A Consistency Framework for Iteration Operations in Concurrent Data Structures

Yiannis Nikolakopoulos*, Anders Gidenstam†, Marina Papatriantafilou*, Philippas Tsigas*
* *Chalmers University of Technology, Gothenburg, Sweden*
{*ioaniko,ptrianta,tsigas*}*@chalmers.se*
† *University of Borås, Borås, Sweden*
anders.gidenstam@hb.se

***Abstract*—Concurrent data structures provide the means to multi-threaded applications to share data. Data structures come with a set of predefined operations, specified by the semantics of the data structure. In the literature and in several contemporary commonly used programming environments, the notion of iteration has been introduced for collection data structures, as a bulk operation enhancing the native set of operations. Iterations in several of these contexts have been treated as sequential in nature and may provide weak consistency guarantees when running concurrently with the native operations of the data structures.**

In this work we study iterations in concurrent data structures in the context of concurrency with the native operations and the guarantees that they provide. 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. Furthermore, by using queues and composite registers as case-studies of underlying objects, we provide a set of constructions of iteration operations, satisfying the properties and showing containment relations. Besides the trade-off between consistency and throughput, we point out and study trade-off between the overhead of the bulk operation and possible support (helping) by the native operations of the data structure.

***Keywords*-concurrency; data structures; iteration; consistency; lock-free**

I. Introduction

Concurrent data structures and in particular collections are an essential part of software libraries, to extend the support for parallelism [1]–[3]. The research community has provided efficient concurrent implementations for the commonly used data structures. However, when such solutions become part of libraries, more functionalities may be added, such as traversing through the data structure elements – commonly called *iteration* or *enumeration* and provided via constructs such as *iterators*, *enumerators* or *generators*.

In programming languages, 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. [4], [30]). Therefore there has been noticeable support in the language level, mainly in two different ways, *object-based* and *control-based* iterators [30]. In the object-based iterators the state of the traversal of the main collection has to be logged in a separate 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.

The main issue in the aforementioned work is that concurrency is not taken under consideration. Suspending an execution is the exact opposite to what we ask for in a concurrent environment. The semantics of iterators change when shifting from sequential to concurrent executions. Furthermore, in data structure implementations, allowing operations to execute concurrently through fine-grain synchronization [17], enables to utilize the system parallelism with anticipated benefits in efficiency.

It is understood that in concurrent data structure implementations, there are non-trivial *trade-offs* among the throughput, the consistency and the ease-of-use by the programmer. Strong guarantees such as linearizability and sequential consistency [17] imply more intuitive usage for the user of the data structure. At the same time, they usually imply larger complexity on the algorithmic implementation of the data structure. Some contemporary implementations in well-used programming environments (cf. Sec. II) provide weaker consistency, but their semantics (consistency properties) have not been described to match definitions in the literature.

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. How can we characterize concurrency-related behavior through consistency specifications? Alternatively, does linearizability have to be very expensive?

In this work we study these questions. After presenting the system and problem model (Sec. III), we propose a set of consistency specifications for iteration operations, including also concurrency-aware properties, by building on Lamport's definitions for registers [20] (Sec. IV). We also observe that universal constructions of lock-free objects provide the means for universal iteration implementations (Sec. V). Further, we show two case-studies (Sec. V): (i) a

simpler one using composite registers where we demonstrate how the specifications proposed apply to already existent solutions of the related problem of snapshots (ii) we further illustrate the framework and the trade-offs through the more interesting case-study of concurrent queues. We present a series of constructions of snapshot-iteration operations for queues, satisfying the proposed specifications and showing containment relations between the latter. Besides the trade-off between consistency and throughput, we point out and study the trade-off between the overhead of the bulk operation and the overhead of possible support (*helping*) by the native operations implementation of the data structure (Sec. VI).

An important issue 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. The semantics of an iteration in a concurrent environment are closely related to the calling application. In the literature, both snapshot-like approaches and raw traversals – with weaker consistency guarantees – are found [2], [29]. Prompted by all the above, here we focus on snapshot iterations that return references to the elements of the data structure. It is up to the library programmer whether these should be copied or directly accessed; the proposed definitions framework is independent of the iteration implementation.

II. Related work and state of the art

Iterators have been an important design pattern in object-oriented languages, to provide sequential access to a collection of objects without exposing the underlying representation. Noble [25] presents a range of designs for iterator objects studying through their encapsulation properties, arguing that iterators can be by definition at odds with encapsulation. Boyland et al [9] discuss iterator validity in terms of ownership rights and alias control.

None of the above though considers multithreaded executions. In light of this observation and the need motivated by definitions of iterations in commonly used programming environments, there is increased attention to the topic, by both the scientific community and practitioners. Prokopec et al [29] are the first to introduce a concurrent data structure integrating an iteration operation based on snapshots. They present a concurrent trie, implemented in Scala using the DoubleCompareSingleSwap (RDCSS) software primitive [13]. Their snapshot operation is considered as constant-time ($O(1)$), given that the trie 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. In a recent work concurrent with the work in this paper[1], Petrank and Timnat [27] present an iteration for set-like data structures of key-value pairs and in particular for implementations using ordered linked-lists. In providing linearizability, the commutativity properties of the operations on sets are exploited, which is not applicable in cases of data structures with non-commutative operations like FIFO queues that this paper addresses. The iteration execution time is bounded by the size of the linked list that constitutes the main data structure, though this list can dynamically increase by updates interfering the iteration. Another aspect of iterators is addressed in [28] by Prokopec et al, introducing a framework for parallelizing iteration operations in collection data structures. Instead of 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 but gives direct support for parallel programming patterns such as map/reduce or parallel looping.

Contemporary programming environments, such as C++, Java and the .NET platform, include in their standard libraries collection data structures that support concurrent operations. These often support iteration over their contents while other operations may concurrently change the data structure. Java's *snapshot style* [2] and .NET's *moment-in-time snapshots* [3] can be expected to be linearizable (or nearly so). The consistency of Java's *weakly consistent* iterators, which vary in detail for each implementation, and the unspecified thread-safe iterators in .NET and TBB [1] provide weaker consistency guarantees that we match with our definition of *weak regularity* (introduced in Sec. IV).

In the context of snapshots, Afek et al [6] use snapshot objects to implement Java's `size` method for concurrent collections. They extend earlier work [5], [19] and exploit the fact that only meta-data, i.e. the size, are needed rather than the data structure itself.

III. System and problem modeling

The system consists of a set of processes that communicate via shared memory. It provides implementations of a concurrent container abstract data type (ADT) that represents a collection of items together with a set of *update* operations to modify them according to its specification. 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 $[s_a, f_a]$ called its *duration* (s_a and f_a being the starting and finishing times of a). There is a precedence relation on operations which is a strict partial order (denoted by $\rightarrow$). For two operations a and b, $a \rightarrow b$ means that f_a occurs before s_b. If two operations are incomparable under $\rightarrow$, they are said to overlap. We consider only runs of complete executions, where there are

[1] A first version of this work [24] had been submitted for reviewing before the publication of [27].

no pending operations. In a *sequential* history ρ, where no operations overlap, we denote as $pref_\rho(a)$, a prefix of ρ ending with operation a. We define $state_\rho(a)$ as the postcondition of the ADT after the operation a, i.e. the items that exist in the collection after the execution of a in ρ.

A history is *linearizable* [18] if it is equivalent to a sequential history – also called a linearization σ – including the same operations, whose total order respects the partial order $\rightarrow$. Thus, a run ρ of a linearizable ADT implementation induces a set of total orders, denoted with $\Rightarrow_\sigma$ for each linearization σ of ρ, that extend the partial order $\rightarrow$ in a compatible way with the sequential semantics of the ADT. For every operation a in a linearizable run ρ we respectively define $state_\sigma(a)$, in a prefix of some linearization σ of ρ that ends with a, as the postcondition of the ADT after operation a. In this notation we will drop the parameter σ when it is clear from the context.

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 $\exists$ a linearization σ of ρ such that there exists a $pref_\sigma(a)$ and $S = state_\sigma(a)$.

The ADT includes *update* operations that can add or remove items in the collection in accordance to the specification of the ADT. 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. We will call these *iteration* operations and define the following sequential specification:

Definition 1. *In a sequential execution ρ, a* valid *iteration Itr returns the items contained in $state_\rho(a)$, where a is the latest update operation preceding Itr, i.e. $Itr : state_\rho(a)$, where $a \rightarrow Itr \wedge \nexists$ update operation a' s.t. $a \rightarrow a' \rightarrow Itr$.*

Given a run ρ, we can define the *reduced run* $\tilde{\rho}$, that does not include the iteration operations.

Regarding progress guarantees of concurrent ADT implementations, we follow the standard definitions in the literature [14], [16], [17]: *Wait-freedom* ensures that any process can complete an operation in a finite number of its own steps, independently of any other process. An implementation is *bounded wait-free* if there exists a bound on the number of steps any process takes to complete an operation. In a *lock-free* object implementation it is ensured that at least one of the contending operations makes progress in a finite number of its own 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*: progress is only ensured for any process that eventually runs in isolation, i.e. in absence of interferences from other operations. It is also interesting to note that there can be distinguishable differences in progress guarantees between the different components of a data structure implementation. For example in Section V, we show that while a queue implementation is lock-free, i.e. at least one of the contending enqueue or dequeue operation makes progress, the iteration operation is obstruction-free with respect to the native enqueue and dequeue operations (i.e. it can be obstructed by them). Overall, the lock-free guarantee still stands (cf. Theorem 7). As indicated by Michael [21], careful selection of such designs can be proved valuable from the application perspective.

IV. Consistency specifications

We define the following properties, building on the consistency-related definitions by Lamport [20] and Herlihy and Wing [18]. We assume that the reduced run $\tilde{\rho}$ that does not include the iteration operations is linearizable and thus for each linearization σ of $\tilde{\rho}$ the respective $\Rightarrow_\sigma$ is defined.

Definition 2. *(i) Let $Itr \in \rho$ be an iteration operation, not overlapping with any other operation in $\tilde{\rho}$. Itr is **safe** if it returns a valid state $S = state_\sigma(a)$ for some linearization σ of $\tilde{\rho}$ and some operation a, such that: $Itr \nrightarrow a$ and $\nexists$ update operation $a' : a \Rightarrow_\sigma a' \rightarrow Itr$. If Itr is overlapping with any operations of $\tilde{\rho}$, it can return any arbitrary state of the object.*
*(ii) Let $Itr \in \rho$ be an iteration operation possibly overlapping some $a \in \tilde{\rho}$. Itr is **regular** if it returns a valid state $S = state_\sigma(a)$ for some linearization σ of $\tilde{\rho}$ and some operation a, such that: $Itr \nrightarrow a$ and $\nexists$ update operation $a' : a \Rightarrow_\sigma a' \rightarrow Itr$, i.e. S is neither "future" nor "overwritten".*
*(iii) The **monotonicity** property states that for any two iteration operations Itr_1, Itr_2 that return valid states $state_\sigma(a_1)$ and $state_\sigma(a_2)$ respectively for some linearization σ of $\tilde{\rho}$, if $Itr_1 \rightarrow Itr_2$, then $a_1 \Rightarrow_\sigma a_2$ or $a_1 = a_2$.* [2]
*(iv) Itr is **linearizable** if it is* regular *and the run ρ is equivalent to some sequential history, that includes the same operations, whose total order respects the partial order of the original run ρ.*

Intuitively, *safeness* guarantees that an iteration implementation returns recent and consistent states only if it does not overlap with other modification operations, otherwise any arbitrary state might be returned (e.g., empty). *Regularity* improves by ensuring that a meaningful, recent and not future state will be returned. Thus, a regular Itr_1 might or might not include in the returned state the effect of an overlapping modification operation a. It is interesting to note that another regular Itr_2, such that $Itr_1 \rightarrow Itr_2$ but Itr_2 still overlapping a, might return a state according to the definition of regularity but different – even preceding–

[2] Dwork et al [10] in the context of composite registers used two notions of monotonicity, for scans and for updates. Notice that a regular Itr also satisfies the monotonicity of updates property, i.e. for two linearized updates, an Itr that "observes" the effects of the latter update, should also "observe" the effects of the preceding update.

from the one returned by Itr_1. *Monotonicity* is an additional property that clarifies such behavior. Furthermore, motivated by implementations in contemporary programming environments (cf. Sec. II) we define one consistency guarantee lying between safeness and regularity:

Definition 3. *Consider an iteration operation $Itr \in \rho$ and the reduced linearizable run $\tilde{\rho}$. Let a be the latest update operation finished before s_{Itr}, and $pref_\sigma(a)$ the respective prefix for some linearization σ of $\tilde{\rho}$. A* ***weakly regular*** *Itr returns a state S such that $S = state_\tau(b)$, for some operation b in a run $\tau = pref_\sigma(a) \cup ops_{[s_{Itr}, f_{Itr}]}$, that extends the $pref_\sigma(a)$ with $ops_{[s_{Itr}, f_{Itr}]}$, i.e. an arbitrary number of operations that are overlapping with the execution interval $[s_{Itr}, f_{Itr}]$.*

Weak regularity is a more relaxed definition that allows implementations to capture only part of the update operations that are concurrent with an iteration.

V. Properties and trade-offs in the consistency framework

In this section we demonstrate the framework and the tractability and containment of the defined consistency specifications, through the well-understood problem of atomic snapshots of composite registers [5], [8], [10], [12], [17], [19] (and references therein), and then through a test case of a linearizable lock-free concurrent queue [22]. Before that, we relate iteration with universal constructions of concurrent data structures.

By reflecting on the literature on concurrent ADT implementations a natural question is the following: Can universal constructions, such as [15] and [7], 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 linearizable concurrent data object just like any other operation of the sequential data object.*

A. Relating with snapshots on composite registers

A composite register is an array of read/write registers. A snapshot of them is an iteration operation. A first straightforward approach for acquiring a snapshot matches the weakest of the presented specifications:

Lemma 1. *An iteration implemented as a set of reads of all the entries of the composite register array satisfies the* safeness *condition.*

Stronger consistency properties are guaranteed by wait-free snapshot algorithms that use *handshaking* methodology like [10], [19].

Lemma 2. *A* time-lapse *snapshot as in [10], or a snapshot satisfying the atomicity criterion of [19] is* regular *and* monotonic *as defined in Section IV.*

The proof of this lemma is based on a sequence of transformations that match formalism presented in [19] and [10]. Based solely on definition 2 we show the following theorem that stands for any object:

Theorem 1. *An implementation of an iteration operation, where iteration operations do not overlap with each other, is linearizable, if it satisfies the regularity and monotonicity properties.*

Proof: We assume an execution ρ that includes non-overlapping iteration operations satisfying the properties of regularity and monotonicity and its reduced linearizable run $\tilde{\rho}$ that does not include any iteration operations. We can linearize every iteration operation $Itr \in \rho$ within its duration and right after the linearization point of the respective operation a that matches the regularity definition, for some linearization σ of $\tilde{\rho}$. Regularity guarantees that $\nexists a' \in \tilde{\rho} : a \Rightarrow_\sigma a' \rightarrow Itr$, i.e. the state $S = state_\sigma(a)$ that Itr returns is no "future" and not "overwritten" and therefore each iteration operation Itr 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. ■

The latter theorem demonstrates the *containment* relationships of the consistency definitions (cf. Def. 2) as a linearizable iteration is also a regular one. Furthermore, it confirms the correctness criteria and the results of the above algorithms for composite registers (Lem. 2). In cases of possibly overlapping iterations only *regularity* is guaranteed.

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 well known method to achieve this is the *double collect* [26]. If a reader manages to collect twice the same timestamped values of all registers then this collection is a snapshot, though guaranteeing 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 [5] presented a *wait-free* solution. More snapshot constructions can be found in [12], [17].

Lemma 3. *An iteration operation implemented as a snapshot [5] of a composite register implementation, satisfies* linearizability.

Iterating finite-domain add-remove containers One can observe that simple container data structures to store and remove items coming from a finite domain set can be

implemented using composite registers. Every item of the domain can be represented by an entry register of the shared array. The *add* and *remove* operations of the set can merely be implemented through *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. The latter also provides a way to get the items of the container or implement a *contains* operation that is usually provided by the interface of a set ADT. Regarding concurrent iterations on a linearizable set implementation, the reader is referred to [27].

B. Iterating concurrent queues

The next case-study, in which we investigate the trade-offs of the presented consistency definitions, is that of iteration operations in concurrent linked-list based queues. As a running example, we use the lock-free linked-list based queue by Michael and Scott [22], since it is a classic, simple construction that can be efficiently implemented in several platforms. More complex designs deploying flat-combining techniques [11], or allowing modify operations to spuriously *fail* [23], show interesting directions for future work.

The goal of this section and the presented algorithms is to study the *inherent difficulties* in achieving the different consistency levels. The efficiency of the presented algorithms is not the main point and it can be improved once several implementation assumptions are done (e.g. memory management). Some aspects of the latter are discussed in section VI as well as in the discussion part of this section (Sec. V-B4). We present the algorithms at a level of abstraction that can be followed independently of the implementation details of the underlying construction. To overcome differences regarding memory management and whether references to the contained items or copies of them should be returned, in the presentation of our iteration algorithms we consider a simple auxiliary 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. 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 as well as the progress guarantees of each method, we always assume the correctness of the underlying linearizable, lock-free, concurrent queue implementation.

1) 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 naive scanning of the collection and returning its state is ineffective in the general case. Specifically for the queue case we can achieve a *weakly regular* iteration by simply traversing the nodes of the list (Alg. 2).

Algorithm 1 Collect function

```
1: function COLLECT(curHead, curTail, *stateToReturn)
2:    stateToReturn.add(curHead)
3:    curNode ← curHead.next
4:    stateToReturn.add(curNode)
5:    while curNode ≠ curTail do
6:       curNode ← curNode.next
7:       stateToReturn.add(curNode)
```

Algorithm 2 Scan&Return-iteration

```
1: stateToReturn.initialize()
2: curHead ← Head
3: curTail ← Tail
4: COLLECT(curHead, curTail, &stateToReturn)
5: return stateToReturn
```

In the original queue implementation [22] the `Tail` might not be up to date and have fallen behind. For presentation simplicity we ignore this in the presented algorithms, as it is easy to be addressed since the difference will be only one node. Notice that during the traversal any newer dequeues will not be captured, while any enqueue operations linearized before the first read of `Tail` will be reflected to the state returned.

Theorem 2. *Algorithm 2 implements a weakly regular iteration operation on the concurrent queue implementation.*

Proof: In absence of interferences the algorithm will return a valid state of the data structure, $state_\sigma(a)$, the one after the most recent linearized enqueue or dequeue operation a, for some linearization σ of the reduced run not containing the iteration operation. Otherwise, the $pref_\sigma(a)$ will be extended as follows: in case enqueue operations are linearized in the duration of the iteration operation: any nodes enqueued between the execution of lines 2 and 3 will also be returned; enqueue operations linearized after the execution of line 3 will not be reflected in the returned state. Any nodes dequeued concurrently by updates that are linearized after line 2 will actually be returned and the dequeue operations will not be reflected in the returned state. ■

Theorem 3. [3] *Algorithm 2 implements a wait-free iteration operation on the concurrent queue implementation.*

Note that Scan&Return is not *bounded* wait-free as there is no fixed bound for the queue size[4]. Furthermore, since Alg. 2 does not interfere with the enqueue and dequeue operations of the original queue implementation, and given the previous theorem it is easy to see that:

Theorem 4. *The lock-free concurrent queue implementation extended with the iteration operation of Algorithm 2 remains lock-free.*

[3] Due to space constraints some of the proofs had to be omitted.

[4] According to the strict definition of bounded wait-freedom it may not be possible to have bounded wait-free iterations on unbounded size data structures.

Algorithm 3 Double-Collect-based-iteration

```
1: while True do
2:    stateToReturn.initialize()
3:    curHead ← Head
4:    curTail ← Tail
5:    COLLECT(curHead, curTail, &stateToReturn)
6:    if curHead = Head ∧ curTail = Tail then
7:       return stateToReturn
```

2) Linearizable Double-Collect-based iteration: This algorithm (Alg. 3) is inspired by classic snapshot constructions with double collect [5], [17], 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` pointers. If they have not changed, then due to the inherent order of the queue's elements the rest of the queue will not have changed.

Theorem 5. *Algorithm 3 implements a linearizable iteration on the concurrent queue implementation.*

Lemma 4. *Algorithm 3 will fail to return only if it is interfered by an unbounded number of update operations.*

Theorem 6. *Algorithm 3 implements an obstruction-free iteration operation on the concurrent queue implementation.*

Proof: Lemma 4 implies that algorithm 3 will terminate if it runs solo long enough. ■

Theorem 7. *The lock-free concurrent queue implementation extended with the iteration operation of Algorithm 3 remains lock-free.*

Proof: No shared variables of the original data structure implementation are modified by the iteration in algorithm 3. Thus, the iteration operation will not interfere with any of the original enqueue or dequeue operations of the underlying queue implementation. Therefore, at least one process operating on the data structure will make progress (enqueue, dequeue, iteration). ■

By observing theorems 2, 5, 6 and the respective algorithms we can see that:

Corollary 1. *A linearizable obstruction-free iteration operation on the lock-free queue construction can be implemented using a weakly regular wait-free iteration of the same queue construction.*

The latter corollary implies *containment* in the definition sets, as algorithm Scan&Return-iteration is used in algorithm Double-Collect-based-iteration. Intuitively, a linearizable implementation is also a weakly regular one.

3) Linearizable iteration with helping techniques: The trade-off between no interference and helping by enqueue and/or dequeue will be explored in the next algorithms. The high level approach is to use some form of *handshaking* between the modification operations and the iteration operations, so that the latter will be able to distinguish

Algorithm 4 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 5 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
```

the appropriate nodes to return according to their specific linearization point.

For presenting these algorithms we need to involve parts of the actual construction of our test case based on [22]. Still, for presentation simplicity we skip some checks and parts of the code that are used for optimization but we include the key points. The helping algorithms make use of two synchronization constructs, the DoubleCompareSingleSwap (`DCSS`) and MultipleCompareDoubleSwap (`MCDS`) (Algorithms 4 and 5) that can be implemented using the CompareAndSwap hardware primitive (`CAS`), e.g. following the lines of the lock-free constructions in [13]. These constructions conditionally update, in an atomic manner, one or two memory words based on the value of a control word and the values of the memory words. We further make use of the FetchAndAdd (`FAA`) hardware primitive that atomically increments the value of a counter, returning the old value to the calling thread.

Enqueuers helping iteration: In this construction, presented in Algorithms 6 and 7, the enqueue operation provides information through the queue nodes in order to help the iteration operations. This enables the latter to decide whether the node should be included in their result or not. A shared counter serves as a logical timestamp, that is increased by each iteration.

An *enqueue* operation (Alg. 6) differs from the original [22] in the part where in addition to the `Tail` pointer, the value of the shared `counter` is also read (line 7). This value is used to tag the appropriate field in the new node (line 8), which is then enqueued using the `DCSS` primitive. `DCSS` in line 9 assures that the node is connected to the list while the value of the shared `counter` has not changed. The latter remains the linearization point as it is when the enqueue operation takes effect. An *iteration* operation (Alg. 7) 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 (lines 1-4). The iterator will stop as soon as it encounters 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 operation. If instead of DCSS only a single CAS was used for connecting the node, then possible differences between the time the enqueue took effect and the timestamp value could result in iteration operations returning inconsistent states not compatible with the sequential semantics of the operation.

Theorem 8. *Algorithm 7 implements a linearizable iteration and the linearization point is the last call of* FAA *on line 3.*

Lemma 5. *Algorithm 7 will fail to return only if it is interfered by an unbounded number of dequeue operations, while executing the loop in lines 1-4.*

Theorem 9. *Algorithm 7 is an obstruction-free implementation of an iteration operation on the modified concurrent queue implementation.*

Proof: Lemma 5 implies that algorithm 7 will terminate if it runs solo long enough. ■

Theorem 10. *The concurrent queue implementation integrating the modified enqueue Algorithm 6 and the iteration operation of Algorithm 7 is lock-free.*

Proof: We assume the lock-free property of the original queue implementation. The enqueue operation will retry, additionally to the conditions in the original implementation, if the DCSS in line 9 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. 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. ■

By helping the iteration operation the possibilities that it will retry are reduced, compared to Alg. 3; this will happen only if dequeue operations make progress. However this comes at the cost of the iteration obstructing enqueue operations. Unbounded interferences of iterations may starve an enqueue operation.

Helping by both enqueue and dequeue: In this construction, in addition to the enqueue operation (Alg. 6) also the dequeue assists the iteration (Alg. 8 and 9 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, though increasing the communication overhead.

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. 6 in the previous construction. 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.

Algorithm 6 Enqueue operation with helping

```
1: initialize newNode
2: while True do                                  ▷ Repeat until success
3:    lastNode ← Tail
4:    if lastNode.next ≠ null then                ▷ Help update Tail
5:       CAS(&Tail, lastNode, lastNode.next)
6:       continue
7:    localTS ← read counter
8:    newNode.enqueueTag ← localTS
9:    if DCSS(&counter, localTS, &lastNode.next, null,
      newNode) then
10:      CAS(&Tail, lastNode, newNode)            ▷ Try update Tail
11:      return
```

Algorithm 7 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
```

Algorithm 8 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 9 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
```

The timestamps 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.

Theorem 11. *Algorithm 9 implements a linearizable iteration and the linearization point is on line 2.*

Theorem 12. *Algorithm 9 implements a wait-free iteration operation on the modified concurrent queue implementation.*

Lemma 6. *A dequeue operation (Alg. 8) may be forced to retry due to an arbitrary overlapping iteration operation at*

most once.

Lemma 7. *An enqueue operation (Alg. 6) may be forced to retry due to an arbitrary overlapping iteration operation at most once.*

Theorem 13. *The concurrent queue implementation integrating the modified enqueue and dequeue Algorithms 6 and 8 respectively, along with the iteration operation of Algorithm 9 is lock-free.*

The proof of the latter theorem is easily derived by the the fact that the original implementation is lock-free and using lemmas 6 and 7. Therefore, at least one contending process will make progress (enqueue, dequeue, iteration).

4) Discussion: The goal of the presented algorithms is to explore the algorithmic and communication (helping) trade-offs that exist, in an implementation oblivious manner. Their efficiency can be greatly improved once specific assumptions are made, especially regarding aspects like memory management. Collecting a chain of nodes can be significantly easier in a garbage collected environment since we can assume that once we keep a reference to a node we can reach it even in case of arbitrary delays. Thus we can have more efficient versions of algorithms like 2 or 3 (cf. Sect. VI). Under such assumptions more designs can be provided, e.g. simplifying and improving Alg. 7 by using a reference to the current head as a timestamp instead of a counter, or using an atomic read for both the `Head` and `Tail` pointers. The former case would still require a `DCSS` for the enqueue, while the latter would serialize enqueue, dequeue and iteration operations. The assumptions required for such cases are not general enough for the approach of this paper.

With these algorithms we complete the picture of the trade-offs involved in enhancing a concurrent data structure with iteration operations. Taking a step back we can observe that the concurrent queue has two different points of contention, the `Head` and the `Tail`. These are the two points that allow for concurrent operations to take place. When no helping methods are involved, the stronger the consistency guarantees we try to provide, the more is the cost that has to be beared by the iteration in terms of progress guarantees and possibilities of retrying. When we try to improve this with helping techniques, the modification operations are forced to communicate with the iteration, thus inherently limiting the concurrency of the data structure (even if `DCSS,MCDS` were available in hardware). Even under assumptions that would allow us an atomic read of `Head` and `Tail`, such an operation would be serialized between modifications of `Head` and `Tail`.

Finally, we showed the tractability of the proposed framework in different case studies, possible trade-offs, and containment properties of the definitions. The landscape of iteration implementations, by both practitioners and researchers, further motivates this work by already including relaxed forms of iteration [1]–[3] (cf. Section II), while the broader distributed systems area has showed that weaker consistency can be proved useful in several contexts.

VI. Implementations and experimental study

In order to study the properties of the implementations following the framework, we implemented iteration methods for the case of a concurrent queue. The goal is to investigate, from a practical perspective also, the qualitative impact of the different consistency guarantees on the throughput performance of iteration operations. Furthermore we are interested in the way the iteration implementations affect the performance of the native operations of the data structure, especially across the different iteration implementations that may or may not use helping techniques.

Implementations: Our implementations consist of the lock-free queue by Michael and Scott [22] extended with iterator constructions presented in section V-B, implemented in Java. Specifically, we implemented two versions of the linearizable double-collect based iteration as seen in section V-B2, the `DCOLLECT` and `DCOL-FULL`. The first exploits the garbage-collected environment of Java for easier collection of the nodes. References only to `Head` and `Tail` are collected, since this is enough to prevent the chain of nodes in between from being garbage collected and thus being able to traversed after the return of the method. The `DCOL-FULL` (full double collect) traverses the entire chain of nodes during the iterator's construction (as in Alg. 3), in order to emulate the delays that would be present in an unmanaged memory environment, where it would be required to explicitly hold references to all the nodes to be returned. For the Scan&Return method presented in section V-B1, we provide the `SNR` and `SNR-FULL` implementations, that exploit garbage collection and emulate a full traversal respectively. For iterations with helping techniques, limited to hardware support, we used a software version of the `DCSS` primitive based on [13] and implemented the iteration with helping from the enqueue operations (`HELP-ENQ`) that we presented in section V-B3. Finally we also compared with Java's implementation of the Michael and Scott queue, `ConcurrentLinkedQueue` in the `concurrent` package. In this case the iterator provided, to which we will refer as `JAVA-SNR`, is implemented following the Scan&Return style too, but with differences regarding the management of nodes that result into "gaps" in the returned state. In particular when a node is dequeued it is disconnected from the linked list by referencing itself. `JAVA-SNR` collects the `Head` and advances through the `next` pointers until the end of the queue is reached. In case the current node is dequeued (and possibly nodes following), `JAVA-SNR` moves on to the current `Head`, skipping this way all nodes dequeued in between (cf. Fig 3). Thus, there is a *qualitative* difference between the two implementations, since `SNR` returns a state

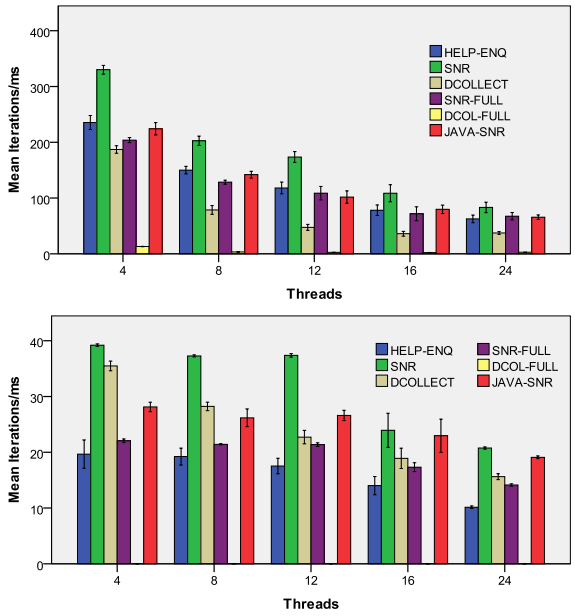

Figure 1. Iterations per millisecond in high contention, with a queue initialized empty (upper) and with 2000 items (lower).

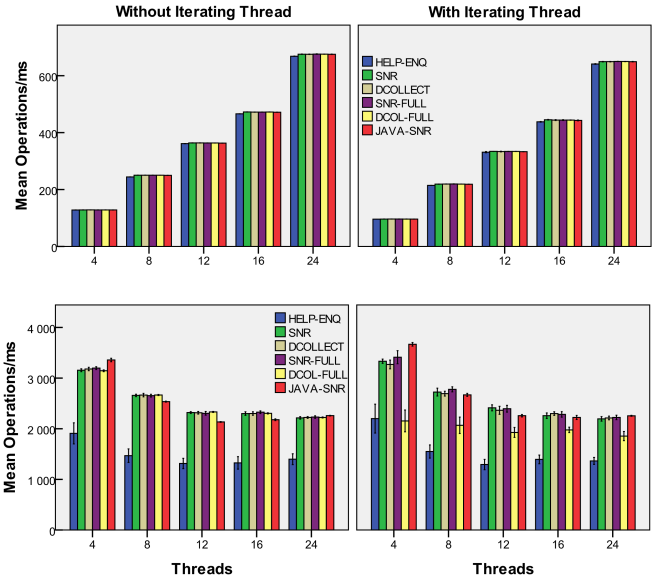

Figure 2. Throughput (enq. and deq.) of queue implementations in medium (upper) and high contention (lower), with a queue initialized with 2000 items. On the right column, one of the threads runs iteration operations.

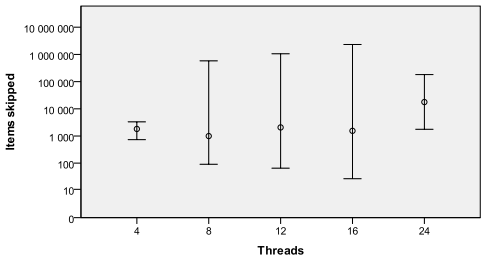

Figure 3. Min, max and $median$ of skipped items in 10 runs of JAVA−SNR in high contention and a queue initialized with 2000 items.

that is a logically continuous chain of nodes as they were connected during the queue's lifetime duration.

Experiment setup: For more consistent comparison the iteration methods 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 method until the iteration was finished. Meanwhile, $N-1$ "worker" threads were running enqueue and dequeue operations at random with equal probability and 1 thread continuously iterated the queue. Each experiment was run for 10 seconds, for values of $N = 4, 8, 12, 16$ and 24, on a workstation with 2 sockets of 6-core Xeon E5645 processors with Hyper Threading (24 logical threads in total), running version 3.2 of the Linux kernel. Other parameters that we varied in the experiments, are the size of the initial queue (empty, 2000, 5000 items) and the level of contention (by introducing dummy work between data structure operations). We present the mean of 10 repetitions of each experiment along with 95% confidence intervals.

Summarising the observations from the experiments, in each implementation the existence of a thread iterating the queue did not have a significant impact in the usual throughput of enqueue and dequeue operations (Fig. 2). Iteration throughput, as expected, comes at a cost of consistency guarantees (Fig. 1). Exploiting the underlying memory management can lead to impressive performance improvements (SNR, SNR−FULL) or even revive otherwise unusable techniques (DCOLLECT, DCOL−FULL). When both consistency and progress are important, helping methods (HELP−ENQ) can assist in balancing the linearizable iteration cost and provide fairer progress for all the contending processes on the data structure. The overhead induced on the native helping operations is non-negligible and appears mainly in the cases of high contention (Fig. 2). In fact, cases of lower contention can benefit from stronger consistency with different methods depending on the implementation environment. Finally weaker consistency implementations (SNR,JAVA−SNR) can have different qualitative characteristics. Figure 3 shows the effect of "gaps" in the returned state for JAVA−SNR, i.e. skipped nodes during the iteration.

VII. Conclusion and Future Work

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 include collection data structures in their standard libraries that support concurrent operations and iterators for some of them The semantics of the iterators in the presence of concurrency are often imprecise and not universal. Weaker consistency in the broader distributed systems area, as well as weakly consistent iteration implementations that already exist, show the need for a more formal description of such behavior.

In this work we:

i) propose a set of consistency specifications for iteration operations, also paving the way for future research in

bulk operations on data structures

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

iii) show that the proposed definitions support containment relations, also through the algorithmic implementations of the queue case study

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.

Overall, providing strong consistency iteration operations varies from affecting the complexity and progress of the iteration itself, to the native operations of the data structure.

Applying the consistency framework in more complex data structures, where the iteration implementations might be even more challenging, is one of our future work directions. Finally, the tractability of the framework can be extended to other bulk operations that can arise from the continuously increasing parallel and concurrent applications.

ACKNOWLEDGEMENTS The research leading to these results has received funding from the European Union 7th Framework Programme (FP7) under grant agreement 611183 (EXCESS Project, www.excess-project.eu) and by the Swedish Research Council project Contract nr. 621-2010-4801. We are thankful to Panagiota Fatourou for her feedback in earlier stages of this work.

REFERENCES

[1] Intel threading building blocks documentation. http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm. Retrieved 2012-11-27.

[2] Java platform standard edition 7 documentation. http://docs.oracle.com/javase/7/docs/index.html. Retrieved 2012-12-06.

[3] .NET framework class library documentation. http://msdn.microsoft.com/en-us/library/gg145045.aspx. Retrieved 2013-05-10.

[4] Python v2.7.5 documentation. http://docs.python.org/2/library/itertools.html. Retrieved 2013-09-10.

[5] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, Sept. 1993.

[6] Y. Afek, N. Shavit, and M. Tzafrir. Interrupting snapshots and the java size method. *Journal of Parallel and Distributed Computing*, 72(7):880–888, July 2012.

[7] J. Anderson and M. Moir. Universal constructions for large objects. In *9th Int'l Workshop on Distributed Algorithms*, volume 972 of *LNCS*, pages 168–182. Springer, 1995.

[8] J. H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, May 1994.

[9] J. Boyland, W. Retert, and Y. Zhao. Iterators can be independent from their collections. In *IWACO, in conjunction with ECOOP 2007*.

[10] C. Dwork, M. Herlihy, S. Plotkin, and O. Waarts. Time-lapse snapshots. *SIAM J. Comput.*, 28(5):18481874, May 1999.

[11] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. SPAA '11, pages 325–334, New York, NY, USA, 2011. ACM.

[12] F. Fich. How hard is it to take a snapshot? In *SOFSEM 2005: Theory and Practice of Computer Science*, volume 3381 of *LNCS*, pages 28–37. Springer, 2005.

[13] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Distributed Computing*, number 2508 in LNCS, pages 265–279. Springer, 2002.

[14] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov. 1993.

[16] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. ICDCS '03. IEEE Computer Society, 2003.

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

[18] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463492, July 1990.

[19] L. Kirousis, P. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: solutions with linear or sublinear complexity. *IEEE Trans. Parallel Distrib. Syst.*, 5(7):688 –696, jul 1994.

[20] L. Lamport. On interprocess communication. *Distributed Computing*, 1(2):86–101, June 1986.

[21] M. M. Michael. The balancing act of choosing nonblocking features. *Commun. ACM*, 56(9):4653, Sept. 2013.

[22] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on PODC*, pages 267–275, 1996.

[23] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

[24] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. Enhancing concurrent data structures with concurrent iteration operations: Consistency and algorithms. Technical report, Chalmers University of Technology, 2013.

[25] J. Noble. Iterators and encapsulation. In *TOOLS 33. Proceedings*, pages 431–442, 2000.

[26] G. L. Peterson and J. E. Burns. Concurrent reading while writing II: the multi-writer case. In *, 28th Annual Symposium on Foundations of Computer Science, 1987*, pages 383–392.

[27] E. Petrank and S. Timnat. Lock-free data-structure iterators. In *Distributed Computing*, number 8205 in LNCS, pages 224–238. Springer, 2013.

[28] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A generic parallel collection framework. In *Euro-Par 2011*, number 6853 in LNCS, pages 136–147. Springer, 2011.

[29] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky. Concurrent tries with efficient non-blocking snapshots. PPoPP '12, pages 151–160. ACM, 2012.

[30] S. M. Watt. A technique for generic iteration and its optimization. WGP '06, pages 76–86. ACM, 2006.