Concurrent Size*

GAL SELA, Technion, Israel
EREZ PETRANK, Technion, Israel

The size of a data structure (i.e., the number of elements in it) is a widely used property of a data set. However, for concurrent programs, obtaining a correct size efficiently is non-trivial. In fact, the literature does not offer a mechanism to obtain a correct (linearizable) size of a concurrent data set without resorting to inefficient solutions, such as taking a full snapshot of the data structure to count the elements, or acquiring one global lock in all update and size operations. This paper presents a methodology for adding a concurrent linearizable size operation to sets and dictionaries with a relatively low performance overhead. Theoretically, the proposed size operation is wait-free with asymptotic complexity linear in the number of threads (independently of data-structure size). Practically, we evaluated the performance overhead by adding size to various concurrent data structures in Java—a skip list, a hash table and a tree. The proposed linearizable size operation executes faster by orders of magnitude compared to the existing option of taking a snapshot, while incurring a throughput loss of $1\% - 20\%$ on the original data structure's operations.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; **Concurrent algorithms**; • **Theory of computation → Data structures design and analysis**.

Additional Key Words and Phrases: Concurrent Algorithms; Concurrent Data Structures; Linearizability; Wait-Freedom; Size

The full version of this paper is available at [Sela and Petrank 2022a], and the code is publicly available at [Sela and Petrank 2022b].

1 INTRODUCTION

Concurrent data structures are fundamental building blocks of concurrent programming, utilized to leverage modern multi-core processors. There has been substantial work on the design of efficient and scalable concurrent data structures with good progress guarantees, in order to benefit concurrent algorithms at large. A fundamental, widely used, property of a data structure is its size (i.e., the number of elements it contains). In Java, for example, any collection or map class that implements one of the elementary interfaces java.util.Collection or java.util.Map [jav 2022] must implement a size method. Interestingly, implementing an efficient and correct size operation for a concurrent data structure is non-trivial. For a formal treatment, we use linearizability as the correctness criterion of concurrent executions [Herlihy and Wing 1990; Sela et al. 2021a], but the discussion below also applies to other intuitive correctness criteria.

The literature does not offer an acceptable solution to implementing a correct size operation, and existing implementations give up correctness in order to avoid a significant performance

---

Authors' addresses: Gal Sela, Technion, Israel, galy@cs.technion.ac.il; Erez Petrank, Technion, Israel, erez@cs.technion.ac.il.

deterioration. For example, the non-blocking collections and maps in the `java.util.concurrent` package [Lea 2004] implement a non-linearizable `size` method that returns an estimate of the size. The returned estimate may be inaccurate when the object is concurrently modified during the execution of `size`. In contrast, a linearizable `size` operation would tolerate concurrent update operations and retrieve the exact number of elements in the data structure at some point during the execution of the `size` operation.

Existing solutions are incorrect or inefficient. Ignoring concurrency, one can determine the size of a data structure simply by traversing it and counting the number of encountered items. This is the approach taken by the `size` method of Java's `ConcurrentLinkedQueue` and `ConcurrentLinkedDeque`. This approach is fine for a sequential execution, but for a concurrent execution this implementation is not linearizable. The following is a worst-case scenario for this implementation. Consider an execution on a linked list with the single item 1. Assume a thread T, running this `size` implementation, starts the traversal from the node containing 1 and then gets preempted. At this point, the following steps may occur repeatedly: some thread appends a node with the item 2 to the end of the list, increasing the list's size to 2; next, T gets scheduled, resumes its traversal and proceeds to this new node; then, some thread deletes the node containing 1, so the list's size is 1 again. Next, some thread inserts 3 and deletes 2, letting T see a third element, etc.; until eventually—after some item s is appended—the thread T gets to the end of the list before another thread gets the chance to insert an additional item. In this scenario, T will erroneously return a possibly large s as the list size, while in practice the list size never exceeded 2. While this is a worst-case scenario, one can envision many other scenarios in which the returned value would be incorrect.

Alternatively, it is possible to obtain a correct `size` implementation by obtaining a linearizable snapshot of the data structure (e.g., using any of the methods in [Arbel-Raviv and Brown 2018; Nelson-Slivon et al. 2022; Petrank and Timnat 2013; Wei et al. 2021]) and then iterate over the returned snapshot to count the number of elements in the snapshot. While correct, this solution is inefficient, yielding a time complexity linear in the number of elements in the data structure.

If we want to avoid such a high cost for the `size` operation, then we need to keep some metadata that allows computing the size of the data structure efficiently when needed, and let the data-structure operations maintain this metadata. Naively, the metadata would just be the current size value. A natural attempt to implement such a `size` operation would be to keep the size in a designated field of the data structure and let the operating threads update it with each operation that affects the size. An `insert` operation would execute a size increment after inserting its item, and a `delete` operation would execute a decrement of the size field after performing the deletion. Java's `ConcurrentSkipListMap` and `ConcurrentHashMap` use such an implementation. However, the separation between the data-structure update and the metadata update foils linearizability. As an example, consider n threads that are preempted exactly after inserting an element to the data structure and before updating the size field. At this point the size field would be off by n and thus inconsistent with a view of another thread that actually reads the data structure.

A simplified execution for one updating thread is depicted in Figure 1. There, only one update operation is ever executed on the data structure: one thread inserts 1 into an empty structure. Another thread that starts by calling `contains(1)`, sees that the data structure already contains the single element in it, but then it calls `size()` and receives 0. We executed this simple program on Java's `ConcurrentSkipListMap` several times, and we actually witnessed executions that reproduced the contradicting result as depicted in Figure 1. This demonstrates that the `size` method of `ConcurrentSkipListMap` is not linearizable. The core issue is the separation between the actual data structure update and the subsequent update of the metadata.

Furthermore, updating the metadata separately from updating the data structure may yield a `size` execution that returns a negative number. This means that the size operation is not only
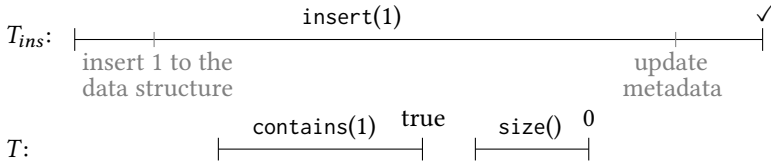
Fig. 1. An execution with conflicting `contains` and `size` results due to the separation between updating the data structure and the size metadata
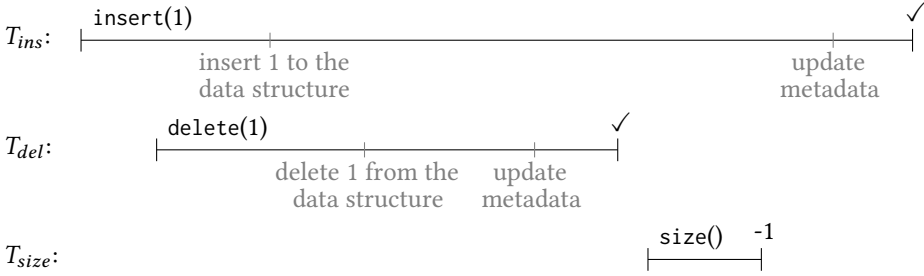

Fig. 2. An execution that yields a negative size due to the separation between updating the data structure and the size metadata

non-linearizable, but it can also not satisfy any correctness criterion that requires method calls to appear to happen in a one-at-a-time sequential order, e.g., it is not quiescently consistent [Aspnes et al. 1994; Herlihy and Shavit 2008; Shavit and Zemach 1996] nor sequentially consistent [Herlihy and Shavit 2008; Lamport 1979]. Consider the following execution (depicted in Figure 2). Thread T_{ins} inserts an item to the data structure, and before it updates the metadata, thread T_{del} deletes that item and updates the metadata, registering its decrement. At this point, thread T_{size} calls size() that returns -1 based on the metadata, which currently reflects only the deletion and not yet the insertion. The separation between the data-structure and metadata updates results here in updating the metadata in a reversed order, which is impossible since the deletion cannot succeed if it happens before the insertion. The returned size exposes this impossible operation order. The method calls in this execution do not appear to happen in a one-at-a-time sequential order since size would never return a negative result in a legal sequential execution.

A more complex metadata maintenance is proposed by Afek et al. for computing the size more efficiently [Afek et al. 2012]. But they, too, update the metadata after the data-structure update, and so their implementation suffers from the same problems. (We elaborate on issues in [Afek et al. 2012] in Appendix A.)

A third alternative for implementing the size operation is to use locks to prevent a size operation from exposing a temporary inconsistency between the data structure's state and the metadata. This too would create a severe bottleneck and deteriorate performance significantly.

In this paper we propose an efficient linearizable size implementation. To the best of our knowledge, this is the first size solution that provides both linearizability and efficiency (namely, not iterating over all elements of the data structure or using coarse-grained locking). We present a methodology for adding a linearizable size operation to concurrent data structures that implement a set or a dictionary. Our methodology yields data structures that satisfy the following attractive theoretical properties:

(1) The time complexity of the size operation is linear in the number of threads.
(2) The size operation is wait-free, namely, a thread running a size operation completes the operation within a finite number of steps, regardless of the activity of other threads.
(3) The (asymptotic) time complexity of the original data-structure operations is preserved.
(4) The progress guarantees of the original data structure are preserved. Namely, wait-free methods of the original data structure remain wait-free in the transformed data structure, and the same goes for lock-free or obstruction-free methods.

To achieve Property (1), we keep always-consistent metadata, from which the size can be correctly computed. To prevent operations from exposing inconsistencies similar to the examples of Figures 1 and 2, we work hard to achieve a single linearization point in which the data structure is modified and the metadata gets updated simultaneously. This is obtained by letting an operation appear as completed to other operations only when the metadata update occurs. Formally, the update of the metadata becomes the single linearization point of the entire data structure operation. Dependent data-structure operations are adapted to comply with the new linearization point, and help completing concurrent operations when necessary. For instance, a delete(k) by thread T_2 that encounters an ongoing delete(k) by another thread T_1 which has already deleted the key from the data structure, must help T_1 to update the metadata in order to complete the obstructing delete(k) before returning a failure. It cannot block and wait for T_1 to update the metadata, since that might change the progress guarantees of the delete operation and foil Property (4).

Helping another operation implies updating the metadata on its behalf. As always with multiple threads helping to execute a single operation, care has to be taken for the operation to be executed only once. We keep per-thread counters as the size metadata, and use a corresponding mechanism that enables helpers to determine whether the metadata already reflects the helped operation, to prevent a wrong double update of the metadata on behalf of the same operation. This mechanism enables helpers to efficiently make a determination and update the metadata if necessary, thus achieving Property (3).

The size of a data structure is a fundamental property of a data set and having a methodology for obtaining an efficient accurate solution for it seems like an important point in the design space, which is currently missing in the literature. Using inaccurate solutions may yield unexpected results, e.g., sizes that the data structure never had and even a negative size. Such results may in turn yield unexpected bugs that may put the reliability of an entire system at risk. A reliable solution is especially desirable in dynamic programming languages that favor correctness over performance, such as Python and Ruby, which use a global interpreter lock in their reference implementations and are expected to behave reliably even in optimized implementations that shed the global interpreter lock to obtain parallelism. This follows the line of previous works [e.g., Daloze et al. 2018; Meier et al. 2016] that present solutions for reliable efficient parallelism.

In order to evaluate the performance overhead of the linearizable size operation, we added the size operation using the methodology described in this paper to various concurrent data structures in Java: a skip list, a hash table and a tree [Sela and Petrank 2022b]. The proposed linearizable size operation executes faster by orders of magnitude compared to counting the elements of a linearizable snapshot. It also demonstrates scalability and insensitivity to the data-structure size. However, obtaining a linearizable size operation does come with some cost, incurring a throughput loss of $1\% - 20\%$ on the original data structure's operations.

The paper is organized as follows. Section 2 introduces some basic terminology. Section 3 surveys relevant work on snapshots. We then describe our methodology, starting with the transformation of a linearizable data structure into one that uses our size mechanism in Section 4, and proceeding with the size mechanism itself: the metadata design is covered in Section 5, and Section 6 describes how

the size is obtained in a wait-free form. We describe possible optimizations to our methodology's implementation in Section 7. In Section 8 we argue about the properties the methodology satisfies. Section 9 presents an evaluation of the methodology applied to different data structures in a variety of workloads. We conclude in Section 10.

2 TERMINOLOGY

An execution is considered *linearizable* [Herlihy and Wing 1990; Sela et al. 2021a] if each method call appears to take effect at once, between its invocation and its response events, at a point in time denoted its *linearization point*, in a way that satisfies the sequential specification of the objects. A concurrent data-structure is *linearizable* if all its executions are linearizable.

A concurrent object implementation is *wait-free* [Herlihy 1991] if any thread can complete any operation in a finite number of steps, regardless of the execution speeds of other threads.

A *set* is a collection of keys without duplicates, supplying the following interface operations: an insert(k) operation which inserts the key k if it does not exist or else returns a failure; a delete(k) operation which deletes k if it exists or else returns a failure; and a contains(k) operation which returns true if and only if k exists in the set.

A *dictionary* (synonymously *map* or *key-value map*) is a collection of distinct keys with associated values, with operations similar to the ones of a set but with values integrated in them. Throughout the paper we will refer only to sets for brevity, but all our claims apply to dictionaries as well.

A compare-and-swap instruction (henceforth CAS) on an object takes an expected value and a new value. It atomically obtains the object's current value and swaps it with the new value if the current one equals the expected value. The return value indicates whether the substitution was performed: its compareAndSet variant returns a corresponding boolean value; its compareAndExchange variant returns the obtained current value.

3 RELATED WORK

A *snapshot object* [Afek et al. 1993] is an abstraction of shared memory made of an array of m cells, supporting two operations: update(i, v) that writes v to the i-th cell of the array, and scan() that returns the current values of all m locations (i.e., a snapshot of the array). The *atomic snapshot* problem is to implement such an object such that its two operations are linearizable and wait-free. Jayanti [2005] presents algorithms that solve the problem with optimal time complexity. We build on the fundamental ideas of Jayanti [2005] in this work to design a wait-free size operation.

However, this scheme is not suited for multiple concurrent scan operations and does not allow other operations (such as reading a specific cell) to occur concurrently. Petrank and Timnat [2013] extend Jayanti's idea and introduce a technique for adding a linearizable wait-free snapshot operation to a concurrent set data structure. In supporting concurrent size operations, we use their method to support multiple concurrent snapshot operations.

An alternative approach by Nelson-Slivon et al. [2022]; Wei et al. [2021] obtains snapshots of concurrent data structures more efficiently, at the cost of higher space overhead. They keep copies of modified nodes and let the snapshot operation advance a timestamp. This timestamp is then used to read the content of the data structure at the time the snapshot was taken. To support such a read of old values, operations on the data structure are responsible to maintaining lists of previous values of mutable fields. Specifically for obtaining the size, one may take a snapshot and use the returned timestamp to traverse the data structure at that time and count elements.

Literature on range queries may be also utilized to take a full snapshot of a data structure. For instance, Arbel-Raviv and Brown [2018] propose to implement range queries using epoch-based memory reclamation.

The above snapshot algorithms can be used to obtain a linearizable size, but using them for this purpose is an overkill. The comparison we make in Section 9 to the algorithms of Petrank and Timnat [2013] and Wei et al. [2021] demonstrates the clear benefit of using our algorithm which is tailored for obtaining the size.

4   DATA-STRUCTURE TRANSFORMATION

In this section we specify how the fields and methods of a linearizable data structure can be modified in order to transform it into a data structure that uses our size mechanism. To efficiently obtain a linearizable size, we keep metadata from which the size may be computed. But unlike previous work, we make the data structure and the metadata change (linearize) simultaneously. The data-structure operations are responsible to maintain the metadata. The main idea is to make sure that updates are not visible to other operations until their metadata is updated. The way to enforce that, is to let each operation complete work for previous related operations, so that it does not view any intermediate states. The details follow.

Successful operations update metadata. The first modification is to let each successful insert or delete operation (i.e., an operation that succeeds to insert a new key or delete an existing key respectively) update the metadata to reflect the operation's effect on the size.

Operations help concurrent operations on the same key update metadata. To prevent operations from exposing inconsistencies similar to the examples of Figures 1 and 2, we linearize data-structure operations that alter the size at the time the metadata is updated to reflect them (informally, linearizing means logically considering them as applied). Dependent data-structure operations are adapted to comply with the new linearization point: if they observe that successful insert or delete operations that they depend on have accomplished their original linearization point, they help them update the metadata so that they reach their new linearization point. For example, a contains(k) that encounters a node with the key k inserted by a concurrent insert(k) cannot return true before ensuring that the insert is reflected in the metadata.

We focus on data structures implementing a set (i.e., a collection of distinct keys) or a dictionary (i.e., a collection of distinct keys with associated values) that provide standard insert, delete and contains operations. In such data structures, an operation on some key logically depends only on operations on the same key. Accordingly, when an operation on some key encounters a node with that key, it acts as follows: if the node is unmarked, it updates the metadata on behalf of the insert operation that inserted the node, to guarantee it is complete (in case the metadata is not yet updated with this insert); if the node is marked as deleted, it ensures the metadata reflects the delete operation that marked the node before proceeding with its own execution.

Successful operations leave a trace for helpers. For operations to help unfinished operations on the same key to update the metadata, they must observe these unfinished operations. To facilitate this, successful insert and delete operations prepare an UpdateInfo object with the information required by helpers for updating the metadata on their behalf, and reference it from the node on which they operate. An insert creates an UpdateInfo object and places a reference to it in the node it is about to link, in a new insertInfo field we add to node objects. A delete also creates an UpdateInfo object, and needs to reference it from the node it deletes. To this end, we rely on a deletion pattern introduced by Harris [2001] and commonly used in concurrent data structures [e.g., Fraser 2004; Harris 2001; Heller et al. 2005; Herlihy et al. 2007; Sundell and Tsigas 2005], where a node is first marked as deleted and then physically unlinked. We install the delete information together with the marking, as demonstrated in the following examples.

When the original marking step already marks the node as deleted by installing an object with information about the delete operation (this is true, for instance, for the binary search tree of Ellen et al. [2010]), then a deleteInfo field referencing the delete's UpdateInfo object may be simply placed inside that object. When the original marking step nullifies the node's value field (as in Java's ConcurrentSkipListMap), in the transformed data structure instead of setting the value field to NULL, it may be set to a reference to the UpdateInfo object. When the original marking step sets a bit in the node's next field (like in Harris's linked list [Harris 2001]), a new deleteInfo field in the node may be set to reference the UpdateInfo object before the marking step.

Metadata is updated before unlinking a marked node. The metadata must be updated on behalf of a delete before the relevant node is unlinked. To see why, assume the metadata is updated to reflect a delete(k) only after it completes to operate on the data structure, including unlinking the node with the key k. In this circumstance, a dependent operation like contains(k) might run in between, and then it will not observe the relevant node and will thus not assist the delete operation update the metadata. Such a contains(k) would return false though delete(k) has not yet updated the metadata, hence, is not yet linearized. Therefore, the metadata is updated before any unlinking attempt: the delete(k) operation itself updates the metadata after marking the node and before unlinking it; and any other operation that attempts to help unlinking the marked node is also required to update the metadata on behalf of delete(k) beforehand.

Adding size functionality. An instance of a SizeCalculator object (described in detail in Section 6.1), responsible for keeping the metadata and computing the size, is referenced from the transformed data structure, and a size method that uses it to retrieve the size is added to the data structure.

4.1 Specific Examples and the SizeCalculator Object

Figure 3 demonstrates how the transformation described above may be applied to standard linearizable linked list, skip list and hash table that implement a set. A similar transformation with minor adaptations will apply to implementations of a dictionary. The transformation may also be applied to search trees with some adaptations.

At the core of our size mechanism stands the SizeCalculator object. We elaborate on this object, responsible for the size calculation, in Section 6.1. For now we just need to be familiar with its interface methods: updateMetadata is called with an UpdateInfo object associated with an insert or delete operation for updating the metadata stored in the SizeCalculator to reflect that operation. This method may be called by both the operation initiator and helpers. We explain in Section 5 how SizeCalculator prevents double update of the metadata on behalf of the same operation. createUpdateInfo is called by insert and delete operations to produce an object that will be published to helpers, with the information required for updating the metadata on their behalf. compute is the method used to retrieve the size of the data structure efficiently (using the metadata).

A SizeCalculator reference field named sizeCalculator is placed in the data structure, and initialized to hold a SizeCalculator instance. Its methods are called in the appropriate places in the data-structure operations, as can be seen in Figure 3. In addition, an insertInfo field referencing an UpdateInfo object is placed in the data structure's node objects. A similar deleteInfo field is placed in the appropriate place, as described above. Since the UpdateInfo record contains the information required for updating the metadata to reflect the associated operation, its content is coupled with the size metadata, so its description is deferred to Section 5.

```
1  INSERT = 0, DELETE = 1
2  class TransformedDataStructure:
3      TransformedDataStructure():
4          initialize as originally
5          sizeCalculator = new SizeCalculator()
6      contains(k):
7          search* for a node with k as originally
8          if not found: return false
9          else if found unmarked node:
10             sizeCalculator.updateMetadata(node.insertInfo, INSERT)
11             return true
12         else: // found marked node
13             sizeCalculator.updateMetadata(node's deleteInfo, DELETE)
14             return false
15     insert(k):
16         search* for the place to insert k as originally
17         if k is already present in an unmarked node:
18             sizeCalculator.updateMetadata(node.insertInfo, INSERT)
19             return failure
20         if k is present in a marked node:
21             sizeCalculator.updateMetadata(node's deleteInfo, DELETE)
22         insertInfo = sizeCalculator.createUpdateInfo(INSERT)
23         allocate newNode as originally with k and the other relevant data, and
                additionally with insertInfo
24         insert newNode as originally (in case of failure proceed as originally)
25         sizeCalculator.updateMetadata(insertInfo, INSERT)
26         return success
27     delete(k):
28         search* for a node with k as originally
29         if not found: return failure
30         if found a marked node:
31             sizeCalculator.updateMetadata(node's deleteInfo, DELETE)
32             return failure
33         sizeCalculator.updateMetadata(node.insertInfo, INSERT)
34         deleteInfo = sizeCalculator.createUpdateInfo(DELETE)
35         mark node with deleteInfo (in case of failure proceed as originally)
36         sizeCalculator.updateMetadata(deleteInfo, DELETE)
37         unlink node
38         return success
39     size():
40         return sizeCalculator.compute()
41 *For each encountered marked node along the search, in case of unlinking it in the
        original algorithm, call sizeCalculator.updateMetadata(node's deleteInfo, DELETE)
        before unlinking it.
```

Fig. 3. A transformed data structure

4.2 Applicability

We focus on a transformation for linearizable data structures that implement the highly prevalent set or dictionary data types. However, the presented ideas may be adapted to other data types. Our transformation recipe requires that the delete operation of the original data structure be linearized at a marking step and not at an unlinking step, to ensure consistency with the size metadata. Otherwise, if operations on k that encounter a marked node with the key k ignore the mark, and consider k as deleted only when its node is unlinked, that might be inconsistent with

the size metadata which is updated to reflect the deletion before the unlinking. Instead, by our requirement, operations on k in the original data structure consider the node as removed when it is marked, and in the transformed data structure they help update the metadata on behalf of the delete(k) that marked the node and only then treat the key k as deleted.

In case of a data structure that linearizes the delete operation at an unlinking step and not in the prior marking step, it is usually not difficult to adjust it to have the marking as the linearization point of delete. We made this adjustment to the binary search tree of Ellen et al. [2010] in order to apply the transformation to it and evaluate its performance.

5 THE SIZE METADATA

In our transformation, operations may help other operations update the metadata. Hence, we must prevent a double update of the metadata on behalf of the same operation. We use metadata which enables threads that operate on the data structure to determine whether the metadata already reflects a certain operation, and update it otherwise. The SizeCalculator object holds the array metadataCounters as the metadata, containing two counters per thread: an insertion counter and a deletion counter, which indicate the number of successful insertions and deletions the thread has performed so far on the data structure. Separating the insertion from the deletion counter allows determining whether an insert (or a delete) operation has been reflected in the counters. If an insert follows a delete, a single counter (incremented on each insertion and decremented on each deletion) cannot indicate if the two operations completed or none of them. Two separate counters allow a simple concise indication of which one of the two operations is reflected in the counters. Next we describe how insertions are handled; deletions are handled similarly.

The per-thread monotonic insertion counters enable to immediately detect whether a certain insert operation by a certain thread is reflected in the metadata, and otherwise ensure that it is reflected via a single CAS: When updateMetadata is called on behalf of a thread T's i-th successful insert operation by either T or helpers, if T's insertion counter is $\geq i$, it leaves the counter as is since the operation is already reflected in the metadata; else, it uses a CAS to increment it from $i - 1$ to i. There is no need to repeat the CAS in case of failure, since that might happen only when another thread succeeds to perform the same update.

To help another operation update the metadata, a helper needs to know on which counter in metadataCounters it should operate and its target value. This dictates the information that the i-th insert operation by thread T leaves for helpers in an UpdateInfo object: T's thread ID, which will be used as an index to the metadataCounters array, and i, which is the counter's target value (which is simply the current value of T's insertion counter in metadataCounters plus 1).

The size may be calculated from metadataCounters as the difference between the sum of insertion counters and the sum of deletion counters. But naively reading the values one by one may result in an inconsistent (non-linearizable) size, because we may obtain a collection of values that never existed simultaneously in the array. We need to obtain a snapshot of values the array counters had at some point in time, but we cannot use locks to achieve this atomicity as we aim for a wait-free size. Next we explain how we manage to achieve that.

6 MECHANISM FOR WAIT-FREE SIZE

The size operation needs to obtain a linearizable snapshot of the metadataCounters array, from which it will be able to compute a consistent size. As the size of this array is twice the number of threads, our solution is the most beneficial (in comparison to computing the size by iterating over a snapshot of the data structure) for applications that usually use data structures with much more elements than the number of threads. If size naively read metadataCounters cell by cell, it could obtain an inconsistent view of the array. For example, consider an execution in which a size

operation starts scanning the array, but after it reads the insertion counter of some thread T, this thread inserts an item and then removes it. Now both T's insertion and deletion counters equal 1, and when the size operation resumes it reads the new value of T's deletion counter and returns -1 as the size. The problem here is that the size operation captured the delete's update of the array but missed the preceding insert's update.

To overcome this problem and obtain a linearizable snapshot of the counters array in a wait-free form, we adopt the basic idea of [Jayanti 2005]'s single-scanner single-writer snapshot algorithm, which is as follows. After an update operation writes to the main array, it checks if a concurrent scan operation is in the process of collecting the main array's values. If so, the scan has maybe already read the relevant cell and missed the new value, thus the update forwards the new value from the main array to a designated second array. The scan operation begins with a collection phase to collect the main array values; before starting the collection it announces it to other operations, and after the collection it announces its completion. In a second phase, the scan retrieves a linearizable view of the array by combining the values it collected with newer values, forwarded to the designated second array by concurrent update operations (namely, each forwarded value is adopted in place of the value that the scan collected from the corresponding cell in the main array). A scan is linearized at the point it announces completing the collection. It might miss values that were written to the main array by some update operations while it was collecting, thus, such operations are retrospectively linearized immediately after the scan's linearization point. We bring the linearization details of update adapted to our context in Section 8.1.

Our size operation acts in the spirit of Jayanti's scan to obtain a view of the metadata array, and data-structure operations that update the metadata array (on behalf of their own operation or to help another operation) act in the spirit of Jayanti's update to inform a concurrent size of a new value it might have missed. However, Jayanti's basic idea supports a single scanner. When multiple size operations execute concurrently, we cannot let each size take an independent snapshot of the metadata array, because the linearization point of a size operation determines the linearization points of updating operations it missed, and concurrent independent size operations might determine contradicting linearization points for concurrent updates. Thus, we need to make sure that concurrent size operations yield the same consistent snapshot of the metadata array.

To this end, we introduce a CountersSnapshot object (on which we detail in Section 6.2). Concurrent size operations coordinate with each other through a CountersSnapshot instance, similarly to concurrent snapshot operations in [Petrank and Timnat 2013] that use a shared object to orchestrate taking a snapshot concurrently. A size operation needs to first obtain a CountersSnapshot instance to operate on. At any given point in time, at most one collecting CountersSnapshot instance (in which the collection has not yet completed) is announced. If a size operation observes such an instance, it operates on it, so that it returns the same size as the size that announced this instance. Otherwise, the size operation produces a new instance, announces it and operates on it.

The CountersSnapshot holds a snapshot array for taking a snapshot of the metadata array. size operations that operate on a certain CountersSnapshot instance collect values into its snapshot array (using a CAS from an initial INVALID value to the value obtained from the metadata array), and operations that concurrently update the metadata array forward their values into the snapshot array. After a collection phase, a size operation needs to compute the size based on the counters in the snapshot array. But the array is not stable—updating operations might be still forwarding values. For all size operations that operate on the same CountersSnapshot instance to agree on the same size, we place a size field in CountersSnapshot, initialized to INVALID. The first size operation to compute a size by traversing the snapshot array and then perform a CAS of the size field from INVALID to its computed size, determines the size value. Concurrent size operations will adopt this

value. Any value forwarded to a counter in the snapshot array after the thread that determined the size read this counter is ignored (and its related operation is linearized after the size).

6.1 SizeCalculator Details

Each transformed data structure holds a SizeCalculator instance associated with it, responsible for calculating the size by holding the metadata and operating on it. The fields of SizeCalculator (as well as the other classes we use) are detailed in Figure 4, and its pseudocode appears in Figure 5.

The SizeCalculator object contains two fields: The first is metadataCounters, holding the size metadata—an array with an insertion counter and a deletion counter per thread, with padding between the cells of each thread and the next one so that the counters of the different threads are placed in separate cache lines to avoid false sharing. The second field is countersSnapshot, that holds the most recent CountersSnapshot instance. In its constructor method (appearing in Line 53), SizeCalculator initializes metadataCounters with zeros, and countersSnapshot with a dummy instance with its collecting flag set to false, to signal that it is not collecting and future size operations should use a new instance.

The compute method is called by the size operation of a transformed data structure. It starts with a collection phase in Lines 58–60. First it needs to announce a new collection if there is no ongoing collection. To this end it calls the private method _obtainCollectingCounters-Snapshot. The latter returns the most recent CountersSnapshot if this instance is still collecting (Lines 63–65), so that the current compute would cooperate with ongoing compute calls. Otherwise, _obtainCollectingCountersSnapshot tries to place a new CountersSnapshot instance in counters-Snapshot using a CAS, and returns the new countersSnapshot value, whether it is an instance placed by itself or an instance placed by another compute call (Lines 66–70). With an activeCounters-Snapshot instance in a collecting mode, compute calls the private method _collect (Line 59), to add all metadataCounters values to activeCountersSnapshot. Then, it unsets activeCountersSnapshot's collecting flag. Now that its collection phase is complete, compute computes the size according to the CountersSnapshot instance maintained in activeCountersSnapshot. This is done using the computeSize method of CountersSnapshot, on which we detail in Section 6.2.

updateMetadata(UpdateInfo(tid, c), INSERT) is called on behalf of the c-th successful insert operation by thread *tid*. We describe how the method handles insertions for convenience; the same applies for deletions by passing opKind=DELETE. The method first updates the relevant counter in the metadata array, i.e., metadataCounters[tid][INSERT], to be c (Lines 78–79), using a CAS to avoid overriding concurrent updates. At this point, the metadata reflects the discussed insertion. Then, according to the described-above scheme, updateMetadata should also forward the counter value c to concurrent size operations that take a snapshot of the metadataCounters array and might

---

```
42  class UpdateInfo:
43      int tid
44      long counter
45  class SizeCalculator:
46      long[][] metadataCounters
47      CountersSnapshot countersSnapshot
48  class CountersSnapshot:
49      long[][] snapshot
50      boolean collecting
51      long size
```

Fig. 4. Classes fields

```
52  class SizeCalculator:
53      SizeCalculator():
54          this.metadataCounters = new long[n][PADDING] // implicitly initialized to
                zeros
55          this.countersSnapshot = new CountersSnapshot()
56          this.countersSnapshot.collecting.setVolatile(false)
57      compute():
58          activeCountersSnapshot = _obtainCollectingCountersSnapshot()
59          _collect(activeCountersSnapshot)
60          activeCountersSnapshot.collecting.setVolatile(false)
61          return activeCountersSnapshot.computeSize()
62      _obtainCollectingCountersSnapshot():
63          currentCountersSnapshot = this.countersSnapshot.getVolatile()
64          if currentCountersSnapshot.collecting.getVolatile():
65              return currentCountersSnapshot
66          newCountersSnapshot = new CountersSnapshot()
67          witnessedCountersSnapshot = this.countersSnapshot.compareAndExchange(
                currentCountersSnapshot, newCountersSnapshot):
68          if witnessedCountersSnapshot == currentCountersSnapshot:
69              return newCountersSnapshot
70          return witnessedCountersSnapshot // our exchange failed, adopt the value
                written by a concurrent thread
71      _collect(targetCountersSnapshot):
72          for each tid:
73              for opKind in (INSERT, DELETE):
74                  targetCountersSnapshot.add(tid, opKind, this.metadataCounters[tid][
                        opKind].getVolatile())
75      updateMetadata(updateInfo, opKind):
76          tid = updateInfo.tid
77          newCounter = updateInfo.counter
78          if this.metadataCounters[tid][opKind].getVolatile() == newCounter - 1:
79            this.metadataCounters[tid][opKind].compareAndSet(newCounter - 1, newCounter)
80          currentCountersSnapshot = this.countersSnapshot.getVolatile()
81          if currentCountersSnapshot.collecting.getVolatile() and
82                this.metadataCounters[tid][opKind].getVolatile() == newCounter:
83            currentCountersSnapshot.forward(tid, opKind, newCounter)
84      createUpdateInfo(opKind):
85          return new UpdateInfo(threadID, this.metadataCounters[threadID][opKind].
                getVolatile() + 1)
```

Fig. 5. SizeCalculator methods

have missed this value. For that, it performs the following steps: (1) obtain the current collecting
CountersSnpashot instance (Line 80); (2) verify it is still collecting (Line 81); (3) obtain the relevant
counter from the metadata array and verify it still holds the value c (Line 82); and then finally, if
these checks pass, (4) call the forward method of the CountersSnpashot instance obtained in the
first step (Line 83). This series of steps is intended to prevent redundant forwarding. Though it is
not yet clear now, it guarantees a constant time complexity for the forward method, as we prove in
Section 8.2.

The last method of SizeCalculator is createUpdateInfo, which is called by insert and delete
operations to obtain an UpdateInfo instance for publication to helpers. createUpdateInfo creates
an UpdateInfo instance with tid=threadID and counter=c, where threadID is the ID of the calling

thread (threadID values are assumed to start from 0, and could be obtained e.g. from a thread-local variable), and c is the current value of the relevant counter (that indicates how many successful operations of the requested kind have been executed by the calling thread so far) plus 1—as the calling thread is about to attempt its c-th operation of this kind.

6.2 CountersSnapshot Details

A new CountersSnapshot instance is announced in SizeCalculator.countersSnapshot each time a new collection phase starts (which happens every time a size operation starts and observes that the last announced CountersSnapshot instance is already not collecting). This instance coordinates the current size calculation among all concurrent size calls that use it to compute the size. Its methods appear in Figure 6 and its fields appear in Figure 4.

The CountersSnapshot object holds a snapshot array called snapshot for taking a snapshot of the metadata array, from which the size will be computed. It also holds a collecting field that indicates whether the collection of values into snapshot is still ongoing, and a size field that will eventually hold the computed size. In its constructor method (appearing in Line 87), CountersSnapshot initializes all its fields. The cells of snapshot are set to INVALID (which may have the value Long.MAX_VALUE for instance), the collecting flag is set to true and size is set to INVALID.

The add method is called by size operations to collect values into the snapshot array. It performs a CAS on the requested cell to the requested value only if the current value is INVALID. Otherwise, another operation has already collected a value to this cell and there is no need to perform another

```
86  class CountersSnapshot:
87      CountersSnapshot():
88          this.snapshot = new long[n][2]
89          setVolatile all cells of this.snapshot to INVALID
90          this.collecting.setVolatile(true)
91          this.size.setVolatile(INVALID)
92      add(tid, opKind, counter):
93          if this.snapshot[tid][opKind].getVolatile() == INVALID:
94              this.snapshot[tid][opKind].compareAndSet(INVALID, counter)
95      forward(tid, opKind, counter):
96          snapshotCounter = this.snapshot[tid][opKind].getVolatile()
97          while (snapshotCounter == INVALID or
                   counter > snapshotCounter): // will execute at most two iterations
98              witnessedSnapshotCounter = this.snapshot[tid][opKind].compareAndExchange(
                       snapshotCounter, counter):
99              if witnessedSnapshotCounter == snapshotCounter:
100                 break
101             snapshotCounter = witnessedSnapshotCounter
102     computeSize():
103         computedSize = 0
104         for each tid:
105             computedSize += this.snapshot[tid][INSERT].getVolatile() -
                       this.snapshot[tid][DELETE].getVolatile()
106         witnessedSize = this.size.compareAndExchange(INVALID, computedSize)
107         if witnessedSize == INVALID:
108             return computedSize
109         return witnessedSize // our exchange failed, adopt the size written by a
                   concurrent thread
```

Fig. 6. CountersSnapshot methods

CAS. Indeed, the value that this size operation fails to add might be missed during the size calculation if it is not forwarded on time by the updating operation associated with it, but this does not foil linearizability, as the updating operation associated with this value is retrospectively linearized after the size.

forward(tid,INSERT,c) is called by updateMetadata that was called on behalf of the c-th successful insert operation by thread tid. It is called after the insertion counter of thread tid in the metadata array is set to c, to ensure that the insertion counter of that thread in the snapshot array contains a value $\geq c$. We prove in Section 8.2 that the forward method shall execute at most two CAS attempts before reaching this goal. forward operates similarly for deletions when called with an opKind=DELETE argument.

The computeSize method is called by the compute method of SizeCalculator (which is called by the data structure's size method), after obtaining a CountersSnapshot instance and completing the collection to this instance, so that its snapshot array is filled with meaningful values (rather than INVALID values). The size is computed as the difference between the sum of insertion counters and the sum of deletion counters in the snapshot array (Lines 103–105). But computeSize may be called by multiple concurrent size operations that operate on the same CountersSnapshot instance, and each of them might compute a different size because values may be concurrently forwarded to the array. Only the first computeSize call to fix the size it computed in the size field (in Line 106), determines the size value that they will all adopt. The rest of them will fail to CAS and will adopt its value.

6.3 Memory Model

The pseudocode brought in this section aligns with our Java implementation [Sela and Petrank 2022b] (evaluated in Section 9) and accesses variables using volatile memory semantics to ensure the visibility required for correctness in accordance with the Java memory model. Read, write and CAS operations on non-final fields of shared objects are performed with volatile semantics (in our Java implementation this is achieved using volatile variables, VarHandles and AtomicReferenceFieldUpdaters). These volatile-semantics accesses are considered synchronization actions, over which the Java memory model guarantees a synchronization order (a total order which is consistent with the program order of each thread, and where a read from a volatile variable returns the last value written to it by the synchronization order). A similar implementation could be designed in C++ according to its memory model guarantees, utilizing the std::atomic library to order accesses to shared memory.

7 OPTIMIZATIONS

The following optimizations may be applied in our methodology, and we apply them in our implementation [Sela and Petrank 2022b] measured in Section 9.

7.1 Eliminate Metadata Update on Behalf of Completed Insertions

When an insertion is complete, there is no need that future operations on the inserted node update the metadata on behalf of that insertion. To this end, after a thread calls updateMetadata to update the metadata on behalf of some insert operation that inserted a node N, it may set N.insertInfo to NULL, to signal that the metadata already reflects the insertion and there is no need to call updateMetadata. Before calling updateMetadata, threads will perform a NULL check to the node's insertInfo to rule if the call is necessary.

We do not propose a similar modification for deletions since deleted nodes are unlinked from the data structure when the deletion completes and cause no more update activity, unlike inserted

nodes which, without the optimization, cause a redundant updateMetadata call on each operation on the node.

7.2 Size Backoff

Each size operation operates on a CountersSnapshot instance it obtains as follows. It collects values into its snapshot array using CAS operations, uses the collected counters to compute the size, and finally sets its size field to the computed size using a CAS, unless another size operation has done that beforehand.

Exponential backoff may be used to reduce contention among concurrent size operations caused by their CAS operations on the snapshot and size fields. Each time a size operation obtains an existing CountersSnapshot instance that was announced by another size operation, it may wait a while to let another size operation complete the size calculation. After waiting, if the calculation is not yet complete (which may be detected by an INVALID value in the size field), it shall collect, compute the size and try to set it on its own.

7.3 Check for an Already-Set Size

There are occasions where we may avoid contention and redundant work by obtaining the size field of CountersSnapshot and returning it in case it does not equal INVALID. This may be done when SizeCalculator's _obtainCollectingCountersSnapshot method observes a concurrent size operation in Lines 65 and 70; at the beginning of CountersSnapshot's computeSize method; and right before computeSize performs a CAS attempt.

8 METHODOLOGY PROPERTIES

8.1 Linearizability

A linearizable data structure transformed according to our methodology to support a size operation, remains linearizable. Recall that an operation has its original linearization point, when its linearization is defined in the original set data structure, but we linearize operations in the transformed data structure only when the metadata is updated. Next, we detail the linearization points of a transformed set's operations, and use them to prove linearizability.

8.1.1 Linearization Points. A size operation is linearized at the announcement of the collection completion. For a successful insert or delete operation, the associated metadata counter is updated to reflect the operation (by either the operation initiator or helpers), and if this update happens when no size is collecting, then the operation is simply linearized at the update. However, if the update is performed while some size is collecting, then the operation is linearized according to that size to comply with its linearization point: if the size takes the operation into account then the operation is simply linearized at the metadata counter update; otherwise, it is retrospectively linearized immediately after the linearization point of that size. Finally, a contains operation and a failing insert or delete operation (namely, one that fails to insert a new key or delete an existing key respectively, and returns a failure), are linearized like in the original data structure, unless the operation they "depend on", namely, the last successful update operation on the same key whose original linearization point precedes their original linearization point (a concurrent successful insert of the same key in case of contains returning true and a failing insert; and a concurrent successful delete in case of contains returning false and a failing delete) is not yet linearized at their original linearization point, in which case they are linearized immediately after this operation is linearized.

In more detail, a size operation is linearized when the collecting field of the CountersSnapshot instance it operates on is set to false for the first time (in Line 60). Regarding a successful insert

operation, the associated metadata counter is updated as follows: a CAS of sizeCalculator.metadata-
Counters[tid][INSERT] to c is performed on behalf of the c-th successful insert operation of thread
tid (in Line 79), where sizeCalculator is the SizeCalculator instance held by the transformed data
structure. For a successful delete operation, the only difference is that DELETE is used as the array
index. As for the linearization point of such an insert or delete operation—if a CountersSnapshot
instance with a collecting field set to true is not announced in sizeCalculator.countersSnapshot
when the metadata counter update is performed, then the operation is linearized at the metadata
counter update (namely, at the CAS in Line 79). Else, the operation is linearized according to this
CountersSnapshot instance: if the size operation that sets its size field read a value $\geq c$ from the
relevant counter (in Line 105), then the operation is linearized at the metadata counter update (as
in the previous case); otherwise, it is linearized immediately after the linearization point of that
size operation.

In the above specification, several operations might be linearized at the same moment—either
operations defined to be linearized immediately after each other, or operations linearized at the
exact same moment (e.g., several size operations operating on the same CountersSnapshot instance).
We order operations that are linearized at the same moment one after the other as follows: size
operations (if any) are placed first; the order among them is arbitrary. Successful update operations
(if any) are placed after the size operations according to their metadata-counter update order. Each
contains or failing insert or delete call that is not linearized at its original linearization point (if
any) is placed right after the successful update operation it depends on; the order among such
operations which are placed after the same successful update is arbitrary.

8.1.2 Linearizability Proof. We prove that our transformation in linearizable using the equivalent
definition of linearizability that is based on linearization points (see [Sela et al. 2021b, Section 7]
and the atomicity definition in [Lynch 1996]). We need to show that (1) each linearization point
occurs within the operation's execution time, and that (2) ordering an execution's operations (with
their results) according to their linearization points forms a legal sequential history.

We prove Property (1) in Claim 8.1 and Property (2) in the full version of the paper [Sela and
Petrank 2022a, Appendix B, Claim B.1].

CLAIM 8.1. *The linearization point of each operation occurs within its execution time.*

PROOF. We begin with the linearization point of a size operation. size calls sizeCalculator.com-
pute, which starts with calling _obtainCollectingCountersSnapshot to obtain a collecting Counters-
Snapshot instance. _obtainCollectingCountersSnapshot returns an instance after its collecting
field has had the value true at some point during this _obtainCollectingCountersSnapshot call:
If this call observes that the current announced instance is collecting (in Line 64), it returns this
instance. Otherwise, this instance cannot be used by the current size because its linearization point
has passed and has possibly occurred before the current size started. Thus, it creates an instance
with collecting set to true, and if it succeeds to announce it using a CAS (in Line 67), it returns this
instance. Else, the failure of its CAS indicates that another thread has in the meanwhile announced a
new instance, with collecting set to true, and the discussed _obtainCollectingCountersSnapshot
call returns such an instance. We showed that in any of the above cases, the collecting field of
the obtained CountersSnapshot instance was still true at some point during the _obtainCollecting-
CountersSnapshot call, hence the size's linearization point does not occur before the compute call
starts. It does occur before it ends, as the collecting field is set to false either when this call
executes Line 60, or before if another compute call has executed this line earlier.

Next, we prove that successful update operations are linearized within their execution time.
A successful insert or delete operation calls updateMetadata with its UpdateInfo instance before

returning. As we prove in Lemma 8.2 below, by the time updateMetadata returns, the operation is guaranteed to be linearized. Additionally, it is not linearized before the operation's execution starts, since it is linearized either at its metadata counter update or at a later point in time, and the update on behalf of a certain operation can only happen after it started and published its UpdateInfo instance.

Lastly, we show that a contains, a failing insert and a failing delete operations are linearized within their execution time. If an operation op of this kind is linearized at its original linearization point, we are done[1]. Otherwise, op is linearized immediately after the linearization point of an operation op_2 it depends on. This happens only in case op observes op_2 and calls updateMetadata on behalf of op_2. By Lemma 8.2, op_2 is linearized by the time this updateMetadata call returns. Hence, op is linearized by that time as well.

<div align="right">□</div>

In the proof of Claim 8.1 we use the following lemmas (Lemma 8.3 is proven in the full version of the paper [Sela and Petrank 2022a, Appendix B]):

LEMMA 8.2. *When a call to* updateMetadata *returns, the operation whose* updateInfo *was passed to the call is guaranteed to be linearized.*

PROOF. Consider a call to updateMetadata on behalf of op, being the c-th successful insert operation by a thread T (a similar proof applies for delete). Denote this call by *updateMetadataForOp*. We need to show that op has been linearized by the time *updateMetadataForOp* returns. By Lemma 8.3, after executing Lines 78–79, the relevant metadata counter's value is $\geq c$. If op is linearized when its related metadata counter is set to c, we are done. Otherwise, the following hold: (1) the counter is set to c when a CountersSnapshot instance with a collecting field set to true is announced in the countersSnapshot field of sizeCalculator (denote this instance by *snapshotAtUpdate*); (2) the size operation that sets *snapshotAtUpdate*'s size field reads, during its size computation, a value $< c$ from the corresponding snapshot counter; and (3) op is linearized immediately after the linearization point of that size operation, namely, immediately after the collecting field of *snapshotAtUpdate* is set to false for the first time. So we need to prove that this collecting field is set to false before the *updateMetadataForOp* call returns. Next, we prove this holds in the various possible scenarios.

If *updateMetadataForOp* obtains a newer CountersSnapshot instance than *snapshotAtUpdate* in Line 80, then we are done, as CountersSnapshot instances announced in SizeCalculator are replaced only after their collecting field is set to false.

Else, if *updateMetadataForOp* observes in Line 81 that *snapshotAtUpdate*'s collecting field value is false, we are done.

Else, if the checks in Lines 81 and 82 pass, *updateMetadataForOp* forwards the value c to the snapshot counter in Line 83. When its forwarding completes, the snapshot counter contains a value $\geq c$. Since we analyze here a case in which the size operation, which sets *snapshotAtUpdate*'s size field, reads a value $< c$ from the snapshot counter, this size must have read the snapshot counter before the forwarding completes, and this read during the size computation occurs only after the *snapshotAtUpdate*'s collecting field is set to false.

The remaining scenario is that *updateMetadataForOp* observes in Line 82 that the metadata counter's value is $\geq c + 1$. We will show that *snapshotAtUpdate*'s collecting field value has been

---

[1]For every linearizable data structure, there exists a selection of linearization points such that each of them is placed during the execution time of the corresponding operation (see the equivalent definition of linearizability based on linearization points in Section 7 in [Sela et al. 2021b]). Each time we refer to the linearization points of the original data structure, we refer to points that satisfy this requirement.

earlier set to false. For the metadata counter to reach the value $c+1$, the thread T must have already started its $(c+1)$-st successful insertion. Prior to that, T has completed op (which is its c-th successful insertion), during which it has called updateMetadata, in a call we denote *updateMetadataByT*. We will next show that by the time *updateMetadataByT* returns, the value of *snapshotAtUpdate*'s collecting field is already false. After *updateMetadataByT* executes Lines 78–79, the metadata counter's value is $\geq c$ by Lemma 8.3. Denote by *currSnap* the value that *updateMetadataByT* obtains in currentCountersSnapshot in Line 80. *currSnap* must be *snapshotAtUpdate*, because otherwise, if it were an earlier CountersSnapshot instance, then when *snapshotAtUpdate* is later announced in the countersSnapshot field of sizeCalculator, the metadata counter's value would already be $\geq c$ as mentioned above, but this contradicts Attribute (2) above which implies that a value $< c$ is collected in *snapshotAtUpdate*. Thus, *currSnap* is *snapshotAtUpdate*. When *updateMetadataByT* checks the value of *snapshotAtUpdate*'s collecting field in Line 81, if it is false then we are done. Otherwise, *updateMetadataByT* calls the forward method to ensure that the value c is forwarded to the snapshot counter. Like in the previous scenario, we analyze here a case in which the size operation, which sets *snapshotAtUpdate*'s size field, reads a value $< c$ from the snapshot counter, hence, this size must have read the snapshot counter before the forwarding completes, and this read during the size computation occurs only after the *snapshotAtUpdate*'s collecting field is set to false. This concludes the proof.

<div align="right">□</div>

LEMMA 8.3. *Consider a call to* updateMetadata *on behalf of op, being the c-th successful* insert *or* delete *operation by a thread T. After this call executes Lines 78–79, the relevant metadata counter's value is $\geq c$.*

8.2 Wait-Freedom and Asymptotic Time Complexity

The size operation in our methodology is wait-free as it completes within a constant number of steps, regardless of other threads' progress. Its time complexity is linear in the number of threads due to its two passes on arrays with per-thread counters: during the collection process (in the _collect method) and the size computation (in the computeSize method).

Our transformation preserves the time complexity and progress guarantees of the data-structure operations, as each call to the udpateMetadata method adds a constant number of steps. This follows from the following claim.

CLAIM 8.4. *Each call to the* forward *method of* CountersSnapshot *(by* updateMetadata*) executes at most two iterations of its while loop.*

Before forwarding, updateMetadata performs several checks in a certain order. Without this careful procedure, delayed threads that run updateMetadata to help old operations could forward stale values to the snapshot array, causing an updateMetadata on behalf of a recent operation to repeatedly fail forwarding. Next, we show how this procedure prevents forwarding stale values.

PROOF. Consider a call to updateMetadata that calls forward and operates on behalf of op, being the c-th successful insert operation by a thread T (a similar proof applies for delete). Denote by *currSnap* the value this call obtains in currentCountersSnapshot (in Line 80) at time denoted t_{obt}. As the snapshot counters are monotonically increasing, it is enough to prove that from t_{obt} and on, only values $\geq c - 1$ may be written to the snapshot counter of *currSnap* that is associated with op.

Note that after obtaining *currSnap* at time t_{obt} and before calling forward, updateMetadata observes that *currSnap* is in a collecting mode (in Line 81), thus, it has been in this mode at t_{obt}. Now, let t_{c-1} be the time in which the metadata counter associated with op is set to $c - 1$. If *currSnap* has been announced in sizeCalculator.countersSnapshot before time t_{c-1}, then it keeps being

announced and in collecting mode at least until time t_{obt}. Thus, the value $c - 1$ is forwarded to the snapshot counter associated with op before time t_{obt}, as otherwise the thread T would not have proceeded from its $(c - 1)$-st successful insert to its c-th successful insert, and we are done since the snapshot counter is monotonically increasing.

Otherwise, *currSnap* is announced in sizeCalculator.countersSnapshot after time t_{c-1}. Two methods write to the snapshot array: add and forward. add is called (in Line 74) with a counter value that is obtained from the metadata array after *currSnap* is announced, hence, after time t_{c-1}, so it writes a value $\geq c - 1$. As for forward, a thread that calls it (in Line 83) to forward a value to the snapshot counter associated with op, performs the following steps in this order: (1) obtain *currSnap* in Line 80—which must happen after *currSnap* is announced and hence after time t_{c-1}; (2) obtain the value of the metadata counter associated with op in Line 82; and then (3) forward this value to *currSnap*'s snapshot array. Since the value is obtained after time t_{c-1}, it must be $\geq c - 1$. □

9 EVALUATION

In this section, we present the evaluation of our methodology on several data structures. The code for the data structures and the measurements is available at [Sela and Petrank 2022b]. We first measure the overhead that the addition of the size mechanism introduces to operations of transformed data structures (in Section 9.1 we break down the overhead by operation type). Then, we demonstrate the benefits of computing a linearizable size in our methodology. We show that it yields a performance better in orders of magnitude than the alternative of taking a linearizable snapshot of the data structure and counting its elements. We further demonstrate the scalability of our methodology and its performance insensitivity to the data-structure size.

Evaluated data structures. We start with three baseline data structures that do not support a linearizable size: a skip list, a hash table and a binary search tree, denoted SkipList, HashTable and BST respectively. The implementation of SkipList is taken from the ConcurrentSkipListMap class of the java.util.concurrent package of Java SE 18. We eliminated methods irrelevant to our measurements and kept the main insert, delete and contains functionality. As for a hash table, we implemented HashTable as a table of linked lists whose implementation is based on the linked list in the base level of SkipList. We use a table of a static size (chosen in a way similar to Java's ConcurrentHashMap to be a power of 2 between 1× and 2× the number of elements; we detail below how we keep the number of elements stable during the measurements). We do not use Java's lock-based ConcurrentHashMap as our hash-table baseline because it deletes items by unlinking without marking (as it does not use a delicate synchronization mechanism but rather coarse-grained locking), thus, our transformation is not applicable to it as is. For BST we use Trevor Brown's implementation [Brown 2018] of the lock-free binary search tree of [Ellen et al. 2010] that places elements in leaf nodes.

We applied our methodology to each of the baseline algorithms, to produce the transformed data structures SizeSkipList, SizeHashTable and SizeBST that support a linearizable size. In the case of the tree, BST linearizes the delete operation at the unlinking and not in the prior marking of the deleted node's parent. Hence, we formed a variant of BST that linearizes delete at the marking step, and then applied our methodology to this variant.

We compare performance of the size operation in the data structures produced using our methodology with a snapshot-based size operation in two data structures supporting snapshots. The first one is SnapshotSkipList—[Petrank and Timnat 2013]'s implementation of a skip list with a snapshot mechanism, which we obtained from the paper's authors. Like our skip list implementations (SkipList and SizeSkipList), it builds on Java's ConcurrentSkipListMap. It uses code from an older version of Java, but the performance degradation incurred by the older version

is negligible and irrelevant to our measurements, due to the immense performance difference between obtaining the size using their snapshot and using our methodology. The size operation is implemented in SnapshotSkipList by taking a snapshot, which produces a snapshot copy of the base level of the skip list, and then iterating it and counting its elements.

The second competitor we compare to is VcasBST-64 [Wei et al. 2021]—a binary search tree with a snapshot mechanism taken from the paper's published implementation [Wei 2021]. It is based on the same implementation by Brown that BST and SizeBST are based on, but uses a modified version of it which batches multiple keys in leaves and stores up to 64 key-value pairs in each tree leaf. To compute the size, we did not use their implementation as a black box, as their interface supplies a snapshot copy of the tree's elements, but such copying is redundant for retrieving the size. Instead, to compute the size we call their snapshot operation that advances the timestamp, and then traverse the tree and sum the number of elements in leaves with a timestamp no bigger than the snapshot timestamp. By this, we save copying all tree elements, and even save iterating the elements one by one—as we simply read the leaf's number of elements (each batched leaf node keeps the number of contained elements). Even though we use this improved size implementation for VcasBST-64, and even though VcasBST-64 uses batched leaves which enables it to perform faster than without them, we will show that our size computation method still outperforms it.

Platform. We conducted our experiments on a machine running Linux (Ubuntu 20.04) equipped with 4 AMD Opteron(TM) 6376 2.3 GHz processors. Each processor has 16 cores, resulting in 64 threads overall. The machine used 64 GB RAM, an L1 data cache of 16 KB per core, an L2 cache of 2 MB for every two cores, and an L3 cache of 6 MB for every 8 cores.

The implementations were written in Java. We used OpenJDK 17.0.2 with the flags -server, -Xms15G and -Xmx15G. The latter two flags reduce interference from Java's garbage collection. We used the G1 garbage collector (using ParallelGC yields similar results).

Methodology. Before each experiment, we fill the data structure with 1M items, except for the experiments that check dependence on the data-structure size, in which we fill the data structure with a varying number of items—1M, 10M or 100M. We chose these sizes in order to measure the performance of the data-structure when it does not fit into the L3 cache.

We run two workloads: an update-heavy workload, with 30% insert operations, 20% delete operations and 50% contains operations, and a read-heavy workload, with 3% insert operations, 2% delete operations and 95% contains operations. These workloads match the read rates suggested by *Yahoo! Cloud Serving Benchmark* (YCSB) [Cooper et al. 2010]—update-heavy workloads with 50% reads and read-heavy workloads with 95% reads (YCSB also suggests a 100%-read workload, but this is less relevant to our case, since it is less likely to have size calls on a data structure that never changes). The left part of Figures 7–13 shows results for the read-heavy workload, and the right part shows results for the update-heavy workload.

Similarly to [Wei et al. 2021], keys for operations during the experiment and for the initial filling are drawn uniformly at random from a range $[1, r]$, where r is chosen to maintain the initial size of the data structure. For example, for $n = 1M$ initial keys and a workload with 30% inserts and 20% deletes, we use $r = n \cdot (30 + 20)/30 \approx 1.67M$.

In all experiments except for the experiments that check how overhead is split by operation type, we repeatedly choose (by the update-heavy or read-heavy proportion) the type of the next operation. However, in the overhead-split measurements (that appear in Section 9.1), we repeatedly choose a uniform type for the next 100 operations, because in these measurements we need to obtain the time it took to execute operations of each type, and obtaining the time it took to execute too few consecutive operations of the same type would impair the time measurement accuracy.

In each experiment, we run w workload threads, performing insert, delete and contains calls according to the update-heavy or read-heavy workloads, and s size threads, repeatedly calling size, except for executions of the baseline algorithms (HashTable, SkipList and BST—evaluated in the overhead and overhead breakdown measurements), for which we run w workload threads only. w and s vary across experiments, and we took $w + s$ to be a power of 2 in most experiments. In each experiment, the threads perform operations concurrently for 5 seconds. Each data point in the graphs represents the average result of 10 runs, after executing 5 preliminary runs to warm up the Java virtual machine. The coefficient of variation was up to 11% in the experiments we present next, and up to 21% in the experiments presented in Section 9.1.
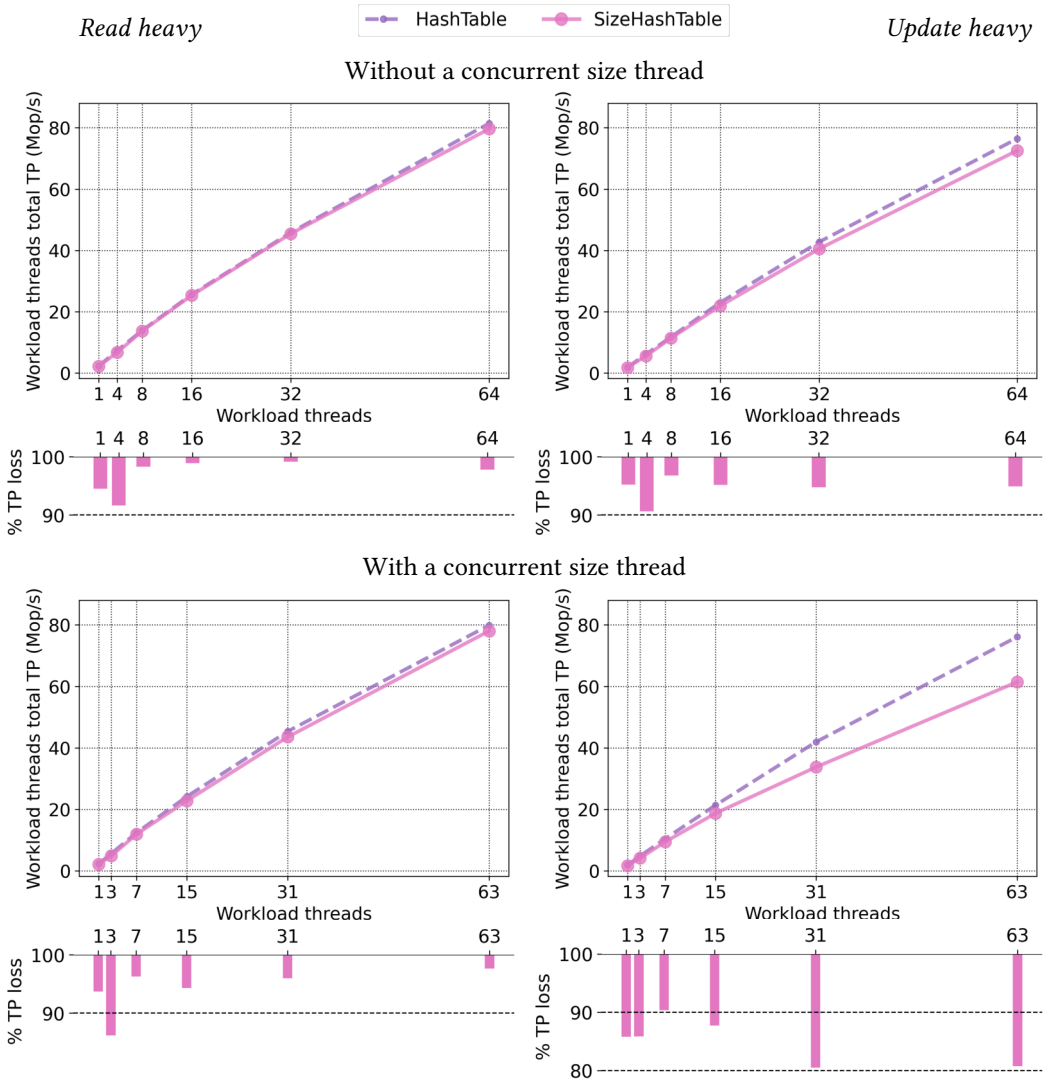

Fig. 7. Overhead on hash table operations

Overhead. We measure the overhead of our methodology on the original data-structure operations by measuring the performance of workload threads—executing `insert`, `delete` and `contains` operations. We compare the total throughput of *w* workload threads, where *w* varies from 1 to 64, for the transformed data structures versus the baseline data structures. The results appear in the top part of Figures 7–9: the results for `SizeHashTable` versus `HashTable` appear in Figure 7, for `SizeBST` versus `BST` in Figure 8, and for `SizeSkipList` versus `SkipList` in Figure 9. They show the overhead when no concurrent `size` operations are executed. To measure the overhead in the presence of `size` calls as well, we similarly run *w* workload threads, where *w* varies from 1 to 63, while also running—for the transformed algorithms only—a concurrent size thread (that executes `size` calls),
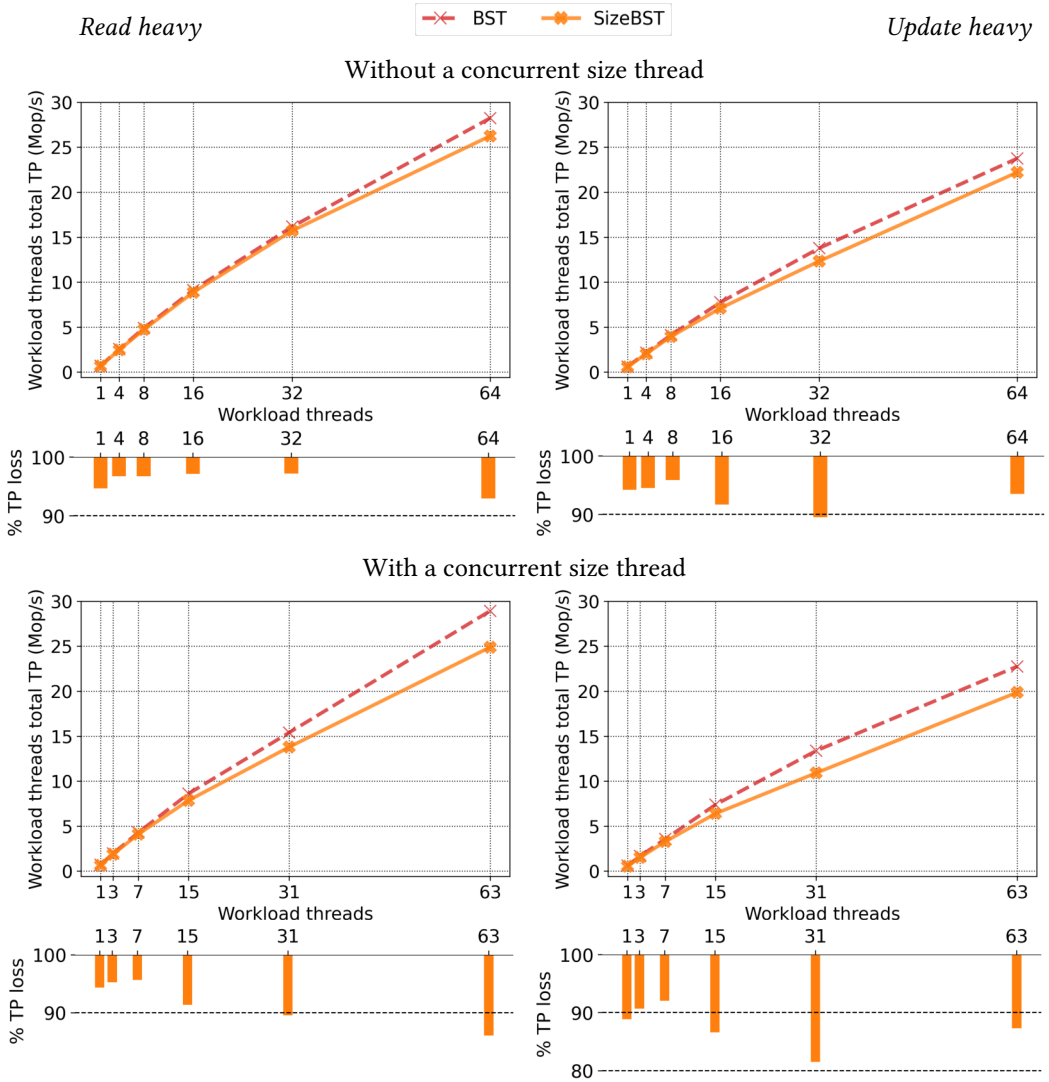

Fig. 8. Overhead on BST operations

and measure the total throughput of the workload threads. The results of these experiments appear in the bottom part of Figures 7–9.

For each experiment, the top graph depicts the number of operations (insert, delete and contains) applied to the data structure per second by the workload threads altogether, measured in million operations per second. The curve of the baseline data structure appears along with the curve of its transformed version with size support. The bottom bar graph shows the throughput of the transformed data structure divided by that of the baseline data structure (in percentages), to demonstrate the throughput loss of the transformed data structure's operations. For instance, 90% signify that the transformed workload threads reach 90% of the throughput of the baseline
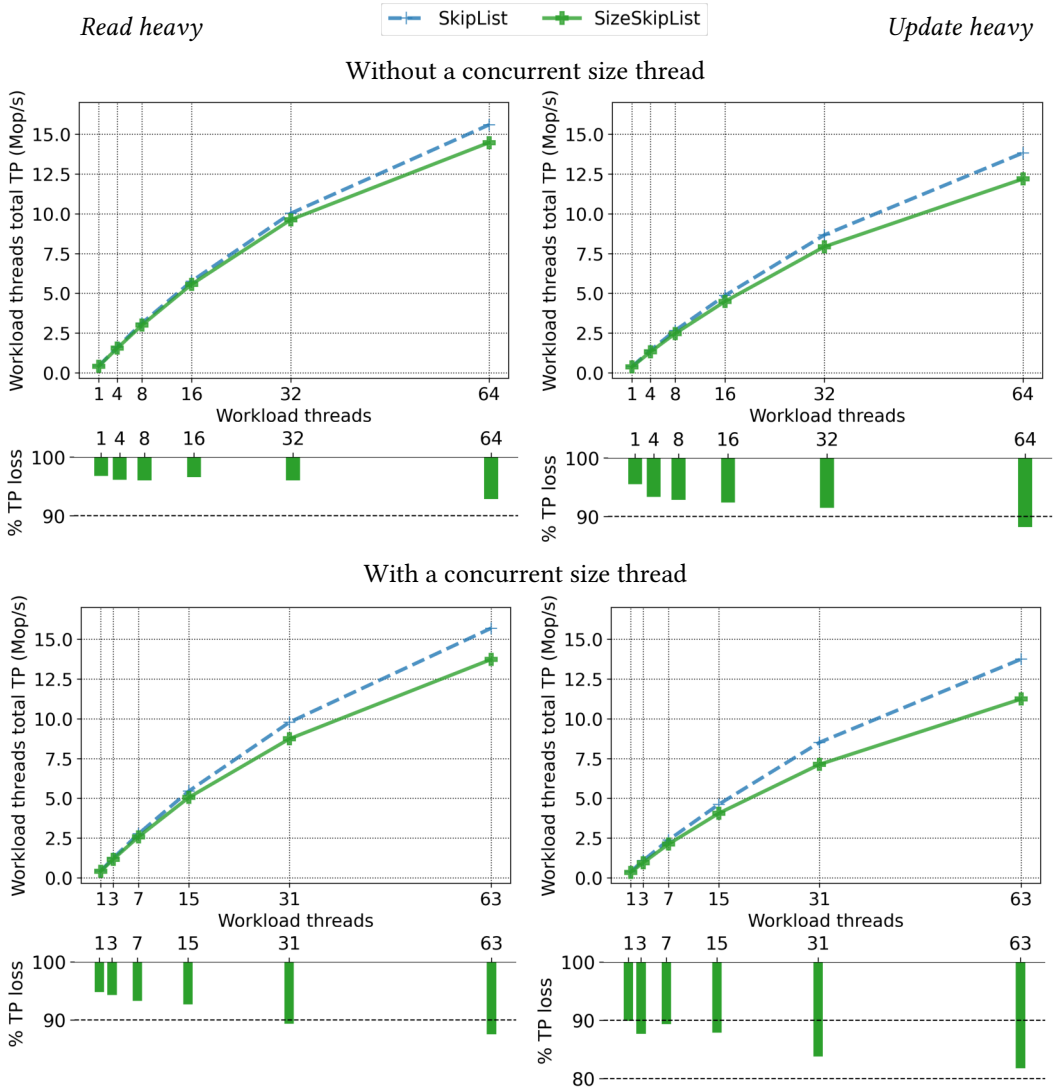

Fig. 9. Overhead on skip list operations

workload threads. The throughput loss is worse for an update-heavy workload than for a read-heavy workload, and worse when a concurrent size is executed. Still, the relative throughput in all experiments varies in the range of 80% to 99%, i.e., a throughput loss of 1% to 20%. We bring a breakdown of the overhead by operation type in Section 9.1.

Varying data-structure size. To measure the effect of the number of elements in the data structure on the size throughput, we run experiments on different data-structure sizes, varying between 1M and 100M, with 32 concurrent threads—one size thread and 31 workload threads. Figure 10 presents the throughput of the size thread, measured in thousand size operations per second. Each curve shows the size throughput for another transformed data structure, per different initial sizes. The results demonstrate that our size-computation methodology is not sensitive to the data-structure size. This is due to the metadata array, on which the size operates instead of traversing the data structure itself. In contrast, obtaining the size using a snapshot-based method causes performance degradation as the size increases, as shown for VcasBST-64 in Figure 11 which presents the corresponding graphs for the competitors. SnapshotSkipList demonstrates a very low size throughput: for a data-structure size of 1M it executes 1.4 size operations per second in average
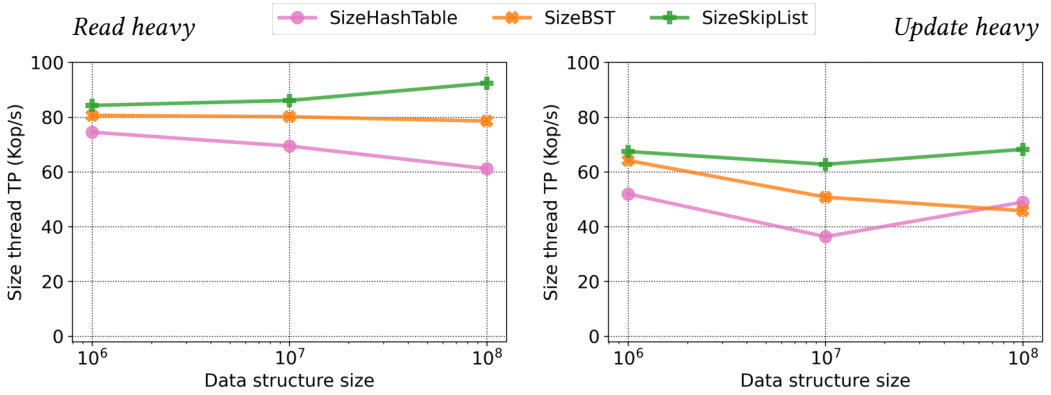

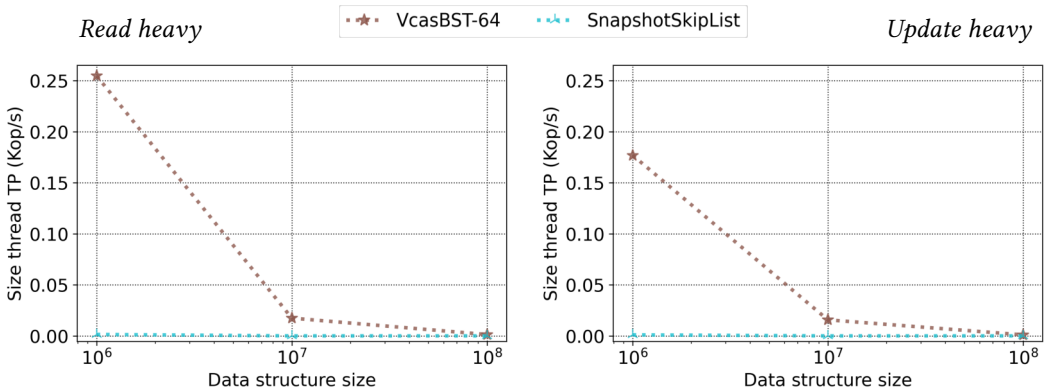Fig. 10. Size throughput as a function of data-structure size

Fig. 11. Snapshot-based size throughput as a function of data-structure size

Fig. 12. Size scalability

for the read-heavy workload and 1 `size` operation per second for the update-heavy workload; and for bigger data-structure sizes it executes less than 1 `size` operation per second.

Scalability. To assess the scalability of the `size` operation, we run s size threads, where s varies between 1 and 16, concurrently with 32 workload threads. Figure 12 presents the total throughput of all size threads, measured in thousand `size` operations per second. It shows results for both our transformed data structures, and the snapshot-supporting data structures which demonstrate inferior performance. For each of our transformed data structures, the throughput improves as number of size threads increases. This demonstrates the scalability of our methodology.

Comparison to snapshot-based size. Our transformed data structures yield a much better throughput than the competitors, as demonstrated in Figures 10–12: `SizeSkipList` demonstrates in these experiments a throughput at least 54806× the throughput of `SnapshotSkipList` (in some experiments, not even a single `size` operation on `SnapshotSkipList` completed within 5 seconds). The throughput of `SizeBST` in these experiments is between $83 - 60423×$ the throughput of `VcasBST-64`. The performance gap between our transformed data structures and `VcasBST-64` is not as large as the gap from `SnapshotSkipList`, because `VcasBST-64` succeeds to improve snapshot performance in comparison to `SnapshotSkipList`, but not without a cost—it pays with higher space overhead.

9.1 Overhead Breakdown by Operation Type

We performed measurements to assess the overhead breakdown by operation type (`insert` / `delete` / `contains`). Similarly to the above overhead measurements, we compare the performance of workload threads for the transformed data structures versus the baseline data structures. But here, in addition to comparing the combined throughput of all three types of operations by all workload threads (i.e., the total number of all operations divided by the total time they ran), we compare also the total throughput of all workload threads *per operation type* (namely, the total number of insertions by all threads divided by the total time the insertions ran, and the same for deletions and for `contains` calls). The results appear in Figure 13. In most measurements, the throughput loss is highest for `insert` operations and lowest for `contains` operations.
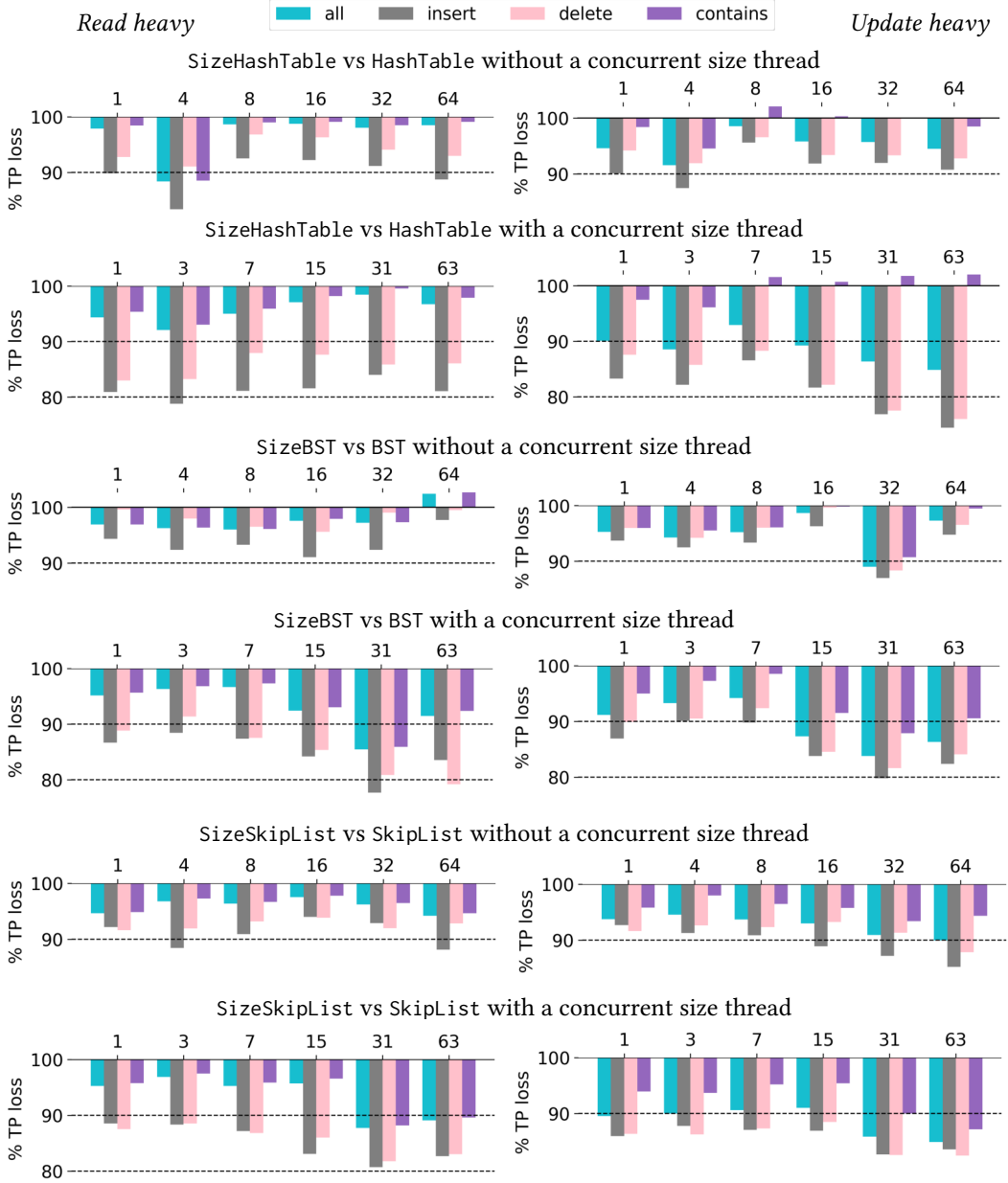
Fig. 13. Overhead breakdown by operation type

10 CONCLUSION

In this work we addressed the problem of obtaining a correct size of a concurrent data structure. We showed that existing solutions in the literature are either inefficient or incorrect (even in a very liberal sense). We then presented a methodology for adding a linearizable size operation to concurrent data structures that implement sets or dictionaries. Our methodology was shown to

yield attractive theoretical properties in terms of progress guarantees and asymptotic complexity. Evaluation demonstrated that while incurring some overhead on the data-structure's original operations, the methodology yields a size operation that provides an orders-of-magnitude performance improvement over existing solutions. We additionally illustrated that the size operation is scalable and insensitive to the data-structure size.

A INACCURACIES OF THE ALGORITHM IN [Afek et al. 2012]

We showed in Section 1 that updating the data structure and the size-related metadata separately, as is done in the algorithm for data-structure size presented in [Afek et al. 2012], makes the algorithm non-linearizable and also not satisfying the weakest correctness principle defined in [Herlihy and Shavit 2008] which requires method calls to appear to happen in a one-at-a-time sequential order. But even if update operations somehow update the data structure and the metadata atomically, the algorithm of Afek et al. [2012] will still not satisfy the above-mentioned correctness principle, due to another issue we elaborate on next. We start with a demonstrating execution, which produces an impossible negative size, hence, no reordering of its method calls forms a legal sequential execution.

Consider an execution with 3 threads executing their operations concurrently: thread T_0 executes an insertion of an item to the data structure, thread T_1 executes a deletion of the same item from the data structure, and thread T_2 executes a size call. First, T_0 and T_1 start executing their operations. T_0 inserts the item to the data structure and then T_1 successfully removes it. After operating on the data structure, they both call the algorithm's $wait\_free\_update$ method to update their values in the array g_mem. During these method calls, they obtain g_seq when its value is 0, and before they proceed to updating g_mem, T_2 starts its size execution. It performs $scan\_seq := FAI(g\_seq)$ which results in $scan\_seq = 1$, and later starts collecting g_mem's values. It obtains $[0, 0]$ from $g\_mem[0]._recent$ and adds 0 to $size$. At this point, T_0 resumes its execution, and writes $[1, 0]$ to $g\_mem[0]._recent$ (this value was already missed by T_2). Then T_1 resumes its execution, and writes $[-1, 0]$ to $g\_mem[1]._recent$. Now T_2 continues scanning the array. It obtains $[-1, 0]$ from $g\_mem[1]._recent$ and accordingly adds -1 to size, and then $[0, 0]$ from $g\_mem[2]._recent$ and adds 0 to size. Subsequently, it returns the incorrect size -1.

To analyze how this happened, we examine the linearization points of the operations on the array g_mem. The linearization point of the size operation by T_2 is when it increments g_seq using FAI; the linearization point of the insertion by T_0 is when it writes to $g\_mem[0]._recent$. The problem stems from the linearization point of the deletion by T_1. It cannot be placed (like erroneously mentioned in [Afek et al. 2012]) when T_1 writes to $g\_mem[1]._recent$, because it must occur before the linearization point of the size operation that observed the deletion. Instead, it is placed in retrospect right before the linearization point of the size operation. This linearization scheme of possibly placing in retrospect linearization points of updates that the size operation observes in its scan long before they write their value, is adopted from the single-scanner algorithm of Riany et al. [Riany et al. 2001], on which the size algorithm in [Afek et al. 2012] is based. The scheme was intended for the atomic snapshot problem, where there are no dependencies between the update operations. However, when handling dependent data-structure operations, they cannot be freely reordered when the size observes them in its scan. In the execution described above, reversing the order of an insertion and a following deletion of the same item is unacceptable, since the deletion cannot succeed if it happens before the insertion, and thus cannot legally decrement the size before the insertion increments it. In conclusion, the correctness problem of the suggested size algorithm stems from the linearization order the size operation dictates: if a size operation S observes during its scan a value, written after S's linearization point by a delayed update U, it dictates in retrospect to place U's linearization point before S's linearization point – which might occur before linearization points of update operations that U depends on.

REFERENCES

2022. *Java Platform Version 18 API Specification*. https://docs.oracle.com/en/java/javase/18/docs/api/index.html

Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *JACM* 40, 4 (1993). https://doi.org/10.1145/153724.153741

Yehuda Afek, Nir Shavit, and Moran Tzafrir. 2012. Interrupting snapshots and the Java™ size method. *J. Parallel and Distrib. Comput.* 72, 7 (2012). https://doi.org/10.1016/j.jpdc.2012.03.007

Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. In *PPoPP*. https://doi.org/10.1145/3178487.3178489

James Aspnes, Maurice Herlihy, and Nir Shavit. 1994. Counting networks. *JACM* 41, 5 (1994). https://doi.org/10.1145/185675.185815

Trevor Brown. 2018. *Java Lock-Free Data Structure Library*. https://bitbucket.org/trbot86/implementations/src/master/java/src/algorithms/published

Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. https://doi.org/10.1145/1807128.1807152

Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of dynamic languages: synchronizing built-in collections. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018). https://doi.org/10.1145/3276478

Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-blocking binary search trees. In *PODC*. https://doi.org/10.1145/1835698.1835736

Keir Fraser. 2004. *Practical lock-freedom*. Ph. D. Dissertation.

Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *DISC*. https://doi.org/10.1007/3-540-45414-4_21

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. 2005. A lazy concurrent list-based set algorithm. In *OPODIS*. https://doi.org/10.1007/11795490_3

Maurice Herlihy. 1991. Wait-free synchronization. *TOPLAS* 13, 1 (1991). https://doi.org/10.1145/114005.102808

Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A simple optimistic skiplist algorithm. In *SIROCCO*. https://doi.org/10.1007/978-3-540-72951-8_11

Maurice Herlihy and Nir Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *TOPLAS* 12, 3 (1990). https://doi.org/10.1145/78969.78972

Prasad Jayanti. 2005. An optimal multi-writer snapshot algorithm. In *STOC*. https://doi.org/10.1145/1060590.1060697

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.* 28, 9 (1979). https://doi.org/10.1109/TC.1979.1675439

Doug Lea. 2004. The java concurrency package (JSR-166). http://gee.cs.oswego.edu/dl/concurrency-interest

Nancy A Lynch. 1996. *Distributed algorithms*. Elsevier.

Remigius Meier, Armin Rigo, and Thomas R Gross. 2016. Parallel virtual machines with RPython. In *DLS*. https://doi.org/10.1145/2989225.2989233

Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling linked data structures for linearizable range queries. In *PPoPP*. https://doi.org/10.1145/3503221.3508412

Erez Petrank and Shahar Timnat. 2013. Lock-free data-structure iterators. In *DISC*. https://doi.org/10.1007/978-3-642-41527-2_16

Yaron Riany, Nir Shavit, and Dan Touitou. 2001. Towards a practical snapshot algorithm. *Theoretical Computer Science* 269, 1-2 (2001). https://doi.org/10.1016/S0304-3975(00)00412-6

Gal Sela, Maurice Herlihy, and Erez Petrank. 2021a. Brief announcement: Linearizability: A typo. In *PODC*. https://doi.org/10.1145/3465084.3467944

Gal Sela, Maurice Herlihy, and Erez Petrank. 2021b. Linearizability: A typo. *arXiv preprint* (2021). arXiv:2105.06737

Gal Sela and Erez Petrank. 2022a. Concurrent Size. *arXiv preprint* (2022). arXiv:2209.07100

Gal Sela and Erez Petrank. 2022b. *Concurrent Size - Artifact for OOPSLA'22*. https://doi.org/10.5281/zenodo.7079982

Nir Shavit and Asaph Zemach. 1996. Diffracting trees. *TOCS* 14, 4 (1996). https://doi.org/10.1145/235543.235546

Håkan Sundell and Philippas Tsigas. 2005. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel and Distrib. Comput.* 65, 5 (2005). https://doi.org/10.1016/j.jpdc.2004.12.005

Yuanhao Wei. 2021. *vcaslib*. https://github.com/yuanhaow/vcaslib

Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time snapshots with applications to concurrent data structures. In *PPoPP*. https://doi.org/10.1145/3437801.3441602