Of Concurrent Data Structures and Iterations

Yiannis Nikolakopoulos[1], Anders Gidenstam[2],
Marina Papatriantafilou[1(✉)], and Philippas Tsigas[1]

[1] Chalmers University of Technology, Gothenburg, Sweden
{ioaniko,ptrianta,tsigas}@chalmers.se
[2] University of Borås, Borås, Sweden
anders.gidenstam@hb.se

Abstract. Bulk operations on data structures are widely used both on user-level but also on programming language level. Iterations are a good example of such bulk operations. In the sequential setting iterations are easy to design on top of an algorithmic construction of a data structure and is not considered as a challenge. In a concurrent environment, such as a multicore system, the situation is completely different and the issue of extending concurrent data structure designs to support iteration operations opens new research challenges in concurrent algorithmic data structure implementations, with respect to consistency and efficiency. In this paper we take a journey through this young and evolving research topic. More precisely we describe recent advances in the area together with an overview of iteration implementations that have appeared in the research literature as well as in widely-used programming environments and we outline a range of application targets and challenging future directions.

Keywords: Iteration · Consistency · Lock-free · Concurrent data structures · In-memory computation · Range-queries

1 Introduction

Algorithms + Data Structures = Programs. Wirths book title [43] has become a famous quote and almost a synonym of what the essential components of a computer program are. It shows how data structures are a crucial part of designing and implementing efficient algorithms. An ideal data structure implementation minimizes the complexity of specific access patterns to data that an algorithm requires and integrates it to the data structure's Application Programming Interface (API) (e.g. FIFO queues, LIFO stacks, heaps).

Concurrent Data Structures

The above requirements are even more pronounced when shifting to a concurrent environment, involving multiple processing entities and units of execution. The shared memory model requires mechanisms to ensure the integrity of the

C. Zaroliagis et al. (Eds.): Spirakis Festschrift, LNCS 9295, pp. 358–369, 2015.
DOI: 10.1007/978-3-319-24024-4_20

data, which can be accessed and modified by several threads or processes. Furthermore, the access patterns to the data may be more complex and involve multiple actors with different synchronization needs e.g. getting access to data under some specific synchronization related conditions. The research community has been studying and providing algorithmic designs and implementations of shared memory data structures, including simple designs that incorporate coarse-grain locking, more complex fine-grain locking techniques, non-blocking implementations and flat combining synchronization techniques [9,15,21,24,40].

The different implementations and methodologies provide a variety of performance guarantees in several quantitative and qualitative metrics, such as throughput, scalability and fairness [8] and may comply with a variety of correctness, consistency and progress requirements, thus introducing interesting trade-offs. Balancing amongst all these requirements according to the needs of the applications that use the data structure, remains a key research and implementation issue as is also emphasized recently by Michael [32].

As with any object in a concurrent environment [17], the algorithmic design and implementation of operations provided by shared memory data structures introduces several challenges regarding the correctness requirements and the provided consistency of the operations. The standard definitions that have prevailed in the literature regarding correctness conditions of *non-blocking* implementations are *sequential consistency* [28] and *linearizability* [25]. These consistency specifications have been the main models used in the research literature for arguing about the correctness of parallel and concurrent programs in shared memory systems. In the data structure context there has been a plethora of concurrent implementations using them, cf. [9,24] and references therein.

Extended API in Concurrent Data Structures

Given the increasing interest and use-cases of concurrent data structures, implementations of them appear as part of wide spread programming frameworks [1–3] (Java, .NET, TBB), either in the language or at the library level. In such cases the API of the data structure is usually complemented with additional methods serving other parts of the programming framework like inheritance reasons, compatibility or extended functionality. As an example, Java's `ConcurrentLinkedQueue`, part of the `java.util.concurrent` package, is an implementation based on Michael and Scott's lock-free queue [33]. The API in this case, besides the `initialize`, `enqueue` and `dequeue` methods that a queue usually has, is extended with a variety of operations including `peek, remove, size, contains, toArray, addAll` and `iterator`. The question that naturally arises is whether the existing consistency specifications are adequate in describing the desired functionality of this extended API.

Specifically, an interesting subset of the above is *bulk operations*, i.e. "meta" operations that consist of a number of operations on the data structure, or a number of primitive operations on sub-components of the data structure. A typical example is *iteration* operations – or *enumeration* as commonly called, where the goal is to gain access to all the items stored in the underlying data structure,

usually in a sequential way, without exposing the internal data structure representation. They are usually provided through the use of constructs like *iterators, enumerators* or *generators*.

Iterations in Sequential Programming

Iterations have been widely supported in object oriented languages in a sequential context. They were used as building blocks for other language functionalities (e.g. [4,41]), as well as for user level convenience, e.g. to create constructs that would assign values to a `for`-loop. Watt [41] characterizes iterators as *object-based* and *control-based*. The former log the state of the traversal in a separate data structure. According to this state the next steps of the iteration are decided and the new state is updated as they proceed. Algorithm 1 shows a simple iterator of single-linked list based FIFO queue. No special language support is needed in this case, but the iterator implementation gets more difficult the more complex the main data structure is. Control-based iterators rely on specific language constructs (e.g. `yield` and `suspend`) that abstract the previous mechanism and assign values to a loop variable, saving the iterator state until another value is needed.

Algorithm 1. Sequential iteration of a FIFO single-linked list based queue.

```
currentNode ← Head.next
while currentNode.next ≠ NULL do
    currentNode ← currentNode.next
```

Iterations in Concurrent Data Structures: Challenging Issues

Typically, none of the above constructs would take concurrency under consideration. In fact, even the iteration semantics may change when shifting to a concurrent execution. For example, protecting the state of an iteration might come in contrast with the goals of a concurrent system. In data structure implementations, the goal is to allow operations from multiple units of execution to execute concurrently, through fine-grain synchronization or lock-free/wait-free methods [24], enabling to utilize the system parallelism with anticipated benefits in throughput and latency (i.e. operations may execute on different cores simultaneously).

Moreover, such concurrent implementations introduce non-trivial *trade-offs* among the performance throughput, the consistency and ease-of-use by the programmer. Strong consistency guarantees like linearizability [25] and sequential consistency [28] are preferred by the programmer using the data structure. On the other hand, they usually come at a cost of larger algorithmic complexity in the design of the data structure. Some implementations in contemporary programming environments (cf. Sect. 4) provide weaker consistency, but with properties that either are unclear or do not match definitions in the literature.

The trade-offs above, along with the fact that iterations are bulk operations on the data structures, raise questions on the cost of iteration operations under specific consistency requirements in a concurrent implementation. How can concurrency-related behavior be characterized through consistency specifications? Do strong consistency properties have to be expensive and what are the alternatives, if any?

A problem that relates with bulk operations and iterations, is that of acquiring a *snapshot* of a shared memory register [5,14,16,27]. The similarity comes in terms of applying bulk read operations on a shared register where multiple processes write, in order to achieve a desired level of consistency for a snapshot to be returned. Consistency specifications for snapshot objects have evolved from context-specific atomicity and correctness criteria [5], to generally applicable definitions such as linearizability or relaxed guarantees like time-lapse properties [13,27].

Paper Outline

In the rest of this article we describe recent advances in problems related with iteration operations in concurrent data structures. Section 2 gives a brief description of the shared memory model. Section 3 describes the necessary consistency definitions for iteration operations, as formed by the authors of this paper in recent work [35]. Section 4 provides an overview of iteration implementations that have appeared in the research literature as well as in widely-used programming environments. Section 5 associates iteration operations with related bulk operations in current and upcoming systems; that section also outlines possible questions for future work.

2 System Model

We consider the asynchronous shared memory model commonly used in the literature, where a set of processes communicates via reading and writing in shared memory, as also used in earlier work on concurrent iterations [34,35]. The model allows to provide concurrent implementations of a container abstract data type (ADT) that represents a collection of items, including a set of *update* operations that modify the collection according to its specification.

A concurrent ADT implementation can satisfy different progress guarantees. Below we describe the standard definitions in the literature [22–24]: *Wait-freedom* ensures that any process can complete an operation in a finite number of its own steps, independently of any other process. An implementation is *bounded wait-free* if there exists a bound on the number of steps any process takes to complete an operation. In a *lock-free* object implementation it is ensured that at least one of the contending operations makes progress in a finite number of its own steps. It is common in lock-free implementations of ADTs that an operation is implemented through fail-retry loops: a retry needs to take place due to one or more *interfering* operations among the contending ones. A weaker guarantee

is *obstruction freedom*: progress is only ensured for any process that eventually runs in isolation, i.e. in absence of interferences from other operations.

We define a *run* ρ (or *history*) as an execution of an arbitrary number of operations on the ADT according to the respective protocol that implements the ADT. For each update operation a of an ADT that exists in a run ρ, we call its *duration* the time interval $[s_a, f_a]$, where s_a and f_a are the starting and finishing times of a. Thus, a precedence relation $\rightarrow$ is defined over the operations, which is a strict partial order. For two operations a and b, $a \rightarrow b$ means that f_a occurs before s_b. If two operations can not be compared under $\rightarrow$, they are concurrent and we say that they overlap. In this work, we consider only runs of complete executions, where there are no pending operations. A *sequential* history ρ, is one where no operations overlap. We denote a prefix of a *sequential* history ρ ending with an operation a, as $pref_\rho(a)$. We define $state_\rho(a)$ as the postcondition of the ADT after the operation a, i.e. the items that exist in the collection after the execution of a in ρ.

A history is *linearizable* [25] if it is equivalent to a sequential history that includes the same operations and the total order of the sequential operations respects the partial order $\rightarrow$. The equivalent sequential history is also called a linearization σ. Thus, a run ρ of a linearizable ADT implementation induces a set of total orders, that extend the partial order $\rightarrow$ in a compatible way with the sequential semantics of the ADT. For each linearization σ of ρ, the respective total order is denoted as $\Rightarrow_\sigma$. As in the sequential case, for every operation a in a linearizable run ρ we respectively define $state_\sigma(a)$ as the postcondition of the ADT after operation a, in a prefix of some linearization σ of ρ that ends with a. In this notation we will drop the parameter σ when it is clear from the context.

For a given linearizable ADT implementation consider the set of all its possible states; a state S from this set is defined to be *valid with respect to a linearizable execution* ρ, if $\exists$ a linearization σ of ρ such that there exists a $pref_\sigma(a)$ and $S = state_\sigma(a)$.

The ADT includes *update* operations that can add or remove items in the collection in accordance to the specification of the ADT. We extend the ADT and linearizable implementations of them, and add *iteration* operations that will return a state of the ADT and in particular the items that are contained in it, with the following sequential specification:

Definition 1. [35] *In a sequential execution* ρ*, a* valid *iteration* Itr *returns the items contained in* $state_\rho(a)$*, where* a *is the latest update operation preceding* Itr*, i.e.* $Itr : state_\rho(a)$*, where* $a \rightarrow Itr \wedge \nexists$ *update operation* a' *s.t.* $a \rightarrow a' \rightarrow Itr$*.*

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

3 Framework of Consistency Definitions for Concurrent Iterations

This section presents consistency definitions for iteration operations described in [34,35], building on the consistency-related definitions by Lamport [29] and

Herlihy and Wing [25]. In the following it is assumed that the reduced run $\tilde{\rho}$ that does not include the iteration operations is linearizable and thus for each linearization σ of $\tilde{\rho}$ the respective $\Rightarrow_\sigma$ is defined.

Definition 2. *(i)* ***Safeness:*** *An iteration operation $Itr \in \rho$, not overlapping with any other operation in $\tilde{\rho}$, is* safe *if it returns a valid state $S = state_\sigma(a)$ for some linearization σ of $\tilde{\rho}$ and some operation a, such that: $Itr \nrightarrow a$ and $\nexists$ update operation $a' : a \Rightarrow_\sigma a' \rightarrow Itr$. If Itr is overlapping with any operations of $\tilde{\rho}$, it can return any arbitrary state of the object.*
(ii) ***Regularity:*** *An iteration operation $Itr \in \rho$, possibly overlapping with some $a \in \tilde{\rho}$, is* regular *if it returns a valid state $S = state_\sigma(a)$ for some linearization σ of $\tilde{\rho}$ and some operation a, such that: $Itr \nrightarrow a$ and $\nexists$ update operation $a' : a \Rightarrow_\sigma a' \rightarrow Itr$, i.e. S is neither "future" nor "overwritten".*
(iii) ***Monotonicity:*** *For any two iteration operations Itr_1, Itr_2 that return valid states $state_\sigma(a_1)$ and $state_\sigma(a_2)$ respectively for some linearization σ of $\tilde{\rho}$, if $Itr_1 \rightarrow Itr_2$, then $a_1 \Rightarrow_\sigma a_2$ or $a_1 = a_2$.*[1]
(iv) ***Linearizability:*** *Itr is* linearizable *if it is* regular *and the run ρ is equivalent to some sequential history, that includes the same operations, whose total order respects the partial order of the original run ρ.*

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

Informally, we can see that a safe iteration implementation guarantees recent and valid returned states only in the case that it does not overlap with any modification operations. Otherwise any arbitrary state can be returned (e.g. empty). A regular iteration improves by guaranteeing a valid and "not future" state to be returned even in the present of concurrent modifications. However, there is no restriction on a regular iteration implementation on whether to actually include in the returned state the effect of one (or more) overlapping modification operations. Thus regularity can allow relaxed enough implementations for the following example to occur: let Itr_1 an instance of a regular iteration, and an overlapping modification operation a. It is interesting to note that another regular Itr_2, such that $Itr_1 \rightarrow Itr_2$ but Itr_2 still overlapping a, might return a different state – even preceding– from the one returned by Itr_1 (if for example Itr_1 includes the effect of a in its state). Monotonicity is an additional property that can clarify such behavior and the combination with regularity can guarantee

[1] Dwork et al. [13] in the context of composite registers used two notions of monotonicity, for scans and for updates. Notice that a regular Itr also satisfies the monotonicity of updates property, i.e. for two linearized updates, an Itr that "observes" the effects of the latter update, should also "observe" the effects of the preceding update.

linearizability for "single-scanner" iterations (cf. Theorem 1 [35]). Finally, iteration implementations in contemporary programming environments (cf. Sect. 4), motivate the need for a weaker definition between safeness and regularity. Weak regularity essentially allows to include or ignore any of the overlapping modification operations, regardless of their respective linearization order.

Correctness conditions related to the above were recently presented by Lev-ari et al. [30], where regularity is extended for single-writer data structures under read-write concurrency. The authors show similar intuition for their respective regularity definition, while they motivate the need for even weaker conditions as the ones presented above [34].

4 An Overview of Iteration Algorithms and Implementations

In the Resarch Literature of Parallel and Distributed Algorithms

Ctrie. The first design of a concurrent data structure that integrated an iteration operation was presented by Prokopec et al. [39]. They present *Ctrie*, a concurrent lock-free hash trie, that besides the usual lookup, insert and remove operations, supports a snapshot operation upon which iteration and size operations are built. *Ctrie* is designed partially as a persistent data structure [36] with immutable states, and the snapshot operation relies on the fact that modification operations will create a newer generation of the data structure content, while the snapshot retains access to the previous one. Thus, the snapshot is considered constant-time ($O(1)$), while the updating of the trie to the newer generation is delegated to the update operations, increasing their constant factor. Nevertheless, to guarantee linearizability, strong synchronization primitives are still required. The authors suggest a variation of a Double-Compare-Single-Swap (RDCSS) software primitive [20], to make sure that no concurrent updates will occur while the generation is changing.

Iterators on Sets. Iteration operations in ordered linked-list based implementations of sets were presented by Petrank and Timnat [37]. The authors build on Jayanti's single scanner snapshot algorithm on composite registers [26], which provides only scan and update operations for the individual registers. They extend it to accommodate multiple iterators as well as the necessary insert, delete and contains operations that the set semantics require. Their method includes an object that holds a list of pointers to nodes in the main data structure and a list of reports of operations that happened during the basic traversal. The commutativity properties of the set semantics are exploited, allowing the snapshot operation to linearize before the linearization points of insert operations that occurred during the traversal. The iteration operation has a complexity of $O(n+r \cdot log(r))$, where r is the size of the report list and n is the size of the main linked list traversed. However, the latter may dynamically increase by updates interfering the iteration.

Iterators in Parallel Collections. Prokopec et al. [38] present a different approach on iterators in collection data structures. They do not address issues of iterations concurrently with modification operations, but focus on the parallelization of iteration operations instead. They develop a framework for parallel programming patterns such as map-reduce or parallel looping for bulk operations, and abstract it by using *splitters*. These are abstractions for iteration operations that can be used from different threads to give access to disjoint parts of the data structure.

Iteration Consistency and the Queue Test-case. A set of consistency specifications for iteration operations (cf. Sect. 3) is proposed by the authors of the present article in [34,35]. The aforementioned work further presents an exploration of the algorithmic design space for iteration operations in shared lock-free queues and provide a set of constructions of iteration operations satisfying the consistency properties. *Weakly regular* iterations are presented based on simple traversals of the queue and by exploiting the inherent structural properties of the linked-list based queues. They are also compared with similar implementations that exist in Java's concurrency library [2]. The authors point out and study the trade-offs on achieving linearizable iteration implementations between the overhead of the bulk operation and possible support (helping) by the native operations of the data structure. *Linearizable* iterations can be achieved by typical read-validate techniques that may starve, providing only *obstruction-freedom*. Concurrent modification operations can help the iteration operation by marking nodes of the queue with appropriate timestamp information. Thus, linearizable iteration algorithms that provide *wait-free* progress guarantees can be designed, with the use of synchronization software primitives like multi-word compare and swap [20]. The reason is that due to the non-commutative nature of the queue's native operations, inconsistencies between the time of insertion and the respective timestamp have to be eliminated.

In Contemporary Programming Environments

Programming frameworks, such as Intel's Thread Building Blocks for C++ (TBB), Java and the .NET platform, include, in their standard libraries, collection data structures that support concurrent operations. These collections often support iteration over their contents while other operations may concurrently change the data structure. What kind of consistency do they offer?

Java: The standard library of the Java Platform Standard Edition 7[2] [2] contains a number of concurrent collection or container data types that support iteration over their contents concurrently with operations that modify them. The documentation classifies the consistency of an iteration of a particular container data type as either *snapshot style*, described as capturing the state of the container at the point in time the iterator was created, or *weakly consistent*, for the `ConcurrentLinkedQueue` described as "returning elements reflecting the state of

[2] Version 1.7.0_09.

the queue at some point at or since the creation of the iterator" and similarly for other data structures. A study of the source code for the `ConcurrentLinkedQueue` reveals that the description is not entirely accurate: the result may be a mixture of the states that occur during the iteration and can include items removed early during the interval together with items added late, i.e. not reflecting the state at any particular point in time.

.NET: The .NET 4.5 Framework Class library [3] contains a number of concurrent container data structures. All of them support iteration of their contents concurrently with operations that modify them. The library documentation classifies what an iteration of a particular container type provides as either a *moment-in-time snapshot* or not a *moment-in-time snapshot*. The container types `ConcurrentBag`, `ConcurrentQueue` and `ConcurrentStack` provide *moment-in-time snapshots* while the `ConcurrentDictionary` type does not. For the latter it is stated that the contents exposed during the iteration may contain modifications made to the dictionary after the iteration started.

Intel Threading Building Blocks: It is a library for parallel programming in C++ [1] that contains a number of concurrent container data structures, some of which support iterations concurrently with other operations on the container. The support is limited to a subset of their operations; only three, namely `concurrent_unordered_map`, `concurrent_unordered_set` and `concurrent_vector` support insertion concurrently with iteration, but do not promise any particular level of consistency.

In summary, Java's *snapshot style* and .NET's *moment-in-time snapshots* can be expected to be linearizable (or nearly so). The consistency of Java's *weakly consistent* iterators varies in detail for each implementation, and the unspecified thread-safe iterators in .NET and TBB are weakly regular.

5 Possible Applications and Research Questions

Iterations form the basis of operations that meet challenges in many of the new, demanding applications in concurrent environments. In the era of Big Data and the Internet of Things, in-memory analytics are becoming more and more important, and parallelism and concurrency are essential tools.

Map and filter operations can in fact be built upon iterations, as also shown in Prokopec et al. [38]. Paradigms of such operations, running over large evolving data sets concurrently with other operations, may be proved valuable for real-time analytics.

In the latest version of Java, the *stream* API is introduced. The idea is to allow parallel programming patterns such as map-reduce computations to run on streams of data that may even be unbounded. As a source of a stream, either another stream is allowed or a collection data structure, possibly concurrent. Streams from the latter are built by using bulk operations that are a generalization of iterations, called *spliterators*. For most of the concurrent collections they provide weak consistency guarantees (as in Definition 3), and they allow also parallelization (similar to the concept of [38]).

Range queries or *partial iterations* for in-memory processing is another example where iterations are useful. Essentially, range queries can be viewed as partial iterations inside a data structure. Traditionally in the database domain, data structures supporting range queries [42] were introduced to handle search queries of multiple multi-dimensional records. Examples of today's use cases are On-Line Analytical Processing systems [11,12]. These delineate algorithmic and consistency challenges for concurrent data structures. Avni et al. [6] explore designs of data structures supporting range queries, based on transactional memory support. Brown and Avni [7] present a non-blocking k-ary search tree that supports linearizable range queries, achieving only obstruction-freedom though. Partial iterations on concurrency- and application-aware data objects, such as e.g. sets, flat-sets and data-streaming-oriented objects [10,18,19,31], can prove very useful in this direction as well.

The questions in this domain are challenging, ranging from consistency definitions that are useful for applications, to algorithmic implementations that enable possibilities for the programmers to smoothly manage efficiency and consistency trade-offs that manifest in applications. It is expected that the consistency framework [35] will be useful for several types of such bulk operations in a wide range of concurrent objects, as it incorporates definitions across several levels of strength, that build on each-other. Besides the weakly regular constructions that already map to some state of the art implementations, *regular* iteration implementations are expected to balance consistency and performance trade-offs, and thus form a challenging direction for future work.

Acknowledgements. The research leading to these results has been partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) through the EXCESS Project (www.excess-project.eu) under grant agreement 611183 and by the Swedish Research Council (Vetenskapsrådet) project "Fine-grain synchronization in parallel programming", contract nr. 2010-4801.

References

1. Intel threading building blocks documentation. http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm. Accessed on 27 November 2012
2. Java platform standard edition 7 documentation. http://docs.oracle.com/javase/7/docs/index.html. Accessed on 06 December 2012
3. .NET framework class library documentation. http://msdn.microsoft.com/en-us/library/gg145045.aspx. Accessed on 10 May 2013
4. Python v2.7.5 documentation. http://docs.python.org/2/library/itertools.html. Accessed on 10 September 2013
5. Anderson, J.H.: Multi-writer composite registers. Distrib. Comput. **7**(4), 175–195 (1994)
6. Avni, H., Shavit, N., Suissa, A.: Leaplist: lessons learned in designing TM-supported range queries. In: Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing, PODC 2013, pp. 299–308. ACM, New York, NY, USA (2013)

7. Brown, T., Avni, H.: Range queries in non-blocking k-ary search trees. In: Baldoni, R., Flocchini, P., Binoy, R. (eds.) OPODIS 2012. LNCS, vol. 7702, pp. 31–45. Springer, Heidelberg (2012)
8. Cederman, D., Chatterjee, B., Nguyen, N., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: A study of the behavior of synchronization methods in commonly used languages and systems. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1309–1320, May 2013
9. Cederman, D., Gidenstam, A., Ha, P., Sundell, H., Papatriantafilou, M., Tsigas, P.: Lock-free concurrent data structures. arXiv:1302.2757 [cs], February 2013
10. Cederman, D., Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: Brief announcement: concurrent data structures for efficient streaming aggregation. In Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2014, pp. 76–78 (2014)
11. Dehne, F., Kong, Q., Rau-Chaplin, A., Zaboli, H., Zhou, R.: A distributed tree data structure for real-time OLAP on cloud architectures. In: 2013 IEEE International Conference on Big Data, pp. 499–505, October 2013
12. Dehne, F., Zaboli, H.: Parallel real-time OLAP on multi-core processors. In: Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), pp. 588–594 (2012)
13. Dwork, C., Herlihy, M., Plotkin, S., Waarts, O.: Time-lapse snapshots. SIAM J. Comput. **28**(5), 1848–1874 (1999)
14. Fatourou, P., Fich, F.E., Ruppert, E.: Time-space tradeoffs for implementations of snapshots. In: Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing, STOC 2006, pp. 169–178, ACM, New York, NY, USA (2006)
15. Fatourou, P., Kallimanis, N.D.: Revisiting the combining synchronization technique. SIGPLAN Not. **47**(8), 257–266 (2012)
16. Fich, F.E.: How hard is it to take a snapshot? In: Vojtáš, P., Bieliková, M., Charron-Bost, B., Sýkora, O. (eds.) SOFSEM 2005. LNCS, vol. 3381, pp. 28–37. Springer, Heidelberg (2005)
17. Gidenstam, A., Koldehofe, B., Papatriantafilou, M., Tsigas, P.: Scalable group communication supporting configurable levels of consistency. Concurrency Comput: Pract. Experience **25**(5), 649–671 (2013)
18. Gidenstam, A., Papatriantafilou, M., Tsigas, P.: NBmalloc: allocating memory in a lock-free manner. Algorithmica **58**(2), 304–338 (2010)
19. Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join enabled by concurrent data structures. Technical Report, Chalmers University of Technology (2014)
20. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 265–279. Springer, Heidelberg (2002)
21. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2010, pp. 355–364. ACM, New York, NY, USA (2010)
22. Herlihy, M.: Wait-free synchronization. ACM Trans. Prog. Lang. Syst. **13**(1), 124–149 (1991)
23. Herlihy, M., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: ICDCS 2003, IEEE Computer Society (2003)
24. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann, Burlington (2008)

25. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Prog. Lang. Syst. **12**(3), 463–492 (1990)
26. Jayanti, P.: An optimal multi-writer snapshot algorithm. In: Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing, STOC 2005, pp. 723–732. ACM, New York, NY, USA (2005)
27. Kirousis, L., Spirakis, P., Tsigas, P.: Reading many variables in one atomic operation: solutions with linear or sublinear complexity. IEEE Trans. Parallel Distrib. Syst. **5**(7), 688–696 (1994)
28. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **C–28**(9), 690–691 (1979)
29. Lamport, L.: On interprocess communication. Distrib. Comput. **1**(2), 86–101 (1986)
30. Lev-Ari, K., Chockler, G., Keidar, I.: On correctness of data structures under reads-write concurrency. In: Kuhn, F. (ed.) DISC 2014. LNCS, vol. 8784, pp. 273–287. Springer, Heidelberg (2014)
31. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002, ACM (2002)
32. Michael, M.M.: The balancing act of choosing nonblocking features. Commun. ACM **56**(9), 46–53 (2013)
33. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, PODC 1996, pp. 267–275. ACM, New York, NY, USA (1996)
34. Nikolakopoulos, Y., Gidenstam, A., Papatriantafilou, M., Tsigas, P.: Enhancing concurrent data structures with concurrent iteration operations: consistency and algorithms. Technical report, Chalmers University of Technology (2013)
35. Nikolakopoulos, Y., Gidenstam, A., Papatriantafilou, M., Tsigas, P.: A consistency framework for iteration operations in concurrent data structures. In: 2015 IEEE 29th International Symposium on Parallel & Distributed Processing (IPDPS) (2015)
36. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, New York (1999)
37. Petrank, E., Timnat, S.: Lock-free data-structure iterators. In: Afek, Y. (ed.) DISC 2013. LNCS, vol. 8205, pp. 224–238. Springer, Heidelberg (2013)
38. Prokopec, A., Bagwell, P., Rompf, T., Odersky, M.: A generic parallel collection framework. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 136–147. Springer, Heidelberg (2011)
39. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent tries with efficient non-blocking snapshots. In: PPoPP 2012, pp. 151–160. ACM (2012)
40. Sundell, H., Tsigas, P.: NOBLE: a non-blocking inter-process communication library. In: Proceedings of the 6th Workshop on Languages, Compilers and Runtime Systems for Scalable Computers, Lecture Notes in Computer Science. Springer Verlag (2002)
41. Watt, S.M.: A technique for generic iteration and its optimization. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Generic programming, WGP 2006, pp. 76–86. ACM (2006)
42. Willard, D.E.: New data structures for orthogonal range queries. SIAM J. Comput. **14**(1), 232–253 (1985)
43. Wirth, N.: Algorithms + Data Structures = Programs. Prentice Hall PTR, Upper Saddle River (1978)