transaction self-aborts (Line 6) and TxCAS returns `false` (Line 15). Otherwise, TxCAS waits for a while and then proceeds to the CAS write. (We explain this delay in § 4.1.)

If the transaction does not abort because of a conflict, or if it aborts because of a conflict that occurs after the nested transaction (i.e., in Lines 10–11), TxCAS retries the transaction (Lines 16–18). Otherwise, TxCAS waits for a while and then verifies that the target location has changed. If so, it returns `false`; otherwise, it retries the transaction. We explain the abort handling logic in § 4.2.

Progress The HTM offers no progress guarantee [20] and so, in principle, TxCAS could fail to terminate because its transactions always abort not due to a conflict. To address this problem, TxCAS falls back to performing a standard CAS after sufficiently many retries of the loop. This fallback makes TxCAS wait-free [14]: every TxCAS returns after a finite number of its own steps. In practice, however, we find that TxCAS operations terminate without requiring this fallback, and so we omit it from the pseudo code.

4.1 Intra-transaction delay

TxCAS places a delay between reading the target location and writing it (Line 7). This delay serves two purposes.

First, the delay increases the chance that the transaction gets aborted by a conflicting TxCAS write before it issues its own write (and corresponding GetM coherence request). Increasing the chance of such pre-write aborts decreases the amount of pending GetM requests, which are issued by TxCASs that ultimately abort (and whose CAS fails) but only after issuing their write. As explained in § 3.3, such pending GetM requests delay future transactions and increase contention on the target memory location.

Second, the delay increases the average number of transactions that conflict with and are aborted by the write of a

successful (committing) TxCAS. The reason is that as the ultimately successful TxCAS—which is the first to arrive—delays, more TxCASs arrive and will be aborted by the write. This delay is especially helpful for scalability in low concurrency settings, where without delaying most TxCASs would succeed, and would thus be serialized like standard CASs.

The disadvantage of the intra-transaction delay is that it slows TxCAS down. Hence, the scalable but relatively high latency observed in Figure 1 for low thread counts.

In our evaluated implementation, we use a delay of approximately 270 nanoseconds, which we empirically found to be optimal for our benchmarks on our evaluation platform.

4.2 Handling aborts

To provide CAS semantics, TxCAS must guarantee that it returns `false` only if another TxCAS has successfully committed (and thus returns `true`). Because transactions can abort for arbitrary reasons, not necessarily due to a conflict, TxCAS cannot simply rely on its transaction aborting as the condition for returning `false`. Instead, TxCAS fails after an abort only if the target location has actually changed; otherwise, it retries the transaction (Line 20 of Algorithm 1).

Checking the target location after an abort needs to be done carefully. The fact that the transaction was just aborted means that there is a GetM request by a writer in flight. Reading the target location at this point would likely trip this writer. To avoid creating a tripped writer problem, TxCAS delays before reading the location (Line 19). The delay is timed to give the writer a chance to complete its GetM request, and thereby avoid aborting it.

TxCAS performs the post-transaction delay only if necessary. Specifically, if the abort is not caused by a conflict (which is determined from the `_xbegin()` return value), the transaction is immediately retried. Similarly, we would like to immediately retry if the abort is caused by a conflict with the write step. In such a case, our aborted transaction may be the tripped writer, and delaying after the abort would be a waste of time.

Unfortunately, the HTM interface—i.e., the abort reasons encoded in the `_xbegin()` return value—does not specify the type of conflict or where it occurred in the transaction. We circumvent this limitation by exploiting a reason that the interface does provide, namely, whether the conflict occurred in a nested transaction. TxCAS performs the read step in a nested transaction. Thus, an abort occurring inside the nested transaction implies that the write step was not executed, and so TxCAS checks if the target location has changed; otherwise, the transaction is immediately retried.

4.3 Implications of the tripped writer problem

Whether the tripped writer problem materializes depends on the likelihood of a remote read coherence request hitting the window in which a TxCAS waits for its write request to complete. When coherence requests are confined to a single multi-core processor, this window is small—around 30–60 cycles. Consequently, a short post-transaction delay—as discussed in § 4.2—suffices to make tripped writers insignificant.

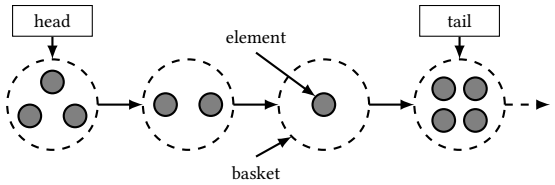


Figure 4. Baskets queue idea. Source: [17]

However, when coherence requests involve different processors in a multi-socket NUMA system, they cross interconnects (such as the QuickPath Interconnect [36, 43]) whose latency is larger than of an on-chip interconnect . Increasing the post-transaction delay to match cross-socket latency would make TxCAS so slow as to obviate its scalability benefits on current hardware. But without the delay, tripped writers cause multiple transaction retries per TxCAS, similarly making TxCAS ineffective.

As a result, in this paper, *we limit the scope of our evaluation to intra-processor synchronization.* This limitation *does not* rule out NUMA executions; it only means that TxCASs of a location should be run on the same processor (§ 6).

5 SBQ: A scalable baskets queue

This section describes SBQ, our scalable, lock-free queue algorithm. SBQ builds on the idea of the baskets queue (§ 5.1). However, the baskets queue fails to scale due to the use of CAS and a non-scalable basket data structure. To address these problems, we abstract the basket data structure, creating a modular baskets queue design (§ 5.2) into which we plug TxCAS and a new scalable basket design (§ 5.3).

5.1 Background: the baskets queue concept

The baskets queue [17] is a variant of the Michael-Scott queue [27] that aims to reduce the queue's CAS contention.

The Michael-Scott queue is comprised of a singly linked list of nodes and head and tail pointers. Its enqueue operation uses CAS to replace the value of the tail node's next pointer with the address of a new node. If the CAS fails, it is retried.

The baskets queue's underlying observation is that a failed CAS indicates that a successful CAS updating the next pointer occurred in the window of time between reading the pointer and attempting the failed CAS. This holds for every enqueue whose CAS fails at the same tail node, which allows to divide all the enqueue operations to equivalence classes. Each class contains a single successful enqueue operation and all the enqueue operations whose CAS fails because of it.

Since all the operations in an equivalence class are guaranteed to be concurrent, their elements may be dequeued at any order without compromising the queue's linearizability. Conceptually, there is an unordered *basket* associated with each node, into which a failed enqueue places its element instead of retrying the CAS (Figure 4).

5.2 Modular baskets queue

We propose to explicitly define an abstract data type (ADT) for the baskets, which enables plugging in different basket implementations. In comparison, the baskets in the original basket queue are implicit: If an enqueue fails to link a node after the tail, it retries the insertion at the same node instead of finding the new tail and linking its node there.

A key benefit of our framework is that it helps crystalizing different basket properties that ultimately imply linearizability of the queue. In our framework, for example, the original baskets queue can be viewed as using a variant of the LIFO Treiber stack [39] as the basket. To maintain linearizability of the queue, this stack variant has the property that once an item is removed from the basket, further insertions are prevented. As we shall see, our proposed scalable basket (§ 5.3) offers more relaxed properties.

5.2.1 The basket interface

A basket is a linearizable implementation of the following sequential specification: The state of a basket is a set B. The basket supports the following operations:

`basket_insert`(x) This operation attempts to insert x to B and returns `SUCCESS` if successful. It is allowed to fail non-deterministically and return `FAILURE` without modifying B.

`basket_extract` This operation removes some $x \in B$ and returns it. If B is empty, it returns `NULL`.

`basket_empty` If B is not empty, returns `false`. Otherwise, the return value can be either `true` or `false`, i.e., false negatives are allowed.

We note that, as we shall see, the basket's interface does not imply the linearizability of the baskets queue. A basket implementation must thus satisfy the condition that plugging it into the baskets queue results in a linearizable queue. The reason that we do not specify a more specific condition is that different basket properties—with associated performance trade-offs—can be used to make the queue linearizable.

5.2.2 Modular queue description

This section describe our formulation of the baskets queue as an algorithm based on explicit (pluggable) baskets.

The queue is represented as a singly linked list of nodes. Algorithm 2 shows the queue data structures. Unlike the original baskets queue, in our framework each node contains a basket that can hold multiple elements. The node's `next` pointer is initialized to `NULL` when the node is created, and will eventually point to the node placed after it. Each node is identified with a unique `index`. The queue maintains the invariant that linked nodes have consecutive indices.

The queue contains `head` and `tail` pointers. Initially, both point to the same empty sentinel node, whose `next` pointer is `NULL`. The algorithm uses a form of epoch-based memory reclamation [8] described later, which relies on the queue's `retired` and `protectors` fields and on node `index` fields.

Algorithm 2 Queue data structures

```
struct node_t {
    basket_t basket;
    node_t* next;
    int index;
};

struct queue_t {
    node_t* head;
    node_t* tail;
    node_t* retired;
    node_t* protectors[N];
};
```

Algorithm 3 Enqueue operation

```
 1: function enqueue(queue_t* Q, T* element, int id)
 2:   node_t* t := protect(&Q→tail, &Q→protectors[id])
 3:   node_t* new_node := allocate_node() ▷ May reuse from last time
 4:   basket_insert(&new_node→basket, element, id)
 5:   loop
 6:     new_node→index := t→index + 1
 7:     status := try_append(t, new_node)
 8:     if status = SUCCESS then
 9:       CAS(&Q→tail, t, new_node)
10:       return
11:     else if status = FAILURE then
12:       t := t→next
13:       if basket_insert(&t→basket, element, id) then
14:         break
15:       end if
16:     end if
17:     while t→next ≠ NULL do
18:       t := t→next
19:     end while
20:     advance_node(&Q→tail, t)
21:   end loop
22:   unprotect(&Q→protectors[id])
23: end function
```

Algorithm 4 Basic `try_append`

```
1: function try_append(node_t* tail, node_t* new_node)
2:   if tail→next ≠ NULL then return BAD_TAIL
3:   return CAS(&tail→next, NULL, new_node)
4: end function
```

Algorithm 5 Dequeue operation

```
 1: function dequeue(queue_t* Q, int id)
 2:   node_t* h := protect(&Q→head, &Q→protectors[id])
 3:   loop
 4:     while basket_empty(&h→basket) and h→next ≠ NULL do
 5:       h := h→next
 6:     end while
 7:     element := basket_extract(&h→basket, id)
 8:     if element ≠ NULL or h→next = NULL then
 9:       break
10:     end if
11:   end loop
12:   advance_node(&Q→head, h)
13:   free_nodes(Q)
14:   unprotect(&Q→protectors[id])
15:   return element
16: end function
```

Algorithm 6 Advancing the queue head/tail

```
1: function advance_node(node_t** ptr, node_t* new_node)
2:   loop
3:     node_t* old_node := *ptr
4:     if old_node→index ≥ new_node→index then return
5:     if CAS(ptr, old_node, new_node) then return
6:   end loop
7: end function
```

We assume that both queue and basket operations take the id of the calling thread, in addition to their standard arguments. For simplicity, we assume that enqueuers and dequeuers are indexed separately.

Enqueue (Algorithm 3) An enqueue operation allocates a new node, inserts the element to that node's basket, and tries to append the node to the tail of the queue. This is done using the try_append function (Algorithm 4), which returns one of the following values:

SUCCESS If the new node was appended to the queue.
FAILURE If another node was appended to the queue.
BAD_TAIL If the tail node already points to another node, making it an invalid (or "stale") tail.

If the try_append succeeds, the enqueuer tries to advance the tail pointer and completes. If the try_append fails, the enqueuer tries to insert its element into the basket of the newly appended node. If this basket insertion succeeds, the operation completes and (to reduce contention) does not advance the tail pointer. If the basket insertion fails or if try_append returns BAD_TAIL, the operation is retried. Before retrying the operation, the new tail of the queue is found by traversing from the current tail, and the enqueuer advances the queue tail at least to that node.

Retrying when try_append returns BAD_TAIL—i.e., when the enqueuer has not observed the next pointer of the current tail to be NULL—is required for linearizability, to prevent the enqueuer from inserting into the same basket it used in a previous enqueue operation that completed without advancing the queue tail.

When an enqueue operation completes without appending its node to the queue, the thread's next enqueue operation reuses the node instead of allocating a new node. Such reuse also resets the node's basket's state (undoing the single element insertion), which we assume takes constant time. This optimization makes basket initialization time amortized $O(B/T)$, where B is the basket size and T is the number of enqueuers, assuming an enqueuer succeeds appending its basket in one out of T attempts.

Dequeue (Algorithm 5) A dequeue operation finds the first node with a non-empty basket, if such a node exists, and tries to extract an element from that basket. If the extraction succeeds, the extracted element is returned. If the extraction fails and the node was the last node in the queue, the queue is considered empty and NULL is returned. Otherwise, the operation finds the next non-empty node and tries again. Before returning, the dequeuer advances the queue's head—swinging it past empty nodes—and attempts to reclaim the memory of nodes which the head has advanced over.

Head/tail advancement (Algorithm 6) The advance_node function is used to make sure that the queue head or tail advances at least to the node passed to it—i.e., that the head/tail points to that node or to a node with a greater index.

Algorithm 7 Memory reclamation functions

```
 1: function protect(node_t** ptr, node_t** p)
 2:   loop
 3:     *p := *ptr
 4:     ▷ On non-SC systems, reordering of the write to *p (line 3)
 5:     ▷ and the read of *ptr (line 7) must be prevented (e.g.,
 6:     ▷ using a memory fence).
 7:     if *ptr = *p then return *p
 8:   end loop
 9: end function

10: function unprotect(node_t** p)
11:   *p := NULL
12: end function

13: function free_nodes(queue_t* Q)
14:   node_t* retired := SWAP(&Q→retired, NULL)
15:   if retired = NULL then return
16:   index := min{p→index | ∃i. p = Q→protectors[i] and p ≠ NULL}
17:   while retired ≠ Q→head and retired→index < index do
18:     node_t* tmp := retired→next
19:     free(retired)
20:     retired := tmp
21:   end while
22:   Q→retired := retired
23: end function
```

Memory reclamation (Algorithm 7) The queue design is compatible with standard memory reclamation schemes, such as epoch-based memory reclamation [8] or hazard pointers [26]. For concreteness, we describe the epoch-based reclamation scheme used in our evaluation, which is adapted from Yang and Mellor-Crummey's wait-free queue [41].

We refer to a node as *retired* when the queue's head advances past it. The queue maintains a retired pointer that initially points to the same sentinel node as head but subsequently lags behind it, pointing to the retired prefix of the queue. The queue contains a protectors array (with an entry per thread) in which a thread announces the earliest node in the queue it might access. Announcements are made and cleared by the protect and unprotect functions, respectively, at the beginning and completion of queue operations.

When a thread completes a dequeue, it attempts to reclaim retired nodes using the free_nodes function. This function advances the retired pointer to the earliest protected node (determined from the indices of the protected nodes) or to the current head (if all retired nodes are unprotected). It also frees the memory of all nodes the retired pointer advances over. Memory reclamation is performed in mutual exclusion: free_nodes updates retired to NULL using SWAP, and immediately returns if retired was already NULL. Like all epoch-based reclamation schemes, the scheme may fail to free memory if a thread stalls indefinitely (either between protect and unprotect calls, or during free_nodes).

Linearizability As stated in § 5.2.1, the basket implementation must somehow guarantee that instantiating the queue with it yields a linearizable queue. For instance, when viewed in our framework, the guarantee of the original basket's queue LIFO basket is that all basket_insert operation fail once an element has been extracted. (Technically, this was

achieved by having the extract set a special "deleted" bit in the `next` pointer, which insertions would then check and fail if the bit was set. We do not discuss the original approach in detail, since our basket uses a completely different approach.)

5.3 SBQ design

We obtain SBQ by improving the scalability of the modular baskets queue from § 5.2 in two ways. First, we replace the use of CAS in the `try_append` function (Algorithm 4) with our TxCAS. Second, we devise a new scalable basket and plug it into the modular queue design.

5.3.1 The SBQ basket

Our basket is designed to avoid contention as much as possible. It consists of an array in which each inserter has a private entry, allowing for synchronization-free insertions. Extractions obtain an index to extract from using FAA. To reduce FAA contention, we use an `empty` bit that, when set, causes extractors to fail without performing the FAA.

Algorithm 8 describes the basket data structure. It consists of an array of pointers, one for each inserter, a counter, and an empty bit. Each array cell may contain either a pointer to a valid element, or one of two reserved values: `INSERT` and `EMPTY`. Each cell is initialized to the value `INSERT` when the basket is created. Additionally, the counter is initialized to 0, and the empty bit is initialized to `false`.

Algorithm 9 shows the basket operations. The `basket_insert` operation uses CAS to attempt to place its value instead of the `INSERT` value in the inserter's cell. If it succeeds, then the operation succeeds too; otherwise, it fails.

The `basket_extract` fails if the `empty` bit is set. Otherwise, it acquires access to some cell by performing a FAA on `counter`. If the index retrieved is outside of the array bounds, the basket is considered empty. Otherwise, the extractor performs an atomic exchange with the value `EMPTY` on that cell. This exchange returns a value previously stored by an inserter, in which case the extract completes, or it prevents a future inserter from writing to this cell. In this case, the extract retries. The extractor that gets an index to the last cell sets the `empty` bit.

The `basket_empty` operation returns the `empty` bit's value.

Basket linearizability & progress Our basket is wait-free and linearizable with respect to the specification of § 5.2.1. We sketch the linearizability proof, due to space constraints. We first define linearization points for the basket operations.

Let the number of inserters (i.e., size of the `cells` array) be N, and let T_N be the time in which the value of the basket's counter is incremented to N. The linearization points of the basket operations are defined as follows:

- A `basket_insert` operation is linearized at the CAS operation.
- A `basket_empty` operation is linearized at the read from the `empty` bit.

Algorithm 8 SBQ basket structure

```
struct basket_t {
    void* cells[enqueuers];
    int counter;
    bool empty;
}
```

Algorithm 9 SBQ basket operations

```
 1: function basket_insert(basket_t* basket,
                           T* element, int id)
 2:   return CAS(&basket→cells[id], INSERT, element)
 3: end function

 4: function basket_extract(basket_t* basket, int id)
 5:   if basket→empty then return NULL
 6:   while (index := FAA(&basket→counter, 1)) < enqueuers do
 7:     if index = enqueuers - 1 then
 8:       basket→empty := true
 9:     end if
10:     element := SWAP(&basket→cells[index], EMPTY)
11:     if element ≠ INSERT then return element
12:   end while
13:   return NULL
14: end function

15: function basket_empty(basket_t* basket)
16:   return basket→empty
17: end function
```

- A failed `basket_extract` operation is linearized at the last FAA the operation performs.
- A successful `basket_extract` operation is linearized at one of the following points, depending on when the CAS inserting the extracted element occurs:
  - If the CAS occurs after T_N, the `basket_extract` operation is linearized at the CAS operation *inserting* the value.
  - Otherwise, the `basket_extract` operation is linearized at the earliest point of either the operation's SWAP or T_N.

We next prove that the linearization point of a successful `basket_extract` operation is within its execution interval.

Lemma 5.1. *A successful* `basket_extract` *operation does not start after* T_N.

Lemma 5.2. *If a* `basket_insert` *operation successfully performs its CAS after* T_N*, there is a pending* `basket_extract` *operation that starts no later than* T_N *and successfully extracts the inserted element.*

Finally, we show that the linearization points induce a linearization of the basket's execution.

Theorem 5.3. *Algorithm 9 is a linearizable implementation of the basket specification.*

5.3.2 SBQ linearizability

We sketch SBQ's linearizability proof, which uses the Aspect-Oriented Linearizability proof framework [13]. According to the framework, the queue implementation is linearizable if every execution of queue operations can be completed

so it does not have any pending (i.e., ongoing) operations, and every such complete history is free of the following violations (assuming uniqueness of enqueued values):

VFresh A value returned by a dequeue operation has not been previously enqueued by an enqueue operation.

VRepeat Two dequeue operations return a value enqueued by the same enqueue operation.

VOrd enqueue(b) is invoked after enqueue(a) completes; some dequeue operation returns b; but either no dequeue operation returns a or the dequeue returning a is invoked after b's dequeue completes.

VWit A dequeue operation returns NULL although there are elements enqueued before the operation which are not yet dequeued.

See [13] for a formal definition of the violations.

That VFresh and VRepeat violations cannot occur follows straightforwardly from the linearizability of our basket. The proof that VOrd and VWit violations are impossible follows from the following property of the basket: Let t be a point in time in which the basket is indicated to be empty, i.e., when some `basket_extract` returns NULL or some `basket_empty` returns `true`. Then any successful `basket_extract` must have started before time t. (In other words, once the basket is indicated as empty, any future `basket_extract` invocations are guaranteed to fail and return NULL.)

Lemma 5.4. *VOrd violations are impossible.*

Proof. Let $P \prec Q$ denote that operation P returns before operation Q is invoked, i.e., they are not concurrent. Proving that VOrd violations are impossible boils down into proving that if two dequeues, $D_1 \prec D_2$, both successfully dequeue from baskets B_1 and B_2 respectively, then $B_2 \not\prec B_1$, where we abuse notation and identify a basket with the enqueue that linked it to the queue. Our basket implementation implies this property, because if $B_2 \prec B_1$ then D_1 traverses through B_2 before reaching B_1 and therefore observes the basket B_2 to be empty. This contradicts the fact that D_2 later successfully dequeues from the basket B_2. □

Lemma 5.5. *VWit violations are impossible.*

Proof. Let D be a dequeue returning NULL. Then at some time t, D observes that the last empty basket is also the last basket in the queue. Our basket's property implies that the dequeue operation of any element inserted into a basket observed empty by D starts before the basket becomes empty, and in particular, before time t. This contradicts the formal definition of the VWit violation [13]. □

5.3.3 SBQ lock-freedom

Lock-freedom of enqueue operations follows from the semantics of CAS (and TxCAS): out of all the enqueuers attempting to append a new node, one must succeed. The dequeue operation is lock-free because if it performs an infinite number of steps then it traverses through infinitely many nodes and fails to extract each time. Since nodes are added to the queue with non-empty baskets, this implies that other dequeues are successful.

5.3.4 Scalability of SBQ operations

Since asymptotically TxCAS has constant latency (§ 3.3), the latency of an SBQ's enqueue operation is asymptotically dominated by the basket's initialization time at the beginning of the operation, which is amortized $O(B/T)$, where B is the basket size and T is the number of enqueuers (§ 5.2.2). For programs that fix B to be the maximum number of threads, the latency therefore monotonically decreases with T and for $T \approx B$ it is $O(1)$. Programs that dynamically set the basket size to T enjoy $O(1)$ enqueue latency at all concurrency levels. (Our evaluation conservatively uses a fixed B.) This analysis ignores contention with dequeues that may lead to multiple basket insertion attempts, which are rare in our experiments.

In contrast, the dominating factor of SBQ's dequeue operations is the basket's contended FAA, whose latency is linear in the number of concurrent dequeuers. This means that SBQ's dequeue performance is comparable to state-of-the-art FAA-based queues [31, 41] but is similarly non-scalable.

6 Evaluation

6.1 Experimental setup

We use a server with two Intel Xeon E5-2699 v4 processors. Each processor has 22 cores, each multiplexing 2 hardware threads, allowing up to 44 threads per processor. Each core has private L1 and L2 caches; the inclusive L3 cache is shared.

In our experiments, each thread acts as either a producer calling enqueue or a consumer calling dequeue. We evaluate three workloads: producer-only, consumer-only, and a mixed producer/consumer workload. Each thread is pinned to some hardware thread, and as explained in § 4.3, all threads of the same type are pinned to the same processor.

We measure the time it takes until each thread completes $4 \cdot 10^6$ operations. We report averages of 5 executions and error bars indicating standard deviation. Contention is consistent throughout each experiment: the relative difference between the longest and shortest thread execution times is $\leq 5\%$. We use the Memkind [4, 18] scalable memory allocator. All implementations are in C11 and use memory reclamation.

We compare SBQ (**SBQ-HTM**) to the following prior queue implementations: (1) **WF-Queue**, Yang and Mellor-Crummey's FAA-based wait-free queue [41]; (2) **CC-Queue**, Fatourou and Kallimanis's combining queue [7]; and (3) **BQ-Original**, the original baskets queue [17]. We use the original authors implementation of CC-Queue and WF-Queue (including their memory reclamation schemes).

To isolate the impact of TxCAS from our scalable basket, we additionally compare to **SBQ-CAS**, a version of SBQ whose `try_append` uses CAS, and has the same delay as TxCAS placed between lines 2 and 3 of the function. In both

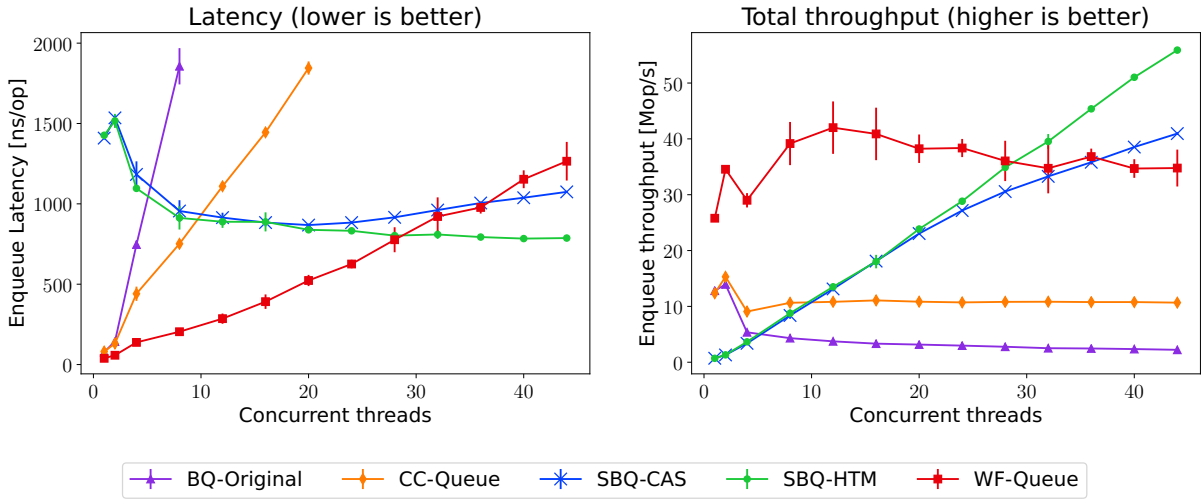


Figure 5. Enqueue operations: latency & throughput.

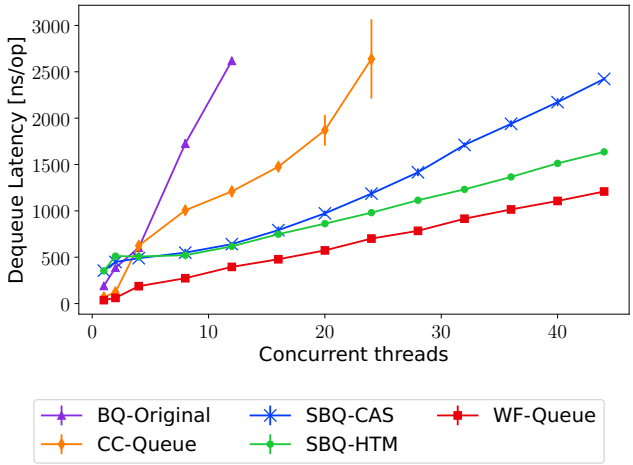


Figure 6. Dequeue operations: latency.

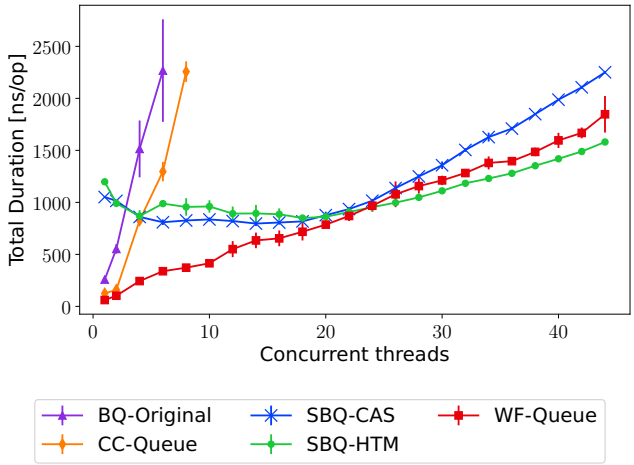


Figure 7. Mixed benchmark: normalized duration.

SBQ versions, the basket size is 44 in all experiments but basket emptiness (Algorithm 9, line 7) is determined using the number of enqueuers in the experiment.

To our knowledge, WF-Queue is the fastest queue in the literature, despite offering a strong wait-free guarantee. The reason is that it uses a fast-path/slow-path approach [23] that triggers costly wait-free helping only when operations fail to make progress. In practice, operations make progress, and so WF-Queue is not penalized by its wait-freedom.

6.2 Experimental results

Producer-only workload Figure 5 shows average enqueue latency when filling an initially empty queue. (We cap the latency graph at a certain latency point, so as to not obscure useful information.) For comparison, we also show the throughput (aggregated operations per second) obtained. SBQ-HTM exhibits linear scalability—beyond 10 threads its latency curve is close to constant. SBQ-CAS behaves similarly at low concurrency, but stops scaling beyond 20 threads. SBQ-HTM latency is dominated by TxCAS delays. As concurrency grows, the probability that when a TxCAS starts a delay there is another pending TxCAS that will soon abort it grows, and thus the time TxCASs spend performing delays decreases as concurrency grows.

In contrast to the SBQ variants, all other queues do not scale, as evidenced by their growing latencies and flat throughput curves. As a result, SBQ-HTM outperforms WF-Queue from 32 threads onwards, obtaining 1.6× its throughput (equivalently, 0.625× its latency) at 44 threads.

Consumer-only workload Here, consumers dequeue from a non-empty queue. (We pre-fill the queue using concurrent producers with enough elements so that it does not get empty.) Figure 6 shows dequeue latency (we omit the throughput graph, due to space constraints). SBQ-HTM outperforms CC-Queue, BQ-Original, and SBQ-CAS, but not WF-Queue. Unlike the enqueue operation, SBQ-HTM's dequeue operation does not scale. The reason is that our basket's dequeue is bottlenecked by a contended FAA, similarly

to WF-Queue. Compared to WF-Queue, SBQ-HTM latency is worse by a constant factor of 1.4× at high thread counts. This happens because an SBQ-HTM dequeue may perform multiple contended FAAs, as it can arrive at an empty basket before its `empty` bit is set. In WF-Queue, in contrast, dequeues perform one FAA per operation, since it uses counters that regulate the entire queue.

Mixed workload Figure 7 reports the average time required to complete a benchmark in which producers enqueue $4 \cdot 10^6$ elements and consumers dequeue $4 \cdot 10^6$ elements (in total) on a queue initially containing $2 \cdot 10^6$ elements. We dedicate one processor for producers and one for consumers.

As before, the SBQ variants and WF-Queue are the best performers. Due to the FAA bottleneck in the SBQ baskets, the scalability trends are similar to the consumer-only workload. However, due to SBQ-HTM's scalable enqueues, it outperforms WF-Queue from 48 threads, and at 88 threads it achieves 1.16× better throughput (or 0.86× better latency).

7 Related work

CAS-based queues The baskets queue [17] is one of several attempts to improve the scalability of the Michael-Scott queue [27], by either reducing the number of CASs performed by enqueues [24], applying the elimination technique [29], or using batching to amortize CAS cost [28]. Kogan and Petrank obtained efficient wait-free CAS-based queues [22, 23]. Still, all these queues rely on contended CASs, and so are not scalable (§ 3).

FAA-based queues Some early queues had both contended FAA and CAS [5, 35, 40]. Morrison and Afek proposed LCRQ, a lock-free queue whose sole contended operation is FAA [31]. We call this property *FAA-only*. Yang and Mellor-Crummey proposed a wait-free FAA-only queue that outperforms LCRQ due to a custom memory reclamation algorithm [41]. While FAA-only queues avoid wasted work due to CAS failures, the use of a contended FAA makes them non-scalable (§ 3).

Combining-based queues Combining is a technique in which a *combiner* thread performs all the operations currently pending by other threads. Flat combining [12] does not use CAS, but the latency of the serial work performed by the combiner exceeds that of a contended RMW. Accordingly, the fastest combining-based queues, SimQueue [6] and CC-Queue [7], are based on contended FAA and SWAP, respectively. These queues are not scalable, and are outperformed by the nonblocking FAA-only queues discussed above.

Relaxed queues Scalable queues can be obtained by relaxing the linearizability correctness condition [1, 10], but such relaxed queues are inapplicable to some applications [30]. Moreover, our focus is the intellectual challenge of obtaining a scalable queue *without* compromising on linearizability.

Scalable synchronization hardware Several works propose hardware support for efficient synchronization, by delaying cache coherence transactions to prevent failures of CAS or lock acquisitions [11, 32] or by forwarding the data accessed in a critical section while lock ownership is being transferred [33]. These proposals eliminate CAS failures but still serialize *all* CAS operations, and therefore do not enable a *scalable* baskets queue implementation.

LL/SC Load-link/store-conditional (LL/SC) instructions [21] are an alternative to atomic RMW instructions. An SC conditionally writes to a memory location previously read from with LL, provided the location was not written to by another core since the LL. An LL/SC implementation has scalable failures if it acquires Shared ownership of the target location on LL and upgrades to Modify ownership on SC. Unlike TxCAS, however, the failure occurs only at the SC instead of when the location gets updated, leading to wasted cycles. This wasted cycles problem has actually motivated proposals for LL to acquire Modify ownership [32], serializing failures.

8 Conclusion & future work

This paper makes a first step towards designing a scalable linearizable queue. Our core insight is that a CAS that is carefully implemented in an HTM transaction inherently scales better than a standard CAS. Based on this insight, we design TxCAS, an HTM-based CAS that realizes these scalability benefits on current Intel processors. We use TxCAS and a new scalable basket in the design of SBQ, a scalable version of the baskets queue. Our empirical evaluation on a dual-processor Intel server with 44 cores (88 hyperthreads) shows that at high concurrency levels, SBQ outperforms the fastest queue today by 1.6× on a producer-only workload and by 1.16× on a producer/consumer workload.

Interesting future work remains to address SBQ's limitations, for example, by designing a basket with scalable dequeue operations and by improving commercial HTM implementations to address the tripped writer problem.

Artifact evaluation results

We wish to clarify that this paper did not receive a *results replicated* artifact evaluation badge not because the artifact experiments produced different results than those we report, but due to technical issues that prevented artifact reviewers from running the experiments. (Namely, reviewers did not have access to Intel machines with HTM, or encountered crashes in setup code—before the experiment runs—that we were not able to debug remotely.)

Acknowledgments

This research was funded in part by the Israel Science Foundation (grant 2005/17) and by the Blavatnik Family Foundation. We thank the anonymous reviewers for their insights.

References

[1] Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-linearizability: Relaxed Consistency for Improved Concurrency. In *OPODIS 2010*. http://dl.acm.org/citation.cfm?id=1940234.1940273

[2] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *PLDI 2008*. http://doi.acm.org/10.1145/1375581.1375591

[3] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. 2013. Robust Architectural Support for Transactional Memory in the Power Architecture. In *ISCA 2013*. http://doi.acm.org/10.1145/2485922.2485942

[4] Christopher Cantalupo, Vishwanath Venkatesan, Jeff R Hammond, and Simon Hammond. 2015. User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies. (2015).

[5] Robert Colvin and Lindsay Groves. 2005. Formal Verification of an Array-Based Nonblocking Queue. In *ICECCS 2005*.

[6] Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A highly-efficient wait-free universal construction. In *SPAA 2011*. http://doi.acm.org/10.1145/1989493.1989549

[7] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the combining synchronization technique. In *PPoPP 2012*. http://doi.acm.org/10.1145/2145816.2145849

[8] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge, Computer Laboratory, University of Cambridge, Computer Laboratory.

[9] J. R. Goodman and H. H. J. Hum. 2004. *MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects*. Technical Report. University of Auckland.

[10] Andreas Haas, Michael Lippautz, Thomas A. Henzinger, Hannes Payer, Ana Sokolova, Christoph M. Kirsch, and Ali Sezgin. 2013. Distributed Queues in Shared Memory: Multicore Performance and Scalability Through Quantitative Relaxation. In *CF 2013*. https://doi.org/10.1145/2482767.2482789

[11] Syed Kamran Haider, William Hasenplaugh, and Dan Alistarh. 2016. Lease/Release: Architectural Support for Scaling Contended Data Structures. In *PPoPP 2016*.

[12] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010*. http://doi.acm.org/10.1145/1810479.1810540

[13] Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. 2013. Aspect-Oriented Linearizability Proofs. In *CONCUR 2013*. http://doi.org/10.1007/978-3-642-40184-8_18

[14] Maurice Herlihy. 1991. Wait-free synchronization. *TOPLAS* 13 (Jan. 1991), 124–149. Issue 1. http://doi.acm.org/10.1145/114005.102808

[15] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA 1993*. http://doi.acm.org/10.1145/165123.165164

[16] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS* 12, 3 (July 1990), 463–492. http://doi.acm.org/10.1145/78969.78972

[17] Moshe Hoffman, Ori Shalev, and Nir Shavit. 2007. The Baskets Queue. In *OPODIS 2007*. http://doi.org/10.1007/978-3-540-77096-1_29

[18] Intel. 2018. Memkind (version 1.8.0). https://github.com/memkind/memkind

[19] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-041. Intel Corporation.

[20] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-069US. Intel Corporation.

[21] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. 1987. *A New Approach to Exclusive Data Access in Shared Memory Multiprocessors*. Technical Report UCRL-97663. Lawrence Livermore National Laboratory.

[22] Alex Kogan and Erez Petrank. 2011. Wait-free Queues with Multiple Enqueuers and Dequeuers. In *PPoPP 2011*. http://doi.acm.org/10.1145/1941553.1941585

[23] Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-free Data Structures. In *PPoPP 2012*. http://doi.acm.org/10.1145/2145816.2145835

[24] Edya Ladan-Mozes and Nir Shavit. 2004. An optimistic approach to lock-free FIFO queues. In *Distributed Computing*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 117–131. http://doi.org/10.1007/978-3-540-30186-8_9

[25] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[26] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE TPDS* 15, 6 (June 2004), 491–504. http://dx.doi.org/10.1109/TPDS.2004.8

[27] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *PODC 1996*. http://doi.acm.org/10.1145/248052.248106

[28] Gal Milman, Alex Kogan, Yossi Lev, Victor Luchangco, and Erez Petrank. 2018. BQ: A Lock-Free Queue with Batching. In *SPAA 2018*. http://doi.acm.org/10.1145/3210377.3210388

[29] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. 2005. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA 2005*. http://doi.acm.org/10.1145/1073970.1074013

[30] Adam Morrison. 2016. Scaling Synchronization in Multicore Programs. *CACM* 59, 11 (Oct. 2016), 44–51. http://doi.acm.org/10.1145/2980987

[31] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *PPoPP 2013*. http://doi.acm.org/10.1145/2442516.2442527

[32] Ravi Rajwar, Alain Kägi, and James R. Goodman. 2000. Improving the throughput of synchronization by insertion of delays. In *HPCA 2000*.

[33] Ravi Rajwar, Alain Kägi, and James R. Goodman. 2003. Inferential Queueing and Speculative Push for Reducing Critical Communication Latencies. In *ICS 2003*.

[34] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *CACM* 53, 7 (July 2010), 89–97. http://doi.org/10.1145/1785414.1785443

[35] Niloufar Shafiei. 2009. Non-blocking Array-Based Algorithms for Stacks and Queues. In *ICDCN 2009*.

[36] Ronak Singhal. 2008. Inside Intel Next Generation Nehalem microarchitecture. In *HotChips 2008*. http://www.hotchips.org/wp-content/uploads/hc_archives/hc20/3_Tues/HC20.26.630.pdf

[37] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. *A Primer on Memory Consistency and Cache Coherence* (1st ed.). Morgan & Claypool Publishers.

[38] Paul Sweazey and Alan Jay Smith. 1986. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. In *ISCA 1986*. http://dl.acm.org/citation.cfm?id=17407.17404

[39] R. K. Treiber. 1986. *Systems Programming: Coping With Parallelism*. Technical Report RJ 5118. IBM Almaden.

[40] Philippas Tsigas and Yi Zhang. 2001. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *SPAA 2001*. http://doi.acm.org/10.1145/378580.378611

[41] Chaoran Yang and John Mellor-Crummey. 2016. A Wait-free Queue As Fast As Fetch-and-add. In *PPoPP 2016*. http://doi.acm.org/10.1145/3016078.2851168

[42] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-performance Computing. In *SC 2013*. http://doi.acm.org/10.1145/2503210.2503232

[43] Dimitrios Ziakas, Allen Baum, Robert A. Maddox, and Robert J. Safranek. 2010. Intel QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *HOTI 2010*. http://dx.doi.org/10.1109/HOTI.2010.24