UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1684

Dynamic Adaptations of Synchronization Granularity in Concurrent Data Structures

KJELL WINBLAD

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018

ISSN 1651-6214
ISBN 978-91-513-0367-3
urn:nbn:se:uu:diva-354026

Dissertation presented at Uppsala University to be publicly examined in room 2446, ITC, Lägerhyddsvägen 2, Uppsala, Friday, 14 September 2018 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Erez Petrank (Computer Science Department, Technion - Israel Institute of Technology).

Abstract
Winblad, K. 2018. Dynamic Adaptations of Synchronization Granularity in Concurrent Data Structures. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 1684. 92 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-0367-3.

The multicore revolution means that programmers have many cores at their disposal in everything from phones to large server systems. Concurrent data structures are needed to make good use of all the cores. Designing a concurrent data structure that performs well across many different scenarios is a difficult task. The reason for this is that the best synchronization granularity and data organization vary between scenarios. Furthermore, the number of parallel threads and the types of operations that are accessing a data structure may even change over time.

This dissertation tackles the problem mentioned above by proposing concurrent data structures that dynamically adapt their synchronization granularity and organization based on usage statistics collected at run-time. Two of these data structures (one lock-free and one lock-based) implement concurrent sets with support for range queries and other multi-item operations. These data structures adapt their synchronization granularity based on detected contention and the number of items that are involved in multi-item operations such as range queries. This dissertation also proposes a concurrent priority queue that dynamically changes its precision based on detected contention.

Experimental evaluations of the proposed data structures indicate that they outperform non-adaptive data structures over a wide range of scenarios because they adapt their synchronization based on usage statistics. Possible practical consequences of the work described in this dissertation are faster parallel programs and a reduced need to manually tune the synchronization granularities of concurrent data structures.

Keywords: concurrent data structures, contention adapting, range queries, lock-freedom, adaptivity, linearizability, ordered sets, maps, key-value stores, concurrent priority queues, relaxed concurrent data structures, locks, delegation locking

Kjell Winblad, Department of Information Technology, Computing Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-0367-3
urn:nbn:se:uu:diva-354026 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-354026)

Dedicated to Meiqiongzi Zhang, Ella Winblad and the rest of my family.

List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I A Contention Adapting Approach to Concurrent Ordered Sets [1]
Konstantinos Sagonas and Kjell Winblad
Published in Journal of Parallel and Distributed Computing, 2018
*(*An extended combination of the publications "Contention Adapting Search Trees, ISPDC'2015" [2] and "Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees, LCPC'2015" [3].)

II More Scalable Ordered Set for ETS Using Adaptation [4]
Konstantinos Sagonas and Kjell Winblad
Published in ACM Erlang Workshop, 2014

III Lock-free Contention Adapting Search Trees [5]
Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson
To appear in SPAA'18: 30th ACM Symposium on Parallelism in Algorithms and Architectures, 2018

IV Delegation Locking Libraries for Improved Performance of Multithreaded Programs [6]
David Klaftenegger, Konstantinos Sagonas and Kjell Winblad
Published in Euro-Par 2014, Proceedings of the 20th International Conference

V The Contention Avoiding Concurrent Priority Queue [7]
Konstantinos Sagonas and Kjell Winblad
Published in Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016

Reprints were made with permission from the publishers.

I am the single primary author for the papers listed above except for Paper IV for which I share the primary authorship with David Klaftenegger.

Other relevant publications written by the author that are not included in this dissertation are listed below:

- On the Scalability of the Erlang Term Storage [8]
  David Klaftenegger, Konstantinos Sagonas and Kjell Winblad
  Published in ACM Erlang Workshop, 2013
- Queue Delegation Locking [9][1]
  David Klaftenegger, Konstantinos Sagonas and Kjell Winblad
  Published in IEEE Transactions on Parallel and Distributed Systems, 2018
- Contention Adapting Search Trees [2]
  Konstantinos Sagonas and Kjell Winblad
  Published in 14th International Symposium on Parallel and Distributed Computing (ISPDC 2015)
- Efficient Support for Range Queries and Range Updates Using Contention Adapting Search Trees [3]
  Konstantinos Sagonas and Kjell Winblad
  Published in Languages and Compilers for Parallel Computing (LCPC 2015)
- Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data [11]
  Kjell Winblad
  Published in 2017 Imperial College Computing Student Workshop (ICCSW 2017)

[1] An early version of the paper "Queue Delegation Locking" has appeared as a brief announcement in the proceedings of SPAA'2014 [10].

Svensk Sammanfattning/Swedish Summary

Flerkärniga processorer finns idag i allt från mobiltelefoner till stora servrar. Flertrådade datastrukturer behövs för att få ut den fulla potentialen från flerkärniga processorer. Det är svårt att utveckla flertrådade datastrukturer som presterar bra under många olika förhållanden. Anledningen till detta är att den synkroniseringsgranularitet och organisation av data som fungerar bäst beror på hur datastrukturen används. Antalet parallella trådar som använder datastrukturen och typen av operationer som används kan till och med variera över tid.

Den här avhandlingen hanterar problemet som beskrivs ovan genom att presentera flertrådade datastrukturer som dynamiskt förändrar sina synkroniseringsgranulariteter och strukturer baserat på användningsstatistik som samlas in under körtid. Två av dessa datastrukturer (en som är låsfri och en som är låsbaserad) implementerar flertrådade mängder med stöd för intervallfrågor (range queries) och andra operationer som arbetar med flera element. Dessa datastrukturer ändrar sina synkroniseringsgranulariteter baserat på detekterade konflikter mellan trådar och antalet element som är involverade i lineariseringsbara (linearizable) operationer som arbetar med mer än ett element. Den här avhandlingen presenterar också en flertrådad prioritetskö som dynamiskt ändrar sin precision baserat på detekterade konflikter mellan trådar.

Experiment som utvärderingar av de föreslagna datastrukturerna i många olika scenarion indikerar att de ofta ger betydligt bättre prestanda än datastrukturer som inte ändrar sin struktur dynamiskt. Datastrukturerna kan tillhandahålla så bra prestanda eftersom de förändrar sig beroende på hur de används. Möjliga praktiska konsekvenser av denna avhandling är snabbare parallella program som kräver mindre manuell finjustering av parametrar för synchroniseringsgranularitet.

Kort Sammanfattning av Artiklar

Artikel I beskriver och utvärderar det låsbaserade konfliktanpassade sökträdet (kallas också “the contention adapting search tree” eller CA-trädet). Den experimentella utvärderingen av CA-trädet visar att denna datastruktur har utmärkt prestanda och skalbarhet i en mängd användningsscenarion: sekventiell användning, endast operationer som involverar ett enda element, intervallfrågor med olika storlekar och intervalluppdateringar. CA-träd kan prestera bra i en mängd användningsscenarion eftersom de kan anpassa sin struktur efter hur de används.

Artikel II diskuterar användningen av CA-träd i minnesbaserade databaser. CA-träd passar väl för att göra sekventiella datastrukturer mer skalbara eftersom mycket av originalimplementationen kan återanvändas. Som ett exempel på detta inkluderar Artikel II en experimentell utvärdering som visar att CA-trädet kan användas för att göra “Erlang ETS `ordered_set`” betydligt mer skalbar än i nuläget.

Artikel III presenterar det låsfria konfliktanpassade sökträdet (kallas också “the lock-free contention adapting search tree” eller LFCA-trädet). LFCA-trädet använder sig av oföränderlighet (immutability) för att göra intervallfrågors konflikttid (längden på den tid då de kan vara i konflikt med andra operationer) kort. Att utnyttja oföränderlighet på detta sätt ger väldigt stark motivation till att dynamisk ändring av synkroniseringsgranulariteten. Anledningen till detta är att flertrådade mängder med grovkornig synkronisering som utnyttjar oföränderlighet har utmärkt prestanda för stora intervallfrågor men väldigt dålig prestanda när det ofta finns parallella uppdateringar medans situationen är omvänd för datastrukturer som använder finkornig synkronisering. Experiment som mäter LFCA-trädets prestanda i en mängd olika senarion presenteras i Artikel III. Dessa experiment visar att LFCA-trädet har överlägsen prestanda jämfört med datastrukturer som använder en bestämd synkroniseringsgranularitet.

Artikel IV diskuterar programmeringsgränssnitt och låsbibliotek för delegationslåsning (delegation locking) samt det arbete som krävs för att ändra en applikation som använder sig av traditionella lås till att använda delegationslåsning. Artikel IV visar också experimentellt de potentiella prestandaförbättringarna som kan åstadkommas genom en sådan förändring av låsningstekniken.

Artikel V beskriver den konfliktundvikande prioritetskön (kallas också “the contention avoiding concurrent priority queue” eller CA-PQ). CA-PQ förändrar sitt beteende, synkroniseringsfrekvens och struktur baserat på detekterade konflikter. Artikel V presenterar också en experimentell utvärdering av en CA-PQ implementation (som använder sig av ett av låsbiblioteken som presenteras i Artikel IV). Att CA-PQ ändrar sitt beteende i olika scenarion bidrar till dess förmåga att prestera bättre än flertrådade prioritetsköer som inte ändrar sig beroende på hur de används.

Acknowledgments

First of all, I would like to thank my main supervisor and co-author of most of my papers Konstantinos (Kostis) Sagonas. Thank you, Kostis, for all the time you have put into making the papers better, and for all good advise. Thank you also for always encouraging me to aim high and pursue meaningful goals. Last but not least, thank you for trusting me and for letting me try out my own ideas.

I would also like to thank my second supervisor and co-author of Paper III Bengt Jonsson. Thank you, Bengt, for giving me encouraging comments and for pointing out that it is essential to have fun. Also, thank you for the enjoyable discussions that we had during the work on Paper III.

David Klaftenegger (co-author of Paper IV) also deserves a big thank you. David, the collaboration that we had during the first one and a half year of our Ph.D. studies has meant a lot to me.

Stavros Aronis, David Klaftenegger, Andreas Löscher and I started our Ph.D. studies at almost the same time and have been in the same research group. Stavros, David, and Andreas: thank you all for being my friends, for helping me with various tasks and for all the happy moments that we have spent together.

I am happy to have shared office with Stephan Brandauer and Elias Castegren during the first years of my Ph.D. studies. Stephan and Elias, thank you for being such friendly people and for all the useful comments I got on papers.

I am also thankful for having had Atis Elsts, Lars-Henrik Eriksson, Kiko Fernandez, Pierre Flener, Olle Gällmo, Alexandra Jimborean, Magnus Lång, Jonatan Lindén, Magnus Norgren, Carlos Pérez Penichet, Joseph Scott, Huu-Phuc Vo, Tjark Weber, Albert Mingkun Yang and many others as colleagues. Thank you all for good collaborations, your company and all the interesting discussions we have had.

My parents, Jan and Birgitta Winblad, thank you for being such good parents, for giving me many opportunities, for being supportive, and for letting me pursue my own path.

Malin and Göran, I am thankful that you are my sister and brother. Thank you for all the support and happiness that I have got from you.

My grandparents, Sven and Karin Eriksson, even though you are no longer with us, I wish you could know that you have meant a lot for me and my Ph.D. studies. Karin, I am very thankful for your persistence in making me better at what I am not so talented at and for your "käpphästar". Sven, you have been an academic role model for me.

Amelie Lind and Martin Viklund deserve a big thank you for designing the beautiful cover of this dissertation. Thanks also to all my other friends and relatives outside the university. You are also very important to me.

Meiqiongzi Zhang, it is difficult to express with words how important you are for me. Thank you for all the support that you have given me, for accepting all my oddities, for calming me down when I need it, for creating a family with me, and for being the love of my life. Thank you for everything.

Funding

This work has been supported in part by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software" and by UPMARC (the Uppsala Programming for Multicore Architectures Research Center).

Contents

1. Overview

This dissertation contains contributions to the field of efficient concurrent data structures. Papers I to V that are attached to the end of this dissertation present the contributions in detail. Titles and publication information for these papers can be found at the beginning of this dissertation. The first section of this chapter (Section 1.1) provides the essential background information and describes the motivations behind the contributions. Section 1.2 contains summaries of the attached papers. The last section (Section 1.3) of this chapter outlines the structure of this dissertation.

1.1 Introduction

Since the early 2000s, the performance improvement rate of a core (processing unit) has slowed down due to physical limitations [12]. Computer manufacturers have started to include several cores in the same chip to compensate, resulting in so-called multicore chips [12].

To get the most out of the new multicore chips and other parallel computers, computer programmers need to consider writing parallel software. Therefore, many abstractions that make the construction of efficient parallel programs easier have been proposed (e.g., synchronization primitives [13–15], programming models [16, 17] and concurrent data structures [18–20]). The efficiency of the algorithms implementing these abstractions is crucial for the scalability and performance of parallel applications[1].

Concurrent data structures

The primary focus of this dissertation is a particular type of abstractions referred to as concurrent data structures. A concurrent data structure is a

[1]The implementation of these abstractions often affect the length of the parts of the program that cannot benefit from increased parallelism (see e.g. [7]). Amdahl's law states that the length of these parts is crucial to the speedup obtainable by increased parallelism [21].

data structure that exposes operations that can be issued by multiple concurrent threads[2]. For example, it is common that a parallel program needs to keep a set of items that can be updated and read by several threads. For this, the program can use a concurrent data structure representing a set. A concurrent set has an interface consisting of operations that can be used to add an item (the INSERT operation), remove an item (the REMOVE operation) and check if an item is in the set (the LOOKUP operation). Concurrent key-value stores (a.k.a. maps) are also common data structures that can be seen as extensions to concurrent sets where each key/item has an associated value. It is trivial to change a data structure for sets so that it becomes a data structure for key-value stores. Papers I, III and V describe novel ways to make concurrent sets and key-value stores. The data structures that are described in Papers I, III and V are so-called concurrent ordered sets, which are concurrent sets where the stored items are ordered internally according to some programmer specifiable order. This internal order of items makes it possible to efficiently support certain operations that will be explained later. But first, some important properties of concurrent data structures will be introduced.

Linearizability

It is straightforward to specify how a sequential data structure should behave. However, specifying the behavior for a concurrent data structure is more involved because one has to consider how overlapping operations should behave. Linearizability is a common correctness criterion for concurrent data structure implementations. A linearizable operation appears to execute atomically (i.e. at one indivisible time point) at some point between the invocation and return of the operation [22]. The point when an operation appears to execute atomically is called linearization point [22]. It is desirable that operations of data structures are linearizable as this is convenient when reasoning about the operations. To see why, note that if an operation does not appear to execute between the invocation and return, then programmers could not rely on that the effects of the operation would be visible after the operation has returned. Furthermore, if opera-

[2]A thread can be seen as a sequential program that may execute concurrently with other threads that can read and write to the same memory. On a multicore chip the threads can be mapped to different cores and may thus run in parallel. Threads running on a single-core chip are also considered concurrent as the operating system typically lets the available threads take turns to execute, which makes them appear concurrent.

tions do not appear to execute atomically, then programmers would also need to reason about the effect of non-completed operations.

Synchronization Granularity

It is also desirable that concurrent data structures enable a high level of parallelism, meaning that many operations can execute in parallel and that conflicts between operations that slow down operations are rare. The degree of parallelism that a concurrent set can provide is correlated with the synchronization granularity of the concurrent set. To define what the synchronization granularity for a concurrent set is, let us first say that an item i is removed at time point t from a concurrent set S if a REMOVE operation call given i as parameter has t as linearization point and i was in S directly before t. Let us also say that an instance S of a data structure D (i.e., an abstract description of a data structure) is a concrete example of D. Now, we can define the synchronization granularity for a concurrent set instance S as the maximum number of items k that are inside S at time point t but that cannot be removed from S at t due to the actions of one particular update operation (INSERT or REMOVE). Furthermore, the synchronization granularity for a concurrent set data structure D is the maximum synchronization granularity of any instance of D. We say that a concurrent set data structure D has changing synchronization granularity if there is no bound to the difference in synchronization granularity of two arbitrary instances of D that both contain exactly the same items. A data structure has fixed synchronization granularity if it does not have changing synchronization granularity. Informally, we say that a small synchronization granularity value means fine-grained synchronization and a large synchronization granularity value means coarse-grained synchronization.

Benefits of Adapting the Synchronization Granularity Based on Contention

To enable a high level of parallelism a plethora of concurrent sets that use a fixed fine-grained synchronization granularity have been described and shown to scale well with the number of cores (e.g. [23–39]). However, unnecessarily fine-grained synchronization often cause overhead both in terms of execution time and memory usage when the contention[3] inside

[3]The contention is high if threads frequently interfere with each other because they need to access or modify the same memory locations.

the data structure is low. To see why, compare a concurrent set which is implemented by a sequential data structure protected by a global lock, and a concurrent set that has a lock for each of its items. The former performs very poorly under parallel access as it sequentializes all accesses but has a low overhead compared to the sequential data structure for uncontended accesses (i.e., essentially only the acquiring and releasing of the lock). The latter can provide much better performance under parallel accesses but has a higher memory footprint and may be slower for uncontended accesses because a smaller part of the data structure can fit in the cache of the processor. Furthermore, the contention level inside a data structure may be impossible to predict when one creates a concurrent program as the number of available cores and the inputs to the program may be unknown. Therefore, it would make sense to automatically change the synchronization granularity at run time based on how much contention is detected. Paper I [1] describes and evaluates a concurrent set called the contention adapting search tree (CA tree for short) that does precisely this. More precisely the CA tree collects information about contention in locks. This contention statistics is used to guide local dynamic changes of the synchronization granularity. Paper II [4] illustrates how the scalability of a real-world key-value store, the Erlang Term Storage (ETS) [8], can be drastically improved with the help of CA trees.

Benefits of Adapting the Synchronization Granularity Based on the Number of Items Accessed by Multi-item Operations

Use cases with low contention are not the only situations that can benefit from coarse-grained synchronization. Concurrent sets that have support for linearizable multi-item operations (operations that operate on multiple items) can benefit from coarse-grained synchronization even though there is contention in the data structure. One example of a multi-item operation that is useful for databases is the range query operation. A range query returns a snapshot of all items in a set that are in a given range (specified by the minimum and maximum key of the range). Range queries and other multi-item operations may benefit from coarse-grained synchronization even more than single-item operations as multi-item operations typically result in a lot of synchronization related overhead in data structures that use fine-grained synchronization (e.g., many locks need to be acquired). Thus conflicting interests exist between single-item operations and

multi-item operations as the former benefit from fine-grained synchronization when there is contention, and the latter benefit from coarse-grained synchronization even when there is contention. Thus, it may be advantageous to not only adapt the synchronization granularity based on the frequency of contended single-item operations but also based on how frequently multi-item operations need to perform synchronization work. Paper I describes CA tree algorithms for several multi-item operations (range queries, range updates, and bulk operations) and how the heuristics for adaptations of synchronization granularity can take the trade-off between single-item operations and multi-item operations into account.

How Immutability Makes the Benefit of Adapting the Synchronization Granularity Even Greater

Let us define the *conflict time* of an operation O_1 as the amount of time in which another operation O_2 can interfere with O_1 in a way that is noticeable for the thread executing O_1 or the one executing O_2. Concurrent sets can exploit immutable data structures to give range queries short conflict times. How this can be done will soon be described but before that immutable data structures need to be introduced.

An immutable data structure is a data structure that cannot be changed [40]. One could naively think that given an immutable data structure instance that represents a set S of items, it would be of linear complexity in the size of S to create a new immutable instance that represents the set $S \cup \{x\}$, where x is an item such that $x \notin S$. Fortunately, in an immutable balanced search tree this can be done in time that is only logarithmic in the size of S, as one only needs to copy nodes on a path from the root of the tree to a leaf to create the new instance [40][4].

One can construct a concurrent set with coarse-grained synchronization from a single mutable reference pointing to an immutable balanced search tree. Let us call such a data structure *Im-Tr*. The INSERT and REMOVE operations of *Im-Tr* change the mutable reference using an atomic compare-and-swap (CAS) instruction[5] so the reference points to a new immutable instance reflecting the update. Using this scheme (which is also described

[4]Section 2.5.2 explains how one can construct efficient update operations for immutable search trees.
[5]The CAS instruction is explained in Section 2.3.

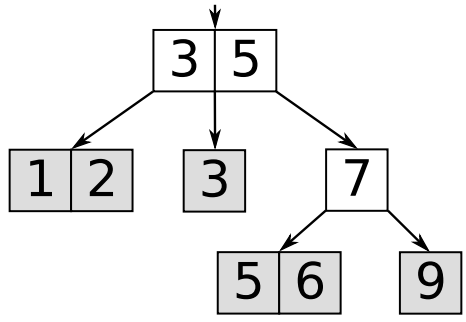


Figure 1.1. The structure of a lock-free *k*-ary search tree.

by Herlihy [41]) it is trivial to perform range queries with constant conflict time as they only need to get a snapshot by reading the mutable reference and then perform the range query in the snapshot. Unfortunately, this scheme does not scale well when there are parallel updates as the updates will compete to update the shared mutable reference and only one can succeed at a time.

The lock-free *k*-ary search tree [42] was the first concurrent set with fine-grained synchronization to also exploit immutable data to make the conflict time of range queries short. Let us illustrate how immutable data can be exploited in concurrent sets with fine-grained synchronization using the lock-free *k*-ary search tree as an example.

Figure 1.1 illustrates the structure of a lock-free *k*-ary search tree. The gray leaf nodes in Fig. 1.1 contain immutable arrays with the actual items that are stored in the set represented by the *k*-ary search tree instance. Internal nodes of the *k*-ary search tree (white nodes in Fig. 1.1) do not contain any actual items but instead hold search keys which are used to direct the search for a particular item. Together the nodes form a search tree. The *k* parameter of the *k*-ary search tree decides the maximum number of items (or search keys) that can be stored in each node. Update operations in the *k*-ary search tree replace a leaf node with a new leaf node corresponding to the update by changing the pointer in the leaf's parent with the help of a compare-and-swap (CAS) instruction[6]. A range query in a *k*-ary search tree collects a list of immutable arrays that may contain items in the range and makes sure that no update operation replaces any of the arrays while the

[6]A leaf node in the *k*-ary search tree may occasionally also need to be split (to avoid more than *k* items in any leaf node) or joined (to avoid too few items in a leaf node) during an update.

arrays were collected. There is no need for the range query of k-ary search trees to scan the items inside the immutable arrays during the conflict time; this scan can happen after the linearization point of the range query.

Both the synchronization granularity of the lock free k-ary search tree and the expected conflict time for range queries depend on the parameter k. Remember that k determines the maximum number of keys/items in the nodes. The synchronization granularity of the lock-free k-ary search tree is proportional to its parameter k, while the expected conflict time for range queries in lock-free k-ary search trees with large enough ranges is approximately inversely proportional to k (i.e. approximately proportional to $\frac{1}{k}$), under the assumptions that the lock-free k-ary search tree is balanced and that all nodes contain k items. Intuitively, one can expect the same kind of relationship between the synchronization granularity and the conflict time of range queries in all concurrent sets that exploit immutability to get short conflict times for range queries similarly to how the lock-free k-ary search tree does this. Thus, such data structures that use a fixed synchronization granularity have a lot to lose when their synchronization granularity does not fit the workload at hand. This is either because they may use too fine-grained synchronization that results in a lot of conflicts between range queries and other operations (due to long conflict times for range queries) or because the synchronization is not fine-grained enough (so that update operations frequently disturb each other.) Therefore, there is a clear motivation for a concurrent set that exploits immutability in a similar manner to the lock-free k-ary search tree and automatically adjusts its synchronization granularity to fit the workload at hand.

The CA tree described in Paper I can be optimized to exploit immutable data structures in a similar manner to how the k-ary does this [11][7]. Paper III [5] describes and evaluates the lock-free contention adapting search tree (LFCA tree) that also similarly exploits immutable data structures.

The LFCA tree adapts its synchronization granularity similarly to the CA tree described in Paper I (i.e. using collected usage statistics that take contention and the number of items covered by range queries into account). However, in contrast to CA trees, LFCA trees do not use locks, and LFCA trees use helping techniques that give them a progress guarantee

[7]The publication [11] that describes this optimization is not included in this dissertation as results for the optimized CA tree are also included in Paper III [5].

called lock-freedom. To appreciate the benefits of the LFCA tree over the lock-based CA tree, let us briefly review why locks may be problematic.

Locks and One of Their Problems

A lock is an abstraction that only permits that one thread holds a particular lock at any point in time and can thus be used to grant exclusive access to some memory locations (which is why locks are also sometimes called mutual execution locks or mutexes). Despite their popularity, locks have some problems associated with them. For example, suppose that thread A is holding a lock L, some other thread B is waiting to acquire L and A gets preempted by the operating system, then neither A nor B will make progress while A is preempted even though no thread is actively using the resource that L is protecting.

Lock-freedom and Wait-freedom

The lock related problem described above is avoided by so-called lock-free algorithms that guarantee that at least one thread can make progress at all times. As already mentioned above, Paper III [5] describes a new lock-free data structure for sets with range query support called the LFCA tree. As is argued in Paper III, LFCA tree's INSERT, REMOVE and range query operations are all lock-free (meaning that these operations guarantee system-wide progress [43]) while its LOOKUP operation is even wait-free (meaning that lookups can always make progress independently of what other operations are doing [43]).

Adaptation to Avoid Contention in Concurrent Priority Queues

Paper V investigates if concurrent priority queues can benefit from synchronization related adaptations based on detected contention. A priority queue is a data structure that represents a set of ordered items and that has an operation for inserting an item (called INSERT) and an operation for deleting and returning the smallest item (called DELMIN). Concurrent priority queues are important for many parallel algorithms (see e.g. [44]). In a traditional linearizable concurrent priority queue, it is inherent that the part of the queue containing the smallest items gets heavily contented when parallel DELMIN operations are competing to remove the smallest item.

Concurrent priority queues with so-called relaxed semantics (e.g. [45–47]) try to deal with this problem. Relaxed semantics in this context means that an item returned by a DELMIN operation may not have been the smallest item in the queue at some point between the invocation and return of the DELMIN operation. Typically, relaxed priority queues provide some kind of bound on how far an item that is removed with the DELMIN operation can be from the actual minimum item [45, 46]. This bound can also typically be controlled with a parameter that is passed to the priority queue when it is created. With relaxed semantics, priority queues can avoid bad performance due to contention, but the application may instead suffer from e.g. wasted work caused by the imprecision of the priority queue. Therefore, it would make sense to turn on relaxations of the semantics only when high contention motivates this. Paper V [7] describes and evaluates the contention avoiding priority queue (CA-PQ) that does precisely this. CA-PQ avoids contention when its relaxed semantics is turned on by changing how often threads synchronize with each other.

The CA-PQ implementation uses a particular type of locks called delegation locks provided by one of the locking libraries described in Paper IV [6]. Delegation locks can provide better performance under contention than traditional locks as they can force subsequent critical sections from different threads into executing on the same core which avoids costly and frequent transferring of data between the private caches of cores [9,48–50]. Paper IV describes and evaluates library interfaces for delegation locks.

I claim that Papers I, III and V strongly support the following thesis which is the central thesis of this dissertation:

Thesis

Concurrent ordered sets that dynamically adapt their structure based on usage statistics can perform significantly better across a wide range of scenarios compared to concurrent ordered sets that are non-adaptive.

1.2 Short Summaries of the Papers

This section contains short summaries of the five papers that are included in this dissertation.

1.2.1 **Paper I**: A Contention Adapting Approach to Concurrent Ordered Sets

Paper I [1] describes contention adapting search trees (CA trees) and uses CA trees to argue that dynamic adaptation of the synchronization granularity inside concurrent data structures can be beneficial. A CA tree is a concurrent lock-based search tree that can be used to represent sets and key-value stores (maps). The distinguishing feature of CA trees is that they dynamically change their synchronization granularity based on heuristics that take contention and the number of items that are needed by operations into account.

Some important properties of CA trees that are discussed at length in the paper are listed below:

- CA trees support single-item set operations (e.g., INSERT, REMOVE and LOOKUP) as well as multi-item operations (e.g., range queries, range updates, and bulk operations). The paper contains detailed proof sketches for that the operations listed above are deadlock free, livelock free and linearizable.
- The paper argues that the expected time complexities for an uncontended CA tree's operations are the same as those expected from an efficient sequential ordered set.
- One can derive new CA tree variants with different performance characteristics by just providing different sequential ordered set implementations. The CA tree's performance characteristics under low contention will essentially be the same as the performance characteristics of this sequential data structure. The usefulness of the flexibility to easily derive CA trees with different performance characteristics is demonstrated in the paper which shows results for two CA tree variants with different sequential data structure components.

The results presented in the paper show that the CA trees outperform many recently proposed data structures over a wide range of scenarios, measuring sequential performance, scalability on a big multicore system, and performance under different types of workloads.

1.2.2 **Paper II**: More Scalable Ordered Set for ETS Using Adaptation

Paper II [4] describes how CA trees can be used to improve the scalability of an existing in-memory database with a complex interface, namely the ordered_set table type of the Erlang Term Storage (ETS). ETS is a part of Erlang/OTP (the most popular implementation of the Erlang programming language [51]). ETS has support for many different operations including single-item set operations, bulk operations and different kinds of range operations [8].

The ETS implementation itself is written in the C programming language. The current implementation of the ordered_set table type is a sequential AVL tree [52] protected by a readers-writer lock. Paper II describes how much of the already existing code for the AVL tree can be reused to implement the complex ETS interface using the CA tree.

Paper II also presents results from performance and scalability measurements of a new prototype implementation of the ordered_set table type using the CA tree. These measurements indicate that the CA tree can greatly enhance the scalability of the ordered_set table type while keeping a sequential performance that is very close to the current single-lock based implementation.

1.2.3 **Paper III**: Lock-free Contention Adapting Search Trees

Paper III [5] describes a new lock-free data structure for sets with range query support called the lock-free contention adapting search tree (LFCA tree for short). The LFCA tree is similar in spirit to the lock-based CA tree presented in Paper I as it also adapts its synchronization granularity based on heuristics that take detected contention and the number of items that range queries need to access into account. However, in contrast to the lock-based CA tree that has blocking operations, LFCA tree's operations are non-blocking. More precisely, LFCA tree's INSERT, REMOVE and range query operations are all lock-free, while its LOOKUP operation is even wait-free.

The LFCA tree uses immutable data structures internally to make the time in which a range query can conflict with other operations (conflict time) short. As is demonstrated in Paper III, combining adaptation of synchro-

nization granularity with the exploitation of immutability to obtain short conflict times of range queries results in substantially better scalability over a wide range of scenarios compared to data structures that use a fixed synchronization granularity.

Paper III presents experimental results that indicate that the LFCA tree not only has progress-related advantages compared to the lock-based CA tree but also that it performs significantly better in a variety of scenarios (especially when thread preemptions are frequent). Still, one can make the case that the lock-based CA tree has some advantages over the LFCA tree in some use cases. For example, the use of locks makes it easy to extend the lock-based CA tree with additional operations and compose operations into new linearizable operations[8]. There are thus use cases for both the lock-based CA tree and the lock-free CA tree.

Some of the techniques used in the LFCA tree operations may be useful for other data structures as well. For example, the experimental results in Paper III indicate that the range query operation developed for the LFCA tree is less prone to starvation than the range query operation suggested for the lock-free k-ary search tree [42].

Paper III gives additional support for the thesis statement by providing experimental evidence that a lock-free concurrent set that dynamically adapts its synchronization granularity based on usage statistics can outperform related non-adaptive data structures over a wide range of scenarios.

1.2.4 **Paper IV**: Delegation Locking Libraries for Improved Performance of Multithreaded Programs

A delegation lock L lets threads delegate critical sections (code that should execute under the protection of the lock) directly to L. As traditional locks, a delegation lock guarantees that only one critical section can run at a time. However, in contrast to traditional locks, which thread is executing a critical section is decided by the delegation locking algorithm and may not be the thread that issues the critical section. Delegation locks have been shown to give substantially better performance than traditional locks as they can

[8]The reason why composing operations can be important for database applications is outlined in the work by Avni *et al.* [53].

avoid frequent transferring of the data protected by the lock between the private caches of cores [48–50]. However, delegation locks require a different interface than traditional locks as delegation locks need to allow the submission of code and associated data to the lock itself. The primary contribution of Paper IV is to discuss delegation locking interfaces, the porting efforts required to port software with traditional locks into using delegation locks and the problems one may encounter when doing such porting.

The first part of Paper IV presents and discusses C and C++ libraries for delegation locking. It is argued that these C and C++ libraries are the first portable libraries for delegation locking. These libraries use the standard libraries introduced in C11 and C++11 for shared memory programming, which makes it possible for them to not contain any platform specific code.

In the second part of Paper IV, a case study is presented. The case study consists in the porting of a real-world application, the Erlang Term Storage (ETS)[8], into using delegation locking. The description of this case study contains the steps that were made to do the porting, the effort required (in terms of changed lines) and solutions to encountered problems. Paper IV also contains experimental results comparing different versions of the ported code with the original code and a recently proposed traditional locking algorithm. These results indicate that delegation locks can substantially improve the scalability of ETS.

1.2.5 **Paper V**: The Contention Avoiding Concurrent Priority Queue

Paper V describes a concurrent priority queue called the contention avoiding priority queue (CA-PQ). Initially, the CA-PQ functions as a traditional linearizable concurrent priority queue. However, the CA-PQ operations collect statistics about contention, and when these statistics indicate that contention is high, relaxations to the semantics of CA-PQ are activated to avoid contention. These relaxations allow the DELMIN operation to pick an item which is currently not the minimum item of the priority queue. The stronger the relaxation is the more items can be between the item picked by the DELMIN operation and the actual minimum item.

CA-PQ's DELMIN operation can relax the semantics of the concurrent priority queue by letting every few DELMIN operations grab several items from the head of the queue. These grabbed items are buffered in a thread-local buffer for future DELMIN operations. The contention in the head of the queue is thus avoided because significantly fewer operations need to access the head.

CA-PQ's INSERT operations can make another relaxation to the semantics of the concurrent priority queue, if the usage statistics indicate that this may be beneficial. This relaxation is done by buffering the items of up to a certain number of INSERT operations locally so that fewer INSERT operations need to access shared memory.

As is argued in Paper V, CA-PQ's relaxations do not affect the correctness of many concurrent priority queue applications. However, unnecessary strong relaxations of the semantics can lead to poor performance due to wasted work which is why it makes sense to only activate the relaxations dynamically when high contention motivates them. Several concurrent priority queues with relaxed semantics have been proposed (e.g. [45–47]). The paper claims that CA-PQ is the first priority queue that adaptively turns on relaxations only when usage statistics indicate that this is motivated.

One application which uses concurrent priority queues, where too strong relaxations can lead to bad performance due to wasted work, is a parallel version of Dijkstra's single source shortest path (SSSP) algorithm. This SSSP algorithm is used for the evaluation of CA-PQ in Paper V. The results from this benchmark show that the CA-PQ provides substantially better performance and scalability compared to several recent concurrent priority queues with relaxed semantics [45–47] and a heavily optimized concurrent priority queue with traditional semantics [26] over a wide range of input graphs. One reason that the CA-PQ performs so well in many different scenarios is its ability to dynamically change the amount of synchronization that its operations do according to detected contention. These results give the thesis statement even more support.

1.3 Organization

Chapter 2 complements Section 1.1 with more background information about concurrent programming for multicores and data structures. Chapters 3 and 4 contain high-level descriptions of the new concurrent data structures that this dissertation proposes. Chapter 3 contains high-level descriptions of the CA tree (Paper I) and the LFCA tree (Paper III). Chapter 3 also shows some highlights from the results presented in Papers I, II and III. Similarly, Chapter 4 provides a high-level view of the CA-PQ as well as highlights from the results presented in Paper V.

Papers I to V contain the main discussions of related work. Chapter 5 complements the related work sections of these papers with more extensive descriptions of some publications that are only briefly mentioned in the papers, and with discussions of some works that have appeared after the publication of the included papers.

Chapter 6 presents some software artifacts that I have developed during the work that lead to this dissertation. Chapter 7 discusses future work. Chapter 8 concludes this dissertation. As already mentioned, Papers I to V are attached at the end of this dissertation.

2. Background

This chapter aims at providing the background information that is needed for understanding the contributions described in Papers I to V.

The chapter starts with a gentle introduction to concurrent programs (Section 2.1). A high-level description of multicore computers is presented in Section 2.2. Section 2.3 describes the atomic compare-and-swap instruction, which many lock-free data structures rely upon (e.g., the one that Paper III describes). The following section (Section 2.4) discusses locks, which are used by the CA tree described in Paper I and the CA-PQ described in Paper V. Section 2.5 explains essential data structure concepts that are relevant for the CA trees. Section 2.6 briefly discusses memory reclamation for concurrent data structures implemented in low-level programming languages. Finally, the chapter ends with some notes about further reading (Section 2.7).

2.1 A Gentle Introduction to Concurrent Programs

A concurrent program executes several sequential processes (from here on called threads) concurrently. Figure 2.1[1] shows code of a concurrent program. The main thread of this program launches two threads. One of these threads prints the string “a” and the other one prints the string “b”. As these two threads can execute concurrently, there are two possible outcomes from running the program. The program may print “ab” or “ba”.

Synchronization and Shared Memory

Let us assume that the non-deterministic behavior of the program described in Fig. 2.1 is undesirable. A deterministic version of the program can be

[1]Throughout this chapter, pseudocode that should be understandable for someone familiar with the C programming language [54] or other similar language will be used.

```
thread A {
  print("a");
};
thread B {
  print("b");
};
spawn(A);
spawn(B);
```

Figure 2.1. The code for a concurrent program.

```
boolean done = false;
thread A {
  print("a");
  done = true;
};
thread B {
  while( ! done ) /* Do nothing */ ;
  print("b");
};
spawn(A);
spawn(B);
```

Figure 2.2. The code for a concurrent program where two threads synchronize with each other using a shared variable.

achieved through synchronization between the threads. One way of achieving such synchronization is by making use of memory which is shared between the threads. The program in Fig. 2.2 illustrates how this can be done: the shared variable `done` is used for synchronization. Thread `A` writes `true` to the `done` variable after "a" has been printed and thread `B` waits until `done` is set to `true` before it prints "b".

Sequential Consistency

In all code examples in this chapter, it is assumed that writes and reads to shared variables take effect at an indivisible point in time (i.e, they happen atomically) and that the operations appear to execute in the order they are presented in the code from the point of view of all threads. This model for the memory is referred to as sequential consistency [55]. Many practical programming languages expose less strict memory models to the programmers (the Java memory model [56] and the C11 memory model [57] are examples of this). Less strict memory models than sequential consistency allow the compiler and the processor to reorder memory operations in ways that make the execution of programs substantially faster. Languages that

use such weaker memory models usually also expose memory operations that let the programmer enforce atomicity and a specific ordering for memory operations when it matters. Sequential consistency is assumed in the code examples of this chapter to keep the focus on the described concepts instead of on their implementation details. However, we note that by relaxing the semantics of some memory operations in these code examples, one could obtain code that runs faster on modern processors. The memory model is not an issue for the pseudocode of Paper I as the CA tree algorithm use locks in a way that enforces the required memory orderings. Similarly, the CA-PQ, which Paper V presents, uses a linearizable concurrent priority component that enforces the necessary memory orderings. In the pseudocode for the lock-free data structure (LFCA tree), which Paper III presents, we explicitly mark the memory operations that should have sequentially consistent semantics.

Concurrent Programming Models

There are different types of programming models or abstractions for writing concurrent programs. The model exemplified in Fig. 2.2, where threads exchange information by reading and writing to the same memory locations, is called shared memory programming. This dissertation proposes algorithms that are expressed through the shared memory programming model because such programs can be translated more or less directly to code that general purpose multicore processors execute.

Other programming models try to provide a higher-level abstraction from the hardware. The Erlang programming language [51] is an example of this. Erlang is utilizing the message passing programming model for concurrent programming. In this model, threads (or processes as they are called in the Erlang terminology) communicate with each other through asynchronous message passing. The message passing programming model may lead to fewer concurrency-related bugs compared to shared-memory programming as the message passing model makes the communication explicit. However, high-level programming models may not be the "best" for implementing efficient concurrent data structures as they make it difficult to control precisely what the hardware will do. This is why the Erlang Term Storage, which is discussed in Paper II, is implemented in the C programming language using shared memory programming instead of being imple-

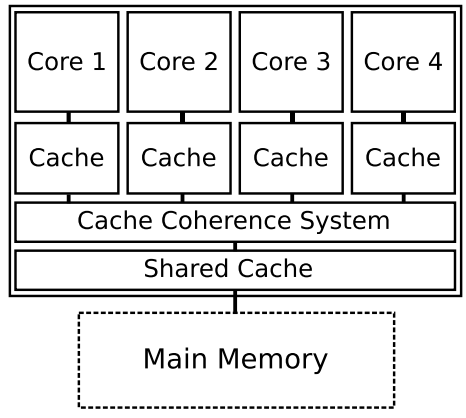


Figure 2.3. Illustration of a multicore processor chip.

mented in Erlang. The reader is referred elsewhere for an introduction to other concurrent programming models (e.g., [58]).

2.2 Multicore Computers

If one wants to understand why one program or data structure performs better than another, it is often crucial to have an understanding of how the hardware works. For example, the concurrent priority queue described in Paper V can provide better performance than related data structures as its DELMIN operation does fewer expensive memory operations compared to related data structures.

Many types of systems can be classified as multicore computers. What they all have in common is that they have more than one core (sometimes also called processing unit) that can execute programs or threads in parallel. The primary target for the concurrent data structures proposed in this dissertation is general purpose cache coherent multicore processors and cache coherent non-uniform memory access (ccNUMA) machines. Therefore, the following text tries to provide a conceptual understanding of how these systems work.

Cache Coherent Multicore Processors

Cache coherent multicore processors are the central component of modern phones, laptops, desktops and server systems [12,59]. Figure 2.3 illustrates

the organization of such processors. The processor in Fig. 2.3 can run four threads in parallel. That is, one thread on each of the cores. Some processors also have cores with support for something called multi-threading (also called hyper-threading). One example of a processor with multi-threading support is the Intel(R) Xeon(R) E5-4650 processor that is used for many of experiments presented in this dissertation. A multi-threading core lets up to a fixed number of threads take turns to execute. This is beneficial as it allows the core to make use of "waiting time". For example, when a core with multi-threading support needs to wait for some data to arrive from the main memory before it can continue to execute a thread T, the core can start to execute another thread for a while instead of being idle until the data that T needs has arrived.

The caches in the processor can cache a fixed number data regions (corresponding to regions in the main memory) in what is called *cache lines* (each cache line can often store 64 bytes of data [59]). Modern multicore chips do not have a single cache but a hierarchy of caches (see Fig. 2.3). In the processor presented in Fig. 2.3, each core has a core private cache, and all cores share a larger but slower shared cache. The system that is synchronizing the cached data between the core private caches is called a cache coherence system. Without a cache coherence system, data that is cached in the private cache of a core could be out of date indefinitely due to a store to the corresponding data region by another core (i.e., without a cache coherence system the private caches are not kept synchronised automatically). The cache coherence system works to make the caches coherent (i.e., the cache coherence system works to make the core private caches present coherent views of the main memory).

Each cache line in the private caches is associated with a state. These states are used by the cache coherence system to decide which data regions need to be transferred between the private caches and which cache lines need to be written to the main memory before they are evicted. For example, in cache systems that use the MESI cache coherence protocol [60], a cache line that is in the state *Shared* can be read but not written to as the memory region stored in this cache line might be present in the private caches of other cores. Before the data in such a shared cache line can be modified, the cache coherence system first has to get this cache line into the *Exclusive* state, which can only happen after the corresponding cache lines in the other private caches have been invalidated (i.e., moved to the state *In-*

valid). If some cached data is already in the *Exclusive* state, the core can go ahead and write to this cached data without any involvement of the cache coherence system because such data cannot be cached in the private cache of any other core.

A *cache hit* happens in cache C when a load or store instruction can be served from cache C without involvement of the lower parts of the memory hierarchy. A *cache miss* occurs in cache C when a load or store instruction cannot be served by cache C. There are several kinds of cache hits with different costs depending on in which cache line the hit occurs and how much work the cache coherence system has to do. The reader is encouraged to have a look at Paul E. McKenney's excellent article [61] about caches and memory barriers, that explains aspects of modern processors which are of importance when designing and evaluating concurrent data structures.

False Sharing in Multicore Processors

So-called *false sharing* can cause scalability issues in concurrent programs running on multicores. False sharing can happen, for example, when two thread local variables belonging to two different threads, A and B, are stored in a memory region that gets mapped to the same cache line. When thread A updates its thread local variable it may invalidate the corresponding cache line in thread B's core even though A has not changed any memory location that is used by B and vice versa. To avoid false sharing, programmers have to be aware of how the data is stored in the memory and sometimes introduce padding to make sure that different variables get mapped to different cache lines.

Cache Coherent Non-uniform Memory Access (ccNUMA) Machines

All papers included in this dissertation include results from experiments that were run on so-called ccNUMA machines. ccNUMA machines combine several processor chips (which may or may not be multicore chips) into a cache coherent multicore system. These systems are referred to as non-uniform memory access (NUMA) because in these systems the access time for a particular main memory location is different for cores located on different chips[2].

[2]More information on ccNUMA machines and their implementations can be found in the book by Hennessy and Patterson [59, page 378].

```
1 bool CAS(value* location, value expected, value new){
2   value old = *location;
3   if ( old == expected ) {
4     *location = new;
5     return true;
6   } else {
7     return false;
8   }
9 }
```

Figure 2.4. The figure shows code that illustrates what the CAS instruction does. A processors with support for the CAS instruction does what the code in the figure does but atomically. That is, it appears as if the CAS instruction executes in one atomic step. Note that the syntax for denoting the memory address of where a value is stored is borrowed from the C programming language.

2.3 The Compare and Swap Instruction

The load and store instructions are often not powerful enough to implement synchronization primitives such as locks efficiently. Herlihy and Shavit explain in their book [43, page 99] why the load and store instructions are not powerful enough to implement such synchronization primitives efficiently. Therefore, an instruction called compare-and-swap (CAS)[3] is often provided by multicore processors. Figure 2.4 illustrates what the CAS instruction does using code.

The CAS instruction takes a memory location, an expected value, and a new value as input. When the instruction is issued, it atomically reads the value stored at the given location, compares this value with the given expected value, and stores the new value at the given memory location only if the expected value was equal to the loaded value in which case the value true is returned. The CAS operation returns false without doing any modification if the loaded value did not match the expected value.

2.4 Locks

The CA tree described in Paper I and the CA-PQ described in Paper V use locks for granting a thread exclusive access to some data. It is assumed

[3]Intel's name for the CAS instruction is CMPXCHG. Some processors provide instructions called load-link/store-conditional instead of a CAS instruction (e.g., PowerPC processors). These instructions can be used in a similar way as the CAS instruction.

that the reader knows what locks are and how to use them. This section discusses aspects about locks that are relevant for the papers included in this dissertation. The term *critical section* refers to a code region that is protected by a lock.

```
1  struct Lock {
2    bool taken = false;
3  }
4  bool try_lock(Lock* lock){
5    return CAS(&lock->taken, false, true);
6  }
7  void lock(Lock* lock){
8    while( ! try_lock(lock) ) /* Do nothing */;
9  }
10 void unlock(Lock* lock){
11   lock->taken = false;
12 }
```

Figure 2.5. The implementation of a test-and-set lock [62].

Different Lock Algorithms and Their Properties

The test-and-set lock [62], outlined in Fig. 2.5, is arguably very simple to implement, but it has some potential problems. First of all, the test-and-set lock is not starvation-free. Starvation-free locks guarantee that all threads that try to acquire a lock L by calling L's `lock` operation eventually acquire L given that all threads that acquire L eventually release L [43, page 24]. If several threads continuously acquire and release a test-and-set lock, one of these threads might be unlucky, so it never gets to acquire the lock. Several starvation-free lock algorithms have been proposed (e.g., [63–67]).

Another problem with the test-and-set lock is its performance under contention. Threads waiting to acquire a test-and-set lock will repeatedly issue a CAS instruction that will fail most of the time. Frequent issuing of the CAS instruction can result in a lot of traffic in the cache coherence system, which slows down the lock-hand-over [68, page 53]. An improved version of the test-and-set lock called the test-and-test-and-set lock [69] deals with this problem to some degree by spin waiting using a load instruction in a loop. Queue based locks [65–67] often perform even better under contention as all waiting threads just keep track of whether the thread before them in the line releases the lock, which avoids a burst of traffic in the cache coherence system when a contended lock is released. Queue based locks are also said to be fair as they order the waiting threads in a first-in first-out fashion.

```
1  struct SeqLock {
2    uint64 number = 0;
3  }
4  bool try_lock(SeqLock* lock){
5    uint64 tmp = lock->number;
6    if(tmp % 2 == 0) {
7      return CAS(&lock->number, tmp, tmp + 1);
8    } else {
9      return false;
10   }
11 }
12 void lock(SeqLock* lock){
13   while( ! try_lock(lock) ) /* Do nothing */;
14 }
15 void unlock(SeqLock* lock){
16   lock->number = lock->number + 1;
17 }
18 uint64 start_read(SeqLock* lock){
19   return lock->number;
20 }
21 bool validate_read(SeqLock* lock, long token){
22   return (token % 2 == 0) && (token == lock->number);
23 }
```

Figure 2.6. The implementation of a sequence lock [72].

Unfortunately, this fairness limits the performance of queue based locks on NUMA systems. The reason is that both the lock-hand-over and the transfer of the data protected by the lock takes much longer time if the thread that is taking over the lock runs on another chip. NUMA-aware locks exploit the structure of NUMA systems to provide better performance at the cost of fairness (e.g., [70, 71]).

Sequence Locks

Something that can make a big difference for the performance of the CA tree that is described in Paper I is if read-only operations can avoid writing to shared memory. Such read-only operations can be accomplished in an optimistic fashion by using sequence locks [72].

Figure 2.6 shows the implementation of a sequence lock. The lock structure consists of just an integer variable called `number` (line 2). It is essential that this integer variable can store enough values so that overflows are unlikely when the variable can only be incremented by 1. (The 64-bit integer that is used in Fig. 2.6 should be enough given the current speed of computers.) The lock is free if the `number` variable stores an even number and the lock

```
1 uint64 token = start_read(account_lock);
2 int sum = var1 + var2; /* Optimistic read-only critical section */
3 if( ! validate_read(account_lock, token) ){ /* Optimistic read failed */
4   lock(account_lock);
5   sum = var1 + var2;   /* Forcing critical section */
6   unlock(account_lock);
7 }
8 ... /* Code that use sum */
```

Figure 2.7. Code that illustrates how a sequence lock can be used to accomplish optimistic read-only critical sections that do not write to shared memory.

is acquired if the `number` variable stores an odd number. The `try_lock` operation (line 4) first loads the current value of the lock's `number` variable. If the `number` variable was even (indicating that the lock is free), an attempt to acquire the lock will be made by using the CAS instruction to atomically add one to the number variable so its value becomes odd. On the other hand, if the `number` variable was odd (indicating that the lock is acquired), the `try_lock` call gives up and returns false. The `lock` operation (line 12) repeatedly tries to acquire the lock until it succeeds. The `unlock` operation (line 12) increments the number variable by 1 to make it store an even number (which releases the lock).

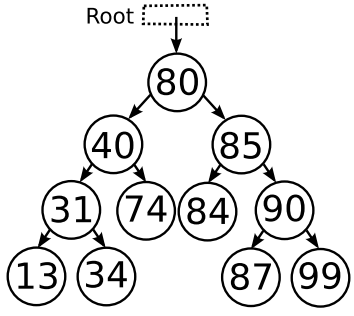
The sequence lock operations for optimistic read-only critical sections that do not write to shared memory are called `start_read` and `validate_read` (lines 18-23). How these operations can be used is illustrated in Fig. 2.7. An optimistic read-only critical section is initiated by saving the token returned by a `start_read` call (Fig. 2.7, line 1). The `start_read` operation returns the current sequence number. A call to `validate_read` with the previously returned token as parameter ends the optimistic read-only critical section (line 3). The `validate_read` operation returns a boolean value indicating whether the optimistic read-only critical section was successful or not (i.e., whether the critical section executed without interference from a mutual exclusion critical section or not). The validation checks that the given token is even (which means that the lock was free when the critical section started) and that the number variable of the lock is the same as the supplied token (which means that the lock was not acquired while the optimistic read-only critical section was executing). If the optimistic attempt fails, the thread has to either retry or acquire the lock in a non-optimistic way to perform the critical section (this is what is done in lines 4–6 of Fig. 2.7).

Readers-writer Locks

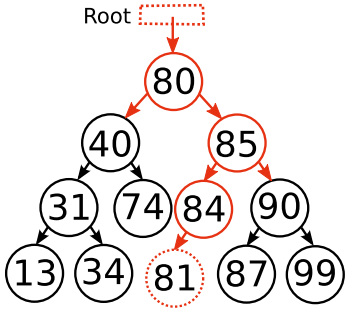
The CA tree implementation that was used to obtain the results presented in Paper I uses a slightly more advanced lock type than the one provided in Fig. 2.6. The sequence lock that the CA tree uses is also a readers-writer (RW) lock. A sequence lock that supports both optimistic and forcing (i.e., that always succeed) read-only critical sections can be constructed using the sequence lock algorithm presented above and one of the generic readers-writer algorithms presented by Calciu *et al.* [73]. Such a lock allows several forcing read-only critical sections to execute in parallel. However, the `read_lock` and `read_unlock` operations that are used to initiate and finish forcing read-only critical sections need to write to shared memory as they need to announce their presence to mutual exclusion critical sections.

Delegation Locks

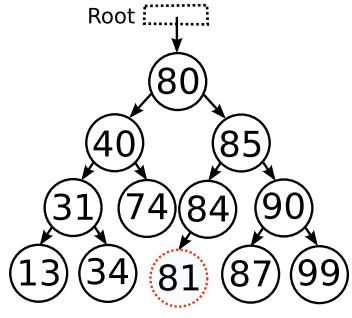
The locks that have been described so far let threads execute their critical sections by themselves. Such traditional locks may be convenient from the perspective of the programmer as they make it easy to use thread local variables inside critical sections [6]. However, it has been demonstrated that delegation locks that let programmers delegate the execution of critical sections to the lock itself can perform substantially better in contended scenarios (e.g., [9, 48–50]). As mentioned earlier, the programming effort that is required to change programs that use traditional locks into using delegation locks is discussed in Paper IV. Delegation locks can enforce that subsequent critical sections frequently execute on the same core. This way, the data that is protected by the lock can stay in the private cache of the same core for relatively long periods of time instead of frequently being transferred to the private cache of a new core when a new critical section is executed. This can decrease the number of expensive data transfers that need to happen in the cache system. The reader is referred to Paper IV and the queue delegation locking paper [9] for more in-depth explanations of delegation locking. Queue delegation locking is the delegation locking algorithm that is used by the concurrent priority queue implementation described in Paper V.



(a) The figure shows the structure of a binary search tree.



(b) A binary search tree created from the tree of Fig. 2.8a with a functional (i.e., does not modify the input data structure) INSERT operation which was given the tree of Fig. 2.8a and the item 81. The nodes marked with red were allocated during the INSERT.



(c) The tree from Fig. 2.8a after the item 81 has been inserted in-place (i.e. the original tree has been modified). Only the red node has been allocated during the INSERT.

Figure 2.8. Figures showing the structure of a binary search tree (Fig. 2.8a) and the effect of functional (Fig. 2.8b) and mutating (Fig. 2.8c) INSERT operations.

2.5 Data Structures

This section aims to explain data structure concepts that are relevant for the papers included in this dissertation at a high-level of abstraction. We will start to look at mutable and immutable search trees (Section 2.5.1 and Section 2.5.2), which are building blocks for the data structures presented in Papers I and III. We will then briefly discuss skip lists (Section 2.5.3) as they are used by one of the CA tree implementations presented in Paper I and the CA-PQ implementation presented in Paper V. Finally, Section 2.5.4 discusses an optimization that makes skip lists and external search trees more cache friendly.

2.5.1 Search Trees

A search tree is data structure for storing a set of ordered items. Figure 2.8a shows the structure of a search tree storing the integers 13, 31, 34, 40, 74, 80, 84, 85, 87, 90 and 99. A *binary search tree* is a search tree where every node in the tree can link to at most two *child* nodes. All items stored under the left pointer of a node N are smaller than N's item and all items stored under the right pointer of N are greater than or equal to N's item. The above property makes it possible to search for a specific item by traversing nodes starting from the one pointed to by the root of the tree (the *root node*) and at every node decide whether to go to the left child or right child depending on the item in the current node. A *leaf node* is a node without any children. A search tree can be *internal* (meaning that all nodes in the tree represent items), *external* (meaning that non-leaf nodes are just used to direct the search, and the actual items that are represented by the tree are stored in the leaf nodes) or *partially external* (meaning that only some internal nodes represent items). The *height* of a search tree is the maximum number of pointers between the root of the tree and any leaf node. As the reader probably already knows, the idea behind search trees is to make it efficient to search for items. A search for an item only needs to traverse the pointers on the path to the desired item or a leaf (if the item is not present in the tree). Thus, the maximum number of pointers that a search for an item needs to traverse is close to $\log n$, if the tree has the minimum possible height. Note that, where log is used we are referring to the binary logarithm ($\log_2$) and n refers to the number of items in the data structure.

Non-balanced and Balanced Search Trees

Unfortunately, there is no guarantee that the height of a *non-balanced* search tree will be close to $\log n$. In a non-balanced search tree, certain sequences of operations can make the search tree's structure resemble a linked list, which gives non-balanced search trees a worst-case time complexity of $\mathcal{O}(n)$ for searches. To improve upon this worst-case complexity, several *self-balancing* (sometimes also called height-balanced) search trees have been proposed (e.g., [52, 74–76]).

The first self-balancing search tree was invented by Georgy Adelson-Velsky and Evgenii Landis and is therefore called the AVL tree [52]. In an AVL tree, all nodes contain information about their balance (i.e., the difference

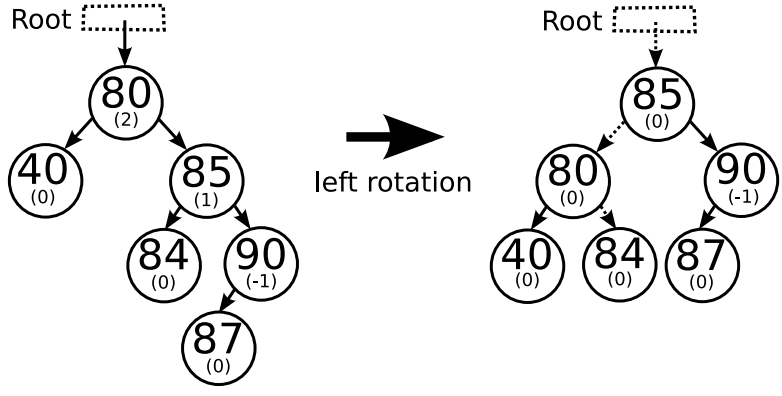


Figure 2.9. The figure illustrates a left rotation at the node storing the item 80. The nodes contain balance information in parenthesis. The balance information for a node is calculated by taking the height of the right subtree rooted in the node and subtracting it with the height of the left subtree rooted in the node.

in height between the right and left subtrees rooted in a node.). In Fig. 2.9, the balance information of the nodes is depicted in parenthesis. All operations of an AVL tree must maintain that the absolute value of the balance of a node never exceeds one. If the insertion or removal of a node results in a violation of the balance requirement described above, then the operation doing the insertion or removal does manipulations of the tree so that the tree satisfies the balance requirement when the operation returns. The manipulations that are done to balance the tree are called rotations. Figure 2.9 depicts one such rotation (called left rotation) that an INSERT operation does to make the tree satisfy the balance requirement after inserting a node with item 87. The LOOKUP, INSERT and REMOVE operations of AVL trees all have the worst-case time complexity $\mathcal{O}(\log n)$ [52]. The AVL tree is one of the main components of one of the CA tree implementations which is experimentally analyzed in Papers I and II.

The LFCA tree implementation that is experimentally analyzed in Paper III uses a type of binary search tree called treap [77] as one of its components[4]. A treap has an expected[5] worst-case time complexity of $\mathcal{O}(\log n)$ for the operations INSERT, REMOVE and LOOKUP. A treap maintains this

[4]We decided to use an immutable treap for the LFCA tree implementation instead of, for example, an immutable AVL tree due to the simplicity of the treap (which makes it easy to implement an immutable treap).

[5]That the expected worst-case time complexity of a data structure operation is X means that one can expect a running time of X on average independently of what the input to the operation is and what other operations have been applied to the data structure previously.

property by making sure that its structure is "random". This randomness is maintained by storing special numbers generated by a pseudo-random generator in every newly added node and making sure that the tree satisfies the treap property. The treap property is that each treap node N must be the root node of a binary search tree and have subtrees with nodes containing special numbers that are smaller or equal to the special number of N. The treap property can be maintained by using only left and right rotations[6] to "bubble" nodes up or down in the tree.

Joining and splitting Search Trees

To be efficiently implemented, the CA tree of Paper I and the LFCA tree of Paper III need an ordered set data structure with efficient support for the join and split operations. The split operation splits an ordered set so that the maximum item in one of the resulting sets is smaller than the minimum item in the other. Ideally, the split should also split the input data structure so that the resulting data structure instances contain approximately the same number of items. The input of the join operation is two instances of the data structure where the minimum item in one of them is greater than the maximum item in the other. The resulting ordered set contains the union of the items of the two input data structures.

The split and join operations can be implemented efficiently (i.e., with worst-case time complexity $\mathcal{O}(\log n)$) in both AVL trees [78, page 474] and treaps [77]. The split operation can be implemented in both of these data structures by taking the subtrees of the root and inserting the item of the root node into the right subtree. Thus, the split is as efficient as the tree's INSERT operation.

The treap's join operation is also simple to implement. An algorithm for the join operation for treaps first creates a dummy node with the treap containing the smaller items as its left child and the treap with the larger items as its right child and then removes the dummy node from the resulting tree with the treap's remove operation. The treap's remove operation will perform rotations to "bubble up" the dummy node to a leaf position before it is removed. These "bubble up" rotations will make the tree a valid treap again.

[6]The right rotation is symmetric to the left rotation which is exemplified in Fig. 2.9.

The join operation for AVL trees is slightly more complicated than the join for treaps. We will only provide a sketch of the algorithm here. A join operation for AVL trees starts by calculating the height of both input trees (this can be done efficiently by making use of the balance information stored in the nodes). Let us assume that the tree A containing the smaller items is the highest (the other case is symmetric). Then the tree B containing the larger items is first joined with the subtree C, where C has its root node along the right spine of A and has the same height as B. The trees C and B are combined by removing the smallest item from B and letting a new node containing the removed item link together C and B. The tree D resulting from joining C and B will have a height that is greater than C by one which may create a violation of the AVL tree balance property when D is replacing the subtree C in A. This balance violation can be fixed in the same way as when such violation occurs during an INSERT operation.

A detailed description of the join operation for AVL trees can be found in e.g., Knuth's book [78, page 474]. Seidel and Aragon have described the join operations for treaps in their paper [77]. Another popular balanced search tree is called the Red–Black tree. An efficient join operation for Red–Black trees can be found in Tarjan's book [79, page 52].

2.5.2 Immutable Search Trees

The LFCA tree implementation presented in Paper III makes use of an immutable treap as one of its main components. This section explains how one can implement efficient immutable versions of search trees. As already discussed in Section 1.1, immutable data structures are data structures that do not change (i.e., instances of the data structures do not change). The update operations of immutable data structures return new instances reflecting the update. Immutable data structures and their operations are sometimes called functional as they are often used in functional programming languages where the use of mutable data is discouraged [40]. Immutable versions of operations for balanced search trees are often asymptotically as efficient as their mutable counterparts [40, 77, 80]. Figure 2.8b illustrates how this is possible. The figure shows the new nodes (marked with red) that need to be created when creating a new search tree based on the search tree in Fig. 2.8a but that also contains the item 81. Only new "copies" of nodes along the path from the root to the added leaf need to

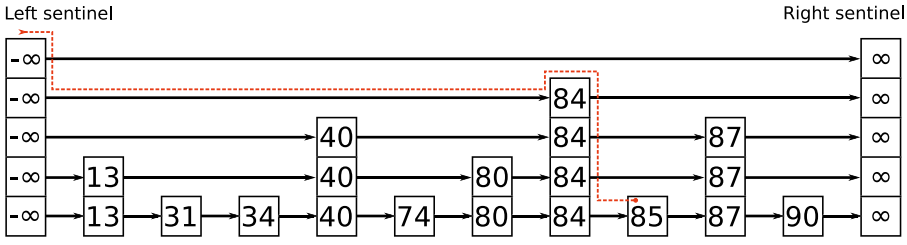


Figure 2.10. The structure of a skip list and, in red, the search path to the item 85.

be created. That is, only about $\log n$ nodes need to be created, where n is the number of nodes in the tree. The remaining nodes (the black nodes in Fig. 2.8b) are shared with the tree on which the new instance is based. Note that even though the worst-case time complexities under the big-O notation for operations in both immutable and mutable search trees are often the same, the immutable versions of update operations often induce more memory management related overheads and can thus be expected to be slightly slower than their mutable counterparts.

2.5.3 Skip lists

A skip list [81] is an alternative to search trees for implementing efficient ordered sets. Figure 2.10 illustrates the structure of a skip list. In the bottom layer, the nodes form a linked list containing all the items stored in the skip list. The other layers can be seen as shortcuts that make searching more efficient.

The red line in Fig. 2.10 illustrates how the search for the item 85 works. A search for the location of a specific item always starts from the left sentinel node as the current node and the top layer as the current layer. The search traverses down in the layers until the current node's pointer P in the current layer is pointing to a node with an item that is smaller than or equal to the item searched for. Then, P is traversed so the current node becomes the node that P is pointing to. This process is repeated until the item is found or until one can conclude that the item is not in the skip list.

The INSERT operation creates a new node. The number of layers that a new node will have is decided using a pseudo random generator so that the probability that a node has x layers is $(1/2)^x$. This way, the skip list will get an expected worst-case time complexity of $\mathcal{O}(\log n)$ for its INSERT,

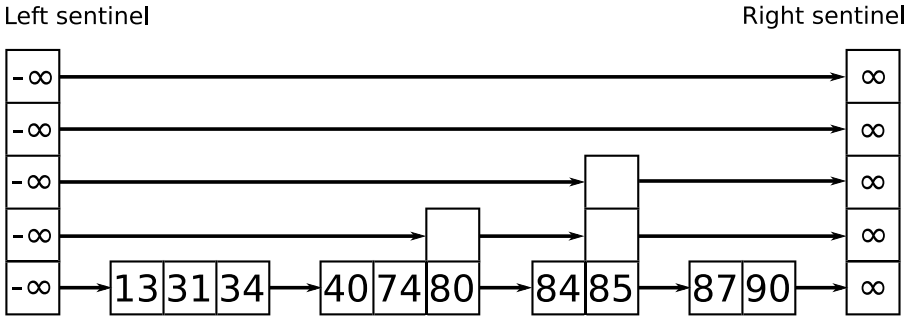


Figure 2.11. The structure of a skip list with fat nodes.

REMOVE and LOOKUP operations [81]. One can also derive efficient split and join operations for skip lists with the same worst-case time complexity. These operations are similar to splits and joins of linked lists, but they are more efficient as they can make use of the skip list's structure.

2.5.4 Skip Lists and External Binary Search Trees with Fat Nodes

The skip lists used in the CA tree implementations of Papers I and V and the treap implementation used by the LFCA tree presented in Paper III use an optimization that makes their operations more cache-friendly (i.e., the optimization makes the data structure operation induce fewer cache misses). Figure 2.11 illustrates this optimization. The figure depicts a version of the skip list that is shown in Fig. 2.10, but with the optimization applied. The optimization consists of putting up to k items in *fat* nodes instead of only storing a single item in every node. This optimization is referred to as list unrolling [82]. Something similar to list unrolling can also be applied to external binary search trees as they may store a range of items in the leaves instead of just a single item.

There are two important performance benefits that can be gained from list unrolling [82]:

Cache Locality List unrolling makes sure that ranges of items are stored close to each other in memory. This is especially beneficial for range queries and similar operations as they scan ranges of items. Let us say that four items can be stored in a cache line. In this case, the op-

timization can reduce the number of cache misses with up to a factor of four for range queries. This constitutes a substantial performance improvement, as a memory operation that induces a cache miss can be several orders of magnitude slower than one resulting in a cache hit [59, page B-3].

Memory Management List unrolling also reduces the number of nodes that are stored in the data structure. This, in turn, reduces the workload for the memory management (i.e., fewer nodes need to be allocated and freed), which may also have a positive impact of the performance of the data structure.

2.6 Memory Reclamation Techniques for Concurrent Data Structures

In the pseudocode presented in Papers I, III and V, data blocks that are removed from the data structures are not freed explicitly. For the pseudocode in these papers, we are assuming that there is some automatic memory reclamation system. Examples of such automatic memory reclamation systems are the garbage collection systems that exist for Java Virtual Machines and similar systems. In low-level programming languages that do not have built-in automatic memory reclamation (e.g., C and C++), it is more tricky to reclaim memory in concurrent data structures than in data structures that are only accessed sequentially. The reason for this is that the same data block may be used by several threads concurrently. A thread T_1 that removes a data block D from a concurrent data structure can usually not deallocate (free) the data block directly as another thread T_2 may be reading the data block concurrently. Several different techniques that can be used in low-level languages have been proposed for delaying reclamation of memory in concurrent data structures until it is safe (i.e., until one can be sure that no thread is reading the memory block to be reclaimed) [83–95]. The reader is recommended to have a look at the comparison article by Hart *et al.* [96] for a description of the techniques presented before 2007 (of which many of the newer algorithms are based).

The CA-PQ implementation that is evaluated in Paper V is implemented in C and uses the epoch based reclamation (EBR) scheme which was proposed in Fraser's Ph.D. thesis [85]. How EBR can be used to reclaim the memory

blocks which are no longer used in a concurrent data structure will be illustrated with a code example. The aim of this code example, which is shown in Fig. 2.12, is to give the reader an idea of what effort is required to use the data structures presented in Papers I, III and V in a low-level programming language. The code implements a concurrent array through a mechanism that is referred to as copy-on-write [97]. The interface of EBR has three functions. The functions `ebr_critical_enter` and `ebr_critical_exit` are used to indicate that a thread enters and exits a region where it accesses the concurrent data structure. The function `ebr_delayed_free` is used to tell the EBR system that a data block that cannot be freed directly (due to the possibility of concurrent readers) should be freed when the system detects that it is safe to do so.

2.7 Further Reading

Herlihy and Shavit's book about multiprocessor programming [43] is an excellent introduction to concurrent data structures and shared memory synchronization. Likewise, Scott's book about shared-memory synchronization [68], that has a slightly heavier focus on synchronization primitives like locks and barriers than on concurrent data structures, is also an excellent introduction to the subject.

Several textbooks describe the design of modern multicore chips and their surrounding components. One of them is the book about computer architecture by Hennessy and Patterson [59]. A more focused description of aspects of modern multicore processors that are relevant for designers and programmers of concurrent data structures can be found in McKenney's article [61]. McKenney's excellent online book [98] is also worth having a look at for those interested in multicore programming.

```
1  typedef struct {
2    int size;
3    long* array;
4  } cow_array;
5  /* Creates a new copy-on-write array */
6  cow_array* new_cow_array(int size) {
7    cow_array* a = malloc(sizeof(cow_array));
8    a->size = size;
9    a->array = malloc(sizeof(long) * size);
10   return a;
11 }
12 /* Sets the value at position pos to new_val */
13 void cowa_update(cow_array* a, int pos, long new_val) {
14   long* new = malloc(sizeof(long) * a->size);
15   ebr_critical_enter();
16   long* snapshot;
17   do {
18     snapshot = a->array; /* atomic */
19     for(int i = 0; i < a->size; i++) {
20       new[i] = snapshot[i];
21     }
22     new[pos] = new_val;
23   } while( ! CAS(&a->array, snapshot, new) );
24   ebr_delayed_free(snapshot);
25   ebr_critical_exit();
26 }
27 /* Reads the values located at indexes [start, end] */
28 void cowa_read_range(cow_array* a, int start, int end,
29                      void (*reader)(long)) {
30   ebr_critical_enter();
31   long* snapshot = a->array; /* atomic */
32   for(int i = start; i <= end; i++) {
33     reader(array_snapshot[i]);
34   }
35   ebr_critical_exit();
36 }
```

Figure 2.12. The code for a copy-on-write based concurrent array that illustrates how memory can be dealoocated in low-level programming languages with the help of epoch based memory reclamation. The operations required by the epoch based memory reclamation are underlined.

3. Contention Adapting Search Trees

This chapter aims to provide high-level descriptions of the lock-based contention adapting search tree (CA tree) described in Paper I and the lock-free contention adapting search tree (LFCA tree) described in Paper III. The chapter begins with two sections providing high-level views of the CA tree (Section 3.1) and LFCA tree (Section 3.2) followed by some highlights from the experimental results presented in Papers I, II and III (Section 3.3).

3.1 A High-Level View of Lock-based CA Trees[1]

As can be seen in Fig. 3.1, CA trees consist of three layers: one containing routing nodes, one containing base nodes and one containing sequential ordered set data structures. Essentially, the CA tree is a partially external search tree where the *routing nodes* are internal nodes whose sole purpose is to direct the search and the *base nodes* are where the actual items are stored. All keys stored under the left pointer of a routing node are smaller than the routing node's key and all keys stored under the right pointer are greater than or equal to the routing node's key. A routing node also has a lock and a valid flag but these are only used rarely when a routing node is deleted to adapt to low contention. The nodes with the invalidated valid flags to the left of the tree in Fig. 3.1 are the result of the deletion of the routing node with key 11; nodes marked as invalid are no longer part of the tree. A base node contains a statistics collecting (SC) lock, a valid flag and a sequential ordered set data structure. When a search in the CA tree ends up in a base node, the SC lock of that base node is acquired. This lock changes its statistics value during lock acquisition depending on whether the thread had to wait to get hold of the lock or not. The thread performing the search has to check the valid flag of the base node (retrying the operation if it is invalid) before it continues to search the sequential data structure inside

[1]Most of the text in this section is copied from the section titled "2. A Brief Overview of CA Trees" in Paper I [1].

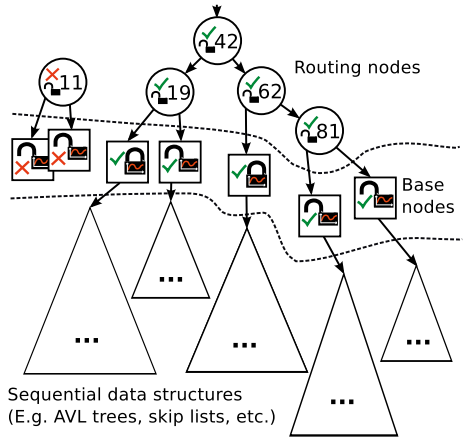
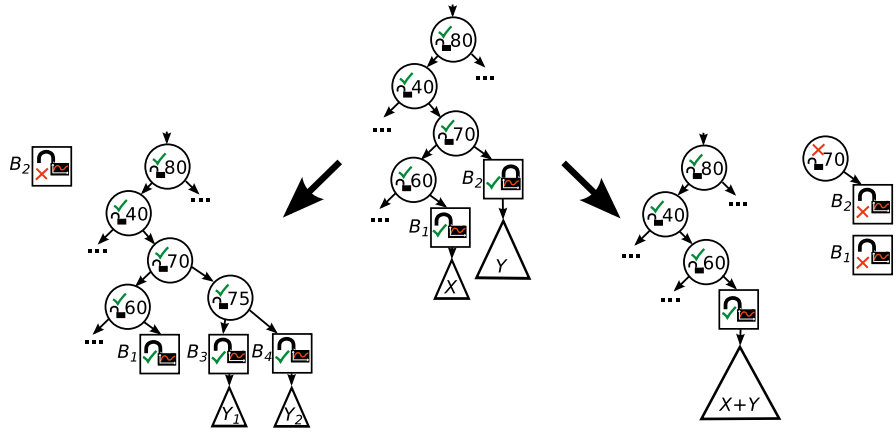


Figure 3.1. The structure of a CA tree. Numbers denote keys, a node whose flag is valid is marked with a green hook; an invalid one with a red cross.

the base node. The statistics counter in the SC lock is checked after an operation has been performed in the sequential data structure and before the lock is unlocked.

When the statistics collected by the SC lock indicate that the contention is higher than a certain threshold in a base node B_2, then the sequential data structure in B_2 is split into two new base nodes that are linked together by a new routing node that replaces B_2 (see Figs. 3.2a and 3.2b). In the other direction, if the statistics counter in some base node B_2 indicates that the contention is lower than a threshold, then B_2 is joined with a neighbor base node B_1 by creating a new base node B_3 containing the keys from both B_1 and B_2 to replace B_1 and by splicing out the parent routing node of B_2 (see Figs. 3.2b and 3.2c).

The CA tree's algorithm for range operations (i.e., operations that operate on all items in a given range) locks all base nodes that may contain items in the range. This locking is always done in a left-to-right order when the CA tree is depicted as in Fig. 3.1 (i.e., the algorithm is locking base nodes containing smaller items first). This strict locking order makes deadlocks impossible. Range operations that lock more than one base node decrease the statistics counters in the locks of the locked base nodes before they are unlocked. This manipulation of the statistics counters makes future joins of these base nodes more likely. The aim is to make future similar range queries cheaper by reducing the number of locks that they need to acquire. When a range operation only needs to acquire one lock, the statis-



(a) After a split *(b)* Initial *(c)* After a join

Figure 3.2. Effect of the split and join operations on the initial CA tree, shown in the middle subfigure.

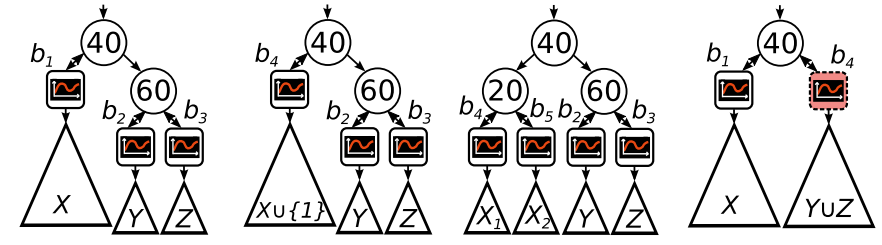
tics counter of this lock is changed in the same way as if the operation would have been a single-item operation (e.g., INSERT, REMOVE or LOOKUP).

3.2 A High-Level View of LFCA Trees[2]

Similarly to the lock-based CA tree, an LFCA tree consists of *route nodes* (round boxes in Fig. 3.3a) and *base nodes* (square shaped boxes in Fig. 3.3a). The route nodes form a binary search tree with the base nodes as leaves. The actual items that are stored in the set represented by an LFCA tree are located in immutable data structures rooted in the base nodes, called *leaf containers*. All operations use the binary search tree property of the route nodes to find the base node(s) whose leaf container(s) should contain the items involved in the operation if they exist.

An update operation (an INSERT or REMOVE) is illustrated by Figs. 3.3a and 3.3b. An update operation uses a compare-and-swap (`CAS`) to attempt to replace a base node b_1 with a new base node b_4 reflecting the update, until the update succeeds. If the first attempt by an update operation to replace a base node with the `CAS` succeeds, the new base node will have a smaller statistics value than the base node it is replacing. Otherwise, the

[2]Most of the text in this section is copied from the section titled "2 A BIRD'S EYE VIEW OF LFCA TREES" in Paper III [5].



(a) Initial *(b)* Update: insert *(c)* Adaptation: split *(d)* Adaptation: join

Figure 3.3. LFCA Trees illustrating various operations.

statistics value of the new base node will otherwise be smaller than the statistics value of the replaced base node.

Before an update operation returns it checks whether the statistics value stored in the updated base node indicates that a structural adaptation should happen. The first kind of adaptation, called *split*, is illustrated by Figs. 3.3a and 3.3c. A split aims at reducing the contention in the LFCA tree and replaces a base node b_1 with a route node linking together two new base nodes (b_4 and b_5) so that approximately half of the original items are in each of them. The second kind of adaptation, called *join*, is illustrated with Figs. 3.3a and 3.3d and aims at optimizing the structure of the LFCA tree for range queries that span multiple base nodes and for situations where the contention is low. A join splices out a base node b_2 and its parent and replaces the base node b_3 where the new location for the items in the spliced out base node is with a new base node b_4 containing the items of both b_2 and b_3. The colored base node in Fig. 3.3d reflects the fact that this base node has a different type than the rest of the base nodes in the figure as a side effect of the join. However, this join type is only of importance when the join is ongoing as it indicates to other operations that they might need to help the join. The joining of base nodes in LFCA trees is fairly complicated as it needs to "mark" several nodes to prevent undesirable modifications as well as letting other operations help in finishing the join to maintain the lock-freedom property.

The range query operation supported by LFCA trees is also complicated for similar reasons as why the join operation is complicated. The range query operation ensures its atomicity by replacing all base nodes that may contain items in the range with base nodes that cannot be replaced for a brief period of time. The lock-free progress guarantee is ensured by letting other operations help in doing these replacements and complete the range query.

Any thread that is helping in completing a range query can finish and linearize the range query once all relevant base nodes have been replaced. The linearization point for a range query is when the join of all leaf containers that may contain items in the range is written to a special storage location, which is unique for the join. After the linearization of a range query, all the base nodes that have been "marked" for the range query will immediately be replaceable again. Similarly to the range operations of the lock-based CA tree, range queries in LFCA trees that span more than one base nodes will try to reduce the number of base nodes that future similar range queries need to traverse by reducing the statistics values in the involved base nodes.

3.3 Highlights from the Experimental Results

Here, some highlights from the experimental evaluations presented in Papers I, II and III will be displayed. This section is only meant to be an appetizer for the more in-depth discussion of the results that are presented in the papers themselves. A lot of details about the experiments are, therefore, left out. The interested reader should be able to find all the relevant details about the experiments in the papers.

The Machines

The benchmark results presented in Paper I come from a machine with four AMD Opteron 6276 chips (each with 16 cores). The benchmark results presented in papers II and III come from a machine with four Intel(R) Xeon(R) E5-4650 CPUs (each with 8 cores and hyperthreading). Thus, both machines have 64 logical cores.

Mixed Operations Benchmark

Papers I, II and III include results from a benchmark that we call mixed operations benchmark. This benchmark measures throughput (operations/μs) of a mix of operations performed by N threads on the same data structure during t seconds. The keys and values for the operations LOOKUP, INSERT and REMOVE as well as the starting key for range operations are randomly generated from a range of size R. The data structure is pre-filled before

the start of each benchmark run by performing $R/2$ INSERT operations. In all experiments presented here $R = 1\,000\,000$, thus we create a data structure containing roughly 500 000 elements. In all captions, benchmark scenarios are described by strings of the form w:A% r:B% q:C%-R_1 u:D%-R_2, meaning that on the created data structure the benchmark performs $(A/2)$% INSERT, $(A/2)$% REMOVE, B% LOOKUP operations, C% range queries of maximum range size R_1, and D% range updates with maximum range size R_2. The size of each range operation is randomly generated between one and the maximum range size.

3.3.1 Some Results from Paper I

In the experimental evaluation presented in Paper I, two version of the CA tree are compared to recently proposed concurrent data structures for ordered sets. All data structures are implemented in Java. The first of the CA tree versions, called CA-AVL, uses an AVL tree as the sequential data structure component. The second of the CA tree versions, called CA-SL, uses a skip list [81] with fat nodes [82] as the sequential data structure component. The reader is referred to Paper I for an in-depth explanation of the recently proposed data structures that these two versions of the CA tree are compared against. Paper I presents results from three categories of scenarios. In the first category of scenarios, only single-item operations (INSERT, REMOVE and LOOKUP) are used. In this category, the CA trees scale similarly to the best of the other data structures in the comparison. This is exemplified in the results for the scenario with INSERT and REMOVE operations shown in Fig. 3.4a, where CA-AVL and a data structure called Snap [29] have the best performance of all data structures in the comparison.

In the second category of scenarios, range queries are also present. CA-SL (the CA tree that uses a skip list with fat nodes as the sequential data structure) is the best performing data structure in this category by a wide margin. Figure 3.4b exemplifies this. The excellent performance of CA-SL in this category can in part be explained by the fat cache friendly nodes of the skip list and the fact that the synchronization granularity of the CA trees is automatically adapted to the workload at hand, which gives them relatively low locking overhead.

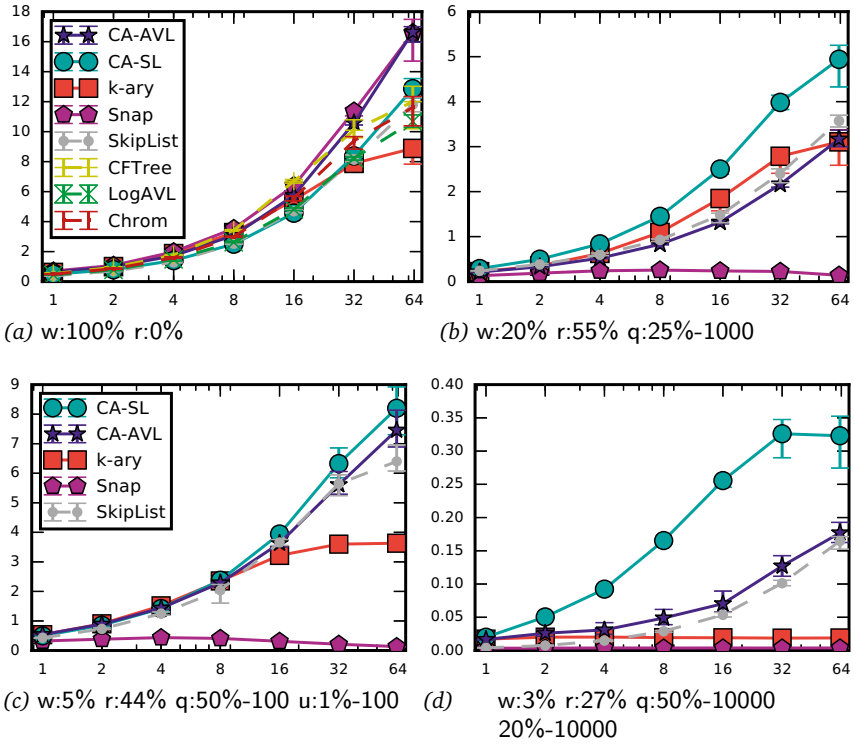
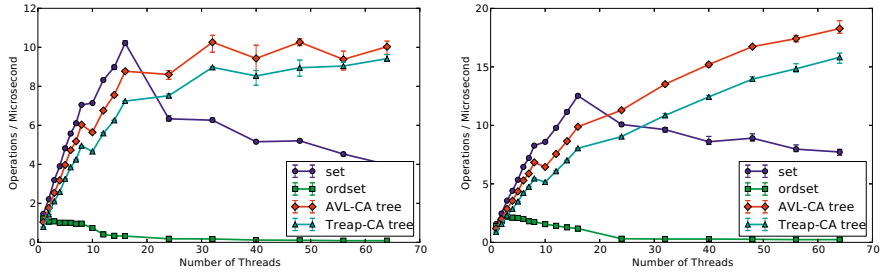


Figure 3.4. Throughput (ops/μs) on the y-axis and thread count on the x-axis. The graphs come from Paper I.

In the third and last category of scenarios presented in Paper I, range updates are also included. The SkipList (ConcurrentSkipListMap from the Java standard library) which is represented with dotted gray lines in the graphs is included in the results even though it only has non-atomic support for range queries and range updates to indicate the cost of atomicity. None of the other data structures that the CA trees are compared to in Paper I have support for range updates out of the box. Furthermore, it would not be trivial to implement scalable support for linearizable range updates in these data structures. Therefore, range updates in these data structures were implemented through a readers-writer (RW) lock[3]. All operations in these data structures except the range updates access the data structure in "read-only" critical sections (and can thus execute in parallel) while range

[3]The RW lock that was used has very good scalability for read-only critical sections as it uses a read indicator which is distributed over as many fields as there are logical cores and every read critical section only need to write to one of these fields (which is specific to the core that the thread is executing on).



(a) w:50% r:50% *(b)* w:10% r:90%

Figure 3.5. Scalability of the CA tree variants in two scenarios compared to `ordset` and `set` with concurrency options activated. The graphs come from Paper II.

updates need to access the data structure in mutual exclusive critical sections. In the scenarios with range updates, the CA trees are substantially more scalable than all other data structures with support for linearizable range updates. Figures 3.4c and 3.4d illustrate this.

3.3.2 Some Results from Paper II

In Paper II, two prototypes for an Erlang Term Storage (ETS) backed by CA trees are experimentally compared to the currently available ETS table types. As mentioned earlier, ETS is an in-memory key-value store that is integrated into the Erlang/OTP implementation of the Erlang programming language [51]. ETS is implemented using the C programming language.

The two CA tree based prototypes differ in the sequential data structure components that they use. One of them is based on the same data structure that is backing the current implementation of the `ordered_set` table type, namely an AVL tree [52].

The `ordered_set` (`ordset` for short) ETS table type is implemented with the above mentioned AVL tree protected by a readers-writer lock. The other ETS table type is called `set` and is implemented by a linear hash table [99] protected by 64 bucket locks. The `ordset` and `set` table types also differ in that the `ordset` guarantees that traversals of key-value pairs will happen in the order of the keys, while the `set` table type does not provide such assurance. [4]

[4]For an extensive description of the current ETS implementation we refer the reader to [8].

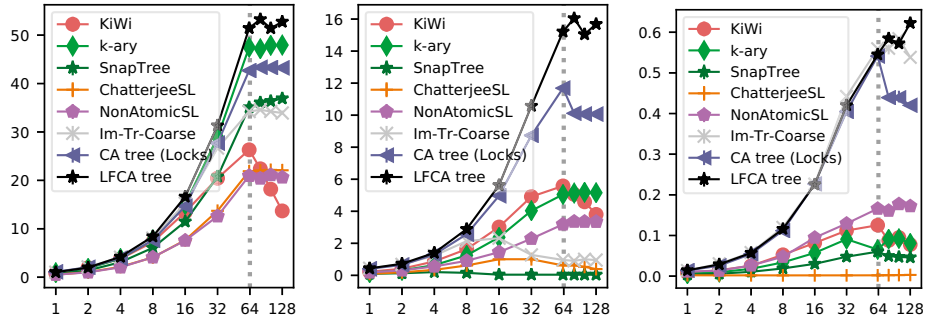


Figure 3.6. Throughput (operations/μs on the y-axis) when varying the thread count (on the x-axis). The graphs in this figure Paper III.

Unsurprisingly, the results included in Paper II show that the CA tree based prototypes are much more scalable than the current `ordset` implementation. A little bit more surprising is that, in some scenarios, the CA tree prototypes perform even better than the `set` table type, which can be explained by the global counters that the `set` table type have to e.g. keep track of the current number of buckets. The results also show that the CA tree prototypes have similar sequential performance compared to the current `ordset` implementation. Figure 3.5 shows some of the results that are presented in Paper II.

3.3.3 Some Results from Paper III

Paper III, which is presenting the lock-free contention adapting search tree (LFCA tree), shows results from experiments that compare the LFCA tree to related data structures using the mixed operations benchmark, as well as a benchmark suggested by the authors of the KiWi data structure [100] that is more suitable for measuring the data structures' ability to handle large range queries under high contention from update operations. All data structures in the comparison are implemented in Java.

Over the wide range of experiments included in Paper III and its appendix, the LFCA tree is the most performant and scalable data structure even though the lock-based CA tree (i.e., a version of the lock-based CA tree that has also been optimized to exploit immutable data) quite closely follows the LFCA tree in several scenarios. The good results for the LFCA

tree can in part be explained by its short conflict times for range queries and in part due to its wait-free LOOKUP operation. Examples of the results from Paper III can be found in Fig. 3.6. Figure 3.6a shows that the LFCA tree can provide outstanding performance even in scenarios without range queries. Figures 3.6b and 3.6c show that LFCA tree can deliver excellent performance both with range queries of maximum size 1000 and with much more extensive range queries of maximum size 100000. LFCA trees' ability to provide excellent performance in a wide range of scenarios without any need to fine-tune parameters can be explained by their ability to automatically change their synchronization granularity to fit the workload at hand. Paper III presents statistics for the number of route nodes in different scenarios that support this claim.

4. The Contention Avoiding Concurrent Priority Queue

Traditionally, a concurrent data structure gets its semantics from the corresponding sequential data structure [101]. That is, an operation in a concurrent data structure appears to happen atomically at some point between the invocation and return of the operation and has the same effect as its sequential variant. However, such *strict* semantics sometimes inherently limits the scalability of concurrent data structures. This has motivated so-called *relaxed* concurrent data structures that have more relaxed semantics than traditional ones (e.g., [45–47,101,102]). For example, in a concurrent priority queue with strict semantics, the DELMIN operation always deletes and returns an item that was the smallest at some point between the invocation of the operation and its return. This semantics inherently limits the scalability of a concurrent priority queue as there may only be one smallest item in the queue and only one of the potentially many parallel DELMIN operations can delete this item. Several publications (e.g., [45–47, 103]) have suggested concurrent priority queues with different types of relaxed semantics. Relaxed semantics in the context of concurrent priority queues means that the DELMIN operation can delete and return an item that has been "reasonable close" to the smallest item instead of always deleting and returning an item that was the smallest at some point between the invocation and return of the operation. These relaxed concurrent priority queues can provide better scalability than traditional ones at the cost of a more imprecise DELMIN operation.

Paper V describes the contention avoiding concurrent priority queue (called CA-PQ). As far as we are aware, the CA-PQ is the first concurrent data structure to automatically turn on and off relaxed semantics depending on the level of contention that it detects. To only have the relaxed semantics turned on under contention makes sense as the relaxed semantics is only motivated in contended scenarios and may cause more harm than good when the contention is low. For example, in a parallel version of Dijkstra's

single source shortest path algorithm, a too imprecise DELMIN operation cause a lot of wasted work which damage performance; cf. Paper V.

Section 4.1 contains a high-level description of the CA-PQ. Section 4.2 presents some highlights from the experimental results presented in Paper V.

4.1 A High-Level View of the CA-PQ[1]

As illustrated in Figure 4.1, the CA-PQ has a global component and thread local components. When a CA-PQ is uncontended it functions as a strict concurrent priority queue. This means that the DELMIN operation removes the smallest item from the global priority queue and the INSERT operation inserts an item into the global priority queue.

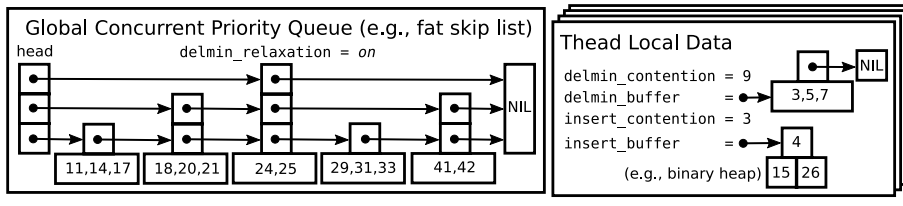
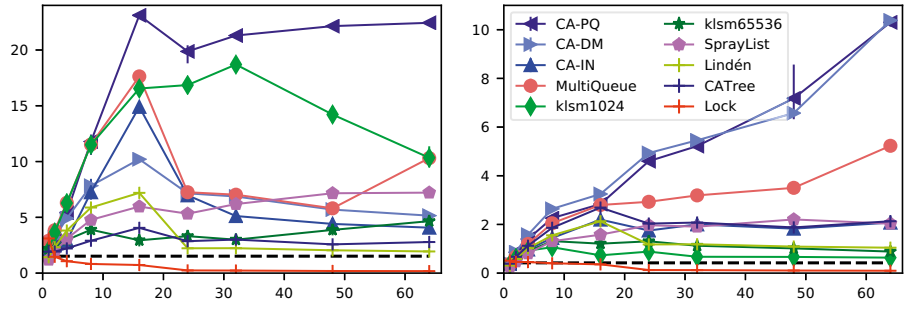


Figure 4.1. The structure of a CA-PQ.

Accesses to the global priority queue detect whether there is contention during these accesses. The counters delmin_contention and insert_contention are modified based on detected contention so that the frequency of contention during recent calls can be estimated. If DELMIN operations are frequently contended, contention avoidance for DELMIN operations is activated. If a thread's delmin_buffer and insert_buffer are empty and DELMIN contention avoidance is turned on, then the DELMIN operation will grab up to k smallest items from the head of the global priority queue and place them in the thread's delmin_buffer. Grabbing a number of items from the head of the global priority queue can be done efficiently if the queue is implemented with a "fat" skip list that can store multiple items per node; see Figure 4.1. Thus, activating contention avoidance for DELMIN operations reduces the contention on the head of the global priority queue by reducing the number of accesses by up to $k-1$ per k DELMIN operations.

[1]Most of the text in this section is copied from the section titled "2. A Brief Overview of the Contention Avoiding Priority Queue" in Paper V [7].



(a) RoadNet [0,1000] *(b)* LiveJournal [0,1000]

Figure 4.2. Graphs showing results from the SSSP experiment. Throughput (# nodes in graph ÷ execution time (μs)) on the y-axis and number of threads on the x-axis. The black dashed line is the performance of sequential Dijkstra's algorithm with a Fibonacci Heap. The graphs are taken from Paper V.

Contention avoidance for INSERT operations is activated for a particular thread when contention during INSERT operations is frequent for that thread. The INSERT contention avoidance reduces the number of inserts to the global priority queue by buffering items from a bounded number of consecutive INSERT operations in the insert_buffer. When at least one of the delmin_buffer and insert_buffer is non-empty, the DELMIN operation takes the smallest item from these buffers and returns it.

4.2 Highlights from the Experimental Results

Paper V includes an experimental evaluation that compares the performance and scalability of CA-PQ to related concurrent priority queues. Three concurrent priority queues that are derived from CA-PQ are also included in the comparison to highlight the effect of the different relaxations on the performance. The concurrent priority queue named CATree has both the relaxations of CA-PQ turned off and is a linearizable concurrent priority queue with traditional semantics. The concurrent priority queue named CA-DM has only the relaxation of DELMIN turned on. The concurrent priority queue named CA-IN only has the relaxation for the INSERT operation turned on.

For the experimental comparison, a parallel version of Dijkstra's single source shortest paths (SSSP) algorithm is used. Using a realistic parallel algorithm

for comparing concurrent priority queues has several advantages compared to using a synthetic micro-benchmark. First of all, with a synthetic micro-benchmark, there is always a risk that the benchmark is measuring scenarios that are unrealistic for real-world applications. Perhaps even more importantly, comparing relaxed concurrent priority queues with a micro-benchmark in a fair way is difficult because one also has to take the precision of the DELMIN operation into account. In the parallel SSSP benchmark, the precision of the DELMIN operation is naturally reflected in the results because a more imprecise DELMIN operation means that the parallel SSSP benchmark will do more wasted work that should make the running time of the algorithm longer.

Examples from the results presented in Paper V can be seen in Fig. 4.2. Figure 4.2a shows performance resulting from the different concurrent priority queues when the SSSP benchmark is applied to a road network. Figure 4.2b shows the same kind of results but for a large social media network. CA-PQ is the top performing concurrent priority in both scenarios. That CA-DM is as good as CA-PQ in the scenario displayed in Fig. 4.2b illustrates that both of CA-PQ's relaxations are not always necessary.

Similarly to the CA tree and LFCA tree, CA-PQ is able to perform well in many scenarios due to its ability to change dynamically to fit the workload at hand based on usage statistics. That is, the CA-PQ can turn off and on the two relaxations depending on how much contention is detected. Another factor that contributes to CA-PQ's excellent scalability is that it needs to access shared memory to a lesser extent than the competing data structures. Paper V contains cache miss statistics (collected by hardware counters) that support this claim.

5. Additional Discussion of Related Work

Papers I to V contain the main discussions of related work. This chapter complements these related work discussions. Section 5.1 provides a more extensive discussion of research related to contention adaptation than the papers. Section 5.2 discusses work on concurrent ordered sets with support for range queries which has appeared after the submissions of the papers included in this dissertation.

5.1 Contention Adapting Data Structures

Here, we discuss contention adapting data structures. Such data structures change themselves according to the level of contention that they detect. We start with a discussion of reactive diffracting trees followed by a discussion of data structures that switch between states optimized for different levels of contention. We end with a short note about distantly related work on adaptation to contention.

Reactive Diffracting Trees

Reactive diffracting trees [104, 105] are tree-shaped data structures designed for the implementation of shared counters and load balancing that change their structure depending on how much contention they detect. They are from one point of view similar to the CA trees (Papers I and III), but they are also entirely different from the CA trees from another point of view. A reactive diffracting tree and a CA tree are similar as they are both tree-shaped data structures that adapt their structures according to detected contention. However, they are also very different since they apply to entirely different applications. Diffracting trees are applicable for applications such as shared counters and load balancing [104–106] while the CA trees are applicable for applications such as ordered key-value stores, sets and priority queues. Furthermore, the implementation of the contention

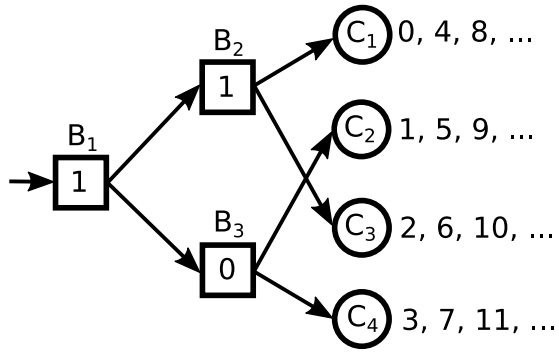


Figure 5.1. Illustration of a diffracting tree.

adaptation mechanisms in CA trees and the reactive diffracting trees have substantial differences. For the reader to better understand the similarities and differences between CA trees and reactive diffracting trees, we first describe how diffracting trees work and then give an overview of the two published reactive diffracting tree variants (i.e., [104, 105]).

We illustrate how reactive diffracting trees work with the shared counter application, which is their main application described in the reactive diffracting tree publications [104, 105]). We first note that a shared counter represents an integer variable that can be accessed and modified with the FETCHANDADD operation. The FETCHANDADD operation increments the counter and returns the value that the counter had before the increment. Figure 5.1 illustrates the structure of a diffracting tree. The tree consists of balancing nodes (the internal nodes) and counting nodes (the leaf nodes). A balancing node contains a toggle bit that can have the values 0 or 1. The counting nodes are protected by locks and contain counter variables that are incremented according to their position in the tree (see Fig. 5.1). A FETCHANDADD operation in a diffracting tree (1) starts at the root balancing node N of the tree, (2) atomically flips the toggle bit of N, (3) goes to the left branch of N if the toggle bit had the value 0 before the flip or goes to the right branch otherwise, (4) continues to traverse the balancing nodes in the same way as the root balancing node is traversed until a counting node C is reached, (5) locks C and, (6) sets the counter variable in C to its next value. This way, a diffracting tree can provide the functionality of a shared counter without relying on a centralized counter variable. Unfortunately, all FETCHANDADD operations still need to toggle the switch in the root node, so the root becomes a scalability bottleneck if a naive shared

variable implementation for the toggle bit is used. Fortunately, one can make use of an elimination array [106] to implement the toggle bit which avoids this bottleneck. The elimination array works by letting threads randomly select array slots where they can pair up with another thread. This way, pairs of threads can agree on that one should go left, and the other should go right. One such pairing between two threads is equivalent to two flips of the toggle bit that thus can be eliminated.

The original diffracting tree had a static height [106]. Obviously, this is not optimal when the contention level in a diffracting tree is unknown or changes dynamically. To solve this issue, Della-Libera and Shavit came up with the first reactive diffracting tree (F-RDT) [104]. We refer to Della-Libera and Shavit's paper [104] for a detailed description of F-RDT and will here mention some interesting differences between the F-RDT and the CA trees. An F-RDT allocates all its nodes at initialization to avoid the cost of memory management (diffracting trees typically have very few nodes [104–106]). An F-RDT node can be in the states `Balancer`, `Counter`, and `Off`, where `Balancer` and `Counter` have the apparent meanings and, `Off` means that a node is currently not used in the tree. In an F-RDT, only pairs of counter nodes that share the same parent can be joined (the joined counter nodes are replaced with their parent that becomes a counter node). The F-RDT uses timing to estimate contention in the counter nodes. That is, the average time for the latest counter node accesses is calculated and when its value exceeds or falls below some thresholds a split or join is triggered. Suitable values for such timing thresholds rely heavily on the system at hand and thus need to be tuned for every new system.

The need for fine tuning the parameters of F-RDT motivated Ha *et al.* to develop the self-tuning reactive diffracting tree (ST-RDT) [105]. Ha *et al.* claim that ST-RDT can react faster to changes in contention than F-RDT and that ST-RDT does not have any parameters that need to be fine-tuned to fit the system at hand. Furthermore, in contrast to F-RDT and the CA trees that all have adaptations that split a leaf into two or join two leaves into one, the high contention adaptation of ST-RDT can expand a leaf into a subtree of a size that depends on the estimated contention in the tree, and its low-contention adaption can collapse a subtree with more than two leaf nodes if the estimated contention levels indicate that this is appropriate. The adaptation strategy of ST-RDT is developed with the help of theory about online algorithms. In particular, ST-RDT's adaptation strategy is inspired

by the threat-based policy described by El-Yaniv *et al.* [107]. An ST-RDT operation increments a contention counter variable in a leaf node when it starts to access the leaf node (i.e., before the operation acquires the counter node lock) and decrements the same counter when it stops to access the leaf node (i.e., when the operation has released the counter node lock). The contention estimate for a leaf node in an ST-RDT is based on the value of the leaf's contention counter (i.e., the number of threads that are accessing or are trying to access the leaf). An ST-RDT operation estimates the optimal height of the tree based on the contention in the leaf that the operation is accessing and when this height differs from the height of the leaf node, an attempt to either expand or shrink the leaf node according to the calculated optimal height is made. The adaptation strategy of ST-RDT is not easily transferable to the CA trees: First of all, the adaptation strategy of ST-RDT assumes that the contention is evenly distributed over the leaf nodes, which may not be the case for CA trees. Secondly, a model for the optimal height of a CA tree should also take the different operations into account (e.g., a range query may benefit from a more shallow tree than a single-item operation). Additionally, such a model should also take into account that the route nodes of a CA tree are not equivalent to the balancing nodes of diffracting trees as the route nodes are used to reduce both the contention and the search space.

Techniques that Switch Between Coarse-Grained and Fine-Grained Synchronization

Österlund and Löwe [108] have described a method that dynamically transforms a concurrent data structure between different structures depending on how much contention is detected in the data structure. They suggest that a concurrent data structure should be represented by a sequential data structure protected by a single lock when it is uncontended and that it should be represented by a data structure with fine-grained synchronization when it is contended. Additionally, they propose an algorithm that can transform a data structure between a state where the data is stored in a sequential data structure protected by a single lock and a state where the data is stored in a lock-free data structure using fine-grained synchronization. This algorithm is not specialized for sets and can be used for other data structures as well; e.g., lists and queues. Österlund and Löwe's motivation for such transformations is similar to the primary motivation for

the CA trees. That is, it is difficult to predict the contention in a data structure so it would be advantageous if the data structure can optimize itself to contention or lack of contention.

Motivated by similar reasoning, Newton *et al.* [109] have proposed a one-way adaptation that transforms an immutable data structure referenced by a mutable reference[1] to a lock-free data structure with fine-grained synchronization. This adaptation is triggered when a high level of contention is detected. Chen *et al.* [110] later extended the work by Newton *et al.* so that adaptation can happen in both directions.

In comparison to the adaptations of CA trees, the transformations between coarse-grained synchronization and fine-grained synchronization proposed by Österlund and Löwe [108], Newton *et al.* [109] and Chen *et al.* [110] are very expensive and slow as the transformations require a scan of all the items in the source data structure, and the reconstruction of the new data structure variant from scratch. In contrast, the CA trees can smoothly transition between different levels of synchronization granularity.

Other Types of Adaptation to Contention

Publications discussing adaption to contention by dynamically changing algorithms and parameters in, for example, locks, combining funnels, and database systems also exist (see, for example, [111–115]). These publications will not be discussed further in this dissertation as we consider them only to be distantly related to the papers included in this dissertation.

5.2 Recent Work on Concurrent Sets with Range Query Support

Here, work related to range queries in concurrent data structures that has appeared after the submissions of Papers I to V is discussed. Note that discussions of earlier such work can be found in Paper I and Paper III.

Arbel-Raviv and Brown [116] recently proposed a general method for extending concurrent data structures with linearizable range query support.

[1] A concurrent data structure that is constructed from a mutable reference pointing to an immutable reference is explained in Section 1.1.

As reported [116], their new method appears to perform substantially better than the general method for linearizable iterators proposed by Petrank and Timnat [117] and the general method for range queries proposed by Chatterjee [118]. With this method, range queries increment a global timestamp variable. Update operations write down the timestamps in the relevant nodes when nodes are inserted and removed. Range queries traverse the data structure to collect nodes with items in the range. Range queries also traverse nodes that have been removed during such a traversal (such nodes can be found in the epoch-based memory reclamation system that the method relies upon [90]). Range queries figure out which of the traversed nodes are relevant for the results based on the timestamps in the nodes. Unfortunately, the global timestamp counter is bound to become a scalability bottleneck once parallel range queries become frequent enough. Furthermore, the global timestamp counter induces an overhead for all update operations (especially when the global timestamp counter is frequently modified) as all update operations have to read this counter. In contrast, the CA trees do not have such a global scalability bottleneck and do not rely on being able to access the internals of the memory reclamation system.

In a recent unpublished technical report, Archita *et al.* [119] describe an extension to the method by Petrank and Timnat [117] for creating linearizable iterators that make the method applicable to more data structures. The results provided by the report [119] indicate that the extended method induces similar performance overhead as the original method.

Also, in a very recent unpublished technical report [120], Fatourou and Ruppert describe an extension to a non-blocking binary search tree [34] that gives it support for linearizable range queries. This extension is similar to the method by Arbel-Raviv and Brown [116] in that range queries need to increment a global counter. On the other hand, the extension differs from the method by Arbel-Raviv and Brown in that it does not piggyback on the used memory reclamation technique but instead lets update operations save links to nodes that have got spliced out from the tree in the nodes that replace the spliced out nodes.

6. Artifacts

The research that has been carried out for this dissertation has resulted in some publicly available software artifacts. These artifacts may be useful for researchers and engineers that find this dissertation exciting and are therefore described in the following sections.

6.1 Java Data Structure Benchmark

When the work on the paper [2] that eventually lead to Papers I started, there was no established framework for benchmarking concurrent data structures written in Java. Therefore, I decided to write a new benchmark framework, called JavaRQBench. This benchmark framework has been used to obtain the results presented in Papers I and III as well as the results presented in some related publications [2, 3, 11]. Older versions of JavaRQBench as well as instructions for reproducing the results presented in the papers listed above can be found through the following website:

`http://www.it.uu.se/research/group/languages/software/ca_tree`

The latest version of JavaRQBench has also been published in a GIT repository on GitHub, to make it easier to receive external contributions to the benchmark framework:

`http://github.com/kjellwinblad/JavaRQBench`

Since my work on the first CA tree paper started, another benchmark framework called Synchrobench [121] has become popular for research related to concurrent data structures (e.g., [33, 100, 122]). JavaRQBench primary focus is on benchmarking data structures for ordered sets (i.e., skiplists and search trees) while Synchrobench is much broader in scope, as it also includes list and hash-based data structures. Functionality wise, JavaRQBench and Synchrobench are similar, but they both have special features

that the other one is lacking. For example, JavaRQBench includes correctness tests for concurrent ordered sets and has support for measuring the performance of workloads that include range updates (i.e., an operation that updates the values associated with the items in a range). These are features that Synchrobench does not have.

The first Java CA tree implementation[1] that was used to obtain the results in the paper "Contention Adapting Search Trees" [2] has been integrated into Synchrobench to make it more convenient for users of Synchrobench to compare data structures with that version of the CA tree.

6.2 CA Tree Implementations

Multiple variants of the lock-based CA tree and the LFCA tree are integrated into JavaRQBench (see the previous section). The ConcurrentSkipListMap is a concurrent ordered set/map implementation provided by the Java standard library. The ConcurrentSkipListMap is probably the most used concurrent ordered set for Java at the moment[2]. As the CA trees have been shown to perform substantially better than ConcurrentSkipListMap [1–3, 5, 11], it would be of great value if one could use a CA tree implementation as a drop in replacement for ConcurrentSkipListMap. The primary goal of the CA tree implementations integrated into JavaRQBench is to investigate the performance of the data structures. Therefore, these implementations do not yet provide an interface which is fully compatible with the interface of ConcurrentSkipListMap. Work has been started on a CA tree implementation that is compatible with the interface of ConcurrentSkipListMap. This work is published here:

`https://github.com/kjellwinblad/ca_trees_java`

CA Tree Models for Concurrency Testing

The pseudo-code included in Paper I has been automatically extracted from Java code written to systematically test the algorithms for concurrency errors and linearizability using Java Pathfinder [123]. Java Pathfinder can

[1] This first Java version of the CA tree only has support for single-item operations.
[2] A search for ConcurrentSkipListMap on `http://github.com` gave 59K code hits (2018-04-28).

systematically explore all interesting interleavings of threads of concurrent programs to find possible bugs using a search technique called dynamic partial-order reduction [124]. The code from which the pseudo-code of Paper I has been extracted as well as instructions for how to test this code with the help of Java Pathfinder can be found through the following URL:

`http://www.it.uu.se/research/group/languages/software/ca_tree`

Similarly, the pseudo-code included in Paper III has been automatically extracted from C code that has been developed to test the algorithm for concurrency errors using Nidhugg [125]. Nidhugg is a concurrency testing tool, which similarly to Java Pathfinder can explore all interesting interleavings of a program in a systematic way. The code, which the pseudo-code of Paper III is derived from, has also been stress tested in many concurrent scenarios to find possible concurrency errors and violations of linearizability. The above mentioned C code, as well as its tests, are also available through the URL above.

6.3 Erlang Term Storage Benchmark

Paper II and an earlier publication [8] about the scalability of Erlang Term Storage (ETS) contain experimental results from a micro benchmark measuring the scalability of ETS under various ETS configurations and distributions of operations. As already mentioned in Section 1.2.2, ETS is an in-memory database or key-value store integrated into the Erlang/OTP system. This ETS benchmark which is written by me together with David Klaftenegger has been integrated into BenchErl [126], which is a scalability benchmark suit for Erlang. BenchErl is available here:

`https://github.com/softlab-ntua/bencherl`

6.4 CA-PQ Implementation and Parallel SSSP Benchmark

The developers of the k-LSM priority queue [46] have made a benchmark framework for concurrent priority queues publicly available [127], called

the k-LSM benchmark suite from here on. The k-LSM benchmark suite contains configurable microbenchmarks as well as implementations of recently published concurrent priority queues. The contention avoiding concurrent priority queue (CA-PQ) that is described in Paper V has been integrated into the k-LSM benchmark suite by me. Paper V contains scalability results for a parallel version of Dijkstra's single source shortest path (SSSP) using different concurrent priority queues. This SSSP benchmark has also been contributed to the k-LSM benchmark suite by the author. The k-LSM benchmark suite is available here:

`https://github.com/klsmpq/klsm`

6.5 `qd_lock_lib`: A Portable Locking Library for Delegation Locking Written in C

As has already been explained in Section 1.2.4, Paper IV describes the interface of two locking libraries, one of them written in the C++ programming language and one written in the C programming language. I am the primary author for the C library, called qd_lock_lib, while my coauthor David Klaftenegger is the principal author of the C++ library.

The purpose of `qd_lock_lib` is to make it easy to use delegation locks in applications written in C and other compatible languages. The lock implementation in `qd_lock_lib` is utilizing the atomics library introduced in the C11 standard for the C programming language. This makes the library portable (i.e., that it compiles with C compilers from different vendors on several operating system and processor architectures.). As of writing, the library makes it possible to chose from the delegation locking algorithms QD lock [9], HQD lock [9] and CCSynch [48] as well as several traditional locking algorithms. `qd_lock_lib` is publicly available at the following URL:

`https://github.com/kjellwinblad/qd_lock_lib`

7. Future Work

In this chapter, some interesting future work directions will be outlined. We will start to discuss some possible improvements to the CA trees (Paper I and Paper III) that would be interesting to investigate and then examine some possible future work directions related to the other papers. In this chapter, we use the term CA tree as a general term to refer to both the lock-based CA tree (Paper I) and the LFCA tree (Paper III) as the issues that are discussed concern both of these data structures in the same ways.

Balancing of the Route Nodes in the CA Trees

The part of a CA tree that is made of route nodes is not necessarily balanced (i.e., the height of two subtrees may differ significantly). One solution to the inefficacy that may come from such imbalance is to limit the depth of the part of the CA tree that is made of route nodes. Paper I discusses this in detail. Another potential solution to the imbalance problem, which does not limit how fine-grained the synchronization granularity of a CA tree can be, is to perform tree rotations to preserve balance when route nodes are removed or inserted similarly to how this is done in concurrent AVL trees (e.g., [29, 128, 129]). Unfortunately, such a balanced CA tree would at least require a different algorithm for range operations as the current algorithms for range operations (presented in Paper I and Paper III) depend on that the only change that can happen to the path from the root to a "locked" and "validated" base node is that a route node get spliced out from the path.

A more promising idea for maintaining balance in the routing layer and still allow for range operations is to use a balanced concurrent search tree similar to the one proposed by Drachsler *et al.* [33] for the routing layer. The balanced concurrent search tree proposed by Drachsler *et al.* explicitly maintains ordering information in the tree (i.e., all nodes contain pointers to their predecessor and successor in the item ordering). With such a structure, range queries and similar operations could traverse the items in their

ranges using the successor pointers despite the fact that balancing rotations could happen concurrently.

Yet another interesting alternative is to use a concurrent skip list for the routing layer. Using a skip list for the routing layer would make it easy to maintain balance and at the same time support range queries. The reason being that the bottom layer of concurrent skip lists [85, 130] always form a sorted list containing the items present in the represented set.

Unnecessary Route Nodes and Base Nodes in the CA Trees

A potential performance issue with the algorithms presented in Paper I and Paper III is that adaptations can only happen in the part of the tree that operations are accessing. Consider a scenario where one region of the tree, containing base nodes and route nodes, is never accessed anymore. Then the base nodes and route nodes in this region take up unnecessary much space (as they could just as well be reduced to a single base node) and may still stay in the CA tree indefinitely even though they are clearly not needed.

A potential solution to this performance problem that would be interesting to investigate is that all operations issue a low-contention join at a random base node with a certain probability. Let us say that one in every 1000 operations issues such a join. Then the average cost per operation would be relatively low and when such joins are not appropriate they will quickly be undone by high-contention splits. Still, such joins would eventually remove all nodes that are not needed to reduce contention.

Integrating the Lock-based CA Tree as an ETS Back-end

The CA tree based prototype for an Erlang Term Storage (ETS) table back-end described in Paper II could be extended so it can be included for general use in the main Erlang/OTP distribution. With such an implementation in-place, it would be interesting to investigate the implications on real world Erlang applications.

Experimenting with Delegation Locking in Large Applications Using the Locking Libraries Presented in Paper IV

The locking libraries presented in Paper IV can be used to integrate delegation locking in real world applications and to investigate the perfor-

mance implications of such integrations. To make this work easier Robert Markovski has started the work on a tool for automatically translating code that uses traditional locking into using delegation locking during his bachelor thesis project [131][1]. An interesting future work direction would be to extend Robert's tool so that it can support the translation of large applications and then experiment with the performance implications of such translations.

CA-PQ in More Applications

Paper V presents the contention avoiding concurrent priority queue (CA-PQ) and shows that CA-PQ can provide outstanding performance and scalability when used in a parallel version of Dijkstra's single source shortest paths algorithm. It would be interesting to investigate if the CA-PQ also works well in other types of applications, such as concurrent scheduling and parallel discrete event simulations.

[1] I was one of the advisers for this bachelor thesis project.

8. Conclusion

This dissertation proposes novel concurrent data structures that dynamically adapt their structure, synchronization granularity or behavior based on detected contention and the type of operations that are accessing the data structures.

Paper I describes and evaluates the lock-based contention adapting search tree (CA tree). The CA tree adapts its synchronization granularity to fit the contention level and the number of items that are accessed by multi-item operations (e.g., range queries). An experimental evaluation shows that the CA tree has state-of-the-art scalability and performance in a varity of scenarios: sequential access, only single-item operations, range queries of various sizes, and range updates. We are not aware of any concurrent ordered set with fixed synchronization granularity that can perform as good as the CA tree over a wide range of scenarios.

Paper II discusses the CA tree as a back end data structure for a general purpose in-memory data base (a.k.a. key-value store). The CA tree is well suited for making concurrent ordered key-value stores based on a sequential data structure more scalable as much of the original implementation can be reused. Paper II illustrates this with an example application use case. The paper contains experimental results showing that the CA tree can be used to make Erlang ETS `ordered_set` substantially more scalable than it currently is.

Paper III presents the lock-free contention adapting search tree (LFCA tree). The LFCA tree makes use of immutability to make the conflict time for range queries short. Such exploitation of immutability makes the case for adaptation of the synchronization granularity very strong: a concurrent set with coarse-grained synchronization that exploits immutability has excellent performance for large range queries and snapshots but very poor scalability when there are frequent parallel updates, while the situation is reversed for a concurrent set with fine-grained synchronization. The experimental results that Paper III presents show that the LFCA tree has supe-

rior performance compared to concurrent sets with a fixed synchronization granularity over a wide range of scenarios.

Paper IV discusses programming interfaces and locking libraries for delegation locking and the effort required to change an application that uses traditional locking into using delegation locking. It also presents experimental results showing the potential performance benefits that can be obtained when doing such a change.

The final paper of this dissertation, Paper V, presents the contention avoiding concurrent priority queue (CA-PQ). A CA-PQ changes its behavior and structure based on detected contention. Paper V presents an evaluation of a CA-PQ implementation (which uses one of the locking libraries presented in Paper IV). The evaluation measures the performance of CA-PQ and related data structures when used in a parallel version of Dijkstra's single source shortest path algorithm. CA-PQ's ability to adapt to different scenarios contributes to its excellent scalability over a wide range of scenarios.

I claim that the above statements make it clear that this dissertation strongly supports the following[1]:

Thesis

Concurrent ordered sets that dynamically adapt their structure based on usage statistics can perform significantly better across a wide range of scenarios compared to concurrent ordered sets that are non-adaptive.

[1]We note that a concurrent priority queues can be seen as concurrent ordered sets as they represent ordered sets of items.

References

[1] Konstantinos Sagonas and Kjell Winblad. A contention adapting approach to concurrent ordered sets. *Journal of Parallel and Distributed Computing*, 115:1 – 19, 2018.

[2] Konstantinos Sagonas and Kjell Winblad. Contention adapting search trees. In *14th International Symposium on Parallel and Distributed Computing*, ISPDC, pages 215–224. IEEE, 2015.

[3] Konstantinos Sagonas and Kjell Winblad. Efficient support for range queries and range updates using contention adapting search trees. In Xipeng Shen, Frank Mueller, and James Tuck, editors, *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC*, volume 9519 of *LNCS*, pages 37–53. Springer, 2016.

[4] Konstantinos Sagonas and Kjell Winblad. More scalable ordered set for ETS using adaptation. In *ACM Erlang Workshop*, pages 3–11, New York, NY, USA, September 2014. ACM.

[5] Kjell Winblad, Konstantinos Sagonas, and Bengt Jonsson. Lock-free contention adapting search trees. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, New York, NY, USA, 2018. ACM. To appear.

[6] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Delegation locking libraries for improved performance of multithreaded programs. In *Euro-Par 2014, Proceedings of the 20th International Conference*, volume 8632 of *LNCS*, pages 572–583. Springer, 2014. Preprint available from `http://www.it.uu.se/research/group/languages/software/qd_lock_lib`.

[7] Konstantinos Sagonas and Kjell Winblad. The contention avoiding concurrent priority queue. In *Languages and Compilers for Parallel Computing: 29th International Workshop, LCPC 2016, Rochester, NY, USA, September 28-30, 2016, Revised Papers*, pages 314–330. Springer International Publishing, 2017.

[8] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. On the scalability of the Erlang term storage. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, pages 15–26, New York, NY, USA, 2013. ACM.

[9] D. Klaftenegger, K. Sagonas, and K. Winblad. Queue delegation locking. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):687–704, March 2018.

[10] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. Brief announcement: Queue delegation locking. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 70–72, New York, NY, USA, 2014. ACM.

[11] Kjell Winblad. Faster Concurrent Range Queries with Contention Adapting Search Trees Using Immutable Data. In Fergus Leahy and Juliana Franco, editors, *2017 Imperial College Computing Student Workshop (ICCSW 2017)*, volume 60 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:13, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[12] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, Feb 2014.

[13] Edsger W Dijkstra. Hierarchical ordering of sequential processes. In *The origin of concurrent programming*, pages 198–227. Springer, 1971.

[14] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[15] Per Brinch Hansen. Structured multiprogramming. In *The origin of concurrent programming*, pages 255–264. Springer, 1972.

[16] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI 3*, volume 3, pages 235 – 245, 1973.

[17] Melvin E. Conway. A multiprocessor system design. In *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*, AFIPS '63 (Fall), pages 139–146, New York, NY, USA, 1963. ACM.

[18] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, New York, NY, USA, 2002. ACM.

[19] William Pugh. Concurrent maintenance of skip lists, 1990. Technical Report CS-TR-2222.1.

[20] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151 – 157, 1996.

[21] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[22] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*,

12(3):463–492, July 1990.

[23] N. Shafiei. Non-blocking patricia tries with replace operations. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 216–225, July 2013.

[24] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008 – 1020, 2008.

[25] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, 1996.

[26] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In Roberto Baldoni, Nicolas Nisse, and Maarten Steen, editors, *Principles of Distributed Systems: 17th International Conference, OPODIS 2013. Proceedings*, pages 206–220. Springer, 2013.

[27] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 161–170, New York, NY, USA, 2012. ACM.

[28] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, May 2006.

[29] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, New York, NY, USA, 2010. ACM.

[30] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems*, 2006.

[31] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *Euro-Par 2013 Parallel Processing - 9th International Conference*, volume 8097 of *LNCS*, pages 229–240. Springer, 2013.

[32] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 317–328, New York, NY, USA, 2014. ACM.

[33] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 343–356, New York, NY, USA, 2014. ACM.

[34] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM*

SIGACT-SIGOPS Symposium on Principles of Distributed Computing, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[35] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 322–331, New York, NY, USA, 2014. ACM.

[36] Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 161–171, New York, NY, USA, 2012. ACM.

[37] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 161–170, New York, NY, USA, 2012. ACM.

[38] Aravind Natarajan, Lee H Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems: 15th International Symposium, SSS 2013*, volume 8255 of *LNCS*, pages 45–60. Springer, 2013.

[39] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 329–342, New York, NY, USA, 2014. ACM.

[40] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

[41] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, PPoPP '90, pages 197–206, New York, NY, USA, 1990. ACM.

[42] Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In Roberto Baldoni, Paola Flocchini, and Ravindran Binoy, editors, *Principles of Distributed Systems: 16th International Conference, OPODIS 2012. Proceedings*, pages 31–45. Springer, 2012.

[43] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[44] Vipin Kumar, K Ramesh, and V Nageshwara Rao. Parallel best-first search of state-space graphs: A summary of results. In *AAAI*, volume 88, pages 122–127, 1988.

[45] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15,

pages 11–20, New York, NY, USA, 2015. ACM.

[46] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-LSM relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '15, pages 277–278, New York, NY, USA, 2015. ACM.

[47] Hamza Rihani, Peter Sanders, and Roman Dementiev. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 80–82, New York, NY, USA, 2015. ACM.

[48] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[49] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[50] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 182–204. World Scientific, 1999.

[51] Joe Armstrong. A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[52] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proceedings of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

[53] Hillel Avni, Nir Shavit, and Adi Suissa. Leaplist: Lessons learned in designing TM-supported range queries. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 299–308, New York, NY, USA, 2013. ACM.

[54] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Prentice hall, 2006.

[55] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept 1979.

[56] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[57] ISO/IEC. Information technology – Programming languages – C. Standard,

International Organization for Standardization, Geneva, CH, March 2011.
[58] Blaise Barney. Introduction to parallel computing.
[59] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
[60] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.
[61] Paul E. Mckenney. Memory barriers: a hardware view for software hackers, 2009.
[62] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, Jan 1990.
[63] M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 234–254, Oct 1979.
[64] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, February 1979.
[65] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.
[66] Travis S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical report, Dept. of Computer Science and Engineering, University of Washington, Seattle, 1993.
[67] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 165–171, Washington, DC, USA, 1994. IEEE Computer Society.
[68] Michael L Scott. Shared-memory synchronization. *Synthesis Lectures on Computer Architecture*, 8(2):1–221, 2013.
[69] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, pages 340–347, New York, NY, USA, 1984. ACM.
[70] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 247–256, New York, NY, USA, 2012. ACM.
[71] Zoran Radović and Erik Hagersten. Hierarchical backoff locks for nonuniform communication architectures. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, pages

241–252. IEEE Computer Society, 2003.

[72] Christoph Lameter. Effective synchronization on Linux/NUMA systems. In *Proc. of the Gelato Federation Meeting*, 2005.

[73] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. NUMA-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 157–166, New York, NY, USA, 2013. ACM.

[74] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, Dec 1972.

[75] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, Oct 1978.

[76] Arne Andersson. Balanced search trees made simple. In Frank Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Algorithms and Data Structures*, pages 60–71, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[77] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4):464–497, October 1996.

[78] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching, vol. 3*. Addison-Wesley, Reading, 2nd edition, 1998.

[79] Robert Endre Tarjan. *Data Structures and Network Algorithms*, volume 14. SIAM, 1983.

[80] Manuel Núñez, Pedro Palao, and Ricardo Peña. A second year course on data structures based on functional programming. In Pieter H. Hartel and Rinus Plasmeijer, editors, *Funtional Programming Languages in Education*, pages 65–84, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[81] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[82] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling lists. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, LFP '94, pages 185–195, New York, NY, USA, 1994. ACM.

[83] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy-update techniques for system V IPC in the Linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309. USENIX, 2003.

[84] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[85] Keir Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge Computer Laboratory, 2004.

[86] Maged M. Michael. Safe memory reclamation for dynamic lock-free

objects using atomic reads and writes. In *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, PODC '02, pages 21–30, New York, NY, USA, 2002. ACM.

[87] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[88] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.

[89] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM.

[90] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, pages 261–270, New York, NY, USA, 2015. ACM.

[91] Dan Alistarh, William M. Leiserson, Alexander Matveev, and Nir Shavit. Threadscan: Automatic and scalable memory reclamation. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 123–132, New York, NY, USA, 2015. ACM.

[92] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 349–359, New York, NY, USA, 2016. ACM.

[93] Nachshon Cohen and Erez Petrank. Automatic memory reclamation for lock-free data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 260–279, New York, NY, USA, 2015. ACM.

[94] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 99–108, New York, NY, USA, 2011. ACM.

[95] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. Interval-based memory reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel*

Programming, PPoPP '18, pages 1–13, New York, NY, USA, 2018. ACM.

[96] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[97] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[98] Paul E. Mckenney. Is parallel programming hard, and, if so, what can you do about it?, 2017. Accessed: 2017-07-26.

[99] Per-Åke Larson. Linear hashing with partial expansions. In *Proceedings of the Sixth International Conference on Very Large Data Bases*, pages 224–232. VLDB Endowment, 1980.

[100] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. KiWi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, pages 357–369, New York, NY, USA, 2017. ACM.

[101] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011.

[102] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328, New York, NY, USA, 2013. ACM.

[103] Dan Alistarh, Justin Kopinsky, Jerry Li, and Giorgi Nadiradze. The power of choice in priority scheduling. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, pages 283–292, New York, NY, USA, 2017. ACM.

[104] Giovanni Della-Libera and Nir Shavit. Reactive diffracting trees. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 24–32, New York, NY, USA, 1997. ACM.

[105] Phuong Hoai Ha, Marina Papatriantafilou, and Philippas Tsigas. Self-tuning reactive diffracting trees. *Journal of Parallel and Distributed Computing*, 67(6):674–694, 2007.

[106] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, November 1996.

[107] R. El-Yaniv, A. Fiat, R. M. Karp, and G. Turpin. Optimal search and one-way trading online algorithms. *Algorithmica*, 30(1):101–139, May 2001.

[108] Erik Österlund and Welf Löwe. Self-adaptive concurrent components. *Automated Software Engineering*, 25(1):47–99, Mar 2018.

[109] Ryan R. Newton, Peter P. Fogg, and Ali Varamesh. Adaptive lock-free maps:

Purely-functional to scalable. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 218–229, New York, NY, USA, 2015. ACM.

[110] Chao-Hong Chen, Vikraman Choudhury, and Ryan R. Newton. Adaptive lock-free data structures in haskell: A general method for concurrent implementation swapping. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 197–211, New York, NY, USA, 2017. ACM.

[111] Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[112] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Reactive spin-locks: a self-tuning approach. In *8th International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN'05)*, pages 6 pp.–, Dec 2005.

[113] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 25–35, New York, NY, USA, 1994. ACM.

[114] Jonathan Eastep, David Wingate, and Anant Agarwal. Smart data structures: An online machine learning approach to multicore data structures. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 11–20, New York, NY, USA, 2011. ACM.

[115] Ashok M. Joshi. Adaptive locking strategies in a multi-node data sharing environment. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 181–191. Morgan Kaufmann, 1991.

[116] Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 14–27, New York, NY, USA, 2018. ACM.

[117] Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205*, DISC 2013, pages 224–238, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[118] Bapi Chatterjee. Lock-free linearizable 1-dimensional range queries. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, pages 9:1–9:10, New York, NY, USA, 2017. ACM.

[119] Archita Agarwal, Zhiyu Liu, Eli Rosenthal, and Vikram Saraph. Linearizable iterators for concurrent sets. `https://arxiv.org/abs/1705.08885`, 2017.

[120] Panagiota Fatourou and Eric Ruppert. Persistent non-blocking binary

search trees supporting wait-free range queries. https://arxiv.org/abs/1805.04779, 2018.

[121] Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[122] Vitaly Aksenov, Vincent Gramoli, Petr Kuznetsov, Anna Malova, and Srivatsan Ravi. A concurrency-optimal binary search tree. In *European Conference on Parallel Processing (Euro-Par 2017)*, pages 580–593. Springer, 2017.

[123] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[124] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[125] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for tso and pso. *Acta Informatica*, 54(8):789–818, Dec 2017.

[126] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E. Venetis. A scalability benchmark suite for Erlang/OTP. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang*, pages 33–42, New York, NY, USA, 2012. ACM.

[127] Jakob Gruber, Jesper Larsson Träff, and Martin Wimmer. Brief announcement: Benchmarking concurrent priority queues:. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 361–362, New York, NY, USA, 2016. ACM.

[128] C. S. Ellis. Concurrent search and insertion in avl trees. *IEEE Transactions on Computers*, C-29(9):811–817, Sept 1980.

[129] K. S. Larsen. Avl trees with relaxed balance. In *Proceedings of 8th International Parallel Processing Symposium*, pages 888–893, Apr 1994.

[130] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *Structural Information and Communication Complexity*, pages 124–138. Springer, 2007.

[131] Robert Markovski. A source-to-source transformer for qd-locking. Bachelor thesis, Uppsala University, October 2017.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 1684

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-354026

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2018