Technical Report No. 2013:06

Enhancing Concurrent Data Structures with Concurrent Iteration Operations: Consistency and Algorithms

YIANNIS NIKOLAKOPOULOS
ANDERS GIDENSTAM
MARINA PAPATRIANTAFILOU
PHILIPPAS TSIGAS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY / GÖTEBORG UNIVERSITY
Göteborg, Sweden, 2013
Abstract
Concurrent data structures provide the means to multi-threaded applications to share data. Data structures come with a set of pre-defined operations, specified by the semantics of the data structure. In the literature and in several contemporary commonly used programming environments, the notion of iterations has been introduced for collection data structures, as a bulk operation enhancing the native set of their operations. Iterations in several of these contexts are treated as sequential in nature and may freeze the data structure while operating or provide a variety of consistency guarantees when running concurrently with the native operations of the data structures. In this work we study iterations in concurrent data structures with respect to their coexistence with the native operations of the data structures. In this work we study iterations in concurrent data structures with respect to their coexistence with the native operations of the latter and the guarantees that they provide under concurrency. Besides linearizability, we propose a set of consistency specifications for such bulk operations, including also concurrency-aware properties by building on Lamport’s systematic definitions for registers. By using queues, fixed-domain sets and composite registers as case-studies of underlying objects, we show a set of constructions of iteration operations, satisfying these properties. Besides the trade-off between consistency and throughput, we demonstrate the trade-off between the overhead of the bulk operation and possible support (helping) by the native operations of the data structure. We show a set of algorithms that demonstrate these and study the implications on the efficiency of the implementations.

1. Introduction
Concurrent data structures are a valuable tool for developing parallel and concurrent applications and systems. In particular, concurrent collections have become an essential part of software libraries in order to extend the support for parallelism and concurrency [1–3]. The research community has provided several efficient concurrent implementations for the commonly used data structures. However, when such solutions become part of software libraries more functionalities may be added depending on the targeted programming language or runtime environment. One such functionality is that of traversing through all the elements of the data structure, commonly called iteration or enumeration and respectively provided via constructs such as iterators, enumerators or generators.

In the programming languages context, iterators have been widely used both for user-level convenience, mainly for assigning values to a for-loop and as building blocks for other language functionalities (e.g. Python [4]). Therefore there has been noticeable support in the language level mainly in two different ways, the object-based iterators and the control-based ones [28]. In the object-based iterators the state of the traversal of the main collection has to be logged in another related data structure. The subsequent steps of the iteration procedure have to be decided based on this recorded state. In these cases no special language support is needed but the more complex the data structure is, the more difficult the implementation of the iterator gets. The control-based iterators on the other hand are based on specific language constructs and mechanisms that give value to a loop variable and suspend the execution until another value is needed. Naturally this introduces a considerable overhead.

The main issue with all the above methods and their interplays is that concurrency is not taken under consideration. The semantics of iterators change when shifting from sequential executions to concurrent ones. Suspending an execution is the exact opposite to what we ask for and expect in a concurrent environment. An important issue that arises is that while in the sequential case it is easy to give access directly to the memory locations of the traversed elements, this is not the case when the data structure is subject to changes by concurrent threads. One solution is to allow read-only operations to access copies of the elements with snapshot like semantics. Recent implementations in environments with immutable objects, provide for "free" access to copies of the data structure that are not yet garbage collected. Prokopec et al [27] in their concurrent tries implementation give a constant time snapshot, bearing though the cost of immutability.
The semantics of an iteration in a concurrent environment are closely related to the calling application. With this in mind and combined with the motivation above, we focus on snapshot iterations that directly return references to the elements of the data structure. It is up to the library programmer to decide whether these objects should be copied or directly accessed at that point.

Regarding data structure implementations, allowing operations to execute concurrently (e.g. by implementing them through fine-grain synchronization methods, employing no locks, AKA lock-free/wait-free methods [17], possibly combined with fine-grain locking) enables to utilize the parallelism in a system (e.g. operations may execute on different processors or cores), with an anticipated benefit in efficiency in terms of throughput and latency.

It is understood that the above introduce non-trivial trade-offs among the throughput, the consistency of the outcomes of the operations on the data structure and the ease-of-use from the programmer’s perspective. Stronger consistency guarantees such as linearizability [18] and sequential consistency [20] which guarantee that all accesses are observed in a total order consistent with the definitions of the sequential specification, may imply more intuitive usage from the point of view of the programmer who uses the data structure. At the same time, they usually imply larger complexity on the algorithmic design and implementation of the data structure’s operations.

Some contemporary implementations in well-used environments (cf. Section 4) provide weaker consistency, but their semantics (consistency properties) have not been described to match properties described in the literature [17].

Taking the above into account, together with the fact that iterations are bulk operations on the data structures, natural questions involve the strength and the cost of the required consistency in the presence of concurrency. Does it make sense to have concurrency-related behavior description for the consistency specifications? Alternatively, does linearizability have to be very expensive?

In this work we study these questions. We propose a set of consistency specifications for iteration operations, including also concurrency-aware properties, by building on Lampert’s definitions for registers [21]. We propose that universal constructions of objects provide the means for universal iteration implementations. Further, we show two case-studies: (i) a simpler one using fixed-domain sets and composite registers where we demonstrate how existing methods in the literature can be adapted to implement iterations satisfying the specifications proposed (ii) we further illustrate the framework and the trade-offs through more interesting case-studies of structures as concurrent queues. We show a set of constructions of snapshot-iteration operations for queues, satisfying the specifications. Besides the trade-off between consistency and throughput, we demonstrate a trade-off between the overhead of the bulk operation and possible support (helping) by the native operations implementation of the data structure.

In sections 2 and 3 the modeling of the system and the studied problem is presented, as well as definitions of consistency specifications. In section 4 we relate with preexistent work in the literature and the state of the art in related methods in commonly used programming environments. We put the problem in the context of universal constructions for concurrent data structures in section 5. Section 6 is a “pedagogical” application of our framework of consistency definitions on bounded-domain sets and composite registers. Our next case study is presented in section 7 where we show iteration algorithms for concurrent queues, satisfying linearizability or concurrency-aware consistency requirements and providing several progress guarantees. In the experimental study in section 8 we study the impact of the iteration on native data structure operations, throughput and consistency differences of different iterators including the implementations of our algorithms based on the operations implementation of the data structure.

2. System model

The system consists of a set of processes that communicate via shared memory. It also provides implementations of a concurrent container abstract data type (ADT) that represents a collection of items. The ADT includes update operations that can add or remove items in the collection in accordance to the specification of the ADT (e.g. FIFO, LIFO).

A run ρ (or history) is an execution of an arbitrary number of operations on the ADT according to the respective protocol that implements the ADT. In such an execution, for each operation a on the ADT there exists a time interval [sa, fa] called its duration. The points sa and fa are the starting and finishing times respectively of a. There is a precedence relation on operations which is a strict partial order (denoted by →). For two operations a and b, a → b means that operation a ends before operation b starts. If two operations are incomparable under →, they are said to overlap. We consider only runs of complete executions, where there are no pending operations.

A history is linearizable [18] if it is equivalent to some sequential history including the same operations, whose total order respects the partial order →. Thus, a run ρ of a linearizable ADT implementation induces a set of total orders that extend the partial order → in a compatible way with the sequential semantics of the ADT. For every operation a in a linearizable run ρ we define stateρ(a) as the postcondition of the ADT after the operation a, in a prefix of a linearization of ρ that ends with a. In this notation we will drop the parameter ρ when it is clear from the context.

3. Problem statement and definitions

Our purpose is to extend the ADT and linearizable implementations of them, adding bulk operations that will return a state of the ADT and in particular the items that are contained in it. This state should comply with at least one out of a variety of consistency requirements. We will call these iteration operations. Depending on the implementation environment, the returned state could be a copy of the items in the ADT or pointers to them.

For a given linearizable ADT implementation consider the set of all its possible states: a state S from this set is defined to be valid with respect to a linearizable execution ρ, if ∃ prefix of ρ ending with an operation a ∈ ρ, denoted as a = vn(S), such that S = statev(a). An iteration operation Itr returns a view of the state of the ADT that is meaningful for the iterator, typically this is the set of items stateItr, that are contained in the ADT. An extended run γ of ρ is the run that extends ρ with the respective iteration operations. Respectively, given such a run γ, we can define the reduced run ρ that does not include the iteration operations.

3.1 Consistency specifications

Building on the consistency related definitions by Lamport [21] and Herlihy and Wing [18] we define the following properties, assuming that ρ is linearizable:

The safeness property states that any iteration Itr ∈ ρ not overlapping with any other operation in ρ must return a valid state S, such that: if a = vn(S) (ρ is the reduced of γ), Itr → a and ρ a′ : a → a′ → Itr. If Itr is overlapping any operations of ρ, it can return any arbitrary state of the object.

Let Itr ∈ γ be an iteration operation possibly overlapping some a ∈ ρ. Itr is regular if it returns a valid state S, which is neither “overwritten” nor “future” in ρ, i.e. if a = vn(S) (ρ is the reduced of γ), Itr → a and ρ a′ : a → a′ → Itr.
The monotonicity property states that for any two iteration operations \( I_{tr_1}, I_{tr_2} \) that return valid states \( state(a_1) \) and \( state(a_2) \) respectively, if \( I_{tr_1} \rightarrow I_{tr_2} \), then \( a_1 \equiv a_2 \).1

\( I_{tr} \) is linearizable if it is regular and the extended run \( q \) is equivalent with some sequential history, that includes the same operations, whose total order respects the total order of the original \( \rho \).

Using the above definitions we can prove the following lemma.

**Lemma 1.** An iteration operation \( I_{tr} \in q \) not overlapping with any other iteration operations is linearizable if it satisfies the regularity and monotonicity properties.

**Proof sketch:** We assume a linearizable execution \( \rho \) extended to an execution \( q \) that includes non-overlapping iteration operations satisfying the properties of regularity and monotonicity. We can linearize every iteration operation right after the linearization point of the operation \( a \) that matches the regularity definition. Regularity guarantees that it is “no future” state, not “overwritten” and therefore each iteration operation \( I_{tr} \) is correctly ordered with any overlapping operation \( a \). In case that one or more operations \( a \) overlap with two subsequent iterations, monotonicity guarantees the total order.

Furthermore, motivated by implementations in contemporary programming environments (cf. Section 4) we define one consistency guarantee lying between safeness and regularity:

A weakly regular \( I_{tr} \) satisfies all the properties of safeness when no overlapping operations exist. In case it does overlap with some operations of \( \rho \), \( I_{tr} \) returns a state \( S \) that is not necessarily valid. \( S \) will result from an execution of the prefix of \( \rho \) before the point \( s_{itr} \), that \( I_{tr} \) started, extended with an arbitrary subset of the update operations in \( q \) that are linearized within the interval \([s_{itr}, f_{itr}]\).

3.2 Progress guarantees

For completeness and self-containment we include standard definitions in the literature (cf. [13, 16]) about progress guarantees in a concurrent ADT implementation. Wait-freedom is the strongest property and ensures that any process can complete an operation on the object in a finite number of its own steps, independently of failures of any other process. In a lock-free object implementation it is ensured that at least one of the contending operations makes progress in a finite number of steps. It is common in lock-free implementations of ADTs that an operation is implemented through fail-retry loops: a retry needs to take place due to one or more interfering operations among the contending ones. A weaker guarantee is obstruction freedom. In this case progress is only ensured in absence of contention among the operations on the shared ADT.

4. Related work and state of the art

Iterators have been an important design pattern in object oriented languages used to provide sequential access to a collection of objects without exposing its underlying representation. Noble [25] presents a range of designs for iterator objects studying through their encapsulation properties, with which iterators can be by definition at odds. Boyland et al [8] discuss iterator validity in terms of ownership rights and alias control. None of the above though takes concurrency and multithreaded executions under consideration.

To the best of our knowledge little work has been done in bridging the gap between iterators and concurrency. Prokopev et al [27] are the first to introduce a concurrent data structure integrating an iteration operation based on snapshots. It is implemented in Scala using the DoubleCompareSingleSwap (RDCSS) software primitive [12] and considered to be a fixed time operation. This stands since it is implemented as a persistent data structure with immutable states where every update needs to recreate the data. Thus, roughly speaking, a snapshot operation borrows “for free” the previous generation of the data structure, which needs to be recreated at every operation. Another aspect of iterators in contemporary systems is addressed in [26] by Prokopev et al, where the authors introduce a framework for parallelizing iteration operations in collection data structures. Instead of providing sequential access to the elements of the collection they enable parallelization allowing multiple threads to access the elements and thus distributing the workload. The framework does not focus on the interaction of the iteration with other concurrent operations of the collection but gives direct support for parallel programming patterns such as map/reduce or parallel looping.

It is not uncommon for contemporary programming environments, such as C++, Java and the .NET platform, to include collection data structures in their standard libraries that support concurrent operations. These collections often support iteration over their contents while other operations may concurrently change the data structure. What kind of consistency do they offer?

**Java:** The standard library of the Java Platform Standard Edition 2 contains a number of concurrent collection or container data types that support iteration over their contents concurrently with operations that modify them. The documentation classifies the consistency of an iteration of a particular container data type as either snapshot style, described as capturing the state of the container at the point in time the iterator was created, or weakly consistent, for the ConcurrentLinkedQueue described as “returning elements reflecting the state of the queue at some point at or since the creation of the iterator” and similarly for other data structures. A study of the source code for the ConcurrentLinkedQueue reveals that the description is not entirely accurate: the result may be a mixture of the states that occur during the iteration and can include items removed early during the interval together with items added late, i.e. not reflecting the state at any particular point in time.

**.NET:** The .NET Framework class library [3] contains a number of concurrent container data structures. All of them support iteration of their contents concurrently with operations that modify them. The library documentation classifies what an iteration of a particular container type provides as either a moment-in-time snapshot or not a moment-in-time snapshot.

**Intel Threading Building Blocks:** It is a library for parallel programming in C++ [1]. The library contains a number of concurrent container data structures, some of which supports iteration over their content concurrently with other operations on the container. However, they only support concurrent use for a subset of their operations and only three, concurrent unordered_map, concurrent_unordered_set and concurrent_vector, support insertion concurrently with iteration but does not promise any particular level of consistency.

In summary, Java’s snapshot style and .NET’s moment-in-time snapshots can be expected to be linearizable (or nearly so), while the consistency of Java’s weakly consistent iterators, which vary in

---

1 Dwork et al [9] in the context of composite registers used two weaker variations of monotonicity, one for scans and one for updates. It is useful to mention that in this context a regular \( I_{tr} \) also satisfies the monotonicity of updates property, i.e. for two subsequently linearized updates, an \( I_{tr} \) that “Observes” the effects of the latter update, should also “Observe” the effects of the preceding update.

2 Due to space limitations, we only provide proof sketches wherever needed. The proofs will appear in the full version of the paper.

3 Version 1.7.0.09.
detail for each actual implementation, and the unspecified thread-safe iterators in .NET and TBB is weakly regular.

5. Iteration and the Universal methodology

By reflecting on the literature on concurrent ADT implementations a natural question is the following: Can universal constructions, such as [14] and [6], that transform any sequential data object to a lock-free or wait-free linearizable concurrent one support iteration?

Observation 1. The universal constructions transform or “lift” the operations of the sequential data object to lock-free or wait-free linearizable operations on the resulting concurrent object. Hence, if a sequential iteration operation can be provided for a particular sequential data object, it, too, can be lifted to the resulting concurrent data object just like any other operation of the sequential data object.

Will the addition of an iteration operation to a data object incur considerable additional overhead when using one of the universal constructions? Observe that these constructions work by applying a sequential operation to a snapshot of the current object state (or relevant parts thereof), thereby producing a new candidate state to be committed (unless another operation concurrently has succeeded in changing the object’s state in an incompatible way). With the memory management used by the original constructions collecting a consistent snapshot of the whole structure, as an iteration would need, could be very expensive. Taking the construction in [14] as an example, it uses counter assisted double collect to read a consistent snapshot of the object which can be very expensive indeed for a large data structure experiencing frequent concurrent updates. However, using more recent developments in non-blocking memory management, such as [15, 22], and sequential operations expressed in functional style (as proposed already in [14]) to produce the new state (with e.g. [15] allowing the new state to reuse/share unchanged parts of the old state) the cost of an iteration operation would be much smaller, possibly no more than reading the shared reference to the current state of the object and then traversing the captured state.

6. Iterations on bounded-domain sets and composite registers

In this section we provide a simpler case study that demonstrates the framework and the tractability of the defined consistency specifications.

Let us consider a fundamental data structure, a set of elements coming from a finite domain. A straightforward implementation of these types of sets can be derived from the well-studied area of composite registers. Every item of the finite set domain can be represented by an entry register of a shared array composite register. Then the add and remove operations of the set are merely update operations on the respective entry register. Thus, the problem of iterating the data structure concurrently with update operations is transformed to that of obtaining a snapshot of the shared memory register [5, 7, 9, 10, 19]. The question that arises now is with which of the presented consistency specifications such iterations comply.

Safety The weakest of these requirements as seen in section 3 is safety. In that case only snapshots that do not overlap with any other operation are guaranteed to be correct. A straightforward implementation would be simply reading and returning the values of the respective registers.

Regularity and Monotonicity Moving to more useful guarantees in practice, regularity combined with monotonicity is the next to come. In the context of snapshots of composite registers it has also been known as the time-lapse property [9]. In fact as seen in lemma 1 this condition satisfies linearizability but only for iterations non-overlapping with each other. In cases of possibly overlapping iterations only regularity is guaranteed. Wait-free protocols satisfying such guarantees can be seen in [9, 19], using handshaking methodology.

Linearizability Finally, linearizability is the strongest guarantee, according to which the iteration operation appears to have occurred instantaneously at some moment in time within its duration. A natural method is the use of the double collect method [5]. If a reader manages to collect twice the same timestamped values of all registers then this collection is a snapshot, guaranteeing though only obstruction freedom. Extending this idea with using the updaters of the register to also do a double collect in order to help the snapshot operation, Afek et al. presented a wait-free solution that guarantees linearizability in [5]. More snapshot constructions can be seen in [10, 17].

7. Iterations on queues

Our next case-study is iteration operations in concurrent linked-list based queues. A representative construction that we will use as a running example is the linked-list implementation of the lock-free queue by Michael and Scott [24].

In this section, the algorithms are presented at a level of abstraction that can be followed independently of the details of the underlying construction. A discussion regarding implementation aspects and how they are connected with our used example will follow when needed in each case.

One of the issues that has to be addressed while lifting the abstraction level of the presentation is that of the returned references to the actual elements that are to be iterated. In the case of an unmanaged or simply managed memory environment, allowing access to the elements of the queue would imply that the accessing thread should have responsibility of deleting the nodes, violating this way the layers of abstraction of the abstract data type. A different solution would be to make a copy of the elements in another data structure in a new memory location which would be returned afterwards. Apparently such solutions introduce a cost linear to the size of the queue. Furthermore any possible memory reclamation techniques currently used should also be considered while designing such a solution.

On the other side, garbage collected environments provide a clearer interface, more flexibility and already paid costs that can be exploited in order to have smart solutions. In linked list based concurrent queue implementations (e.g. [24]), provided that dequeued nodes remain connected with the rest of the list until they are garbage collected, keeping references of two nodes is enough for having the whole chain of nodes that connects them at our disposal in order to either traverse it or copy it. Furthermore, garbage collection helps ensuring that the ABA problem is avoided.

To overcome these differences in the presentation of our iteration algorithms we consider a simple ADT where each of the iterated nodes is added. We will refer to this ADT as stateToReturn and we consider it providing the intuitive add and initialize methods. As explained above, the functionality of this ADT can be mapped to different techniques depending on the implementation environment.

We should also note that in the iteration methods presented in the rest of this section when we argue about the correctness of each method, as well as the progress guarantees of the objects, we always assume the correctness of the underlying linearizable, lock-free, concurrent queue implementation.
7.1 Linearizable Double-Collect-based iteration

This algorithm (Alg. 1) is inspired by classic constructions of snapshot algorithms with double collect [5, 17] (cf. section 6) with the additional observation that the queue’s inherent structure imposes an order amongst its elements. Hence, after the first collect it suffices to check Head and Tail’s next pointer instead of doing a second collect, thus significantly reducing the number of required operations. If the Head and Tail pointers have not changed, then due to the inherent order of the queue’s elements the rest of the queue will not have changed.

Algorithm 1 Double-Collct based linearizable iteration

1: while True do
2:   stateToReturn.initialize()
3:   curHead ← Head
4:   curTail ← Tail
5:   stateToReturn.add(curHead)
6:   curNode ← curHead.next
7:   stateToReturn.add(curNode)
8:   while curNode ≠ curTail do
9:      curNode ← curNode.next
10:     stateToReturn.add(curNode)
11:    if curHead = Head ∧ curTail = Tail then
12:       return stateToReturn

Correctness and progress guarantees

Theorem 1. Algorithm 1 produces a linearizable iteration and the linearization point is on line 4. 

Proof sketch: Any dequeue or enqueue operations linearized after the reads in lines 3 and 4 respectively, will cause the check that takes place in line 11 to fail and the loop to restart. Any enqueue operations that are linearized after the execution of line 2 will not be reflected in the read of line 4. □

Lemma 2. Algorithm 1 will fail to return only if it is interfered by an unbounded number of update operations.

Corollary 1. Algorithm 1 gives an obstruction-free implementation of an iteration operation on a lock-free queue implementation.

Theorem 2. A lock-free concurrent queue implementation extended with the operation described in Algorithm 1 remains lock-free.

Proof sketch: No shared variables of the original data structure implementation are changed by the iteration in algorithm 1. Therefore the iteration operation will not interfere with any of the original enqueue or dequeue operations of the underlying queue implementation. □

It is interesting to note that there can be distinguishable progress guarantees between the different components of a data structure implementation. In the theorems above we showed that while the entire queue implementation is lock-free, the iteration operation is obstruction-free with respect to the native enqueue and dequeue operations. That is, the iteration operation will fail to make progress while being obstructed by the enqueue or dequeue operations (cf. lemma 2). Careful selection of such designs can be proved valuable from the application perspective [23].

Implementation-related aspects In garbage collected environments this design can be further optimized by not actually doing a full collect of the elements but by just collecting the Head and Tail pointers. Since the references to the nodes where Head and Tail point to will be returned, it can be guaranteed that the chain of nodes will not be garbage collected. Building on [24] though (section 8), this does not suffice since there exists a possibility that the Tail pointer has fallen behind. For that reason the conditional statement in line 11 must also check for a non null next pointer of the node pointed by the Tail.

7.2 Weakly regular Scan&Return iteration

Independently of the ADT properties, a simple and intuitive algorithm would be to scan the entire collection, though such a naïve scanning of the collection and returning its state is ineffective in the general case. If the elements are accessed concurrently with the scanning procedure, an inconsistent state may be returned.

Specifically for the queue case we can achieve a weakly regular iteration by traversing the nodes of the list, starting from the head until we reach the tail. During that traversal any newer dequeues will not be captured, while some of the newer enqueue operations might be reflected to the state returned.

Algorithm 2 Scan&Return weakly regular iteration

1: curHead ← Head
2: curTail ← Tail
3: stateToReturn.add(curHead)
4: curNode ← curHead.next
5: stateToReturn.add(curNode)
6: while curNode ≠ curTail do
7:   curNode ← curNode.next
8:   stateToReturn.add(curNode)
9: return stateToReturn

Correctness and progress guarantees

Theorem 3. Algorithm 2 satisfies the weak regularity condition.

Proof sketch: In absence of interferences the algorithm will return a valid state of the data structure, the one after the most recent linearized enqueue or dequeue operation. In case enqueue operations are linearized in the duration of the iteration operation: any nodes enqueued between the execution of lines 1 and 2 will also be returned; enqueue operations linearized after the execution of line 2 will not be reflected in the returned state. Any nodes dequeued concurrently by updates that are linearized after line 1 will actually be returned and the dequeue operations will not be reflected in the returned state. □

Observation 2. An execution of an iteration operation as described in Algorithm 2 takes a number of steps linear to the size of the queue, with Head and Tail as read in lines 1 and 2 respectively.

Theorem 4. A lock-free concurrent queue implementation extended with the iteration operation described in Algorithm 2 remains lock-free.

Proof sketch: No shared variables of the original data structure implementation are changed by algorithm 2. Thus the iteration operation will not interfere with any of the enqueue or dequeue operations of the underlying queue implementation. □

Implementation-related aspects Implementations in garbage collected environments can be optimized again by collecting only the Head and Tail pointers. However this technique includes a risk for memory leak in case for example the first node iterated remains referenced by the calling application. Similar iterators are implemented using different memory management techniques.

An example in contrast with the above is the ConcurrentLinkedQueue in Java, also an implementation based on the lock-free queue
of [24]. When a node is dequeued it is also nullified and disconnected from the linked list by referencing itself. A similar scan and return type iterator is constructed just by collecting the Head and then advancing on the next pointers until the end of the queue is reached. This is also a weakly regular implementation as the state returned may reflect an arbitrary number of overlapping operations. There is though a qualitative difference between the two implementations. Our presented Scan&Return method (Alg. 2) returns a state that is a logically continuous chain of nodes as they were connected during the queue’s lifetime duration. On the contrary, in the iterator of the ConcurrentLinkedQueue, if a node is dequeued and self-referenced while the iterator accesses it, when the next method is called the iteration will move on to the current Head, skipping this way all nodes dequeued in between.

7.3 Linearizable iteration with helping techniques

Up to this point we presented algorithms for iteration that did not interfere with the update operations. Can a linearizable iteration be achieved in a more efficient way than double collect, by using helping techniques? The trade-off between no interference and helping by enqueue and/or dequeue will be explored in the next two algorithms. The iterator must be able to distinguish the appropriate nodes to return according to its specific linearization point. This can be done by using information provided by the enqueue and dequeue operations.

For presenting these algorithms we need to involve the actual construction of our test case based on [24], at a more detailed level than before. Still for presentation simplicity we skip some checks and parts of the code that are used for optimization but we include the key points of the algorithm.

The helping algorithms make use of two higher-level synchronization constructions, the DoubleCompareSingleSwap (DCSS) and MultipleCompareDoubleSwap (MCDS) (Algorithms 3 and 4) that can be implemented using the CompareAndSwap hardware primitive (CAS), e.g. following the lines of the constructions in [12]. These constructions conditionally update one or two memory words based on the value of a control word and the values of the memory words.

Algorithm 3 DCSS semantics

1: function DCSS(addr1, old1, addr2, old2, new2)
2: if *addr1 ≠ old1 ∨ *addr2 ≠ old2 then
3: return False
4: *addr2 ← new2
5: return True

Algorithm 4 MCDS semantics

1: function MCDS(addr1, old1, addr2, old2, new2, addr3, old3, new3)
2: if *addr1 ≠ old1 ∨ *addr2 ≠ old2 ∨ *addr3 ≠ old3 then
3: return False
4: *addr2 ← new2
5: *addr3 ← new3
6: return True

Implementation-related aspects Synchronization constructs like DCSS or fully double word CompareAndSwap CAS2 are rarely implemented in contemporary hardware architectures, but with some overhead they can be implemented in software using normal CAS [12]. The emerging support for hardware transactional memory implementations in newly arrived architectures (e.g. Intel Haswell, IBM System Z) may give the opportunity for MCDS or similar designs to be even more efficiently implemented.

7.3.1 Enqueuers helping iteration

In this construction, presented in Algorithms 5 and 6, the enqueue operation provides extra information in the queue nodes in order to help the iterators. This information enables the iterators to decide whether the node should be included in their result or not. This is accomplished by a shared counter, serving as a logical timestamp, that is increased by each iterator.

In an enqueue operation, in addition to the Tail pointer the value of the shared counter is also read. This value is then used to tag the appropriate field in the new node which is then enqueued using the DCSS primitive. DCSS assures that the node is connected to the list while the value of the shared counter has not changed. Otherwise, the enqueuer will retry.

An iteration operation is associated with a value of the shared counter as it begins. This is done by tying together the read of the Head pointer and the counter’s value with a double collect that confirms that the Head has not changed while atomically fetching and incrementing the counter’s value. The atomic FetchAndAdd (FAA) counter can be easily implemented using the CAS instruction. The iterator will stop returning nodes as soon as it encounters a node with a tag higher than the value fetched when it started or when it reaches the end of the list. As a result, no nodes that were enqueued after the successful update of the shared counter and the read of Head will be returned, making this the linearization point of the algorithm.

Algorithm 5 Enqueue operation with helping

1: initialize newNode
2: while True do
3: lastNode ← Tail
4: localTS ← read counter
5: newNode.enqueueTag ← localTS
6: if DCSS(&counter, localTS, &lastNode.next, null, newNode) then
7: CAS(&Tail, lastNode, newNode)
8: return
9: else
10: CAS(&Tail, lastNode, lastNode.next) ➔ Help update Tail

Algorithm 6 Iteration helped by enqueue operations

1: repeat
2: curHead ← Head
3: localTS ← FAA(&counter, 1)
4: until Head = curHead
5: curNode ← curHead
6: while curNode ≠ NULL ∧ curNode.enqueueTag < localTS do
7: stateToReturn.add(curNode)
8: curNode ← curNode.next
9: return stateToReturn

Correctness and progress guarantees The enqueue operation described in Algorithm 5 differs from the original in [24] in the part that it reads the shared counter and uses that to tag the enqueued node and also in the replacement of the original CAS operation with the DCSS in line 6. The latter remains the linearization point as it is the point where the enqueue operation takes effect.

For the iteration operation we show the following:

Theorem 5. Algorithm 6 produces a linearizable iteration and the linearization point is the last successful call of FAA on line 3.

Proof sketch: The condition check in line 4 will succeed if no dequeue operation interferes (thus changing the Head), after Head
has been read in line 2 and until the check takes place. Any enqueue operation that takes place after line 3 will be ignored by the iterator due to the check in line 6. Similar dequeue operations will be ignored since the node where the Head refers to, will already be copied locally in the last successful call of line 2. Since FAA is atomic no inconsistent states will be seen by possibly overlapping iteration operations.

Lemma 3. Algorithm 6 will fail to return only if it is interfered by an unbounded number of dequeue operations, while executing the loop in lines 1-4.

Corollary 2. Algorithm 6 is an obstruction-free implementation of an iteration operation with respect to the dequeue operations of the lock-free queue implementation.

Theorem 6. A concurrent queue implementation integrating the modified enqueue Algorithm 5 and the iteration operation of Algorithm 6 is lock-free.

Proof sketch: We assume the lock-free property of the original queue implementation. The enqueue operation will loop, additionally to the conditions in the original implementation, if the DCSS in line 6 fails when the shared counter is different from the previously fetched localTS value. That will happen only if an iteration operation has made progress in between. Similarly an iteration operation will fail to exit the loop in lines 1-4 only if a dequeue operation makes progress and updates the Head pointer.

So, by helping the iteration operation we managed to reduce the possibilities that such an operation will retry; this will take place only if dequeue operations make progress. Can we further share the load of helping with the dequeue operations also, in order to achieve even better progress guarantees for the iteration?

7.3.2 Helping by both enqueue and dequeue

In this construction, in addition to the enqueue operation (Alg. 5) also the dequeue assists the iterator (Alg. 7 and 8 respectively). Besides sharing the helping between enqueue and dequeue operations, this further reduces the iterator’s cost by eliminating the need for a double collect of the shared counter and the Head.

An iteration operation first reads the Head pointer and then atomically fetches and increments the value of the shared counter. The latter is the linearization point of the iteration. An enqueue operation is the same as in Alg. 5 in the previous construction. Thus it reads the value of the shared counter in addition to the Tail pointer and uses its value to tag the appropriate field in the node. The new node is then enqueued using DCSS to update the next pointer of the last node and at the same time to verify that the value of the shared counter has not changed. If any of the two has changed, the enqueue will retry. The dequeue operation also reads the counter and writes its value in the dequeueTag of the dequeued node, at the same time as it updates the Head pointer. The initial value of a node’s dequeueTag is set to a special infinity value. The updates are performed atomically by the MCDS operation.

These timestamps will help the iteration operation to ignore the effects of update operations that started after the atomic increment of the shared counter. In order though to make sure that the timestamp values written are the more recent ones, to help for the case the enqueue or dequeue operation is delayed, the CAS of the original enqueue and dequeue algorithms is replaced with DCSS and MCDS respectively.

Correctness and progress guarantees

Theorem 7. Algorithm 8 produces a linearizable iteration and the linearization point is on line 2.

Algorithm 7 Dequeue operation with helping

1: repeat ▷ Repeat until success
2: curHead ← Head
3: newHead ← curHead.next
4: value ← curHead.value
5: localTS ← read counter
6: until MCDS(&counter, localTS, &Head, curHead, newHead, &curHead.dequeueTag, ∞, localTS)
7: return value

Algorithm 8 Iteration helped by both enqueue and dequeue

1: curHead ← Head
2: localTS ← FAA(&counter, 1)
3: curNode ← curHead
4: while curNode ≠ null ∧ curNode.enqueueTag <= localTS do
5: if curNode.dequeueTag > localTS then
6: stateToReturn.add(curNode)
7: curNode ← curNode.next
8: return stateToReturn

Proof sketch: Enqueue operations that appear to take effect after the linearization point on line 2 will be ignored due to the check on line 4. Dequeue operations linearized also after the execution of line 2 will be ignored as the check on line 5 will succeed and the node will be included in the returned state. Since FAA is atomic no inconsistent states will be seen by possibly overlapping iteration operations.

For the progress guarantees of the construction, we show the following:

Lemma 4. A dequeue operation (Alg. 7) may be forced to retry due to an arbitrary overlapping iteration operation at most once.

Proof sketch: An overlapping iterator may atomically fetch and increment the shared counter after the dequeue operation has read it in line 5, causing the MCDS instruction to fail. However the localTS will be properly updated in the next iteration of the loop as the same iteration operation will not increase the shared counter again.

With a similar argument the respective lemma for the enqueue operation can be easily proved.

Lemma 5. An enqueue operation (Alg. 5) may be forced to retry due to an arbitrary overlapping iteration operation at most once.

Observation 3. The iteration operation (Alg. 8) finishes in a bounded number of its own steps as it loops only for a number of times bounded by the size of the queue at the time that line 2 was executed.

Theorem 8. A concurrent queue implementation integrating the modified enqueue and dequeue Algorithms 5 and 7 respectively along with the iteration operation of Algorithm 8 is lock-free.

The proof is easily derived by the the fact that the original implementation is lock-free and using lemmas 4 and 5 and observation 3.

8. Experimental study

In order to evaluate also the practical aspects of the presented constructions we performed a set of experiments that compare the time performance and the consistency behavior of the proposed iterators.

At first we want to investigate how the iteration operation impacts the execution of the "native" concurrent operations in the
shared data structure i.e. whether any slowdown is caused compared to the usual throughput of the data structure. Furthermore we want to study the relations of throughput of the different algorithmic implementations, achieved under different congestion conditions and with implementations that emulate different memory management techniques. Moreover, given that some of the presented iteration techniques fulfill different consistency requirements, we would like to study whether the outcome of the iteration operations is affected by such differences in practice.

8.1 Experiment setup

The experiments involve \( N - 1 \) “worker” threads running enqueue and dequeue operations at random with equal probability on a shared queue and 1 thread continuously iterating the queue. The experiments were run for values of \( N = 4, 8, 12, 16 \) and 24. Each of the “worker” threads had its own probability distribution assigned in all the experiments, deciding the operation that will be performed each time. As a very simple application while iterating the queue, the iterating thread reads the content of the nodes; the latter was also used to verify the consistency guarantees or the absence of them. We introduced some local workload in the “worker” threads between their operations in order to vary the contention levels among low, medium and high when no local work was done. In order to investigate how the size of the queue affects the behavior and performance of each iteration method in connection to the expected consistency and progress guarantees we also used as a parameter in the experiments the initial size of the queue. Specifically we used 3 different settings: empty, 2000 nodes and 5000 nodes, as with smaller sizes during runtime - and depending on the scheduling of the multiple “worker” threads - the queue could be emptied. The duration of every such experiment was 10 seconds.

We implemented the constructions in Java version 1.6.0.27 and specifically used the IcedTea6 1.12.6 OpenJDK Runtime Environment. The experiments were run on a workstation with 2 sockets of 6-core Xeon E5645 (Nehalem) processors with Hyper Threading (24 logical threads in total) running version 3.2 of the Linux kernel. We used methodologies well established in the literature [11] for our measurements. We performed 10 repetitions of each experiment and unless otherwise mentioned, we present the mean of these values along with 95% confidence intervals.

Our tested implementations consist of the lock-free queue by Michael and Scott [24] extended with the iterator constructions that are being presented in section 7. For easier and more consistent comparison the iteration methods presented were used in order to construct an object that implemented Java’s Iterator interface. Afterwards the thread that had created the Iterator consumed the iterated nodes by calling the appropriate next method until the iteration was finished.

More specifically we implemented and tested two versions of the linearizable double collect based iteration as seen in section 7.1, the DCOLLECT and DCOL-FULL. The first exploits the garbage collected environment of Java and collects only the references to Head and Tail since this is enough to prevent the chain of nodes from being garbage collected. The DCOL-FULL (full double collect) implementation emulates the delays that would be present in an unmanaged memory environment by putting the iterator to traverse the entire chain of nodes during its construction. Similarly the SNR and SNR-FULL implementations exploit garbage collection and emulate a full traversal respectively, of the Scan&Return method presented in section 7.2. For iterations with helping techniques, given the hardware support of the underlying architecture, we used a software version of the DCSS primitive based on [12] and implemented the iteration with helping from the enqueue operations (HELP-ENQ) that we presented in section 7.3.1. Finally we also compared with Java’s implementation of the Michael and Scott

\[ \text{Queue, ConcurrentLinkedQueue in the concurrent package. In this case too, an iterator is implemented following the Scan&Return style but with differences that we already discussed in section 7.2. We will refer to this implementation as JAVA-SNIR.} \]

8.2 Results discussion

**Overhead imposed to native operations** The first question that we investigated is how much an overhead for the performance of the data structure an iterator induces. In Figure 1 we compare the throughput of all the implementations, i.e. the number of successful enqueues and dequeues per millisecond, in high contention, when no iterator thread operates on the queue versus the case where one of the running threads iterates the queue continuously. One of the first observations is that with or without an iterator thread running, the HELP-ENQ implementation in all the cases achieves the lowest throughput. This is quite expected as the original queue protocol is modified to use the DCSS software primitive instead of the CAS.

![Figure 1. Throughput of queue implementations in high contention, with and without active iterators](image1)

![Figure 2. Throughput of queue implementations in medium contention, with and without active iterators](image2)
hardware primitive. Besides this overhead though, its performance does not change when iterations occur. In the case of empty initial queues, all the implementations are not significantly affected by the running iteration. In fact, in the cases of 4 and 8 threads, the throughput slightly improves when the iterator thread is running as there are 3 and 7 “worker” threads respectively. However when the queue is initialized with 2 or 5 thousand nodes the throughput of the DCOL-FULL construction is significantly affected. The reason is that in such large queue lengths and especially in high contention, the time spent to traverse the entire queue before checking for the second time the value of Head and Tail is enough for one of them to change and cause the traverse to restart. In fact in several experiments no DCOL-FULL iteration can complete. In low and medium contention no significant overhead from the iterator is noticed, thus we indicatively include only the empty and 2000 elements of the medium contention case in Figure 2.

Iteration performance In Figure 3 the performed iterations per millisecond in an initially empty queue under high contention environment are described. Comparing with Figure 4 (note the different scale) we can see that the initial size of the queue is inversely related with this measure. In the 2 and 5 thousand elements cases the queue is large in its entire life duration. Moreover, after every Iterator is constructed all the elements are consumed. Because of these, less number of iterations will be performed in the same time duration compared to the empty queue cases. As expected SNR manages the highest number of iterations per millisecond as it has the less overhead in constructing an iterator. It is interesting that while JAVA-SNR is a similar method (cf. Sect. 7.2) its throughput is lower than that of SNR. JAVA-SNR continues traversing the tail until it finds a null node. Our implementation of SNR finishes on the tail that was read in the beginning of Algorithm 2. Thus, as seen in the next paragraph there is a difference in the number of returned items between these two methods. The DCOL-FULL as expected consistently achieves the lowest throughput and even more so in the large queue sizes where it is quite common not to manage to achieve even one iteration. In the case of an initially empty queue the DCOLLECT also achieves fewer iterations per millisecond compared to the remaining ones, as it is more probable to fail and restart during an iteration. When the queue is initialized though (Figure 4), the overheads of full traversal in SNR-FULL and of the software primitive implementation in HELP-ENQ show up giving worse results. In lower contention the trends are similar with SNR and DCOLLECT usually achieving the best results and the DCOL-FULL the worst but still decent when compared with the high contention case.

Returned items Trying to get an insight of the quality of the iterations we also captured the number of returned items in order to get an average of the returned items per iteration. When the queue is not initialized (Figure 5) the most noticeable difference is that of DCOL-FULL, which due to the earlier mentioned problems completes an iteration more easily when the queue is small at that moment in time. In Figure 6 the results for the larger queue sizes can be observed. There we only considered runs of the DCOL-FULL where at least one iteration was completed, resulting in large confidence intervals due to the small sample (otherwise the number of returned items would be a non realistic average value). Another important observation is that both the weak regular iteration methods that include full traversal (SNR-FULL and JAVA-SNR) returned consistently larger queues since they also captured the items enqueued while traversing the full length of the queue. For the low and medium contention cases, besides some lower results of the DCOL-FULL in 16 and 24 threads cases, no other statistically significant differences were observed.

During every iteration the content of the nodes was read appropriately in order to identify the number of nodes that have been skipped within that iteration if any. The total number of skipped items in each experiment is presented in Figure 7. As expected from the consistency guarantees of the methods the only iteration that presented such gaps was the JAVA-SNR. Out of the 10 repetitions of each experiment we present the min, max and median of
the observed values in logarithmic scale. This shows the very large variation of gaps that may appear.

Summarising the observations from the experiments, it is seen that iteration throughput, as expected, indeed comes at a cost of consistency guarantees in both small and large queue sizes. Exploiting the underlying memory management can lead to impressive performance improvements (SRH, SRH-FULL) or even revive otherwise unusable techniques (DCOLLECT, DCOL-FULL). When consistency is important, helping methods (HELP=ERQ) can assist in balancing the linearizable iteration cost and provide more fair progress for all the components of the data structure. The overhead induced on the native helping operations is existent, but not dramatic, even in the cases of very high contention in both native operations and iterations.

9. Conclusion

Iterators have been an important design pattern in object oriented languages used to provide sequential access to a collection of objects without exposing its underlying representation. Contemporary programming environments, such as C++, Java and the .NET platform, include collection of data structures in their standard libraries that support concurrent operations and iterators for some of the data structures. The semantics of the iterators in the presence of concurrency are often imprecise and not universal. In this work we:

i) propose a set of consistency specifications for iteration operations

ii) show that universal lock-free or wait free constructions provide inherent means for such implementations, at the cost induced by them

iii) investigate whether efficient alternatives exist at a variety of consistency guarantees that can be desired or adequate

iv) experimentally study the trade-off between consistency and throughput, and the overhead imposed by the bulk operation to the native operations implementation of the data structure.

References


Figure 7. \textit{Min},\textit{max} and median of the numbers of skipped items in 10 runs of JAVA\textemdash{SIIR} in high contention.