Efficient Concurrent Size

Hen Kas Sharir

Efficient Concurrent Size

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Hen Kas Sharir

Submitted to the Senate
of the Technion — Israel Institute of Technology
Av 5784 Haifa September 2024

This research was carried out under the supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

The author of this thesis states that the research, including the collection, processing and presentation of data, addressing and comparing to previous research, etc., was done entirely in an honest way, as expected from scientific research that is conducted according to the ethical standards of the academic world. Also, reporting the research and its results in this thesis was done in an honest and complete manner, according to the same standards.

Acknowledgements

I would like to thank, first and foremost, my advisor, Prof. Erez Petrank, whose expertise, guidance, and encouragement have been invaluable throughout this journey. I am also profoundly grateful to Gal Sela, with whom I worked closely throughout this thesis. Your collaborative spirit, commitment, and insights have greatly enriched my research. To my family, I am immensely thankful for your unwavering support and belief in me. Your encouragement and patience have been the foundation that allowed me to pursue this work.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

The size of collections or maps, and data structures in general, constitutes a fundamental property that plays a crucial role in various programming paradigms. An accurate and efficient implementation of the size method is required in most programming environments to ensure correct functionality and optimal performance. However, in a concurrent environment, integrating a linearizable concurrent size operation introduces a significant overhead on all operations of the data structure, even when the size method is not explicitly invoked during execution. This overhead can impact the overall system performance, making it a critical consideration in the design and implementation of concurrent data structures. In this work, we present a comprehensive study of synchronization methods aimed at improving the performance of data structures in concurrent environments. Specifically, we analyze and compare the effectiveness of a handshake technique commonly used in conjunction with concurrent garbage collection, an optimistic technique, and a traditional lock-based technique. Our evaluation against the state-of-the-art size methodology demonstrates that the overhead associated with concurrent size operations can be significantly reduced by selecting the appropriate synchronization approach. However, it is important to note that there is no one-size-fits-all solution; different scenarios and levels of contention require different synchronization strategies, as rigorously shown in this study. Our findings align with general trends in concurrent computing, where in scenarios characterized by low contention, optimistic and lock-based approaches tend to be the most effective, while in high contention environments, the handshake approach and the original wait-free method prove to be more efficient.

Abbreviations and Notations

CAS	:	Compare And Swap
BST	:	Binary Search Tree
YCSB	:	*Yahoo! Cloud Serving Benchmark*
op	:	Operation

Chapter 1

Introduction

With the proliferation of multicores in modern computing architectures, the significance of concurrent programming has become acute. Concurrent data structures are a key component of concurrent systems, making their correctness and performance crucial. A fundamental property of data structures is their `size`, which is the number of elements they contain. The `size` method is widely used and its implementation is required for collections in many programmming environments. For example, in Java, in order to implement the elementary `Collection` and `Map` interface classes, one is required to implement the `size` method.

In spite of the importance of the `size` method, until recently it was not known how to compute the `size` of a data structure concurrently efficiently and correctly. Traditional methods were either very slow or incorrect. For example, taking a concurrent snapshot of the data structure and traversing it to count its elements is correct but infeasible (for large data structures). In contrast, naively maintaining a global `size` variable and updating it with each operation is not linearizable (i.e., it is incorrect). The latter is currently in use in some Java libraries with an adequate warning that the result may be inaccurate. See for example collections and maps within the `java.util.concurrent` package, such as `ConcurrentSkipListMap`, and `ConcurrentHashMap`.

Recently, Sela and Petrank [SP22a] proposed a new mechanism for computing a linearizable concurrent `size` for sets and dictionaries, significantly improving efficiency compared to previous methods. While their algorithm represents the first relatively efficient solution to the problem of concurrent size computation, improving upon earlier non-linearizable or inefficient methods, it still incurs certain overheads. Specifically, their mechanism incurs a performance cost of 1–20% on the standard operations of the data structure (i.e., `insert`, `delete`, `contains`). This overhead comes from the requirement that data structure operations take steps to cooperate with a potential concurrent `size` method. The cooperation affects performance of the standard operations even when the `size` operation is not used at all, albeit at a lower overhead. Such an overhead could be problematic, especially for workloads that execute the `size` operation infrequently, or not at all.

The goal of this thesis is to provide the first study of diverse synchronization methods aimed at mitigating the overhead required for supporting a linearizable concurrent `size`. While following the general scheme presented by Sela and Petrank, our study covers a variety of synchronization methods, including handshake synchronization—that is often used with concurrent garbage collection, optimistic synchronization, and lock-based synchronization. All examined methods provide a correct (linearizable) concurrent `size`, and our goal is to identify the best scheme for widespread scenarios (i.e., workload characteristics). We examine the overhead that a concurrent `size` places on standard data structure operations (such as `insert` or `contains`), as well as the efficiency of the concurrent `size` operation itself. These synchronization methods are compared with the original wait-free synchronization proposed by Sela and Petrank [SP22a] (denoted `SP`), which is the most efficient linearizable `size` method existing in the literature.

Let us first clarify why the naive method described above, currently implemented in Java and other languages, does not provide correctness. This method maintains a global size variable, which is updated with each update of the data structure. The primary issue with this approach in a concurrent environment is that the the data structure and the size variable are updated separately and not atomically. Therefore, after updating the data structure and before updating the size variable, there is a period of time of inconsistency between the two that might be exposed by a concurrent `size` operation. To illustrate this, consider a scenario in which four threads insert items into an initially empty data structure (with a size variable initially set to zero). However, they are temporarily preempted before updating the size variable. Then, three other threads remove three objects and decrement the size variable three times. Calling the `size` operation at this point will return a negative size of -3. This is one example of a `size` operation yielding a value that fails to represent the data structure's state at any specific point in time. Other methods for computing the size (that are used in real systems) may exhibit even more severe inaccuracies. We focus on using size metadata to efficiently compute the size, while using appropriate synchronization methods to solve the above issue and yield correct solutions.

Sela and Petrank introduced a (correct) mechanism for concurrent size, which solves the above issue. For a more detailed overview of their method see Chapter 3. But in short the mechanism works as follows. Each thread maintains two local size variables representing the number of objects it has inserted into the data structure and the number of the objects it has removed. This allows a swift local update of the appropriate variable when an update is executed. When an `insert` or `delete` operation updates the data structure, the thread updates its corresponding local size variable. A crucial aspect of the mechanism is ensuring that the updates to the data structure and the local size variables appear atomic to all other threads. Failure to achieve atomicity may lead to inconsistencies, resulting in an incorrect concurrent size calculation. To achieve atomicity in a lock-free manner, a helping mechanism is used. While this incurs some overhead on all data structure accesses, it guarantees correctness in the form of

linearizability. Finally, when a size needs to be computed, a snapshotting mechanism is applied to the local size variables of all threads, in order to obtain a consistent (atomic) view of them all. The snapshot mechanism is crucial for correctness, but it also introduces additional overhead on all data structure operations. This approach is applicable to a variety of data structures and has been tested by Sela and Petrank on three commonly used ones: a hash table, a skip list, and a binary search tree.

While the size mechanism of Sela and Petrank is correct and offers reasonable efficiency for the `size` operation, it carries an overhead on the other data structure operations that may discourage users from adopting it. Notably, an overhead of up to more than 10% on data structure operations is observed even when the `size`() method is not invoked. A natural question that arises is whether alternative synchronization methods could offer a reduced overhead. The primary focus of this thesis is a study of various alternative synchronization methods with the aim of minimizing this overhead. Specifically, we investigate handshake synchronization, optimistic synchronization, and lock-based synchronization techniques. For each of these methods, we present a correct (linearizable) design for concurrent `size`, and we proceed to implement and evaluate each algorithm in comparison to Sela and Petrank's original method, which is the most efficient linearizable option available before this research. We denote Sela and Petrank's original method as the *wait-free* method according to its synchronization method.

The first synchronization method that we study is handshakes, a concept initially introduced in the context of concurrent and on-the-fly garbage collection [DG94, DL93, DKL$^+$00, LP01]. Handshakes allow threads to adapt their behavior to different execution phases. Originally, handshakes were devised to enable threads to cooperate differently when a concurrent garbage collection was active or non-active. In a similar spirit, we harness the handshake mechanism to allow threads to operate (almost) normally (on a fast path) when no `size` operation is active, while requiring them to cooperate fully (on a slow path) when a `size` operation is invoked. This synchronization method offers a tradeoff: It reduces the overhead on all other operations when no `size` operation is in progress. However, the `size` operation itself suffers a reduced efficiency due to the need to invoke the handshake mechanism and wait for the cooperation of all threads before beginning an execution.

The second synchronization method we study is optimistic. In this approach modifications to the data structure are carried out with almost no cooperation, except for announcing each modification. In tandem, the `size` operation optimistically assumes that no updates of the data structure occur concurrently and attempts to optimistically collect the local size variables of all threads in order to sum them and compute the overall size. If the collection is interrupted by a concurrent modification, the collection is retried. After several attempts, the method transitions to a non-optimistic approach, specially designed for this circumstance.

Finally, we investigate the utilization of lock synchronization. In this context, the objective of the synchronization is to prevent the `size` method from executing con-

currently with data structure modifications. To improve performance, we employ a reader-writer lock, with the `size` operation acquiring the write lock and updating operations acquiring the read lock. This setup allows concurrency between the updating operations, while preventing concurrent execution of the `size` operation, thereby ensuring correctness.

The approaches we examine are primarily centered on linearizable data structures implementing set or dictionary data types, but the underlying principles can potentially be applied to various other data types. The optimistic approach and the lock-based one may be applied to any linearizable set or dictionary, while the handshake-based approach imposes an additional requirement detailed in Chapter 2. All our methodologies produce linearizable size-supporting data structures.

As for progress guarantees, the handshake-based methodology preserves the original progress guarantees of `insert`, `delete` and `contains`. 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. The other methodologies preserve the original progress guarantee of `contains` while making the `insert` and `delete` blocking. The `size` operation itself is blocking in all our methodologies, unlike the wait-free guarantee for `size` provided by the original method of [SP22a].

We have carefully designed and implemented a concurrent `size` method with each of the aforementioned synchronization methods (along with additional methods that did not perform well and have thus been omitted from this report). The results appear in Section Chapter 5. It turns out that there is no one-size-fit-all solution. Different scenarios call for different synchronization methods. The observed results vary by the chosen data structure, the contention levels, the frequency of utilizing the `size` operation, and by the workload (read intensive or update intensive). Nevertheless, our findings align with general trends in concurrent computing. In low contention scenarios, optimistic and lock-based approaches exhibit good performance, but their efficacy diminishes when contention increases. In such cases, wait-free and handshake approaches work best.

Hash table operations execute noticeably faster than the BST or skip list operations, resulting in a higher overhead when coordinating with the `size()` computation for the hash table. Specifically, the average overhead of utilizing `size` with the BST is 2.4% (with wait-free synchronization), while for the skip list it is 4.4% (with the handshakes approach). For the hash function, on typical workloads of 5% updates and 95% read operations incur a similar overhead of 4% (with an optimistic approach). However, in write-oriented workloads, this overhead increases to 10% (with wait-free synchronization).

Additionally, we evaluated the scalability of the `size` operation when executed concurrently with data structure updates. Similar trends emerge in this context as well. For specific scenarios, synchronization methods that optimize the `size` scalability of the `size` operation may differ from those that improve the performance of data structure

operations. We expect data structure performance to be a priority for users in the majority of the cases, and thus its performance will probably guide the choice of the adopted synchronization method.

In Chapter 2 we define the basic terms used in this work. Previous solutions are discussed in Chapter 3. A design of `size` with each of the three synchronization methods is described in Chapter 4. The evaluation appears in Chapter 5 and we conclude in Chapter 6. Moreover we bring a correctness proof for the involved handshake mechanism in Section 4.1.5.

Chapter 2

Preliminaries

An execution is said to be *linearizable* when each of its operations appears to be completed in an instant, occurring between its invocation and response and termed its *linearization point*, in accordance with the sequential specification of the data structures. A concurrent data structure is *linearizable* if all its executions meet this criterion [HW90].

A concurrent object is said to be *lock-free* if, at any point in time, at least one of the threads trying to access the object makes progress within a finite number of steps, even if other threads are paused or delayed. Furthermore, a concurrent object framework is *wait-free* when any operation by any thread can be completed within a finite number of steps, without being influenced by the operational speed of other threads [Her91].

A *set* represents a unique assemblage of keys. This assemblage offers specific operational functions: the `insert`(k) function inserts the key k unless it is already present, in which case it signals an error. Conversely, the `delete`(k) function removes k if found; otherwise, it returns an error. The `contains`(k) function verifies the presence of k within the set.

A *dictionary*, also known as a map or a key-value map, represents a distinctive collection of keys, each paired with a corresponding value. Its operations parallel those of a set, with the distinction of incorporating values. We will mainly discuss sets in this thesis, yet the observations and assertions made here are also true for dictionaries.

All our methodologies may be applied to linearizable sets and dictionaries, while the handshake-based methodology also imposes the following additional requirement (which is the same requirement that the original wait-free mechanism imposes on the data structures it may transform to size-supporting data structures [SP22a]): The `delete` operation in the original data structure must perform a marking step before physically unlinking. Crucially, the linearization point of the `delete` operation within the original data structure must be at this marking step.

A *snapshot* of an object provides a captured state of that object at a specific point in time. For a set, this is essentially an atomic view of all the elements currently in it.

A *readers-writer lock* is a synchronization primitive allowing multiple threads to

read or write to shared data without interfering with each other. It is made of two locks: a read lock and an exclusive write lock. Multiple threads can hold the read lock at the same time. If a thread has the read lock, no other thread can acquire the write lock. Only one thread can hold the write lock at any given time. If a thread has the write lock, no other thread can acquire either the read lock or the write lock.

The *get-and-add* operation is an atomic operation available in many concurrent architectures. It atomically retrieves the current value of a variable and adds a specified number to it (which can be negative).

Lastly, *compare-and-swap* (`CAS`) is an atomic operation available in concurrent architectures. It checks an object's current value against an expected value. If they match, it updates the object with a new specified value. The outcome of this action is given in two ways: its compareAndSet version yields a boolean value indicating the success or failure of the operation, whereas its compareAndExchange version returns the obtained current value.

Chapter 3

Previous size solutions

Traditional solutions for determining the size of data structures include solutions that are either non-linearizable or highly inefficient. A simple solution that is occasionally used in practical settings is to traverse the data structure and count its elements. However, this approach can lead to significant errors and is highly inefficient for large data structures. A second approach currently utilized in some systems is to let each updating operation report the resulting number of elements in metadata accessible to the `size` method for size computation. This solution is not linearizable. An extension proposed in [AST12] also falls short of linearizability [SP22a]. Although linearizability is the standard correctness model, other models can provide more relaxed guarantees, such as offering upper and lower bounds on the size. For instance, this problem could be addressed using other linearizability criterias such as the one proposed in [RK23], which would allow the `size` method to return any value within a legitimate range of linearizable values.

Finally, a solution that traverses a snapshot of a data structure using a linearizable snapshot [PT13] does yield a linearizable solution but not an efficient one.

Recently, Sela and Petrank [SP22a] introduced a novel linearizable solution with significant improvements upon efficiency compared to prior methods. This approach, denoted `SP` in what follows, utilizes two local variables per thread, one of which tracks the number of insertions and the other the number of deletions applied by the thread to the data structure. Each update of the data structure begins by executing the actual update, followed by an update to the associated count (of inserts or deletes). To obtain linearizability, an operation does not take effect until the count is properly updated. If a thread encounters an inserted node, for which the inserting thread has not yet updated its count, it helps updating the count before using this node. A similar action is taken with a node marked for deletion. The result of this helping mechanism is that operations can be regarded as being linearized at the time when the thread's local count is updated. To ensure linearizability, the `size` method needs to take a linearizable snapshot on all local counts to obtain a consistent view of them and compute the overall size of the data structure. They utilize a specialized snapshot mechanism for that,

which requires adjustment of the `insert`, `delete` and `contains` operations to support size snapshots. Further details appear in [SP22a].

An essential implementation component in the `SP` approach is the `UpdateInfo` class. An object of this class is installed by `insert` and `delete` operations in the nodes on which they operate, posting information about themselves, to enable other operations to observe modifications being made to these nodes and detect whether they should assist updating the associated metadata and how. Each inserted node and each node marked for deletion is associated with an `UpdateInfo` object. Particularly, the installment of an `UpdateInfo` object in a `deleteInfo` field in a node signifies its marking for deletion.

Chapter 4

A Study of Synchronization Methods for size

This thesis studies three synchronization methods: handshakes, optimistic, and lock-based for `size`. For each of these synchronization methods, we design a linearizable `size` method focusing on improving efficiency. We then evaluate all of them against the original wait-free method of [SP22a]. In this section, we describe the design of size with each of these synchronization methods.

4.1 Handshakes

4.1.1 Overview

In this section, we propose a handshake-based approach for evaluating the size of a data structure. The underlying idea is that many applications often perform data structure operations without requiring the data structure's size. Consequently, it appears inefficient to impose an overhead continuously for a potential `size` operation, especially when it is infrequently executed. To address this, we aim to segment the program into distinct phases, ensuring that the overhead for the `size` operation is only incurred when it is actually executed. When no `size` operation is triggered, the program should execute with minimal overhead. Essentially, our goal is to establish a fast path, resembling normal data structure operations when no `size` operation is invoked, and a slow path that manages concurrent `size` operations, incurring the necessary overhead.

Technically, the design involves creating both a fast path and a slow path and establishing a mechanism for transitioning between these execution phases. We have opted for a straightforward approach: the fast path employs the original data structure operations without incurring overhead, while the slow path executes the implementation from [SP22a]. However, transitioning between these phases is complex and involves some overhead on both the slow and fast paths. The mechanism for phase transitions, particularly upon starting to execute a `size` operation, entails notifying all active threads

that a `size` operation is imminent, prompting them to shift to executing the slow path. This can be done by halting all operations until all threads complete their currently executing operations and switch to the slow path. After completing the `size` operation, all threads halt again until they can collectively revert to the fast path. This strategy ensures that either all threads are engaged in a fast path with no concurrent `size`, or all threads execute the slow path while `size` concurrently executes. However, these phase changes are costly, as threads pause and remain idle while waiting for all other threads to complete their operations and switch to the required mode of operation. This cost becomes acute if phase changes occur frequently.

In the realm of concurrent garbage collection, a similar synchronization issue arises where threads must coordinate their actions during the execution of garbage collection, yet can continue on a fast path when the collector is inactive. We adopt a method commonly employed in memory management known as *handshakes* [DG94, DL93, DKL$^+$00, LP01]. The concept involves an initiator prompting a phase change by incrementing a global counter, thereby asking all other threads to acknowledge the phase changed. Each thread periodically checks this variable and responds by setting its local thread counter to match the global counter. Once all threads have responded, the handshake concludes, enabling the initiator to proceed, assured that all threads are aware of the phase change.

Notably, after responding to the handshake (by setting its local counter), a thread can continue to execute operations (on the slow path), with no need to wait for other threads, avoiding stalls required in the simple method. However, this setup introduces a complication. As threads respond to the handshake individually and continue to execute operations, concurrent threads operate in different modes (fast and slow paths) simultaneously. Managing these concurrent thread executions in different modes requires careful handling. In fact, it turns out that in order to guarantee correctness, prior to initiating the execution of the `size` operation, we need to run two handshakes. While the necessity for two handshakes might not be intuitively evident, we will demonstrate that a single handshake does not guarantee correctness. Subsequently, we will prove that executing two handshakes is sufficient for ensuring correctness[1].

An important optimization in this context is to disregard idle threads that are not engaged in operating on the data structure. These threads do not impact the size of the data structure and as a consequence the `size` operation does not need to wait for them to acknowledge a phase change that marks the beginning of a `size` execution. On the one hand this is beneficial due to the time saved by not waiting for these threads. On the other hand, this requires a thread that becomes active (i.e., starts an operation) to notify all other threads that it is not idle anymore. Such notifications (including a memory fence) incur a non-negligible cost on very fast operations such as operations on

[1] In the domain of garbage collection, a study in [DL93] initially used two handshakes, but subsequent research in [DG94] identified the need for a third handshake in the presence of multithreading to guarantee correctness.

the hash table. However, we cannot do without treating idle threads specially, because threads cannot cooperate with a handshake when they are engaged in other activity, unrelated to the data structure at hand.

4.1.2 A size design with handshakes

The process of implementing handshakes for concurrent size entails designing the fast and slow paths, along with the mechanism to transition between different phases of execution. We will run two handshakes before starting the size calculation. During the first handshake, each thread will respond and start using the slow path. In the second handshake threads will only respond, acknowledging the handshake, without taking any further action. A thread that starts an operation after the first handshake was initiated, leaving an idle state, will use the slow path, maintaining the invariant that all threads execute only in the slow path after the first handshake completes. Once the size calculation is completed, a corresponding announcement by the `size` suffices to let the threads revert to using the fast path. In Section 4.1.5 we will explain why a single handshake is insufficient at the beginning of a `size` execution and then argue that two handshakes are enough.

4.1.3 Data-structure transformation

To implement the fast and slow paths we employ two metadata arrays: `fastMetadataCounters` designated for fast path operations and `metadataCounters` for slow path operations. They are both fields of a `HandshakeSizeCalculator` object we will later describe. When an operation of a thread T executes the fast path, it is guaranteed that no `size` operation can execute concurrently. Therefore, it is guaranteed that, during the fast path execution of the operation, the fast path metadata associated with the thread T is accessed only by T. Consequently, no synchronization is required for updating this part of the metadata, enhancing the efficiency of the fast path. Conversely, slow path operations are executed in a manner similar to the wait-free design of [SP22a], denoted `SP` in what follows. This design takes extra care to allow data structure operations and updates to the slow path metadata to run concurrently with a `size` operation.

Our data structure transformation scheme, detailed next, is illustrated in Figure 4.3. For each `insert` or `delete` operation performed on the data structure, we have it first announce starting an operation (leaving the idle mode), then execute one of two new operations—`slow_op` or `fast_op`, and eventually announce returning to idle mode. The `fast_op` operation executes the original operation and then updates the metadata associated with fast operations on a successful `insert` or `delete`. The `slow_op` executes the code of the `SP` algorithm.

The slow path and fast path use different methods to mark an object as deleted. The slow path (following the `SP` algorithm) installs an `UpdateInfo` object in a `deleteInfo` field in order to mark the node, whereas the fast path, following the code of the original

data structure, uses a simpler mark (typically setting the value field to NULL or setting the next field to point to a marker node). Since slow and fast operations might run concurrently during the execution of the first handshake and at the end of executing size, we further adjust both slow and fast operations of the data structure to treat a node as marked when either the associated deleteInfo field is not NULL or when the node is marked according to the original data structure's marking scheme. This change allows both slow and fast operations on the same key to execute concurrently. To complete the adjustment, we need to address the following issue: in the SP methodology, the data structure's operations call this.sizeCalculator.updateMetadata to help concurrent delete operations whenever they encounter marked nodes that they need to unlink. Slow operations in our transformation follow the same behavior, but only for nodes that slow operations marked (by installing an UpdateInfo object in a deleteInfo field in the node).

We do not modify the contains operation to use two different paths; it always runs in a "slow" mode, following the design of the SP algorithm. We did explore building slow and fast paths for contains, but an evaluation indicated no performance enhancement. By exclusively employing the slow path, threads executing contains can bypass the necessity to engage in handshakes, which are typically utilized for transitioning between fast and slow paths. Consequently, this eliminates the requirement to announce their departure from the idle phase at the beginning of a contains operation. It turned out that employing a slow path for contains did not exhibit a noticeable slowdown compared to using a fast path that announces non-idle status at the beginning of the operation.

For the size calculation we employ the HandshakeSizeCalculator and HandshakeCountersSnapshot objects, which include both methods from SizeCalculator and CountersSnapshot of the SP methodology (_collect, updateMetadata and createUpdateInfo of SizeCalculator, and add and forward of CountersSnapshot) which we do not repeat here, and new methods that appear in Figures 4.1, 4.2 and 4.4.

In the SP methodology, a size operation calculates the size by calling SizeCalculator.compute, which performs the calculation using a CountersSnapshot object. It first obtains a CountersSnapshot instance that has the value true in its collecting field (because otherwise the instance is associated with a size operation whose linearization point has already passed). It then performs some collection process of the metadata values into that CountersSnapshot instance. At this point it sets the collecting field of this instance to false and then computes the size using this instance. The moment of setting collecting to false is the linearization point of the size operation. Further details appear in [SP22a]. Our HandshakeSizeCalculator.compute adds transitioning from the slow path to the fast path and back as well as handling the fast metadata, as follows.

After obtaining a HandshakeCountersSnapshot instance with collecting=true, if the size operation was the one to install this instance in the HandshakeSizeCalculator, it proceeds to initiate a sequence involving two handshakes with other threads concurrently accessing the data structure for insertion or deletion (Line 30). In practical terms, these

```
1  class HandshakeSizeCalculator:
2      HandshakeSizeCalculator():
3          this.sizePhase = 4
4          // The 3 following arrays are implicitly initialized to zeros
5          this.fastMetadataCounters = new long[n]
6          this.metadataCounters = new long[n][2]
7          this.opPhase = new int[n]
8          this.countersSnapshot = new HandshakeCountersSnapshot()
9          this.countersSnapshot.collecting.setVolatile(false)
10     fastUpdateMetadata(opKind):
11         tid = ThreadID.threadID.get()
12         if opKind == INSERT:
13             this.fastMetadataCounters[tid].setVolatile(1+this.
                   fastMetadataCounters[tid].getVolatile())
14         else:
15             this.fastMetadataCounters[tid].setVolatile(-1+this.
                   fastMetadataCounters[tid].getVolatile())
16     setOpPhase(opPhase):
17         tid = ThreadID.threadID.get()
18         this.opPhase[tid] = opPhase
19     setOpPhaseVolatile(opPhase):
20         tid = ThreadID.threadID.get()
21         this.opPhase[tid].setVolatile(opPhase)
22     getSizePhase():
23         return this.sizePhase.getVolatile()
24     compute():
25         currentCountersSnapshot = this.countersSnapshot.getVolatile()
26         if not currentCountersSnapshot.collecting.getVolatile():
27             newCountersSnapshot = new HandshakeCountersSnapshot()
28             witnessedCountersSnapshot = this.countersSnapshot.
                   compareAndExchange(currentCountersSnapshot,
                   newCountersSnapshot)
29             if witnessedCountersSnapshot == currentCountersSnapshot:
30                 currentSizePhase = _doFirstAndSecondHandshakes()
31                 _collect(newCountersSnapshot)
32                 fastSize = this._computeFastSize()
33                 newCountersSnapshot.collecting.setVolatile(false)
34                 c = newCountersSnapshot.computeSize(fastSize)
35                 this.sizePhase.setVolatile(currentSizePhase + 2)
36                 return c
37             currentCountersSnapshot = witnessedCountersSnapshot
38         return _waitForComputing(currentCountersSnapshot)
```

Figure 4.1: `HandshakeSizeCalculator` interface methods

```
39      _performHandshake(sizePhase):
40          for each tid:
41              wait until this.opPhase[tid].getVolatile() == IDLE_PHASE or this.
                    opPhase[tid].getVolatile() ≥ sizePhase
42      _doFirstAndSecondHandshakes():
43          wait until (currentSizePhase = sizePhase.getVolatile()) % 4 == 0
44          this.sizePhase.setVolatile(currentSizePhase + 1)
45          _performHandshake(currentSizePhase + 1)
46          this.sizePhase.setVolatile(currentSizePhase + 2)
47          _performHandshake(currentSizePhase + 2)
48          return currentSizePhase + 2
49      _computeFastSize():
50          fastSize = 0
51          for each tid:
52              fastSize += this.fastMetadataCounters[tid].getVolatile()
53          return fastSize
54      _waitForComputing(currentCountersSnapshot):
55          while true:
56              currentSize = currentCountersSnapshot.size
57              if currentSize != INVALID_SIZE:
58                  return currentSize
```

Figure 4.2: `HandshakeSizeCalculator` auxiliary methods

handshakes are facilitated through the use of two fundamental components: a `sizePhase` field, to which `size` operations write the phase with which they wish the other threads will synchronize, and an `opPhase` array field, sized to accommodate all running threads, where they publish their current phase for `size` operations to inspect.

When a thread is not actively engaged in either an `insert` or a `delete` operation on the data structure, it is in an `IDLE_PHASE` status. This state essentially informs the `size`-performing threads that they can disregard the phase of the thread in question. Conversely, during an `insert` or a `delete` operation, a thread indicates its activity to the `size`-performing threads by setting its corresponding cell in the `opPhase` array to the appropriate value: If it identifies no concurrent `size` operation (according to a `sizePhase` value that reflects no ongoing `size` operations), it takes the fast path after setting its cell to `FAST_PHASE` using a volatile write (to make sure `size` operations—which accordingly perform volatile reads of the `opPhase` values—see it). Else (if it identifies a concurrent `size` operation), it takes the slow path after setting its cell to the phase value that was published by a concurrent `size` operation in the `sizePhase` field, in order to communicate its acknowledgment of the `sizePhase`.

Back to the `size`-performing thread, to initiate a handshake it increments the `sizePhase` field by 1. Subsequently, it awaits the synchronization of all other threads with this `sizePhase` by inspecting their respective cells in the `opPhase` array. Synchronization is achieved when the value in each thread's cell is either equal to `IDLE_PHASE` or it is greater than or equal to `sizePhase`.

The `size`-performing thread carries out two such handshakes one after another. After all threads have successfully synchronized with it in the second handshake,

```
INSERT = 0, DELETE = 1
IDLE_PHASE = 0, FAST_PHASE = 1
class TransformedDataStructureWithHandshakes:
    TransformedDataStructureWithHandshakes():
        Initialize as originally.
        this.sizeCalculator = new HandshakeSizeCalculator()
    fast_op(k):
        Perform the original operation*. For an insert or a delete operation
            that succeeded call this.sizeCalculator.fastUpdateMetadata with
            the relevant opKind.
        Return the result of the original operation.
    slow_op(k):
        Perform the transformed operation defined in [SP22a]** and return its
            result.
    op(k): // this transformation is for insert/delete operations
        this.sizeCalculator.setOpPhaseVolatile(FAST_PHASE)
        currentSizePhase = this.sizeCalculator.getSizePhase()
        if currentSizePhase%4 == 0:
            ret = fast_op(k)
        else: // Some thread runs size()
            this.sizeCalculator.setOpPhase(currentSizePhase)
            ret = slow_op(k)
        this.sizeCalculator.setOpPhase(IDLE_PHASE)
        return ret
    contains(k):
        return slow_op(k)
    size():
        return this.sizeCalculator.compute()
*Consider also nodes with a non-NULL deleteInfo field as marked for deletion.
**Do not call this.sizeCalculator.updateMetadata before unlinking nodes
    marked using the original data structure's marking scheme (call it only
    for nodes marked using a non-NULL deleteInfo field).
```

Figure 4.3: A transformed data structure with handshakes

```
class HandshakeCountersSnapshot:
    HandshakeCountersSnapshot():
        this.snapshot = new long[n][2]
        setVolatile all cells of this.snapshot to INVALID
        this.collecting.setVolatile(true)
        this.size.setVolatile(INVALID)
    computeSize(fastSize):
        computedSize = fastSize
        for each tid:
            computedSize += this.snapshot[tid][INSERT].getVolatile() -
                this.snapshot[tid][DELETE].getVolatile()
        this.size.setVolatile(computedSize)
        return computedSize
```

Figure 4.4: `HandshakeCountersSnapshot` methods

the `size`-performing thread can safely perform the size computation: It first collects slow metadata values into the obtained `HandshakeCountersSnapshot` instance according to the `SP` scheme by calling `_collect`, then sums the fast metadata values by calling `_computeFastSize`. At this point it sets the `collecting` field to `false` (which constitutes the linearization point of the operation similarly to the `SP` method), and then completes

the size calculation—computes the size derived from slow operations (by summing slow metadata snapshot values following the snapshot mechanism of the SP methodology) and adds it to the previously-computed size derived from fast operations.

Upon completion of the computation process, the size-performing thread increments the sizePhase field by 2, to inform the other threads that the size operation has finished, and they can return to a fast path of execution in their following operations.[2]

When a size operation observes another ongoing size operation, it does not follow the execution scheme described thus far for size operations; instead, it calls the _waitForComputing method to wait for the other size to complete its computation and adopt its computed size value once it becomes available.

The handshake-based methodology preserves the original progress guarantees of insert, delete and contains, as it adds a constant number of non-blocking instructions to each of them.

4.1.4 Optimization: size operations join the previous handshake

We could further improve the performance of size operations in this methodology by allowing concurrent size operations to join the previous handshake when possible. This improvement is possible when the previous size operation has already set the collecting field of its HandshakeCountersSnapshot object to false in Line 33 and has yet to exit the second handshake by incrementing the sizePhase value by 2 in Line 35. In that case, a later concurrent size that has successfully installed its HandshakeCountersSnapshot object in Line 28 and has observed a sizePhase value $\equiv_4 2$ in Line 43 can attempt to increase the sizePhase by 4 using a compareAndSet operation. This modification to the sizePhase value together with the altering of the write in Line 35 to use a compareAndSet operation with an expected value of currentSizePhase ensures that if the size that wants to join the previous handshake successfully modifies the sizePhase then the previous size (who was yet to execute Line 35) will not modify sizePhase's value again. By successfully adding 4 to the sizePhase value, the size that wants to join the previous handshake maintains a sizePhase $\equiv_4 2$, thus maintaining correctness as the only insert and delete operations that may run are ones on the slow path which did not run concurrently with fast operations. This optimization will be implemented such that instead of executing Line 43, a size operation will obtain sizePhase only once and then check if the obtained value is $\equiv_4 2$. If the condition is not met, it will proceed as before, performing two handshakes one after the other. Otherwise, it will try to write sizePhase+4 to sizePhase using a compareAndSet call with an expected value of sizePhase. If successful, there is now no need for two handshakes and it can proceed directly to Line 31. If the compareAndSet call fails, then the previous size managed to execute Line 35 and once again, size has to

[2] The decision to increment the sizePhase by 2 is a straightforward yet practical optimization. It allows the determination of the size operation phase using (sizePhase mod 4), instead of (sizePhase mod 3). Computing the remainder of a number after division by 4 is highly efficient in hardware, involving a bitwise and operation with the constant 3.

perform two handshakes from the beginning as before. Along with these modifications, the value of the variable `currentSizePhase` in the `_doFirstAndSecondHandshakes` function should be maintained to uphold the current value of the `sizePhase` field.

4.1.5 Correctness of the handshake-based methodology

4.1.5.1 Two handshake rationale

In the `SP` approach, dependent operations help the update operation they depend on to update the metadata before carrying out their own operations. In our handshake-based methodology that involves a fast path, on the other hand, we have fast operations which do not verify that the metadata is updated on behalf of operations they depend on before operating on the data structure. Thus, if a `size` operation executes the size calculation concurrently with an update operation that in turn ran concurrently with a fast operation that depends on that update operation, the fast operation might update the metadata before the update operation it depends on updates the metadata, and the size calculation might run between those metadata updates thus taking into account only the fast operation and not the operation it depends on, which could hinder linearizability. An example for such non-linearizable execution is given in Figure 4.5.

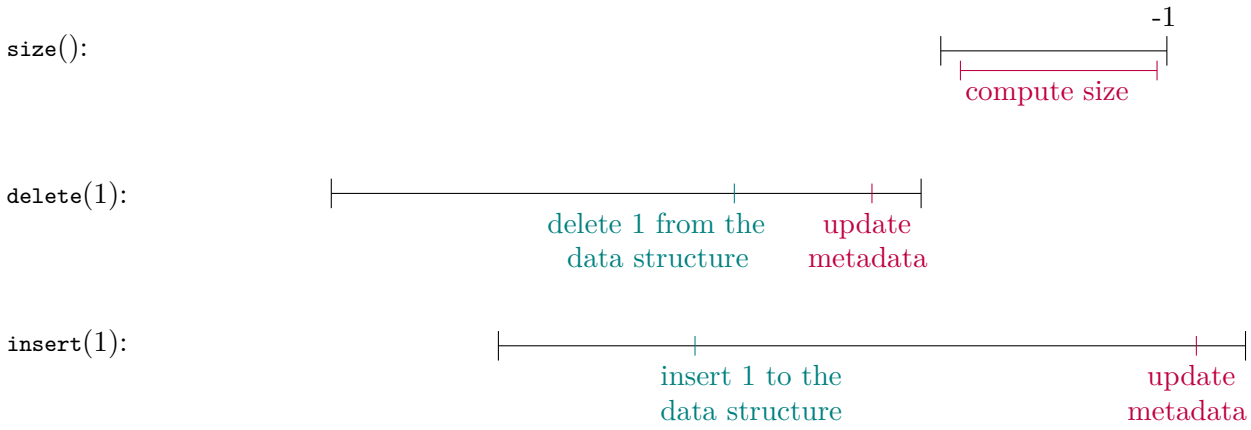

Figure 4.5: `size` concurrent with an `insert` operation that ran concurrently with a dependent `delete` operation

To ensure linearizability we must prevent `size` operations from calculating the size from the metadata concurrently with an update operation that in turn ran concurrently with a fast operation that depends on that update operation. We will have `size` operations calculate the size only when slow operations that have not run concurrently with fast operations are the only ones to run, as is the case in the `SP` approach. Therefore, our size calculation will be correct like the `SP` approach.

To this end, a thread executing a `size` operation initiates a handshake with the other threads, to wait for them to complete their ongoing operations (if any) and guarantee that their following operations will be slow. But the size calculation can still not be

performed after completing this handshake, as there could be threads that acknowledge the handshake and start running slow operations before other threads complete their fast operations that were unaware of the handshake. This scenario is demonstrated in Figure 4.6, where an `insert` operation acknowledges the handshake initiated by the `size` operation and starts running in slow mode, while a concurrent `delete` operation that started operating in fast mode before the beginning of the handshake is still running. The `size` operation must wait until such slow operations complete. The first handshake initiated by a `size` operation guarantees that no fast operations run once it is finished, but slow operations that ran concurrently with fast operations may still be running. A second handshake then waits until all ongoing operations complete and guarantees that once finished, no more operations that ran concurrently with fast operations are executed (but rather only slow operations that have not run concurrently with fast operations).

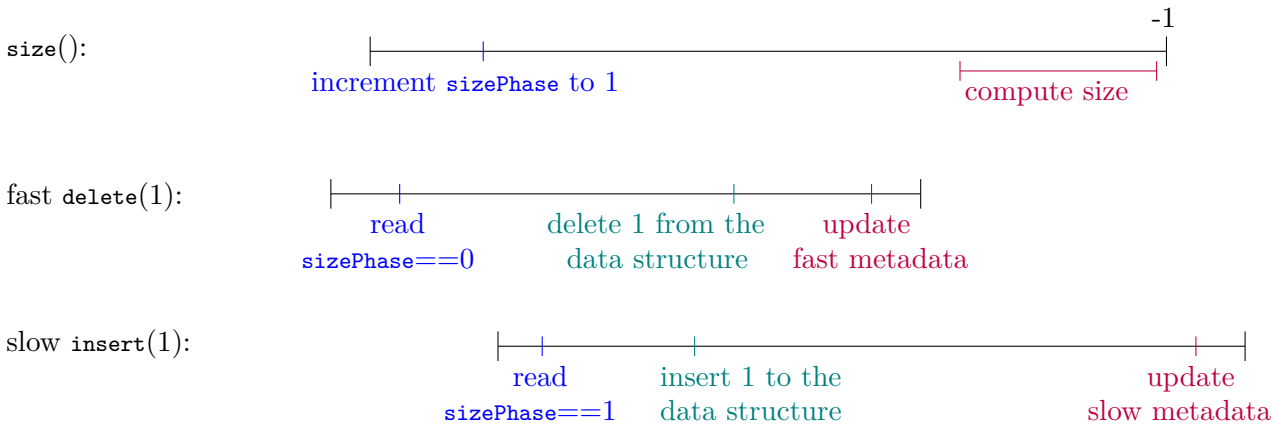

Figure 4.6: An execution with a single handshake in which the size computation is concurrent with a slow `insert` that ran concurrently with a fast dependent `delete`

4.1.5.2 Linearization points

Our handshake-based methodology is linearizable. We detail the methods' linearization points next, and bring the linearizability proof in Section 4.1.5.3.

A `size` operation that has managed to successfully install its `HandshakeCountersSnapshot` object in Line 28 is linearized as in [SP22a]. Otherwise, a `size` operation which had to wait on another `size`'s `HandshakeCountersSnapshot` object (i.e., called `_waitForComputing`) is linearized at the linearization point of the `size` which installed the `HandshakeCountersSnapshot` object it obtained in Line 25.

Fast operations are always linearized according to the original linearization point. Any successful slow `insert` or `delete` operation that has seen the phase number of the first handshake (i.e., has read `size_phase` $\equiv_4 1$ in Line 72) is linearized according to its original linearization point. Otherwise, a successful slow `insert` or `delete` operation that has seen

the number of the second handshake phase (`size_phase` $\equiv_4 2$ in Line 72) is linearized according to the linearization point defined in [SP22a] unless a depending concurrent fast operation is linearized between its original linearization point and the linearization point in [SP22a] in which case it is linearized at the latter of the original linearization point and right after the linearization point of the last concurrent `size` operation that does not take the operation into account (due to a scenario we elaborate on in the next paragraph). Note that two slow operations can be linearized at the same time (immediately after a `size` operation), in which case we order them according to the order of the linearization points defined in [SP22a]. The linearization points of `contains` and failing slow `insert` or `delete` operations follow a methodology similar to that presented [SP22a] with the slight modification that we now linearize them based on the new linearization points of `insert` or `delete` operations. In detail, an operation *op* which is a `contains` or a failing slow `insert` or `delete` is linearized at the original linearization point unless the operation it depends on (namely, the last successful update operation on k whose original linearization point precedes *op*'s original linearization point) is a slow operation that has yet to be linearized (according to the new linearization point defined above) at *op*'s original linearization point, in which case we linearize *op* immediately after that operation is linearized.

The scenario in which the metadata related to a successful slow `insert` or `delete` operation, is updated later than the linearization point of a dependent fast operation, requires special handling of the slow operation's linearization point. It cannot be linearized like in [SP22a] at its metadata update as it occurs after the linearization point of the dependent fast operation. Instead, we linearize it beforehand as we demonstrate on the scenario illustrated in Figure 4.7, where a slow `insert`(1) updates the metadata only after a dependent fast `delete`(1) is linearized. The `size` operation in the figure is the one that set `sizePhase` to the value obtained by the `insert`. This is the last `size` operation that does not take the `insert` into account (this `size` operation does not take the `insert` into account since it must have completed before the fast `delete` ran which happened in turn before the `insert` updated the size metadata; following `size` operations perform handshakes before computing the size and may hence compute the size from the metadata only after the `insert` completes including its metadata update). This `size` operation may be linearized either before the `insert` inserts 1 to the data structure or afterwards. If it is linearized first, then we linearize the `insert` when it inserts 1 to the data structure. Else, we linearize the `insert` right after the linearization point of the `size`. In both cases, the chosen linearization guarantees that the `insert` is linearized after the `size` which did not take it into account and before any dependent operation.

4.1.5.3 Linearizability proof

In this section we prove the linearizability of the handshake-based methodology presented in Section 4.1, using the linearization points stated in Section 4.1.5. We will

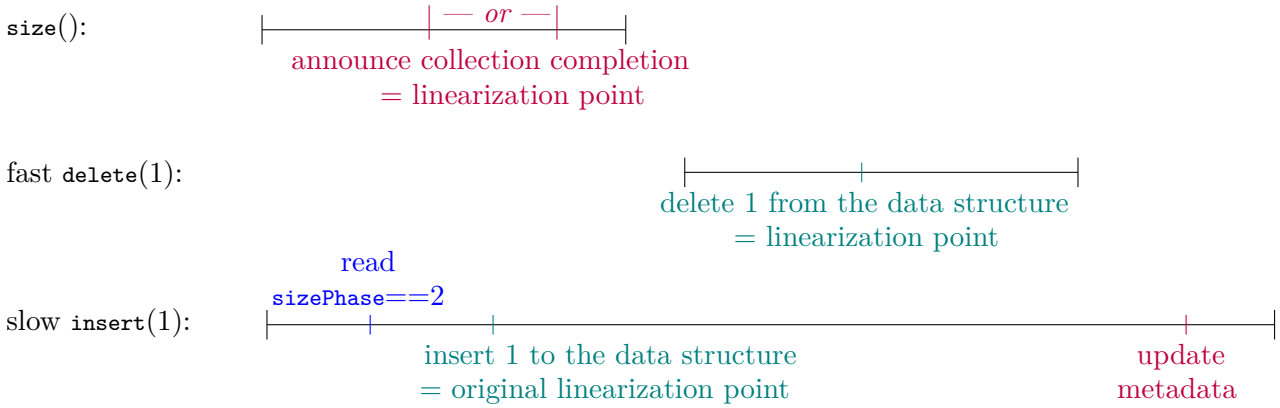

Figure 4.7: An execution with a slow `insert` that takes part in a second handshake with `size` and updates the metadata after a dependent fast `delete` executes

prove linearizability in a manner similar to [SP22a]. This requires us to show (1) each linearization point occurs within the operation's execution time, and (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 4.1.1 and Property (2) in Claim 4.1.7.

Compared to the proof presented in [SP22a], the main distinction in this context lies in having to consider new linearization points as well as to closely examine the handshake mechanism. It is essential to establish and prove significant observations related to this mechanism. These observations are important to determining which types of operations (fast path operations and slow path operations) can be executed concurrently to a `size` operation and which can not. Our linearizability proof will rely closely on these observations.

Linearization points occur within operations' intervals

Claim 4.1.1. *The linearization point of each operation occurs within its execution time.*

Proof. For fast `insert`, fast `delete`, slow `insert`, slow `delete` and `contains` operations that are linearized according to the original linearization point, the claim follows from the linearizability of the original data structure.[3] For `size` operations, they are linearized in the same manner as in [SP22a]. As the `CountersSnapshot` object each `size` operation holds is obtained in the same way as in [SP22a] and the operation is linearized according to [SP22a], the same arguments in *Claim 8.1* in [SP22a] can be applied and the claim is valid. For successful slow `insert` or slow `delete` that are not linearized at

[3]There is a selection of linearization points for every linearizable data structure such that each of them is placed within the execution period of the relevant operation. We only use linearization points that meet this criterion when we discuss linearization points of the original data structure.

the original linearization point, from Lemma 4.1.4 and Lemma 4.1.8 we conclude that the linearization point of such an operation occurs within its execution time. It is left to show the claim that for `contains` and failing slow `insert` or slow `delete` operations that are not linearized according to the original linearization point. Let *op* be such an operation. Since *op* was not linearized at its original linearization point and from the way we defined our linearization points we know that there must exist some slow operation op_2 that *op* depends on and has yet to be linearized at *op*'s original linearization point. In this case, *op* is linearized immediately after the linearization point op_2. Since op_2 was yet to be linearized at the time of *op*'s original linearization, we know that op_2's linearization must occur after *op*'s original linearization point. Moreover, because op_2 is a successful slow operation by Lemma 4.1.4 we conclude that *op* observes op_2's metadata and calls `updateMetadata` on behalf of op_2. By Corollary 4.1.3, op_2 must be linearized by the time `updateMetadata` returns and thus *op* is linearized by that time as well. Therefore, *op* is linearized within its execution time. □

When proving Claim 4.1.1 we rely on the validity of *Claim 8.1* from [SP22a], this validity holds only if *Lemma 8.2* in [SP22a] still holds. We next show in Corollary 4.1.3 that this is indeed the case. To show this we recall the following lemma from [SP22a], the correctness of these lemma follows from the proof of *Lemma 8.2* in [SP22a]:

Lemma 4.1.2. *When a call to `updateMetadata` returns, the operation whose `updateInfo` was passed to the call is guaranteed to have reached the linearization point defined in [SP22a].*

Now, as a direct conclusion from Lemmas 4.1.2 and 4.1.4:

Corollary 4.1.3. *When a call to `updateMetadata` returns, the operation whose `updateInfo` was passed to the call is guaranteed to be linearized.*

The following lemmas are also used as part of Claim 4.1.1's proof.

Lemma 4.1.4. *The linearization point of any successful slow `insert` or `delete` operation occurs no later than the linearization point of that operation as defined in [SP22a].*

Proof. Denote by *op* some successful slow `insert` or `delete` operation. If *op* is linearized at its original linearization point or at the linearization point as defined in [SP22a] then by Lemma 4.1.6 the lemma holds.

Otherwise, *op* is linearized immediately after a concurrent `size` operation that does not take it into account and that is linearized after *op*'s original linearization point: If the `size` operation is collecting when *op* is performing its metadata update, then by definition *op*'s linearization point as defined in [SP22a] is immediately after that `size` and we are done [4]. Otherwise, if the `size` operation is not collecting when *op* is

[4]Note that because we order successful update operations that are linearized at the same time in the same manner as in [SP22a] then our linearization point will be the same as the one in [SP22a].

performing its metadata update, then the *op*' linearization point as in [SP22a] will be at the metadata update. This metadata update must occur after `size` finished collecting; otherwise, by Lemma 4.1.5 `size` should have seen the update. Thus, because `size` is linearized when the collecting field is set to false, then the `size` operation must be linearized before *op*'s linearization point as in [SP22a], and consequently *op* will be linearized before its linearization point as in [SP22a]. □

Lemma 4.1.5. *Consider a call to `updateMetadata` on behalf of op, which is the c-th successful slow `insert` or `delete` operation by a thread T. After this call executes Lines 78-79 in [SP22a], the relevant metadata counter's value is $\geq c$.*

Lemma 4.1.6. *The linearization point of each successful insert or delete operation as defined in [SP22a] occurs after its original linearization point.*

Lemma 4.1.5 correlates to *Lemma 8.3* in [SP22a] and Lemma 4.1.6 is shown as part of *Lemma B.3* in [SP22a], the proofs of both of these Lemmas remain the same.

The linearization is legal In what follows, we denote the set's i-th successful `insert`(k) operation (by *i-th* we refer to the linearization order, namely, to the i-th successful `insert`(k) to be linearized) by `insert`$_i(k)$, its linearization time by $t_{insert_i(k)}$, the time of its original linearization by $orig\_t_{insert_i(k)}$ and the time of its linearization point defined in [SP22a] by $g\_t_{insert_i(k)}$ (this is defined only for slow operations). We further denote the analogous `delete` operation and its related times by `delete`$_i(k)$, $t_{delete_i(k)}$, $orig\_t_{delete_i(k)}$, and $g\_t_{delete_i(k)}$.

Claim 4.1.7. *Consider a sequential history formed by ordering an execution's operations (with their results) according to their linearization points defined in Section 4.1.5. Then operation results in this history comply with the sequential specification of a set.*

Proof. The correctness of the results of successful update operations follows from Corollary 4.1.11. Now, let us examine the results of `contains` operations and failing update operations. Let *op* be such an operation on a key k, and let the operation it depends on be $insert_i(k)$ for some $i \geq 1$, a similar proof can be made for a `delete` operation. Because $insert_i(k)$ is an insertion then *op* must be a failing insertion or a `contains` returning true. Our goal is to show that `insert`$_i(k)$ is indeed the last successful operation on key k to be linearized before *op*. Let $orig\_t_{op}$ be the original linearization moment of *op* and t_{op} its actual linearization point according to Section 4.1.5. If *op* is a slow operation that is linearized immediately after $t_{insert_i(k)}$ then we are done. Otherwise, *op* must be linearized according to its original linearization point. We begin by showing that $t_{insert_i(k)} < t_{op}$. If $insert_i(k)$ is linearized according to its original linearization point then $t_{insert_i(k)} = orig\_t_{insert_i(k)} \overset{\text{the way we chose } i}{<} orig\_t_{op} = t_{op}$. Otherwise, $t_{insert_i(k)}$ must be a slow operation and because *op* is linearized at its original linearization point we know that by time t_{op} it must be that $insert_i(k)$ is already linearized

(this is due to the way we defined our linearization points in Section 4.1.5). Therefore, $t_{insert_i(k)} < t_{op}$. We are left to show that if there is a successful delete operation linearized after $\texttt{insert}_i(k)$ then $t_{op} < t_{delete_i(k)}$. If there is no such operation then we are done, otherwise, from the way we chose i and due to the linearizability of the original data structure it holds that $t_{op} = orig\_t_{op} < orig\_t_{delete_i(k)} \overset{Lemma\ 4.1.8}{\leq} t_{delete_i(k)}$. $\implies$ In all cases it holds that $t_{insert_i(k)} < t_{op} < t_{delete_i(k)}$ and we are done.

Finally, we analyze the linearization of a `size` operation. Denote such an operation by *op*. From Corollary 4.1.21 we know there are no concurrent fast operations when `fastCompute()` is executing thus the fast counter array values stay untouched when `fastCompute()` is performed. Therefore, any fast operation that *op* has seen must be linearized before *op* and any fast operation that *op* didn't see must be linearized after *op*.

Moreover, from Corollary 4.1.21 all slow operations executed during `compute()` have seen the second handshake phase and thus are linearized accordingly. Let j be the value that *determiningSize* obtained from the insertion counter of some thread T in *countersSnapshot*.`snapshot`. We will prove that T's j-th successful slow `insert` is linearized before *op* and T's $(j+1)$-st successful slow `insert` (if such operation occurs) is linearized after it (the proof for `delete` is the same). We will denote T's j-th successful slow `insert` by $\texttt{insert}_{T,j}$, its linearization time as $t_{insert_{T,j}}$, its original linearization time as $orig\_t_{insert_{T,j}}$ and its linearization point according to [SP22a] as $g\_t_{insert_{T,j}}$. The same notations are used accordingly for T's $(j+1)$-st successful slow `insert` as $\texttt{insert}_{T,j+1}$, $t_{insert_{T,j+1}}$, $orig\_t_{insert_{T,j+1}}$, $g\_t_{insert_{T,j+1}}$.

According to the linearizability proof in *Claim B.1* in [SP22a] we know that $g\_t_{insert_{T,j}} < t_{op}$ and together with Lemma 4.1.4 we get that $t_{insert_{T,j}} < t_{op}$.

If T's $(j+1)$-st successful slow `insert` has not seen a second handshake then it is linearized according to the original linearization point and from the linearizability proof in *Claim B.1* in [SP22a] we know that $t_{op} < g\_t_{insert_{T,j+1}}$. Therefore, using Corollary 4.1.21 and since $g\_t_{insert_{T,j+1}}$ happens during $insert_{T,j+1}$'s execution of $slow\_op(k)$ we conclude that $t_{op} < t_{insert_{T,j+1}}$.

In the case where $insert_{T,j+1}$ had a concurrent fast operation which was linearized between $orig\_t_{insert_{T,j+1}}$ and $g\_t_{insert_{T,j+1}}$ then $insert_{T,j+1}$ is linearized at the latter of $orig\_t_{insert_{T,j+1}}$ and sz_t where sz_t is immediately after the linearization of the last concurrent `size` operation that did not see $insert_{T,j+1}$. From the linearizability proof in *Claim B.1* in [SP22a] we know that $t_{op} < g\_t_{insert_{T,j+1}}$.

If $insert_{T,j+1}$ and *op* are not concurrent then from Claim 4.1.1 and *Claim 8.1* in [SP22a] which states that $g\_t_{insert_{T,j+1}}$ happens within $insert_{T,j+1}$'s execution we conclude that $t_{op} < t_{insert_{T,j+1}}$.

If $insert_{T,j+1}$ and *op* are concurrent then because we know *op* does not see $insert_{T,j+1}$ and because $insert_{T,j+1}$ is linearized at the latter of 2 points one of which being im-

mediately after the linearization of the last concurrent `size` operation that did not see $insert_{T,j+1}$ we conclude that $insert_{T,j+1}$ must be linearized after op thus $t_{op} < t_{insert_{T,j+1}}$. □

The proof of Claim 4.1.7 uses the following:

Lemma 4.1.8. *The linearization point of a successful $insert_i(k)$ or $delete_i(k)$ happens in its original linearization point or after it.*

Proof. For fast and slow operations which are linearized according to their original linearization point the lemma holds immediately. Otherwise, the operation is either linearized at the linearization point from [SP22a] which is shown to occur after the original linearization point as part of *Lemma B.3* in [SP22a], or it is linearized at the latter of two options one of which being the original linearization point.

⇒ For each key k and each $i \geq 1$: $orig\_t_{insert_i(k)} \leq t_{insert_i(k)}$ and $orig\_t_{delete_i(k)} \leq t_{delete_i(k)}$. □

Observation 4.1.9. *The original linearization points of successful insertions and deletions of each key k are alternating.*

This follows from the linearizability of the original data structure and the sequential specification of a set.

Lemma 4.1.10. *For each key k and each $i \geq 1$:*

$$orig\_t_{insert_i(k)} \leq t_{insert_i(k)} < orig\_t_{delete_i(k)} \leq t_{delete_i(k)} < orig\_t_{insert_{i+1}(k)}$$

Proof. From Lemma 4.1.8 we know that $orig\_t_{insert_i(k)} \leq t_{insert_i(k)}$ and $orig\_t_{delete_i(k)} \leq t_{delete_i(k)}$. Thus it is only left to show that $t_{insert_i(k)} < orig\_t_{delete_i(k)}$ and $t_{delete_(k)} < orig\_t_{insert_{i+1}(k)}$.

$t_{insert_i(k)} < orig\_t_{delete_i(k)}$:
If $insert_i(k)$ is a fast operation or a slow operation that has seen a 1st handshake, then it is linearized according to its original linearization point. Therefore, $t_{insert_i(k)} = orig\_t_{insert_i(k)}$ and since $orig\_t_{insert_i(k)} < orig\_t_{delete_i(k)}$ by Observation 4.1.9, then $t_{insert_i(k)} < orig\_t_{delete_i(k)}$. Otherwise, $insert_i(k)$ is a slow operation that has seen a second handshake and there are 2 cases:

- $delete_i(k)$ is a slow operation - from *Lemma B.3* in [SP22a] we know $g\_t_{insert_i(k)} < orig\_t_{delete_i(k)}$ and together with Lemma 4.1.4 we get that $t_{insert_i(k)} < g\_t_{delete_i(k)}$. Therefore, $t_{insert_i(k)} < orig\_t_{delete_i(k)}$.
- $delete_i(k)$ is a fast operation - from Lemma 4.1.13 and observation 4.1.9 we conclude that $t_{insert_i(k)} < orig\_t_{delete_i(k)}$.

The proof for $t_{delete_(k)} < orig\_t_{insert_{i+1}(k)}$ is similar with minor changes.

□

Corollary 4.1.11. *The linearization points of successful insertions and deletions of each key k are alternating.*

Lemma 4.1.12. *Let countersSnapshot be a* `HandshakeCountersSnapshot` *instance. Any non-*`INVALID` *value written to a counter in the countersSnapshot.*`snapshot` *array must have been written to the corresponding counter in the* `metadataCounters` *array (of the* `SizeCalculator` *instance held by the set)* before *the countersSnapshot.*`collecting` *field is set to* `false`*.*

The proof of Lemma 4.1.12 remains the same as in [SP22a].

Lemma 4.1.13. *For any successful* `insert` *or* `delete` *operation op_s that has obtained* `sizePhase` $\equiv_4 2$ *in Line 72 and for any depending fast operation op_f it holds that: If $orig\_t_{op_s} < orig\_t_{op_f}$ then $t_{op_s} < orig\_t_{op_f}$*

Proof. Let op_s be some successful `insert` or `delete` operation that has obtained `sizePhase` $\equiv_4 2$ in Line 72 and let op_f be some fast operation such that $orig\_t_{op_s} < orig\_t_{op_f}$. If op_s was linearized according to its original linearization point then we are done. Otherwise, from the definition of our linearization points we know there are 2 cases:

- op_s was linearized as in [SP22a]. In this case, since we know that there is no depending fast operation whose original linearization point is between $orig\_t_{op_s}$ and t_{op_s} and because every fast operation is linearized at its original linearization point. Then, $orig\_t_{op_s} \leq t_{op_s} < orig\_t_{op_f}$.
- op_s is linearized at sz_t where sz_t is immediately after the linearization of the last concurrent `size` operation does not take op_s into account and whose linearization point comes after $orig\_t_{op_s}$. op_s has obtained `sizePhase` $\equiv_4 2$ in Line 72. Therefore, it must have executed Line 72 after some `size` operation has executed Line 46 and because $orig\_t_{op_s}$ happens during Line 77 then $orig\_t_{op_s}$ happened after that `size` operation executed Line 46. By definition, sz_t must have happened before that `size` executed Line 35 (because `size` is linearized at `compute()`). Finally, because $orig\_t_{op_s} < sz\_t$ then both $orig\_t_{op_s}$ and sz_t happened when some `size` has finished executing Line 44 and has yet to execute Line 35. Therefore, from Lemma 4.1.22 we conclude that op_f could not have been linearized between $orig\_t_{op_s}$ and $sz\_t = t_{op_s}$ and $orig\_t_{op_s} \leq t_{op_s} < orig\_t_{op_f}$.

□

In the next following lemmas and observations we will undertake a detailed analysis of the handshake mechanism which is essential for some of the lemmas and claims part of the linearization proof above.

Observation 4.1.14. *Every* `HandshakeCountersSnapshot` *object is initialized in Line 27 and is installed onto the* `HandshakeSizeCalculator.countersSnapshot` *field exactly once in Line 28.*

Lemma 4.1.15. *For any* `size` *operation denoted as* op_{size} *that had a successful* `compareAndExchange` *in Line 28: no successful writes to the* `HandshakeSizeCalculator.countersSnapshot` *field by other* `size` *operations were performed when* op_{size}*'s next line to execute pointed to one of Lines 26–28.*

Proof. If any successful writes to the `HandshakeSizeCalculator.countersSnapshot` field by other `size` operations were performed when op_{size}'s next line to execute pointed to one of Lines 26–28 then from Observation 4.1.14 the value of the `HandshakeSizeCalculator.countersSnapshot` field would have changed from the time op_{size} has read it in Line 25 and therefore op_{size}'s invocation of the `compareAndExchange` operation in Line 28 would fail in contradiction to the way we defined op_{size}. □

Corollary 4.1.16. *From Lemma 4.1.15 we conclude that no two* `size` *operations that had a successful* `compareAndExchange` *in Line 28 (i.e successfully wrote to* `HandshakeSizeCalculator.countersSnapshot`*) can have their next line to execute point to Lines 26–28 simultaneously. In other words, at a given moment there is at most one* `size` *operation whose* `compareAndExchange` *is about to succeed and whose next line to execute points to one of Lines 26–28.*

Lemma 4.1.17. *At any given moment, at most one* `size` *operation has their next line to execute point to one of Lines 30–33.*

Proof. We prove by contradiction. Let op_1, op_2 be two `size` operations whose next operation to execute points to one of Lines 30–33 at the same time. w.l.o.g assume op_1 got to Line 26 before op_2 (i.e op_1's next line to execute pointed to Line 26 before op_2's next line to execute pointed to it) and we choose op_2 to be the first operation whose next line to execute reaches Lines 30–33 after op_1.

For a `size` operation to reach Lines 30–33 it must enter the `if` clause in Line 29. Therefore, its `compareAndExchange` operation in Line 28 must have succeeded. Therefore, from Corollary 4.1.16 when op_1's next line to execute pointed to one of Lines 26–28 then op_2's next line to execute did not point to any of Lines 26–28.

Therefore op_2's next line to execute reached Line 26 only after op_1's next operation to execute has pointed to Line 29 which is after op_1 executed its `compareAndExchange` operation in Line 28 and successfully wrote to `HandshakeSizeCalculator.countersSnapshot`. Therefore, op_2 must have read in Line 25 the object op_1 wrote to `HandshakeSizeCalculator.countersSnapshot`. Because we chose op_2 to be the first operation after op_1 to reach Lines 30–33 and since op_1 could not have reached Line 33 (because its next line to execute needs to be one of Lines 30–33 when op_2 reaches Line 30) then the value of the `countersSnapshot.collecting` of the object op_2 read in Line 25 must be true until op_2 reaches Line 30. Therefore, op_2 does enter the `if` clause in Line 26 which is in contradiction to the fact that op_2 reaches Lines 30–33 during its execution. □

Lemma 4.1.18. *At any given moment, at most one* `size` *operation has their next line to execute point to one of Lines 44–48, 31-35 .*

Proof. We prove by contradiction. Let op_1, op_2 be two `size` operations whose next line to execute points to one of Lines 44–48, 31-35 at the same time. w.l.o.g assume op_1 got to Line 30 before op_2 (i.e op_1's next line to execute pointed to Line 30 before op_2's next line to execute pointed to it) and we choose op_2 to be the first operation whose next line to execute reaches Line 30 after op_1.

From Lemma 4.1.17 we determine that when op_2's next line to execute points to Lines 43–35 then op_1's next line to execute must point to one of Lines 34–35. From Lemma 4.1.17 and because we chose op_2 to be the first operation whose next line to execute reaches Line 30 after op_1 we conclude that from the moment op_1's next line to execute reached Line 34 and until op_2's next to line to execute reached Line 43 no writes to `sizePhase` were made. Therefore, the first value read by op_2 in Line 43 must be the value written by op_1 in Line 46. Now, from Line 43 we know that the value of `currentSizePhase` in op_1's execution of Line 46 must be $\equiv_4 0$ therefore the value written in Line 46 by op_1 must be $\equiv_4 2$. Therefore, op_2 has read in Line 43 a value $\equiv_4 2$ and from Lemma 4.1.17 we know that no `size` operation other then op_1 and op_2 can write to `sizePhase` until op_2's next line to execute reaches Line 34. Therefore, since op_1's next line to execute must point to one of Lines 34–35 until op_2's next line to execute reaches Line 44 then no writes at all can be made to `sizePhase` when op_2's next line to execute points to Line 43 making it so that the value of `sizePhase` will stay $\equiv_4 2$ and op_2 will never reach Line 44 in contradiction to the assumption that op_1's and op_2's next line to execute points to one of Lines 44–48, 31-35 at the same time. □

Lemma 4.1.19. *While there is some `size` operation whose next line to execute points to Line 46 then all concurrent `insert` and `delete` operations whose next line to execute is one of Lines 73–78 must have obtained `sizePhase` $\equiv_4 1$ in Line 72.*

Proof. Denote op_{size} some `size` operation whose next line to execute points to Line 46. Let `size_ph` be the `sizePhase` set by op_{size} in Line 44. From Line 43 we know we could only reach Line 44 if `currentSizePhase` $\equiv_4 0$, therefore, `size_ph` $\equiv_4 1$. From Lemma 4.1.18 we know that there can not be any other `size` operation whose next line to execute points to one of Lines 44–48, 31-35 . Therefore, op_{size} is the only operation that can write to `sizePhase`.

Let op be some `insert` or `delete` operation whose next line to execute points to Lines 73–78 while op_{size}'s next line to execute points to Line 46.

- If op's next line to execute has reached Line 72 before op_{size}'s has executed the iteration for op's thread ID in Line 41 when executing Line 45 then op has changed its relevant `opPhase` entry to `FAST_PHASE`. Unless this `opPhase` entry is changed again then when op_{size} executes the iteration for op's thread ID in Line 41 it will read `FAST_PHASE` forever making it so that it never continues to the next iteration and so that op_{size} never reaches Line 46 in contradiction to op_{size}'s definition. Therefore, op must have changed its thread ID's entry in `opPhase` to `IDLE_PHASE` or to some

other value x such that $x \geq$ `size_ph`. If *op* changed its `op_phase` to `IDLE_PHASE` then its next line to execute must have reached Line 79 before op_{size}'s next line to execute reached Line 46 in contradiction to the way we defined *op*. Otherwise, *op* must have changed its `op_phase` to some value x such that $x \geq$ `size_ph`. Then, because op_{size} is the only operation that can write to `sizePhase` while op_{size}'s next line to execute points to one of Lines 44–48 we conclude that *op* must have read `size_ph` $\equiv_4 1$ in Line 72.

- Otherwise, if *op*'s next line to execute has reached Line 72 after op_{size}'s has executed the iteration for *op*'s thread ID in Line 41 when executing Line 45 then op_{size}'s next line to execute must have reached Line 45 before *op*'s next line to execute reached Line 73. Therefore, because op_{size} has executed the write in Line 44 before *op*'s next line to execute has reached Line 73 then *op*'s read in Line 72 must have obtained `size_ph` $\equiv_4 1$.

□

Lemma 4.1.20. *While there is some* `size` *operation whose next line to execute points to any of Line 48, Lines 31–35 then all concurrent* `insert` *and* `delete` *operations whose next line to execute is one of Lines 73–78 must have obtained* `sizePhase` $\equiv_4 2$ *in Line 72.*

Proof. Denote op_{size} some `size` whose next line to execute points to any of Line 48, Lines 31–35. Let `size_ph` be the `sizePhase` set by op_{size} in Line 46. From Line 43 we know we could only reach Line 46 if `currentSizePhase` $\equiv_4 0$, therefore, `size_ph` $\equiv_4 2$. From Lemma 4.1.18 we know that there can not be any other `size` operation whose next line to execute points to one of Lines 44–48, 31-35 . Therefore, op_{size} is the only operation that can write to `sizePhase`.

Let *op* be some `insert` or `delete` operation whose next line to execute points to Lines 73–78 while op_{size}'s next line to execute points to any of Line 48, Lines 31–35.

- If *op*'s next line to execute has reached Line 72 before op_{size}'s has executed the iteration for *op*'s thread ID in Line 41 when executing Line 47 then *op* has changed its relevant `opPhase` entry to `FAST_PHASE`. Unless this `opPhase` entry is changed again then when op_{size} executes the iteration for *op*'s thread ID in Line 41 it will read `FAST_PHASE` forever making it so that it never continues to the next iteration and so that op_{size} never reaches Line 48 in contradiction to op_{size}'s definition. Therefore, *op* must have changed its thread ID's entry in `opPhase` to `IDLE_PHASE` or to some other value x such that $x \geq$ `size_ph`. If *op* changed its `op_phase` to `IDLE_PHASE` then its next line to execute must have reached Line 79 before op_{size}'s next line to execute reached Line 48 in contradiction to the way we defined *op*. Otherwise, *op* must have changed its `op_phase` to some value x such that $x \geq$ `size_ph`. Then, because op_{size} is the only operation that can write to `sizePhase` while op_{size}'s next line to execute points to one of Lines 44–48 we conclude that *op* must have read `size_ph` $\equiv_4 2$ in Line 72.

- Otherwise, if *op*'s next line to execute has reached Line 72 after op_{size}'s has executed the iteration for *op*'s thread ID in Line 41 when executing Line 47 then op_{size}'s next line to execute must have reached Line 47 before *op*'s next line to execute reached Line 73. Therefore, because op_{size} has executed the write in Line 46 before *op*'s next line to execute has reached Line 73 and because op_{size} does not execute any other writes to `sizePhase` until its next line to execute reaches Line 36 then *op*'s read in Line 72 must have obtained `size_ph` $\equiv_4 2$.

□

Corollary 4.1.21. *From Lemma 4.1.20 we conclude that when some `size` operation's next line to execute points to any of Line 48, Lines 31–35, then only slow operations that have seen the second handshake and are linearized accordingly can reach their operation execution (namely, execute slow_op).*

Lemma 4.1.22. *While there is some `size` operation whose next line to execute points to one of Lines 46–48, 31-35 then all concurrent `insert` and `delete` operations whose next line to execute is one of Lines 73–78 must have obtained `sizePhase` $\not\equiv_4 0$ in Line 72.*

Proof. The only case that is not covered by Lemma 4.1.19 and Lemma 4.1.20 is that in which some `insert` or `delete` operation *op*'s next line to execute points to Line 73 and to Line 78 while there is some `size` operation (denote as op_{size}) whose next line to execute points to Line 47. For this to happen, it must be that *op* starts and finishes executing Lines 72–78 while op_{size}'s next line to execute points to Line 47. Therefore, from Lemma 4.1.18 and because op_{size} has already executed Line 44 by the time *op* executed Line 72 we conclude that *op* must have read the value written by op_{size} in Line 44. From Line 43 we know we could only reach Line 44 if `currentSizePhase` $\equiv_4 0$, therefore, the value written by op_{size} in Line 44 must be $\equiv_4 1 \not\equiv_4 0$. □

4.2 Optimistic Approach

In this section, we present in detail an optimistic scheme for evaluating the size of a data structure. The main idea of this approach is to allow the `size` operations to execute optimistically, with no locks and minimal interference on other operations. In this scheme, each thread maintains a local metadata counter that represents the number of inserts it has executed minus the number of deletes. This metadata counter effectively captures the impact of this thread on the size of the data structure. Although a local metadata counter could potentially be negative (indicating more removals than insertions by that thread), the aggregate sum of all local metadata counters depicts the current size of the data structure and is always non-negative when no update operation is running. The objective of the optimistic synchronization is to read all these local metadata counters and collectively sum them up at a time when no concurrent update is executing. Achieving this enables the computation of a linearizable size.

To support the optimistic `size` operation the other threads cooperate in order to allow detecting whether the `size` operation executes without any concurrent update operations. To this end, each thread maintains a local flag indicating whether it is presently involved in updating the data structure, and a local counter that keeps track of the overall number of update operations the thread has executed. This information is compactly stored within a single word, denoted the activity counter, wherein the least significant bit represents the flag, while the remaining bits represent the updates counter. When a thread initiates an update operation, it increments its activity counter. Upon completion of the update operation, the thread increments its activity counter again. Therefore, an odd number in the activity counter indicates that the thread is currently performing an update operation, while the value of the activity counter divided by 2 reveals the count of overall number of updates executed by that thread.

A `size` operation first reads the activity counters of all threads (non-atomically) to ensure none of them are currently engaged in updating the data structure, i.e., verifying that all activity counters are even. If any read activity counter is odd, signifying that a thread was actively modifying the data structure during the read, the size operation waits until it becomes even. After all activity counters are read as even, indicating no ongoing updates, the `size` operation proceeds by reading the metadata counters of all threads (non-atomically), sums them, and then re-reads the activity counters (non-atomically) and confirms that they remained unchanged (otherwise it restarts). The fact that the activity counters were even and were not changed, ensures that no concurrent update operation occurred during the reads of the metadata counters. Consequently, the sum of the metadata counters provides a linearizable size of the data structure (the `size` operation may be linearized at any moment during summing the metadata counters—between the completion of the first read of the activity counters and the start of their second read).

The issue with the method described above in its raw form is its potential to endlessly restart due to a second read that does not match the first read. To address this, after multiple restarts from the `size` operation, it signals a flag to cease updating operations temporarily and request the updating operations to assist in calculating the size. Once the size computation is finished, all operations can resume. While this approach effectively handles exceptional cases, it may impact performance adversely, especially during high contention periods when data structure update operations occur frequently.

4.2.1 Data-structure transformation

The data structure transformation for the optimistic approach uses an `OptimisticSizeCalculator` object whose methods appear in Figures 4.8 and 4.9 to calculate the size. Next we bring the transformation details (the full pseudocode appears in Figure 4.10). Like in the other methodologies, an array named `metadataCounters` with per-thread size metadata

is maintained and updated upon a successful `insert` or `delete` operation. The time gap between updating the data structure and updating the `size` metadata in `insert` or `delete` operations can lead to non-linearizable size results for `size` operations that observe the `size` metadata during this period. To prevent this, we maintain an activity counter per thread in an array named `activityCounters`. Each thread performing an `insert` or `delete` increments its cell in the `activityCounters` array before making any changes to the data structure, and increments it again after updating the `size` metadata regardless of whether the operation was successful or not. Using this activity counter array, a `size` operation can determine whether the metadata was updated during its execution, and if so, it can retry the operation, as follows.

To calculate the size, a `size` operation calls the `_tryComputeSize` method, which starts by making a copy named `status` of the `activityCounters` array (Line 156). It is important to note that this copy is not obtained using a snapshot mechanism. For any obtained odd value, which means that the corresponding thread is executing an `insert` or a `delete` operation, the cell in the `status` array is re-read until obtaining an even activity counter value. Once obtaining a `status` array with no odd values, the `size` operation proceeds to calculating the size by summing up the values in the `metadataCounters` array (Lines 157 and 159). Then the `size` operation accesses the `activityCounters` array again and compares its values with the values previously obtained in the `status` array (Line 160). If they do not match, it restarts. Otherwise, the `size` operation finishes and returns the computed size.

To prevent the `size` operation from restarting indefinitely, we set a limit named `MAX_TRIES` on its number of retries, which determines the maximal number of attempts the `size` will go through before making concurrent `insert` and `delete` operations assist it. Once this limit is reached, the `size` operation increments a counter called `awaitingSizes` (Line 135), which it will later decrement before it returns (Line 143). The `insert` and `delete` operations check this counter before they start operating. In case its value is positive, they help the `size` operation by trying to compute the size themselves in a similar fashion to `size` operations—by obtaining `activityCounters` values before and after the computation (see the `helpSize` method in Figure 4.8). The `MAX_TRIES` variable has a big effect on the transformed data structure's performance. If it is too small, `insert` and `delete` operations may be interrupted frequently by `size` operations requiring them to help before performing their operation and therefore harming their performance. If it is too big, `size` operations may take a long time to complete, deteriorating the performance of `size` operations.

Helping a `size` operation compute the size (both by `insert` and `delete` operations and by other `size` operations) is coordinated using a shared object named `SizeInfo`, which has a single field named `size` initialized to `INVALID_SIZE` and intended to hold the result of a `size` operation. `size` operations install such an instance in `OptimisticSizeCalculator.sizeInfo`, and concurrent `size`, `insert` and `delete` operations that observe an installed instance with `INVALID_SIZE` size value attempt to compute the size and write the obtained size onto

the size field. The reason a `size` operation needs to obtain a `SizeInfo` instance, installed in `OptimisticSizeCalculator.sizeInfo` by itself or by a concurrent `size`, is to be able to retrieve from it a size value computed by another thread, as the `size` operation might keep failing to obtain two identical copies of even activity counters and compute a correct size on its own. After obtaining a `SizeInfo` instance, the `size` operation keeps attempting to obtain two such activity counters copies and compute the size in between. On a successful attempt, it returns the computed value while also writing it to the obtained `SizeInfo` instance for helping others. On a failing attempt, if another thread succeeded and wrote its computed size to the `SizeInfo` instance, it returns this computed size. However, a size written by another thread to the first `SizeInfo` instance obtained by `size` may not be returned, since it might have been computed (by summing the `countersMetadata` values) before this `size`'s interval (and so the `size` operation would have been linearized outside its interval). Once observing that the `size` field in the first obtained `SizeInfo` instance is set, a new instance should be installed and obtained, and the size value—that will be later computed and written to it—may be legally returned.

The optimistic methodology does not maintain the progress guarantees of `insert` and `delete` due to the blocking wait in Line 148. It does maintain them for the `contains` operation as it does not modify it.

```
class OptimisticSizeCalculator:
    OptimisticSizeCalculator():
        MAX_TRIES = 3
        this.metadataCounters = new long[n]
        this.activityCounters = new long[n]
        this.awaitingSizes = 0
        this.sizeInfo = new SizeInfo()
    incrementActivityCounter():
        tid = ThreadID.threadID.get()
        this.activityCounters[tid].setVolatile(1+this.activityCounters[tid].
            getVolatile())
    helpSize():
        if this.awaitingSizes.getVolatile() == 0: return
        currentSizeInfo = this.sizeInfo.getVolatile()
        while true:
            if currentSizeInfo.size.getVolatile() != INVALID_SIZE:
                break
            size = _tryComputeSize()
            if size != INVALID_SIZE:
                activeSizeInfo.size.compareAndSet(INVALID_SIZE, size)
                break
    updateMetadata(opKind):
        tid = ThreadID.threadID.get()
        if opKind == INSERT:
            this.metadataCounters[tid].setVolatile(1+this.metadataCounters[tid
                ].getVolatile())
        else:
            this.metadataCounters[tid].setVolatile(-1+this.metadataCounters[
                tid].getVolatile())
    computeSize():
        count = 0
        <activeSizeInfo, isReturnableSizeInfo> = _obtainActiveSizeInfo()
        while true:
            if (size = activeSizeInfo.size.getVolatile()) != INVALID_SIZE:
                if isReturnableSizeInfo: break
                else:
                    <activeSizeInfo, _> = _obtainActiveSizeInfo()
                    isReturnableSizeInfo = true
            if count == MAX_TRIES:
                this.awaitingSizes.getAndAdd(1)
            if count <= MAX_TRIES:
                count++
            size = _tryComputeSize()
            if size != INVALID_SIZE:
                activeSizeInfo.size.compareAndSet(INVALID_SIZE, size)
                break
        if count == MAX_TRIES + 1:
            this.awaitingSizes.getAndAdd(-1)
        return size
```

Figure 4.8: `OptimisticSizeCalculator` interface methods

```
145         _readActivityCounters():
146             status = new long[n]
147             for each tid:
148                 wait until ((status[tid] = this.activityCounters[tid].
                        getVolatile())%2 == 0)
149             return status
150         _retryActivityCounters(status):
151             for each tid:
152                 if status[tid] != this.activityCounters[tid].getVolatile():
153                     return false
154             return true
155         _tryComputeSize():
156             status = _readActivityCounters()
157             sum = 0
158             for each tid:
159                 sum += this.metadataCounters[tid].getVolatile()
160             if _retryActivityCounters(status):
161                 return sum
162             return INVALID_SIZE
163         _obtainActiveSizeInfo():
164             currentSizeInfo = this.sizeInfo.getVolatile()
165             if currentSizeInfo.size.getVolatile() == INVALID_SIZE:
166                 activeSizeInfo = currentSizeInfo
167                 isNewlyInstalledSizeInfo = false
168             else:
169                 isNewlyInstalledSizeInfo = true
170                 newSizeInfo = new SizeInfo()
171                 witnessedSizeInfo = this.sizeInfo.compareAndExchange(
                        currentSizeInfo, newSizeInfo)
172                 if witnessedSizeInfo == currentSizeInfo:
173                     activeSizeInfo = newSizeInfo
174                 else:
175                     activeSizeInfo = witnessedSizeInfo
176             return <activeSizeInfo, isNewlyInstalledSizeInfo>
```

Figure 4.9: `OptimisticSizeCalculator` auxiliary methods

```
177   class TransformedDataStructureOptimistic:
178       TransformedDataStructureOptimistic():
179           Initialize as originally
180           this.sizeCalculator = new OptimisticSizeCalculator()
181       contains(k):
182           Perform the original contains operation
183       insert / delete(k):
184           this.sizeCalculator.helpSize()
185           Search as originally for the place to insert k in case of insert /
                for a node with key k in case of delete
186           Return on failure (if k is present in an unmarked node in case of
                insert / not present in case of delete)
187           this.sizeCalculator.incrementActivityCounter()
188           Perform the original modification attempt, if successful perform
                this.sizeCalculator.updateMetadata(INSERT / DELETE)
189           this.sizeCalculator.incrementActivityCounter()
190           Return the result of the original modification attempt
191       size():
192           return this.sizeCalculator.computeSize()
```

Figure 4.10: A transformed data structure with an optimistic scheme

4.3 Locks

In this section, we describe the final synchronization method studied: lock-based synchronization. As detailed in Section 2, locks provide mutual exclusion by allowing only one thread to hold the lock at any given time. However, simply blocking all update operations to let only one execute at a time could be detrimental to scalability and performance. To address this, we employ advanced locks known as read-write locks, which enable multiple reader threads to execute concurrently, whereas a writer thread executes alone, preventing any other thread (reader or writer) from acquiring the lock concurrently with a writer thread that holds the writer lock.

Using these locks, all update operations acquire the reader lock, enabling them to execute concurrently. The `size` operation acquires the writer lock, ensuring it executes alone without concurrent updates. To facilitate quick execution of `size` and clear the path quickly for other operations, the updating threads maintain their local metadata counters. The metadata allows the `size` operation to quickly execute, as it only needs to read the metadata counters of all threads (rather than traversing the whole data structure to count elements), and it can then release the write lock, allowing all threads to resume execution. The `size` operation returns the sum of all metadata counters. Moreover, concurrent `size` executions cooperate, allowing one execution to perform the size computation and others to utilize the result of this computation.

4.3.1 Data-structure transformation

Next we detail the data structure transformation to make it support our lock-based size mechanism (the full transformation pseudocode appears in Figure 4.11). We add a readers-writer lock to the data structure in the form of a field named `readWriteLock`

placed in a `LocksSizeCalculator` object (see Figure 4.12 for its full method pseudocode). Different implementations of such a lock can be used; we used Java's `StampedLock` class from the `java.util.concurrent.locks` package in our evaluation as it provided the best results out of the tested lock implementations. Additionally, we add an array named `metadataCounters` to the `LocksSizeCalculator` object, with a cell per thread to keep track of the size metadata for each thread.

An `insert` operation starts with a search to find the insertion point. If an unmarked node with the required key is already found, it returns a failure. Otherwise, the read lock is acquired by invoking the `readLock()` method on the `readWriteLock` object. Following this, an insertion attempt is executed as in the original data structure. If it concludes successfully, the current thread's cell in the `metadataCounters` array is incremented by 1. To wrap up the process, the read lock is released by calling the `readUnlock()` method on the `readWriteLock` object, and the result of the insertion attempt is returned.

Similarly, a `delete` operation begins by searching for a node with the key it wishes to delete. If such a node is not located, the operation promptly returns a failure. However, if found, the read lock is acquired by invoking the `readLock()` method on the `readWriteLock` object. The operation then advances to execute a deletion attempt like in the original data structure. If successfully completed, the current thread's cell in the `metadataCounters` array is decremented by 1. Finally, the read lock is unlocked by calling the `readUnlock()` method on the `readWriteLock` object, and the outcome of the deletion attempt is returned.

The `contains` operation remains as in the original data structure. Lastly, the `size` operation sums the values of all the cells in the `metadataCounters` array. To do so, the write lock is acquired by invoking the `writeLock()` method on the `readWriteLock` object. The operation then iterates over the array and sums the values of all the cells. Once the summation is complete, the write lock is released using the `writeUnlock()` method on the `readWriteLock` object and the result of the summation is returned. The acquisition of the write lock ensures that no `insert` or `delete` operation is in an inconsistent state while the summation is executed.

The updates and summation of the `metadataCounters` array are not executed using a special snapshot algorithm. This is because due to the mutual exclusion guaranteed by the acquisition of the write lock, when the `size` operation is accessing the array, no other thread can update it. This makes a simple pass over the array sufficient to correctly compute the size.

To allow multiple `size` operations to be performed concurrently in an efficient manner, we place a field holding a shared object of type `SizeInfo` in the `LocksSizeCalculator` object, which has a single field for holding the computed size. At the start of a `size` operation, it checks if the `SizeInfo` instance currently installed in that field has a valid size value written to it. If not, the operation waits until a valid size value is written to the `SizeInfo` instance and then returns that value. Otherwise, the operation attempts to replace the existing `SizeInfo` instance with a new one with a size field initialized

to `INVALID_SIZE` using `compareAndExchange`. If the `compareAndExchange` fails, the operation waits until a valid size value is written to the `SizeInfo` instance and then returns it. If the `compareAndExchange` succeeds, the `size` operation is responsible for computing the size by acquiring the write lock, summing the metadata array, releasing the write lock and writing the computed size value into the `SizeInfo` instance; it then returns the computed size.

```
193  class TransformedDataStructureWithRWLock:
194      TransformedDataStructureWithRWLock():
195          Initialize as originally
196          this.sizeCalculator = new LocksSizeCalculator()
197      contains(k):
198          Perform the original contains operation
199      insert / delete(k):
200          Search as originally for the place to insert k in case of insert /
                 for a node with key k in case of delete
201          Return on failure (if k is present in an unmarked node in case of
                 insert / not present in case of delete)
202          this.sizeCalculator.readWriteLock.readLock()*
203          Perform the original modification attempt, if successful call
                 this.sizeCalculator.updateMetadata(INSERT / DELETE)
204          this.sizeCalculator.readWriteLock.readUnlock()*
205          Return the result of the original modification attempt
206      size():
207          return this.sizeCalculator.computeSize()
208  *The locking and unlocking scheme depends on the implementation of the
         lock used and may look different (e.g. when using a StampedLock).
```

Figure 4.11: A transformed data structure with a readers-writer lock

```
209 class LocksSizeCalculator:
210     LocksSizeCalculator():
211         this.metadataCounters = new long[n]
212         this.readWriteLock = new ReadWriteLock()
213         this.sizeInfo = new SizeInfo()
214     updateMetadata(opKind):
215         tid = ThreadID.threadID.get()
216         if opKind == INSERT:
217             this.metadataCounters[tid].setVolatile(1+this.metadataCounters[tid
                    ].getVolatile())
218         else:
219             this.metadataCounters[tid].setVolatile(-1+this.metadataCounters[
                    tid].getVolatile())
220     computeSize():
221         currentSizeInfo = this.sizeInfo.getVolatile()
222         if currentSizeInfo.size.getVolatile() != INVALID_SIZE:
223             newSizeInfo = new SizeInfo()
224             witnessedSizeInfo = this.sizeInfo.compareAndExchange(
                        currentSizeInfo, newSizeInfo)
225             if witnessedSizeInfo == currentSizeInfo:
226                 size = _computeSize()
227                 newSizeInfo.size.setVolatile(size)
228                 return size
229             currentSizeInfo = witnessedSizeInfo
230         return _waitForComputing(currentSizeInfo)
231     _computeSize():
232         sum = 0
233         this.readWriteLock.writeLock()*
234         for each tid:
235             sum += this.metadataCounters[tid].getVolatile()
236         this.readWriteLock.writeUnlock()*
237         return sum
238     _waitForComputing(currentSizeInfo):
239         while true:
240             currentSize = currentSizeInfo.size
241             if currentSize != INVALID_SIZE:
242                 return currentSize
```

Figure 4.12: `LocksSizeCalculator` methods

Chapter 5

Evaluation

We implemented all of the presented methodologies for computing size in Java, closely corresponding to the algorithms and pseudocode described across Sections 4.1–4.3. This implementation includes all described optimizations in Sections 4.1.4 and 5.1.2. In this section, we present the evaluation of all methodologies compared to the methodology from [SP22a]. Two primary aspects were chosen for testing: (1) the additional overhead each methodology incurs on the operations of the original data structure and (2) the performance of the `size` operation, evaluated by testing its scalability. Finally, we try to give recommendations on which methodology should be used in each scenario.

Platform. All experiments were executed on a system operating on Linux (Ubuntu 20.04.5 LTS), powered by two Intel(R) Xeon(R) Gold 6338 CPUs @2.00GHz, each with 64 threads, summing up to a total of 128 threads. The system is equipped with 32GB of RAM. The methodologies were implemented using Java, employing OpenJDK version 21. As in [SP22a], the G1 garbage collector was deployed and the flags -server, -Xms31G, and -Xmx31G were utilized to improve performance and minimize disruption of Java's garbage collection.

Data structures. We evaluated the methodologies on three different data structures: SkipList, Binary Search Tree (BST) and HashTable. The implementations for the baseline data structurse are taken from the public implementation of [SP22a], available in [SP22b]. These implementations in turn are based on prior work. The SkipList builds on Java's `ConcurrentSkipListMap` from the `java.util.concurrent` package in Java SE 18. The BST builds on Brown's implementation [Bro18] of the lock-free binary search tree of [EFRvB10] that places elements in leaf nodes. The HashTable was implemented in [SP22a] based on the linked list in the base level of Java's `ConcurrentSkipListMap`. Since the BST implementation in [Bro18] does not linearize the `delete` operation at the marking step, to comply with the restrictions of the handshake-based methodology, we used a variant of BST that linearizes the deletion at the marking step as in [SP22a]. Since the lock-based and optimistic methodologies do

not pose that restriction, we used the unmodified BST implementation from [Bro18] for these transformations.

Methodology. For the most part, we use the same testing methodology as in [SP22a]. This methodology involves initializing the data structure with 1M items prior to each experiment. Subsequently, two distinct workloads are executed: an update-heavy workload, comprising 30% `insert` operations, 20% `delete` operations, and 50% `contains` operations; and a read-heavy workload, consisting of 3% `insert` operations, 2% `delete` operations, and 95% `contains` operations. These workloads align with the recommended read rates outlined in the *Yahoo! Cloud Serving Benchmark* (YCSB) [CST+10]. YCSB also proposes a 100%-read workload; however, this scenario is less pertinent to our case as the likelihood of `size` calls on a data structure that remains unchanged is negligible. The results that correspond to the read-heavy workload are displayed on the left side of Figures 5.2–5.10, while those related to the update-heavy workload are presented on the right side.

The keys utilized for operations during the experiment as well as for the initialization of the data structure, are selected uniformly at random from a specified range $[1, r]$ like in [SP22a]. The value of r is determined to ensure the target size of the data structure is maintained. Furthermore, in all experiments, the type of the subsequent operation is determined iteratively based on the specified update-heavy or read-heavy workload proportions. Each experiment involves the concurrent execution of w workload threads, which engage in `insert`, `delete`, and `contains` operations according to the characteristics of the workload, alongside s size threads, which repeatedly invoke the `size` operation with a delay between each two invocations. We set the delay time between `size` operations in overhead measurements to either 0 or 700µs (microseconds) to represent continuous or occasional invocations of `size`. In the rest of the measurements we kept the delay at 0. We chose 700µs to represent an execution of `size` at about 10% of the clock time, depending on the methodology used. For baseline algorithms, only w workload threads are employed. The values of w and s vary across experiments, with the constraint that $w + s$ is predominantly chosen as a power of 2. Experiments are run for 5 seconds. Each reported data point in the graphical representations is the average outcome of 10 runs, following an initial warm-up phase consisting of 5 preliminary runs to stabilize the Java virtual machine. To reduce the variance between experiments, we disabled hyper-threading, leaving us with 64 threads to utilize. Additionally, in experiments involving up to 32 threads, we employed the "taskset –cpu-list 0-31" command to ensure that the entire experiment was executed on a single CPU node to reduce variability.

5.1 Implementation details

5.1.1 Thread registration

Each methodology we study utilizes a metadata array to effectively track the count of insertions and deletions on a per-thread basis. Within this metadata array, every thread is allocated a distinct cell. To allocate a cell to each thread, we incorporate a registration mechanism, assigning a unique ID to each thread that aligns with a cell in the array. To facilitate this thread identification and management in a concurrent environment, we introduce the `ThreadID` class presented in Figure 5.1, in which we have implemented a mechanism to manage thread registration. Within this class, an `AtomicInteger` variable is utilized to keep track of the next thread ID that has not yet been used. A pool is maintained to store the thread IDs that have been released by other threads, ensuring reuse of these IDs for subsequent thread registrations. Before performing any operation on the data structure a thread must call `ThreadID.register()`. When it is done using the data structure it should call `ThreadID.deregister()` to release its thread ID allowing other threads to use it.

It is important to note that the reassignment of a thread ID from one thread to another does not compromise the correctness of the size operation. A thread should perform deregistration only after it is done operating on the data structure, and each operation on the data structure returns only after it has been finalized. Consequently, a specific thread ID is allocated to one operating thread only at any given time, and if a new thread is allocated a previously used ID, the data structure continues to reflect the cumulative effects of all operations conducted under that ID. Therefore, the data structure maintains its integrity and correctness even as thread IDs are dynamically allocated and deallocated among different threads.

We used Java's `PriorityBlockingQueue` class from the `java.util.concurrent` package to serve as our concurrent pool, ensuring the management of concurrent accesses. This class implements the `poll()` method to allow extraction of an element from the pool and the `add()` method to allow insertion of a new element into the pool. In addition, we utilized Java's `AtomicInteger` from the `java.util.concurrent` package to keep track of the next available thread ID in an atomic manner.

5.1.2 General Optimizations

5.1.2.1 Avoid false sharing

To prevent false sharing among threads while accessing arrays that hold per-thread data, for each array of this kind utilized in the different methodologies (`meatadataCounters`, `fastMeatadataCounters`, `opPhase` and `activityCounters`), we pad its cells so that the data of each thread occupies a full cacheline.

```
244    class ThreadID:
245        MAX_THREADS = 128
246        this.threadID = ThreadLocal<Integer>()
247        this.pool = PriorityBlockingQueue<Integer>(MAX_THREADS)
248        this.nextId = AtomicInteger(0)
249        register():
250            if this.threadID.get() is not null:
251                throw new RuntimeException("Thread already registered")
252            tid = this.pool.poll()
253            if tid is null:
254                tid = this.nextId.getAndIncrement()
255                if tid >= MAX_THREADS:
256                    throw new RuntimeException("Too many threads")
257            this.threadID.set(tid)
258        deregister():
259            tid = this.threadID.get()
260            if tid is null:
261                throw new RuntimeException("Thread not registered")
262            this.pool.add(tid)
263            this.threadID.set(null)
```

Figure 5.1: `ThreadID` class methods

5.1.2.2 Partial array iteration

The usage of the registration scheme described in Section 5.1.1 enables the determination of the maximum number of threads that have been operating concurrently on the data structure. This number is represented as the value of the `nextId` variable. Consequently, when going over any metadata array sized to the number of threads in each methodology (for example, the `activityCounters` array in the optimistic methodology), it is possible to go over only the first `nextId` cells and not iterate over the entire array. This can improve performance in cases where the maximal number of concurrent active threads is fewer than the maximal number of threads (which determines the size of the `size` metadata array). To implement this, in places where we iterate over the `size` metadata array (not including the initialization of this array), we first read the value of the `nextId` variable. Then, we iterate over only the first few cells of the array based on the value we read from `nextId`. In order to address the race condition in which a new thread has registered causing `nextId` to increment after we have read the value of the `nextId` variable allowing that new registered thread to modify a cell which we are not aware exists, we read this value again when we have finished iterating. If the value has changed (which would only be an increase), we repeat the process by reading the `nextId` variable again. We will only finish once we reach an iteration where this value has not changed. This verification loop is not necessary in all cases, for example, in the optimistic methodology when executing the `_readActivityCounters` function there is no need to verify the value of `nextId` as if we missed a thread's registration - in the case that thread has modified the `activityCounters` array we will find out about it in the `_retryActivityCounters` function, causing the `size` attempt to retry.

5.1.2.3 Usage of tailored `opKinds`

In situations where it is not necessary to differentiate between the `size` metadata for `insert` and `delete` operations (such differentiation is needed only for the slow operations' metadata in the handshakes methodology), we can make usage of specific `opKind` values to allow us eliminate some conditional statements from the implementation. This approach enables us to eliminate the `if` condition when updating the `size` metadata, as appears in Lines 12, 121 and 216. To achieve this, we define new, different `opKind` values to be `INSERT=1` and `DELETE=-1` (where previously `INSERT` was 0 and `DELETE` was 1 to allow access to the `size` metadata array while separating the `insert` metadata from the `delete` metadata). When updating the metadata, rather than checking whether the `opKind` indicates an `insert` or a `delete` operation to decide whether to increase or decrease the counter, we simply add the `opKind` value directly to the corresponding cell in the `size` metadata array.

5.1.3 Memory Model

In our Java implementation, volatile semantics are consistently used in read, write, and `CAS` operations on non-final fields of shared objects. This approach is carried out through the usage of volatile variables, `VarHandle`s and `AtomicReferenceFieldUpdater`s.

Under the Java memory model, any access that utilizes volatile semantics is treated as a synchronization action. The model, in turn, commits to a synchronization order for these actions. This allows for a total order that seamlessly aligns with each thread's program order. Moreover, any read operation executed on a volatile variable is promised to fetch its most recently written value, in accordance with the synchronization order.

5.2 Overhead of size

The graphs in Figures 5.2–5.4 present throughput measurements of the various synchronization methods with the skip list (Figure 5.2), the BST (Figure 5.3), and the hash table (Figure 5.4). Specifically, for each data structure we execute the original version of the data structure, the `SP` method of [SP22a] (denoted *SP*), the optimistic method (denoted *optimistic*), the handshake method (denoted *handshake*), and the lock-based method (denoted *stampedLock*). Each figure contains three rows. The first row presents an execution with no concurrent `size` execution, representing just the overhead for having size available. The second row presents an execution with a continuous execution of `size` by a concurrent thread, representing the overhead for always cooperating with the execution of `size`. Lastly the third row presents an execution of operations when the `size` operation occasionally runs concurrently. As stated above, we let the thread running `size` execute a delay of 700µs after each `size` execution, to represent a cooperation with a `size` method at approximately 10% of the time, depending on the efficiency of `size` with the specific synchronization method.

The X-axis represents the number of threads running operations concurrently (1, 4, 8, 16, 32, and 64 when no concurrent `size` executes, and 1, 3, 7, 15, 31, 63 when one concurrent thread executes `size` on a separate thread). Each graph has an upper part and a lower part. The throughput is depicted in the upper part, with the Y-axis showing the number of million operations executed per second. On the lower part of each graph we depict the overhead percents compared to the throughput of the original data structure (that does not support a `size` execution). The Y-axis shows 100% when no throughput loss is demonstrated, and lower percentage when overhead is witnessed. To prevent long bars interfering with small ones, we cut long overhead bar at 80% and write the lost percentage below the specific bar.

It turns out that there is no one-size-fit-all method. Different scenarios call for different synchronization methods. The observed results vary by the chosen data structure, the contention levels, the frequency of utilizing the `size` operation, and the workload (read intensive or update intensive). Notably, the hash function exhibits exceptionally fast operations, making the relative overhead on cooperation with size much higher when compared to both the skip list and the BST. In scenarios characterized by a write-heavy workload, the original `SP` approach emerges as the recommended strategy for the hash table. Its overhead averages around 10%, and generally within the range of $0 - 20\%$. Conversely, for read-intensive workloads, the optimistic approach proves optimal, showcasing an average overhead of around 4% and fluctuating between $0 - 14\%$.

The skip list and BST perform comparably, demonstrating commendable performance with both the lock-based and optimistic methods, particularly in read-heavy workloads. However, their efficacy diminishes significantly under write-heavy workloads, especially during heightened contention and when the `size` operation is actively employed. The optimistic approach is somewhat less harmful in this scenario due to its more gradual degradation. Consequently, when anticipating write-heavy workloads or when usage patterns are uncertain, both the `SP` and handshake approaches exhibit superior performance. Notably, the handshake approach on the skip list maintains an average overhead of approximately 4.4%, surpassing the `SP` approach by about 1%, and ranging between $0 - 12.9\%$. On the other hand, for the BST, the `SP` approach consistently outperforms the other approaches, showcasing an average overhead of 2.4% and ranging between $0 - 7.1\%$ across all scenarios.

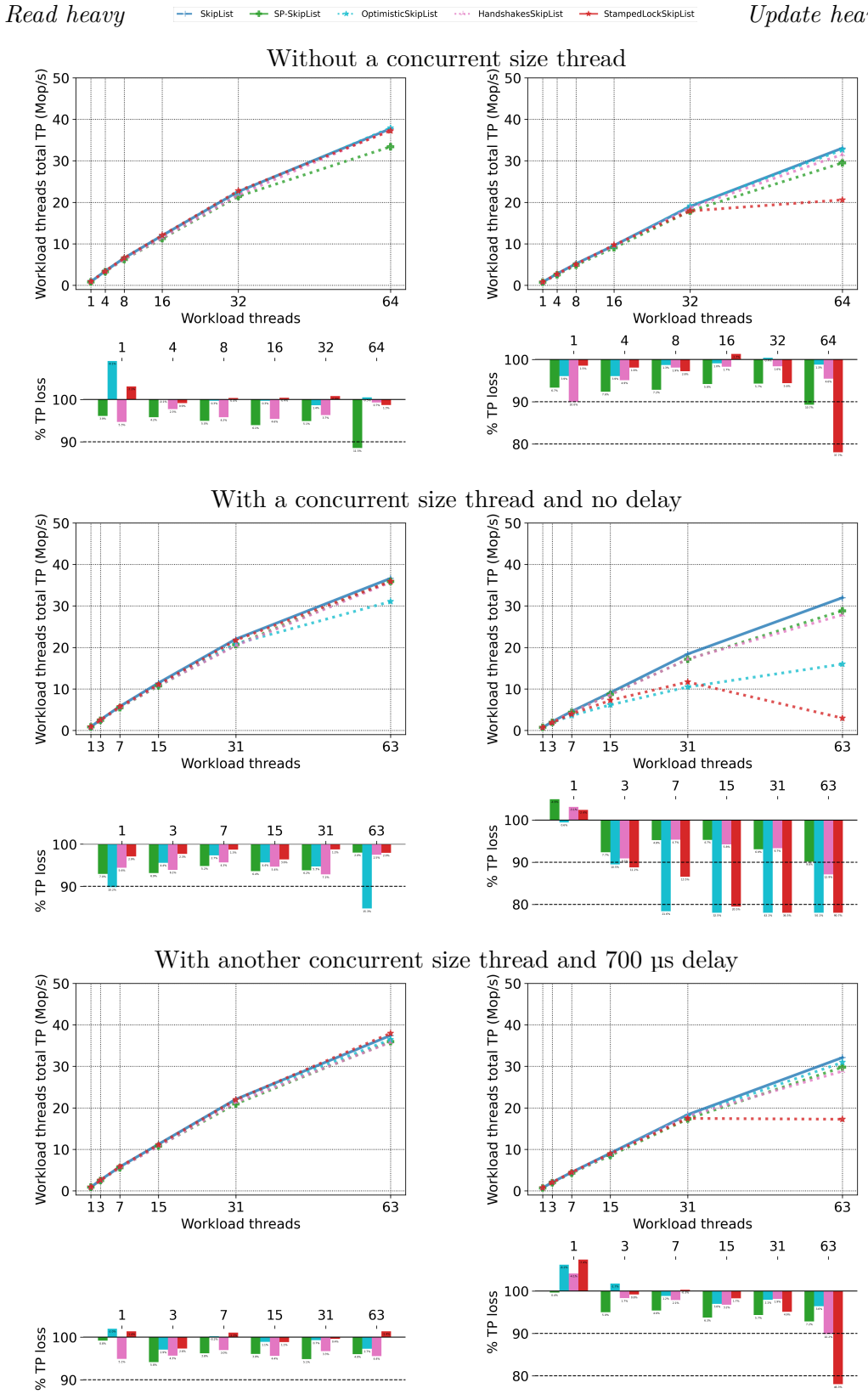

Figure 5.2: Overhead on skip list operations

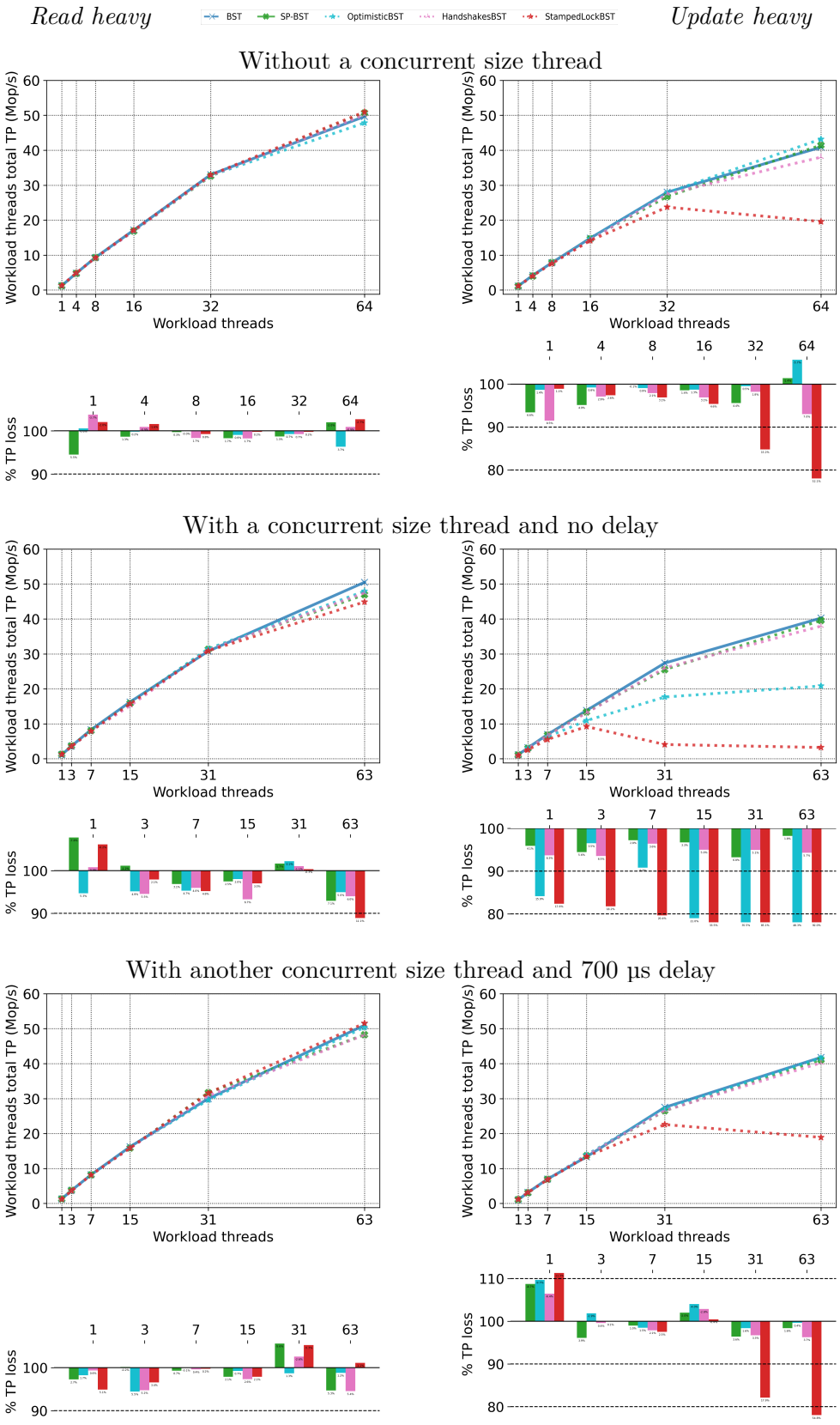

Figure 5.3: Overhead on BST operations

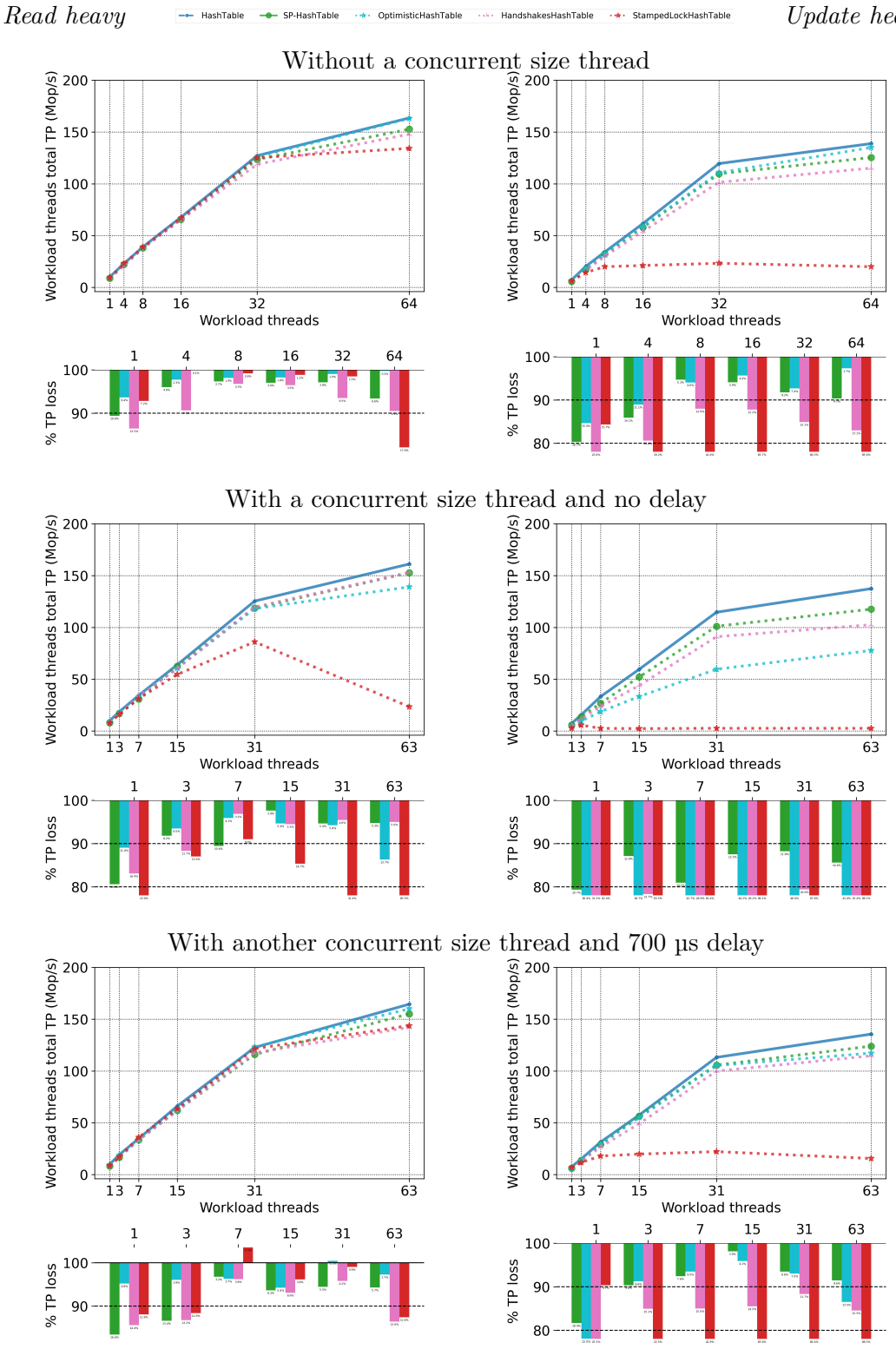

Figure 5.4: Overhead on hash table operations

5.3 Size Scalability

Next, we study the scalability of the `size` operation across the studied synchronization methods. We measured the total throughput of threads executing the `size` operation in each data structure, both in a read-oriented and a write-oriented workload. We ran 32 workload threads concurrently with `size`-executing threads, whose number varies between 1 to 32 (so the overall number of running threads was bounded by 64). The results are depicted in Figures 5.5–5.7.

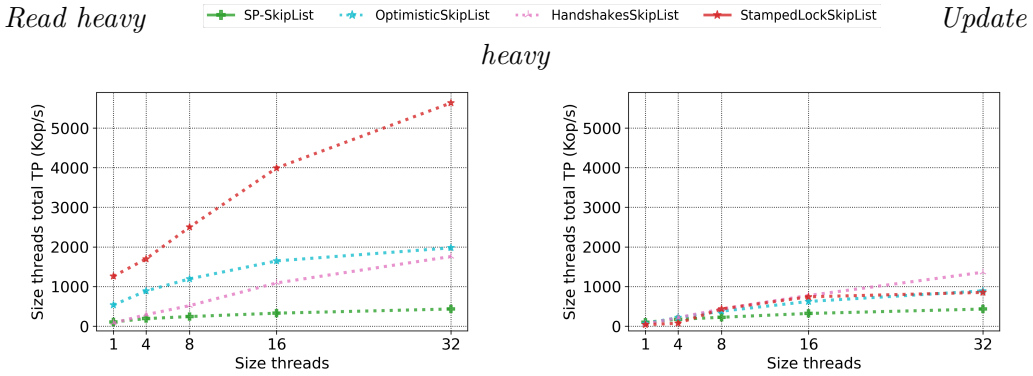

Figure 5.5: size scalability in skip list

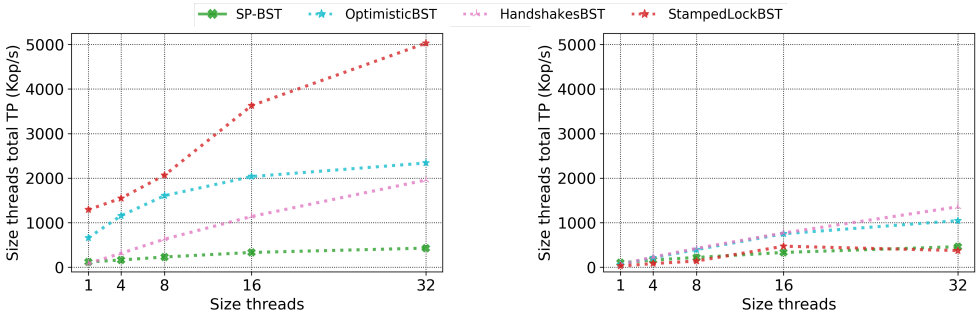

Figure 5.6: size scalability in BST

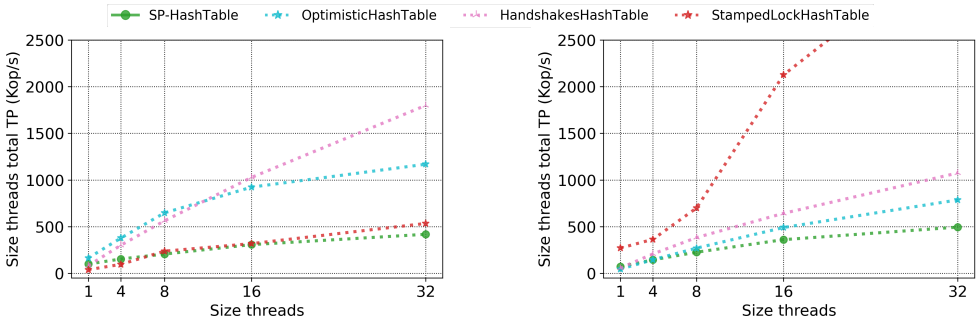

Figure 5.7: size scalability in hash table

Again the results differ between the hash table and the other two data structures. For the BST and the skip list on a read-oriented workload, the lock-based method does significantly better. A `size` operation that grabs the write lock can execute very fast with no interference from the data structure operations. In contrast, when the data

structure is updated frequently, the wait for acquiring the lock becomes dominant and the handshake synchronization wins, albeit not with such a significant advantage as locks have on a read-heavy workload. So if the scalability of the `size` is of high importance then the lock-based method should be favored with read-oriented workloads, and for write-heavy workloads the handshake synchronization does somewhat better than the lock-based synchronization. Normally, we expect the overhead on the operations themselves to be the more important consideration, as they typically occur more frequently.

Similarly to the investigation of overheads, the hash table behaves completely different. The handshakes synchronization wins on the read-heavy workload and the lock-based synchronization wins on the write-heavy workload. Lock-based synchronization is not recommended for hash table as the `size` operations take over the lock and allow almost no data structure operations to execute, as can be seen in the middle row, right side, of Figure 5.4. The handshake approach or `SP` approach synchronization methods may thus be the best overall choice for this case.

5.4 MAX_TRIES measurements

To determine the effect of the `MAX_TRIES` variable on the optimistic method, we performed additional measurements that compared the overhead on the original data structure and the scalability of the `size` operation with `MAX_TRIES` values ranging from 2 to 16. Graphs presenting the effect of different `MAX_TRIES` values on the performance of the optimistic method appear in Figures 5.8–5.10.

Read heavy *Update heavy*

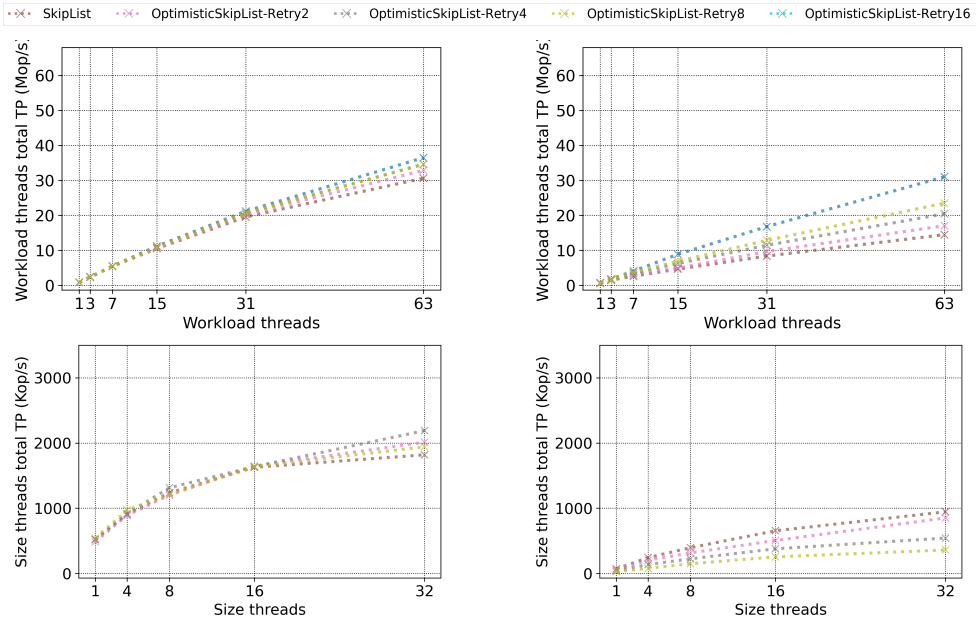

Figure 5.8: MAX_TRIES overhead and scalability in skip list

Read heavy *Update heavy*

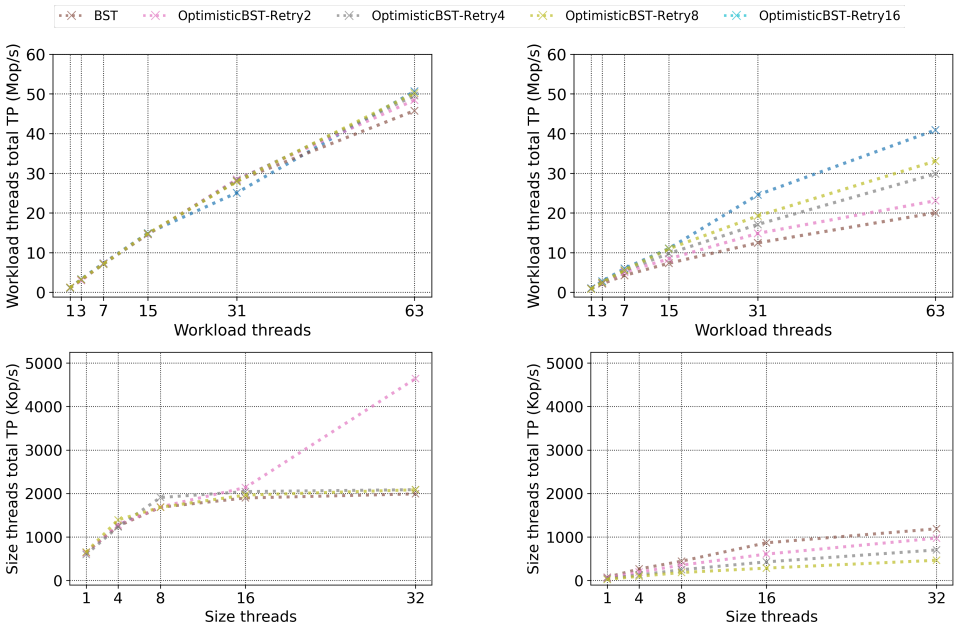

Figure 5.9: MAX_TRIES overhead and scalability in BST

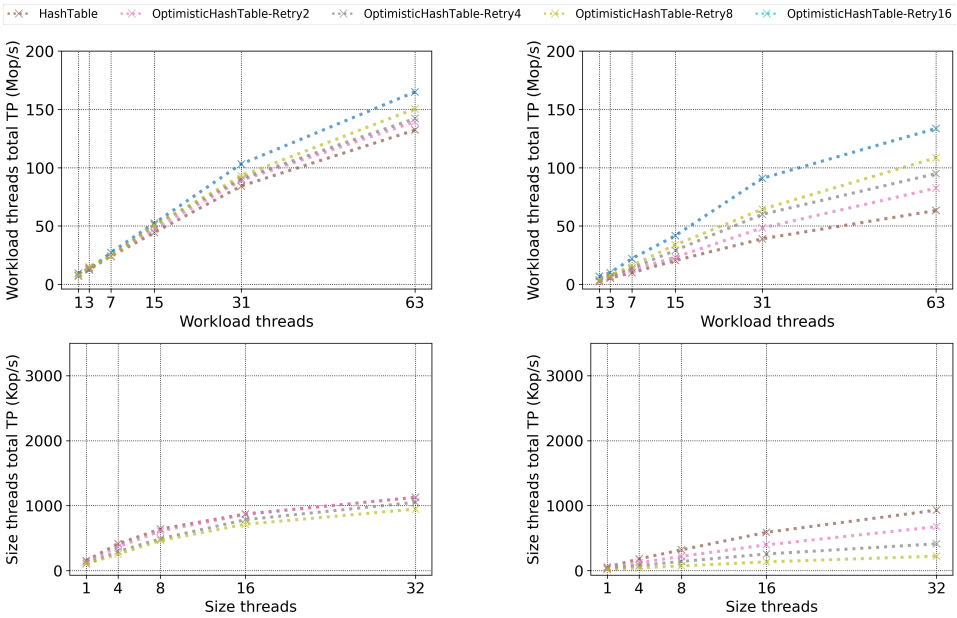

Figure 5.10: MAX_TRIES overhead and scalability in hash table

5.5 Progress Guarantees

Table 5.1 presents a comparison of the progress guarantees for the synchronization methods researched and presented in this thesis for concurrent size computation. The "Size progress guarantees" column indicates whether each method provides strong progress guarantees, such as wait-freedom. The remaining columns evaluate whether each size computation methodology preserves the progress guarantees and asymptotic complexities of the set operations in the original data structures

As shown in the table, the handshakes approach preserves the original progress guarantees and complexity but does not offer specific guarantees for size computation. The optimistic and locks based methodologies, on the other hand, compromise both progress guarantees and complexity. In contrast, the `SP` algorithm [SP22a] provides a wait-free guarantee for the size computation while maintaining both the original progress guarantees and asymptotic complexity of the data structure.

	Size progress guarantees	**Maintaining original progress guarantees**	**Maintaining original asymptotic complexity**
Handshakes	–	✓	✓
Optimistic	–	–	–
Locks	–	–	–
`SP` [SP22a]	Wait-Free $O(n)^*$	✓	✓

* Where n is the maximal number of threads in the system.

Table 5.1: Progress guarantees comparison of synchronization methods for concurrent size computation.

Chapter 6

Discussion and Conclusion

This thesis investigated the challenge of accurately and efficiently determining the size of a concurrent data structure. We proposed three new approaches to incorporate a linearizable `size` operation into concurrent data structures that implement sets or dictionaries. These approaches were tested and compared to existing methods, with our findings highlighting the complexity of selecting the optimal synchronization technique. As demonstrated, there is no single scheme that provides the best performance across all scenarios, reinforcing the notion that different conditions necessitate different synchronization methods.

Our evaluation revealed that the effectiveness of synchronization techniques depends on several factors, including the data structure in use, contention levels, the frequency of `size` operations, and the workload distribution between read and update operations. In low contention environments, both optimistic and lock-based synchronization methods perform well, but their efficiency declines as contention rises. In such high-contention scenarios, the handshakes method and the `SP` approach emerged as the most effective solutions.

Notably, the performance differences between data structures were significant. Hash table operations, being faster overall, presented higher overhead when integrated with the `size` method.

We also evaluated the scalability of the `size` operation when executed concurrently with data structure updates, observing that synchronization methods optimizing `size` scalability may differ from those enhancing data structure operation performance. Given that users are likely to prioritize overall data structure performance, this factor should guide the selection of synchronization techniques.

In conclusion, this study aligns with general trends in concurrent computing, illustrating that no single approach provides universally superior performance. The choice of synchronization method for the `size` operation should be considerate of the specific requirements of the workload and data structure characteristics, balancing the trade-offs between `size` computation efficiency and overall data structure performance.

Bibliography

[AST12] Yehuda Afek, Nir Shavit, and Moran Tzafrir. Interrupting snapshots and the javatm size method. *Journal of Parallel and Distributed Computing*, 72(7):880–888, 2012. https://doi.org/10.1016/j.jpdc.2012.03.007.

[Bro18] Trevor Brown. Java lock-free data structure library, 2018.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[DG94] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 70–83, New York, NY, USA, 1994. Association for Computing Machinery. https://doi.org/10.1145/174675.174673.

[DKL+00] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Eliot E. Salant, Katherine Barabash, Itai Lahan, Yossi Levanoni, Erez Petrank, and Igor Yanorer. Implementing an on-the-fly garbage collector for java. *SIGPLAN Not.*, 36(1):155–166, oct 2000.

[DL93] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, page 113–123, New York, NY, USA, 1993. Association for Computing Machinery. https://doi.org/10.1145/158511.158611.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *PODC*, 2010.

[Her91] Maurice Herlihy. Wait-free synchronization. *TOPLAS*, 13(1), 1991.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[LP01] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for java. In *Proceedings of the 16th ACM SIGPLAN Conference*

on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '01, page 367–380, New York, NY, USA, 2001. Association for Computing Machinery.

[PT13] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *DISC*, 2013.

[RK23] Arik Rinberg and Idit Keidar. Intermediate value linearizability: A quantitative correctness criterion. *J. ACM*, 70(2), April 2023.

[SP22a] Gal Sela and Erez Petrank. Concurrent size. *PACMPL*, 6(OOPSLA2), 2022.

[SP22b] Gal Sela and Erez Petrank. Concurrent size - artifact for oopsla'22, 2022.

במחקר שלנו, אנו מבצעים הערכה מקיפה של שיטות הסנכרון הללו ומשווים אותן למתודולוגיות חישוב גודל עדכניות. הממצאים שלנו מראים כי ניתן להפחית באופן משמעותי את העומס הכרוך בחישוב גודל על ידי בחירת שיטת הסנכרון המתאימה ביותר לסביבה הספציפית. עם זאת, חשוב לציין שאין פתרון אוניברסלי שמתאים לכל המקרים; סביבות שונות דורשות אסטרטגיות סנכרון שונות, כפי שמודגם בבירור במחקר זה.

הממצאים שלנו תואמים את המגמות הכלליות בתכנות מקבילי. בסביבות בעלות התנגשויות מעטות בגישה למשאבים, השיטות האופטימיסטיות והמבוססות נעילה מתגלות כיעילות ביותר. לעומת זאת, בסביבות בעלות התנגשויות רבות בגישה למשאבים, שיטת לחיצת היד ושיטות קיימות חסרות נעילה מוכיחות את עצמן כמתאימות יותר. באמצעות בחירה נכונה של שיטת הסנכרון ניתן להשיג שיפור משמעותי בביצועים תוך שמירה על תקינות מבני הנתונים המקביליים, ובכך למקסם את הביצועים הכוללים של המערכת תוך צמצום העומסים שנובעים מחישוב הגודל.

תקציר

גודלם של אוספים, מפות ומבני נתונים באופן כללי, מהווה מאפיין בסיסי אשר משחק תפקיד מהותי במגוון רחב של פרדיגמות תכנות. יישום מדויק של חישוב גודל נדרש ברוב סביבות התכנות, משום שפעולות רבות מתבססות על מידע זה. לדוגמה, ניהול זיכרון, חלוקת עומסים ואופטימיזציה של אלגוריתמים מתבססים על מידע זה. יחד עם זאת, בסביבה מקבילית שבה מספר תהליכים פועלים בו זמנית, שילוב של שיטת חישוב גודל מקבילית ונכונה עשוי להוסיף עומס משמעותי על כל הפעולות במבנה הנתונים, גם כאשר חישוב הגודל אינה מנוצל כלל במהלך ההרצה. עומס זה עלול לפגוע בביצועים הכוללים של המערכת, מה שהופך את שיטת חישוב הגודל לשיקול קריטי בעיצוב ויישום של מבני נתונים מקביליים.

במחקר זה, אנו מציגים מחקר מקיף על שיטות סנכרון שונות שנועדו לשפר את ביצועי מבני הנתונים בסביבות מקביליות. מטרת המחקר היא להבין כיצד ניתן לצמצם את העומס שנובע מחישוב גודל במקביל, ולבחור בשיטת הסנכרון היעילה ביותר בהתאם למאפייני הסביבה. באופן ספציפי, אנו מנתחים ומשווים את היעילות של שיטת "לחיצת יד" בה משתמשים לעיתים קרובות במנגנוני איסוף זבל מקביליים, שיטה אופטימיסטית ושיטה מסורתית מבוססת מנעולים.

שיטת לחיצת היד מבוססת על תיאום בין תהליכים שונים כדי להבטיח שכל הפעולות על מבנה הנתונים מתבצעות בצורה מתואמת, במיוחד כאשר חישוב הגודל מנוצל במהלך הריצה. שיטה זו היא אפקטיבית במיוחד בסביבות בעלות התנגשויות רבות בגישה למשאבים, שבהן תהליכים רבים ניגשים למבנה הנתונים בו זמנית. השיטה מפחיתה את הסיכון לקונפליקטים ומבטיחה שהגודל המחושב יהיה עקבי ונכון. יחד עם זאת, לשיטה זו יש גם חסרון: היא עלולה להוסיף זמן המתנה לפעולות אחרות, כיוון שיש צורך בתיאום בין כל התהליכים לפני שניתן להמשיך בביצוע הפעולה.

השיטה האופטימיסטית יוצאת מנקודת הנחה שקונפליקטים בין תהליכים הם נדירים. גישה זו מאפשרת לתהליכים לבצע פעולות על מבנה הנתונים ללא נעילות, תוך הסתמכות על מנגנוני אימות לזיהוי ופתרון קונפליקטים במידת הצורך. שיטה זו מתאימה במיוחד לסביבות עם מעט התנגשויות בגישה למשאבים, שבהן העלות של נעילה גבוהה יותר מהתועלת שהיא מספקת. במקרים בהם ישנם קונפליקטים רבים, השיטה האופטימיסטית עלולה להוביל לניסיונות חוזרים ונשנים ולעיכובים, מה שמקטין את היעילות הכוללת שלה.

השיטה המבוססת על נעילה מהווה גישה מסורתית יותר, שבה כל פעולה על מבנה הנתונים מתבצעת תחת מנגנון נעילה שמבטיח גישה בלעדית. שיטה זו מבטיחה עקביות, אך עלולה להוביל לצווארי בקבוק כאשר תהליכים רבים מנסים לגשת למבנה הנתונים במקביל. במקרים כאלו, הנעילה עשויה להאט את ביצועי המערכת ולפגוע ביעילות הכוללת שלה.

המחקר בוצע בהנחייתו של פרופסור ארז פטרנק, בפקולטה למדעי המחשב.

מחבר/ת חיבור זה מצהיר/ה כי המחקר, כולל איסוף הנתונים, עיבודם והצגתם, התייחסות והשוואה למחקרים קודמים וכו', נעשה כולו בצורה ישרה, כמצופה ממחקר מדעי המבוצע לפי אמות המידה האתיות של העולם האקדמי. כמו כן, הדיווח על המחקר ותוצאותיו בחיבור זה נעשה בצורה ישרה ומלאה, לפי אותן אמות מידה.

תודות

ברצוני להודות תחילה למנחה שלי, פרופ' ארז פטרנק, שמומחיותו, הכוונתו והתמיכה שלו היוו עבורי מקור השראה ותרומה עצומה לאורך הדרך. ארצה להודות גם לגל סלע על שיתוף הפעולה, המחויבות ועל העצות והתובנות הנהדרות אשר העשירו את המחקר שלי באופן משמעותי. לבסוף, ארצה להודות למשפחתי על התמיכה הבלתי מתפשרת והאמונה בי לאורך כל הדרך.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

חישוב גודל של מבני נתונים לנוכח עדכונים מקביליים באופן יעיל

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

חן קאס שריר

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
אב תשפ״ד חיפה ספטמבר 2024

חישוב גודל של מבני נתונים לנוכח עדכונים מקביליים באופן יעיל

חן קאס שריר