THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Efficient concurrent data structure access parallelism techniques for increasing scalability

Adones Rukundo

CHALMERS
UNIVERSITY OF TECHNOLOGY

Division of Networks and Systems
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2023

Efficient concurrent data structure access parallelism techniques for increasing scalability

Adones Rukundo

ISBN: 978-91-7905-837-1
Series number: 5303
in the series Doktorsavhandlingar vid Chalmers tekniska högskola.
ISSN 0346-718X

Division of Networks and Systems
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg, Sweden
Phone: +46 (0)31 772 1000
www.chalmers.se

Printed by Chalmers Reproservice
Gothenburg, Sweden, May 2023

I dedicate this thesis to my dear mother Saddress Twinomucunguzi
and my dear sister Victoria Talent Musiime

Abstract

Multi-core processors have revolutionised the way data structures are designed by bringing parallelism to mainstream computing. Key to exploiting hardware parallelism available in multi-core processors are concurrent data structures. However, some concurrent data structure abstractions are inherently sequential and incapable of harnessing the parallelism performance of multi-core processors. Designing and implementing concurrent data structures to harness hardware parallelism is challenging due to the requirement of correctness, efficiency and practicability under various application constraints. In this thesis, our research contribution is towards improving concurrent data structure access parallelism to increase data structure performance. We propose new design frameworks that improve access parallelism of already existing concurrent data structure designs. Also, we propose new concurrent data structure designs with significant performance improvements. To give an insight into the interplay between hardware and concurrent data structure access parallelism, we give a detailed analysis and model the performance scalability with varying parallelism.

In the first part of the thesis, we focus on data structure semantic relaxation. By relaxing the semantics of a data structure, a bigger design space, that allows weaker synchronization and more useful parallelism, is unveiled. Investigating new data structure designs, capable of trading semantics for achieving better performance in a monotonic way, is a major challenge in the area. We algorithmically address this challenge in this part of the thesis. We present an efficient, lock-free, concurrent data structure design framework for out-of-order semantic relaxation. We introduce a new two-dimensional algorithmic design, that uses multiple instances of a given data structure to improve access parallelism.

In the second part of the thesis, we propose an efficient priority queue that improves access parallelism by reducing the number of synchronization points for each operation. Priority queues are fundamental abstract data types, often used to manage limited resources in parallel systems. Typical proposed parallel priority queue implementations are based on heaps or skip lists. In recent literature, skip lists have been shown to be the most efficient design choice for implementing priority queues. Though numerous intricate implementations of skip list based queues have been proposed in the literature, their performance is constrained by the high number of global atomic updates

per operation and the high memory consumption, which are proportional to the number of sub-lists in the queue. In this part of the thesis, we propose an alternative approach for designing lock-free linearizable priority queues, that significantly improve memory efficiency and throughput performance, by reducing the number of global atomic updates and memory consumption as compared to skip-list based queues. To achieve this, our new design combines two structures; a search tree and a linked list, forming what we call a Tree Search List Queue (TSLQueue).

Subsequently, we analyse and introduce a model for lock-free concurrent data structure access parallelism. The major impediment to scaling concurrent data structures is memory contention when accessing shared data structure access points, leading to thread serialisation, and hindering parallelism. Aiming to address this challenge, a significant amount of work in the literature has proposed multi-access techniques that improve concurrent data structure parallelism. However, there is little work on analysing and modelling the execution behaviour of concurrent multi-access data structures especially in a shared memory setting. In this part of the thesis, we analyse and model the general execution behaviour of concurrent multi-access data structures in the shared memory setting. We study and analyse the behaviour of the two popular random access patterns: shared (Remote) and exclusive (Local) access, and the behaviour of the two most commonly used atomic primitives for designing lock-free data structures: Compare and Swap, and, Fetch and Add.

Keywords: Data structure, lock free, concurrency, semantic relaxation, design framework, parallelism, performance modelling, performance analysis, multi-core processor, multi-access, stack, FIFO queue, counter, priority queue, search tree.

Acknowledgements

I would like to convey my heartfelt gratitude to my supervisor Prof. Philippas Tsigas who has supported and mentored me all these years with great insight and wisdom. His mentorship has not only made this work possible but has also guided my research career path. I am also grateful to my co-supervisors Dr. John Ngubiri and Prof. John Businge.

I owe a debt of gratitude to my co-authors and collaborators that have directly or indirectly contributed to this work. In no particular order, thank you, Aras Atalar, Ivan Walulya, Fazeleh Hoseini, Charalampos Stylianopoulos, Tommie Månsson, Magnus Almgren, Christian Marx and York Ostermeyer.

I am honoured to have Prof. Peter Sanders as the faculty opponent during the thesis defence. I would like to acknowledge members of the grading committee: Prof. Jesper Larsson Träff, Prof. Idit Keidar, Prof. Stefanos Kaxiras and Prof. Ioannis Sourdis. I also wish to thank my examiner Prof. Thierry Coquand, and the follow-up committee for their support during my PhD studies.

I take this opportunity to thank the SIDA coordinators, with special thanks going to Prof. Engineer Bainomugisha, Prof. Michel Chaudron, Prof. Buyinza Mukadasi and Nestor Mugabe for their tremendous support. I would also like to extend a token of appreciation to my managers Prof. Tomas Olovsson at Chalmers and Dr. Evarist Nabaasa at Mbarara University.

Many thanks to all past and present members of the Chalmers Network and Systems group, Mbarara University Faculty of Computing and the Academic Registrar's department that have contributed to such a great working environment, not forgetting the SIDA PhD Scholars 2016. I consider myself particularly lucky to be able to call several of you, not just colleagues, but friends. Thank you.

Last but not least, I would like to appreciate my family and friends. You have been such a support pillar towards realising this work.

This work received funding from; SIDA/BRIGHT project grant number 317 under the Makerere-Sweden bilateral research programme 2015–2020, the Chalmers Area of Advance: Energy, the Swedish Civil Contingencies Agency (MSB) through the projects "RICS" and "RIOT", the Swedish Foundation for Strategic Research, Sweden, proj. "Future factories in the cloud (FiC)", the Sweden East African University Collaboration (SWEAFUN) and Mbarara University of Science and Technology.

List of Publications

This thesis is based on the following publications:

[A] **Adones Rukundo**, Aras Atalar, Philippas Tsigas, "Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2D design framework". *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary.* *https://doi.org/10.4230/LIPIcs.DISC.2019.31.*

[B] **Adones Rukundo**, Philippas Tsigas, "TSLQueue: An Efficient Lock-free Design for Priority Queues". *27th International Conference on Parallel and Distributed Computing, Euro-Par 2021: Parallel Processing, September 1-3, 2021, Proceedings, Lisbon, Portugal.* *https://doi.org/10.1007/978-3-030-85665-6_24.*

[C] **Adones Rukundo**, Aras Atalar, Philippas Tsigas, "Performance Analysis and Modelling of Concurrent Multi-access Data Structures". *34th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2022, July 11-14, 2022, Philadelphia, USA.* *https://doi.org/10.1145/3490148.3538578.*

Other publications by the author, not included in this thesis, are:

[D] Tommie Månsson, **Adones Rukundo**, Magnus Almgren, Philippas Tsigas, Christian Marx, York Ostermeyer, "Analysis of door openings of refrigerated display cabinets in an operational supermarket". *Journal of Building Engineering*, Volume 26, 2019.
https://doi.org/10.1016/j.jobe.2019.100899.

[E] **Adones Rukundo**, Aras Atalar, Philippas Tsigas, "Brief Announcement: 2D-Stack - A Scalable Lock-Free Stack Design that Continuously Relaxes Semantics for Better Performance". *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, PODC 2018, July 23-27, 2018, Egham, United Kingdom.
https://doi.org/10.1145/3212734.3212794.

Research Contribution

I contributed to Paper A, Paper B, Paper C and Paper E as the lead designer, main implementer and lead author. In Paper D, I was the lead implementer of the sensor network that monitored and streamed data for door openings of refrigerated display cabinets in an operational supermarket. I also collaborated with all the authors in the writing of the manuscript.

Contents

Part I

Overview

CHAPTER 1

Introduction

1.1 Multi-Processor Systems

The computing industry has gone under a revolution, certainly a vigorous shaking-up. More and more transistors have been fit into the same space (*Moore's Law*), but their clock speed cannot be increased without overheating. Overheating can require complex cooling systems negating the possible performance advantages of increased clock speed. The major chip manufacturers have, for the time being at least, given up trying to make single processors run faster.

Instead, manufacturers are turning to *multi-processor* architectures, in which multiple processors (*cores*) communicate directly through shared hardware resources. Herein, the term core refers to an independent computational unit of a *processor*. Therefore a *multi-core* processor will have more than one core, whereas a multi-processor system will have more than one processor that are typically multi-core processors as depicted in Figure 1.1. However, processor and core are sometimes used interchangeably in the literature to refer to the same thing (independent processing unit). Similarly, we use the terms *process(es)* and *thread(s)* interchangeably to refer to an active execution of a set

of computational instructions by a core.

A multi-processor system is defined as a system with more than one independent processing units linked together to enable parallel processing to take place. The key objective of a multi-processor is to boost a system's execution speed without necessary having to increase processor clock speed. Multi-processor systems make computing more effective by exploiting parallelism: harnessing multiple processing units to work on a single task. To keep up with the increasing processor count, *coherent* multi-core processors are configured as multiple nodes with complex memory hierarchies, including several levels of private/local and shared memory.

Shared Memory

Shared memory can be simultaneously accessed by multiple threads with an intent to provide communication among the threads. Shared Memory is an efficient means of sharing data between different cores/processors. Shared memory can either be uniformly distributed as in the *Uniform Memory Access (UMA)* architecture [1] or non-uniformly distributed as in the *Non Uniform Memory Access (NUMA)* architecture.

In a NUMA system, the response time of a memory access depends on the memory location relative to the core location. Under NUMA, a core can access it's own local memory faster than remote memory, where, remote memory can be local memory that belongs to another core or shared memory as depicted in Figure 1.1. In contrast to the NUMA system, under the UMA system, the response time for memory accesses is the same for all cores irrespective of their location away from memory.

Memory Hierarchy

Multiprocessor systems typically have a great deal of complexity associated with their memory hierarchy. There is often a small amount of fast memory such as registers, augmented with increasingly larger amounts of slower memory such as *cache*. Whereas registers are private to each core, cache levels can be private to a single core or shared among a set of cores as depicted in Figure 1.1. Each core typically has a cache hierarchy composed of private and shared caches. The cores store temporary copies of data in the cache hierarchy for faster access than reading from main memory.

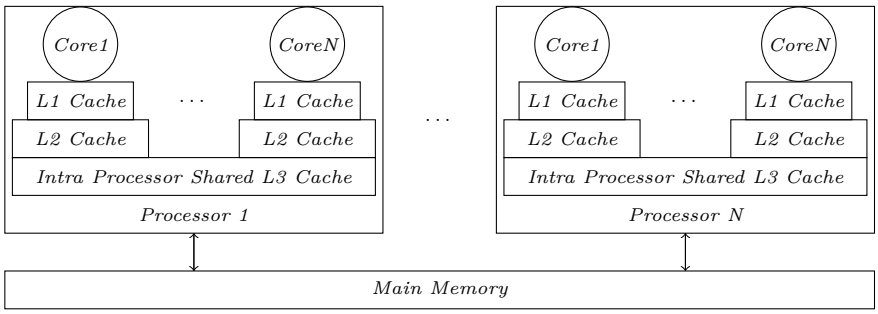

Figure 1.1: An illustration of a typical multiple multi-core processor with three cache levels, L1 and L2 being private to each core, while L3 being shared among all cores on the same processor. Typically, multiprocessors will not have a shared cache between processors and will mostly inter communicate through the main memory or through a special interconnection.

Caches create an illusion of fast high-bandwidth memory by exploiting *locality*. Programs generally access small portions of memory at any small interval in time; either an address is accessed repeatedly, and the accesses are close in time *temporal locality* or adjacent addresses are accessed close in time *spatial locality*. Effectively, bandwidth demands on main memory are reduced, allowing multiple processors to access data more efficiently.

Unfortunately, when cores store copies of shared data in caches, reasoning about executions by different processes on different cores is not straight forward. Replicated copies of data in different caches may not be up-to-date; accordingly, processes may have different views of shared memory locations creating the possibility of inconsistency among cached copies. This gives rise to cache coherence concerns.

Cache Coherence

If multiple caches are allowed to simultaneously have copies of a given memory location, a mechanism must exist to ensure that all copies remain consistent when the contents of a given memory location are modified. For example, imagine a dual-core processor where each core brought a copy of a given block of memory `M` into its private cache, and then one core modifies the content

of `M`. When the second core attempts to read the content of `M` from its cache, its copy will not have the most recent version of `M` since the content will have since been modified. In some systems, a software approach is taken to prevent the existence of multiple copies by marking shared blocks as not to be cached, and by restricting or prohibiting task migration. An alternate approach is to allow multiple copies of given memory blocks be cached simultaneously and to rely on a cache coherence mechanisms to maintain consistency.

In general there are two mechanisms for cache coherence; a *snooping* mechanism and a *directory-based* mechanism.

- **Snooping:** Snooping mechanisms work with bus-based[1] systems, and uses a number of states to determine whether or not it needs to update cache entries, and whether it has control over writing to the block. Snooping depend on cache controllers observing the bus transactions of all other cache processes in the system and taking appropriate actions to maintain consistency [2]–[5]. The state of each memory location in the system is encoded in a distributed way among all cache controllers.

- **Directory-based:** In a directory-based coherence mechanism, a directory is used to hold information about which memory locations are being shared in multiple caches, and which are exclusively cached in one core's cache. The directory knows when a block needs to be updated or invalidated. [6]–[9].

Among multi-processor systems, the snooping mechanism is mostly used together with the *MESI* cache coherence protocol. MESI maintains cached memory blocks (*cache-lines*) in either of the four states; modified, exclusive, shared and invalid [4]. A cache-line in modified state means that it is cached only in the current cache, and has been modified from the value in main memory. The cache-line is required to be written back to main memory at some time in the future, before granting any other core read requests to the cache-line. Writing back a cache-line in modified state, changes the cache-line to the shared state. Shared state means that the cache-line maybe cached in other caches and also matches the value in main memory. Exclusive state applies to a the cache-line that is only cached by the current cache and matches the value in main memory. It may be changed to the Shared state in response

[1] The bus is a single set of wires connecting several devices, each of which can observe every bus transaction.

to a read request from another core. Alternatively, it may be changed to the Modified state when written by either the current core or another core. Invalid state indicates that the cache-line is not in use and therefore has to be fetched from main memory.

While cache coherence hides the complexity of the system from the programmer, it also hides opportunities for performance improvement, making it difficult to exploit the full capabilities that these processors provide. A lot of research has been dedicated to understanding the performance limitations of cache coherence protocols, and how concurrent systems can be tuned to overcome such limitations and extract utmost performance [10]–[15].

1.2 Process Synchronization

The need for process synchronization arises in every area where multiple processes have to agree or commit to a given set of steps. In order to cooperate, concurrently executing processes must communicate and synchronize. Process synchronization refers to information exchange among concurrent processes either via shared memory or a message passing mechanism. In shared memory systems, concurrent processes can share information by updating variables and make them visible to other processes through shared memory locations. In contrast, a message passing mechanism allows processes in separate memory spaces to share information with each other through messages passed across the different memory spaces.

Process synchronization is important in concurrent processing to prevent race conditions. A race condition occurs when the timing or order of process events affects the correctness of a program. This can be due to a data race condition, where, a process accesses a mutable object while another concurrent process is writing to it. Race conditions can be detrimental, especially for concurrent data structures whose sequential semantics must be maintained. Race conditions are commonly avoided through the notion of *critical section*. A critical section can be a particular program segment (sequence of instructions) that need to be executed by a single process at a time in order to maintain program correctness.

Process synchronization ensures that concurrent processes do not simultaneously execute a critical section by guaranteeing some notion of atomicity. A guarantee of atomicity prevents the critical section from being partially

executed. As a consequence, a process executing the critical section cannot be observed to be in progress by another concurrent process, in that, a given sequence of instructions executed within a critical section appears instantaneously to other concurrent processes [16]–[19].

As a requirement, process synchronization mechanisms should be correct by satisfying both the *safety* and *liveness* properties. Informally, a safety property states that "bad" things never happen, while, liveness property (progress guarantees) states that "good" things eventually happen [20], [21]. Some examples of safety properties are *mutual exclusion* and *deadlock freedom*, whereas for liveness are *starvation freedom* and *live-lock freedom*.

- **Mutual Exclusion:** in any execution, at most one process is in the critical section – "bad thing" happening, is two or more processes executing in a critical section leading to a race condition.
- **Deadlock Freedom:** if a process attempts to enter a critical section, then eventually some process executes inside the critical section – "bad thing" happening, is a deadlock where no process cannot eventually enter the critical section, hindering progress.
- **Starvation Freedom:** a process makes progress in an infinite execution – the "good thing", is making progress.
- **Live-lock freedom:** not all processes run forever without progress – the "good thing" is at least one process makes progress.

Processes' access to a critical section is controlled by using synchronization mechanisms popularly implemented using hardware *atomic primitives*. Synchronization mechanisms can be split into two categories; *blocking* and *non-blocking*, that can further be split into sub-categories as discussed below.

Atomic Primitives

Atomic primitives are mostly hardware-assisted operations that atomically update data at a given memory location. Given hardware instructions conduct a set of steps (*read*, *modify* and *write*) atomically at the hardware level. Atomic primitives are required for efficient implementation of synchronization, and have been utilised extensively to implement synchronization mechanisms in an efficient way. Utilising the atomic primitives of a processor to access a

shared memory location atomically is key to the correctness and feasibility of concurrent software systems. For this reason, many modern multi-processor hardware systems provide a number of atomic primitives as part of their instruction set [22]–[24].

Some of the most popular atomic primitives for synchronization include:

- **Compare and Swap (`CAS`):** `CAS` typically takes three arguments say `A`,`B` and `C`, where `A` is the memory location whose value needs to be atomically updated, `B` is the value expected to be currently in location `A` and `C` is the new value that needs to be written into location `A` [25]. `CAS` compares the contents of `A` with the given value `B` and, only if they are the same, `CAS` modifies the contents of `A` by writing the new given value `C` to `A`. If the value had been updated by another processor in the meantime, the `CAS` write instruction would fail thus the `CAS` being unsuccessful. This is done as a single atomic operation guaranteeing that the new value is calculated based on up-to-date information. On a successful `CAS` write, the `CAS` operation is a *read-modify-write* operation, whereas if the `CAS` write fails, the `CAS` operation is a *read* operation. The result of the `CAS` operation must indicate whether it performed the *read-modify-write* operation; this can be done either with a simple boolean response, or by returning the value read from `A` before the `CAS` write. `CAS` typically operates on memory location of size equivalent to one word, however, there are other variants such as the Double-word Compare and Swap (`DCAS`) the can operate on a double word size memory location.
- **Fetch and Add (`FAA`):** `FAA` typically takes two arguments say `A` and `B`, where `A` is the memory location whose value needs to be atomically incremented and `B` is the value that needs to be added to the value in location `A`. `FAA` reads the value in location `A`, modifies the value by adding to it the specified value `B`. That is, `FAA` increments the value at memory location `A` by `B` and return the original value at `A`. This is done as a single atomic operation guaranteeing that the increment is calculated based on up-to-date information in such a way that if this operation is executed by one process in a concurrent system, no other process will ever see an intermediate result. Unlike the `CAS` operation that can either be as a *read-modify-write* or *read* operation, `FAA` always succeeds to increment the given memory location, making it a *read-modify-write* operation. There are other variants of atomic primitives

that follow the same procedure as FAA , these include; Add and Fetch, Fetch and Subtract, Subtract and Fetch, Fetch and Xor, and Xor and Fetch.

- **Test and Set (TAS):** TAS typically takes one argument say A, which is a memory location. TAS reads the value at memory location A and modifies it by writing one to it (setting it to one). That is, TAS sets the value of a given memory location to one and returns the old value read before the TAS write instruction. This is done as a single atomic operation, in that if multiple processes may access the same memory location A, and if a process is currently performing a TAS , no other process may begin another TAS until the first process's TAS is finished. As described above, TAS always sets the given memory location to one despite whether it is already set or not. Test Test and Set (TTAS) is an extension of TAS that adds an initial test before TAS instructions are executed. In the case of TTAS , the given memory location A is read and checked if it is set or not, if the memory is set, TTAS retries without modifying the memory location. Otherwise, if the memory is not set, TAS operation is executed to try and set the memory to one.

Apart from the commonly available atomic primitives described above, some multi-processor hardware systems provide a pair of instructions called Load-Link / Store-Conditional (LL/SC) as an alternative to the CAS operation [25]. LL/SC typically takes one argument as a memory location. Load-Link reads and returns the current value in the given memory location, while a subsequent Store-Conditional to the same memory location by the same process will store a new value only if no updates have occurred to that memory location since the Load-Link. Similar to the CAS , on a successful LL/SC write, the LL/SC operation is an atomic *read-modify-write* operation, whereas if the Store-Conditional write fails, the LL/SC operation is only a *read* operation. LL/SC is some what similar to the CAS operation in terms of conditionally updating a given memory location. However, the CAS operation can be affected by the so called *ABA* problem, a problem which does not occur with LL/SC . ABA problem occurs during synchronization, when a process say X, reads a given memory location A twice and returns the same value a for both reads. However between the two reads, another process(es) say Y, can update the value to b (A $\leftarrow$ b), do some work and then update the value back to a (A $\leftarrow$ a), thus fooling X into thinking "nothing has changed" even though Y

could have done work that violates that assumption. The `LL/SC` operation can instead detect any concurrent update on `A` between the time interval of a Load-Link and Store-Conditional pair, independent of the value held by `A`. Unfortunately, the real hardware implementations of `LL/SC` available are rather weak, where the Store-Conditional operation can fail spuriously due to shared accesses to `A` or even undefined reasons. Also, LL/SC pairs may not be nested by the same processor. Versioning memory updates is one of the common techniques used to support `CAS` operations overcome the ABA problem [26], [27].

Blocking Synchronization

A general way to synchronize multiple accesses to shared resources is to use blocking approaches also referred to as *mutual exclusion*. Mutual exclusion can be implemented in several ways including disabling process interrupts, message passing where a token system distributes accesses and critical sections guarded by locks. In shared memory concurrent systems, mutual exclusion is most commonly achieved by using critical sections guarded by locks. A process has to request and acquire the given lock before it can enter the critical section. As a requirement, a process never enters a critical section while another process is already executing inside the critical section [28]. This implies, that at most one process executes instructions inside the critical section at any given point in time. Any other process that requests for the critical section lock held by another process (executing inside the critical section) will be blocked until the lock is released. The blocked process can either busy wait or yield [29], thus guaranteeing that instructions executed in the critical section appear atomic to other processes. Locks are simple to implement in most use cases and are widely available in most platforms and operating systems.

However, locks have some significant drawbacks that need to be taken care of to avoid associated pitfalls:

- **Blocking/Convoying:** Once a given process holds a critical section lock, all other eligible process trying to enter the same critical section have to block. If the process is delayed within the critical section, the blocked processes can queue up as they wait on the lock to be released. When the lock is released, the queued processes form a convoy as they acquire the lock to gain exclusive access to the critical section [17], [30].

Blocking makes the computation of worst-case response times more complicated, and the currently used computation methods are quite pessimistic.

- **Deadlocks:** If locks are improperly used, circular dependencies might arise leading to a deadlock. A circular dependency occurs when a process (say `P1`) holding a critical section lock (say `L1`) blocks while trying to enter a critical section locked by a lock (say `L2`) held by another process (say `P2`). Meanwhile, `P2` is also blocked trying to enter the critical section locked by `L1` held by blocked `P1`. This therefore implies that neither `P1` or `P2` will proceed leading to a deadlock.

- **Priority inversions:** The exclusion of other tasks while one low priority task is holding the lock can cause a high priority task to actually have to wait for middle priority tasks to finish. With preemptive scheduling, a high priority process can yield the processor to a low priority process that holds a lock required by the high priority process. Then a middle priority process that does not need the lock preempts the low priority process leading to a priority inversion between high priority and middle priority processes [31].

Non-blocking Synchronization

Non-blocking synchronization mechanisms provide an alternative to achieving atomicity without using mutual exclusion. Non-blocking synchronization states that an attempt by a process to access a shared resource cannot block or be blocked by another process, regardless of the state of the system. However, it does not specify the outcome of the attempt. As non-blocking synchronization does not involve mutual exclusion, shared resource accesses can be executed concurrently. Generally, non-blocking mechanisms are optimistic; each process attempts to execute independently or locally for as long as possible and publish their modifications using atomic instructions. An optimistic execution of a process can be invalidated by a concurrent modification, at which point publication of the modifications will fail, and the process will have to repeat the local execution and try to publish.

Being concurrent, the criteria for consistency correctness of non-blocking synchronization is a bit more complex than that for the respective blocking implementation. In general, the correctness condition used for concurrent

operations is called *linearizability* [32]. Informally, linearizability states that in every execution, each supporting operation appears to take effect instantaneously at some point (*linearization point*) between the operation's invocation and response. This basically means that for each real concurrent execution there exists an equivalent sequential execution that preserves the partial order that is legal according to the sequential semantics of the given data structure. The fulfilment of the linearizability property enables concurrent shared resources to still be accessed in a predictive manner. The operation can be viewed by other concurrent processes as it occurred at a unique instant in time, i.e. the effect of two operations can not be viewed as taking place at the same time.

Non-blocking mechanisms can be classified according to the progress guarantees that they provide, as follows:

- **Wait-freedom:** A synchronization mechanism is wait-free if it guarantees that every process continues to make progress regardless of arbitrary delays or failures of other processes [33]. Wait-freedom guarantees individual progress; combines non-blocking progress with starvation freedom.

- **Lock-freedom:** A synchronization mechanism is lock-free if it guarantees that at least some process makes progress [34]. Ensures system-wide progress without ensuring starvation freedom.

- **Obstruction-freedom:** A synchronization mechanism is obstruction-free if it guarantees that a process will eventually make progress if executed in isolation.

Non-blocking synchronization mechanisms are more resilient to pitfalls associated with mutual exclusion. Strong progress guarantees such as wait-freedom may be required for systems with real-time constraints and resiliency requirements. However, they are non-trivial to achieve efficiently [35], thus, on modern shared-memory multi-core systems, weaker progress guarantees such as lock-freedom and obstruction-freedom generally suffice, and are easier to implement efficiently [36]. In contrast to wait-freedom and lock-free, obstruction-freedom is dependent on the operating system scheduler.

1.3 Concurrent Data Structures

Data structures are an important component of efficient and well structured programs. Data structures organise data in a way to allow efficient access. In shared memory multi-core computing, data structures can be shared among multiple threads to exploit parallelism available on multi-core systems. Concurrent data structures are shared data structures that allow threads to access the data structure concurrently. Concurrent data structures are essentially adaptations of abstract data types, defined for sequential data structures to support concurrent operations. Concurrent accesses require synchronized access to guarantee consistency with respect to the given data structure sequential semantics [37], [38].

Thread synchronization of concurrent accesses is generally achieved by guaranteeing some notion of atomicity, where, an operation appears to occur at a single instant between its invocation and its response. A concurrent data structure is typically designed around one or more synchronization *access-points*. An access-point is a memory location from where threads compute, consistently, the current state of the data structure. As an example, a stack has one access-point referred to as the *top*, from where concurrent threads can add (*push*) or remove (*pop*) an item from the stack. The state of the stack will change when an item is pushed or popped at the top of the stack. Almost similar to the stack, a FIFO[2] queue has two access-points, through one access-point (*tail*) a thread can add (*enqueue*) an item to the queue, and through the other access-point (*head*) a thread can remove (*dequeue*) an item from the queue. Although the FIFO queue has two access-points, each type of operation is tied to only one access-point. There are also data structures such as a tree and a skip list that have multiple access-points for either adding or removing items from the data structure.

Thread synchronization is vital to achieving consistency and cannot be eliminated [39]. Whereas this is true, synchronization mechanisms usually result in poor performance because they produce large amounts of memory and interconnection network *contention* and, more significantly, because they produce *convoy* effects [40], [41]. When threads concurrently access a shared resource, one thread succeeds and others incur stalls waiting to gain access, one by one (convoying).

[2]First In First Out semantics

The necessity of reducing contention at the synchronization access-points, and consequently improving scalability, is and has been a major focus for concurrent data structure researchers. Techniques like; elimination [42], [43], combining [44], dynamic elimination-combining [45] and back-off strategies have been proposed as ways to improve scalability. To address, in a more significant way, the challenge of scalability bottlenecks of concurrent data structures, it has been proposed that the semantic legal behaviour of data structures should be extended [46]. This line of research has led to the introduction of an extended set of weak semantics including; weak internal ordering, weakening consistency and semantic relaxation.

Correctness

Unfortunately, concurrent data structures are difficult to design. There is a kind of conflict between correctness and performance: the more one tries to improve performance, the more difficult it becomes to reason about the resulting data structure correctness. In contrast to the sequential data structures, concurrent data structure correctness has two aspects: safety, guaranteeing that nothing bad happens, and liveness, guaranteeing that eventually something good will happen.

The safety aspects of concurrent data structures impose the need to argue about the many possible interleavings of processes. It is more intuitive to specify how abstract data structures behave in a sequential setting, where there are no interleavings. Thus, the standard approach to arguing the safety properties of a concurrent data structure is to specify the data structure's properties sequentially, and then map its concurrent executions to these "correct" sequential properties. There are various approaches for doing this, called consistency conditions. Some familiar conditions are serializability, linearizability, sequential consistency, and quiescent consistency. Linearizability [32] is widely accepted as the strongest correctness condition of concurrent data structures. Informally, linearizability states that in every execution of the data structure implementation, each supporting operation appears to take effect instantaneously at some point (*linearization point*) between the operation's invocation and response.

Concurrent execution processes are modelled by a *history*. A history `H` is a finite or infinite sequence of operation invocations and responses. A history is sequential if; the sequence starts with an invocation and a matching

response immediately follows each invocation. A history is admissible if it is a subset of the data structure's sequential specification. Thus, an execution is linearizable if there exists a sequential history `S` of operations in the execution that respects the object's sequential specification, and observes the real-time ordering of events for all processes. `S` is referred to as a linearization of `H`. Typically, to show that a concurrent data structure execution is linearizable, one defines a linearization point for every operation in the execution history. Intuitively, the order induced by a sequence of linearization points preserves the real-time ordering of non-overlapping operations. Additionally, linearizability is composable; a composition of linearizable histories is linearizable. This property is fundamental, concurrent data structures can be designed, verified and implemented independently then combined to make a larger, but still linearized data structure.

Semantic Relaxation

Semantic relaxation is one popular way of improving concurrent data structure scalability on the ever growing number of multi-core processor cores. However, this is achieved at the expense of relaxing correctness, by redefining the semantics of the data structures [47]. By relaxing the semantics of a data structure, a bigger design space, that allows weaker synchronization and more useful parallelism, is unveiled. One of the main definition of semantic relaxation proposed and used in the literature is *k-out-of-order* [27], [47]–[52]. k-out-of-order semantics allow operations to occur out of order within a given k bound, for example, a pop operation of a k-out-of-order stack can remove any item among the k topmost stack items. By allowing a pop operation to remove any item among the k topmost stack items, the semantics do not anymore impose a single access-point. Thus, by relaxing the stack semantics, we allow for potentially more efficient stack designs with reduced synchronization overhead, which is the motivation for concurrent data structure semantics relaxation.

The general idea behind most semantic relaxation techniques is to increase the number of access-points from which concurrent threads can access the data structure and complete their operations in parallel [49], [50], [53]–[55]. Increasing the number of access-points of a concurrent data structure has the potential to improve parallelism, and thus harness the high throughput performance capabilities of the highly parallel multi-core processors [56]. Al-

though relaxing semantics has been studied and shown to significantly improve throughput performance, it has also been shown, that semantic relaxation, through increasing the number of access-points, is inversely proportional to the data structure *accuracy* (degree of relaxation) [27], [57], [58]. In the context of semantic relaxation, data structure accuracy is the measure of how far away the relaxed version of given data structure is from the it's exact sequential semantics. In other words, the degree of relaxation. Increasing the number of data structure access-points also has a memory consumption trade-off. Memory consumption increases with the increase in the number of access-points, which in turn increases the cost of data structure access [13], [27].

Scaling throughput performance is as important as data structure accuracy and memory efficiency. As the number of access-points increases beyond a certain point, the trade-offs can out weigh the performance gain [27]. Therefore, understanding the liveliness of the trade-offs is key to the designing of scalable concurrent data structures without counteracting the performance benefits achieved through increasing the number of access-points. Modelling and analysing the practical performance of concurrent data structures with multiple access-points can give an insight into how to balance the various trade-offs [59].

Access-point Search Overhead

Having multiple access-points means that a thread has to search and select a given access-point among the available access-points from which to read or update the state of the data structure. Searching for an access-point may involve reading multiple shared memory resources (cache-lines) before finally selecting a given access-point. Having to read multiple cache-lines increases the memory latency, in turn increasing the operation cost of accessing the data structure. Towards minimising the search cost, design approaches have been proposed in the literature, for both semantically relaxed and non relaxed concurrent data structures. One popular design approach is to introduce temporal locality.

Temporal locality is mostly achieved by assigning a thread exclusive ownership of a given access-point for given number of operations or depending on a set of conditions [27], [52], [53], [60]. A thread can therefore operate on the same access-point without having to incur the cost of searching for another

access-point. However, for temporal locality to be effective, there has to be a mechanism through which threads can efficiently share tasks or access-points. Such mechanisms include work stealing [61], [62] and controlled thread access-point acquisition [27]. Recall in cache based multi-core systems, a cached copy of a shared memory location changes state between thread accesses from different cores. When a cached copy of a given access-point `A` is written to by a given thread `X` on a given core, the cache-line copy of `A` belonging to `X` core (A_X) state changes to modified state. `X` can consecutively access `A` cheaply without having to fetch data remotely[3] for as long as A_X is in modified state. Without temporal locality, another thread on a different core may access and write to `A` effectively changing the state of A_X to invalid. The next time `X` tries to access `A`, `X` will have to fetch a fresh copy of `A` remotely since it's cached copy A_X would be invalid. Fetching data remotely is costly in terms of memory latency depending on where the data is located within the memory hierarchy [10]–[14], [59], [63], [64]. Allowing a thread to utilise the same access-point for multiple operations without interference from other threads improves parallelism and cache efficiency.

[3]Remote data access is when a thread has to request for a valid copy of data from the main memory or other cache levels other than it's core local cache levels.

CHAPTER 2

Summary of included contributions

This chapter provides a summary of the included contributions.

2.1 Contribution A

Adones Rukundo, Aras Atalar, Philippas Tsigas
Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2D design framework

There has been a significant amount of work in the literature proposing semantic relaxation of concurrent data structures for improving scalability and performance. By relaxing the semantics of a data structure, a bigger design space, that allows weaker synchronization and more useful parallelism, is unveiled. Investigating new data structure designs, capable of trading semantics for achieving better performance in a monotonic way, is a major challenge in the area. We algorithmically address this challenge in this contribution. We present an efficient, lock-free, concurrent data structure design framework for out-of-order semantic relaxation. We introduce a new two dimensional algo-

rithmic design, that uses multiple instances of a given data structure. The first dimension of our design is the number of data structure instances operations are spread to, in order to benefit from parallelism through disjoint memory access; the second dimension is the number of consecutive operations that try to use the same data structure instance in order to benefit from data locality. Our design can flexibly explore this two-dimensional space to achieve the property of monotonically relaxing concurrent data structure semantics for better performance within a tight deterministic relaxation bound. We show how our framework can instantiate lock-free out-of-order queues, stacks and counters. We provide implementations of these relaxed data structures and evaluate their performance and behaviour on two parallel architectures. Experimental evaluation shows that our two-dimensional design significantly outperforms the respected previous proposed designs with respect to scalability and performance. Moreover, our design increases performance monotonically as relaxation increases.

2.2 Contribution B

Adones Rukundo, Philippas Tsigas
TSLQueue: An Efficient Lock-free Design for Priority Queues

Priority queues are fundamental abstract data types, often used to manage limited resources in parallel systems. Typical proposed parallel priority queue implementations are based on heaps or skip lists. In recent literature, skip lists have been shown to be the most efficient design choice for implementing priority queues. Though numerous intricate implementations of skip list based queues have been proposed in the literature, their performance is constrained by the high number of global atomic updates per operation and the high memory consumption, which are proportional to the number of sub-lists in the queue. In this contribution, we propose an alternative approach for designing lock-free linearizable priority queues, that significantly improve memory efficiency and throughput performance, by reducing the number of global atomic updates and memory consumption as compared to skip-list based queues. To achieve this, our new design combines two structures; a search tree and a linked list, forming what we call a Tree Search List Queue (TSLQueue). The leaves of the tree are linked together to form a linked list of leaves with a

head as an access point. Analytically, a skip-list based queue insert or delete operation has at worst case $O(\log n)$ global atomic updates, where n is the size of the queue. While the TSLQueue insert or delete operations require only 2 or 3 global atomic updates respectively. When it comes to memory consumption, TSLQueue exhibits $O(n)$ memory consumption, compared to $O(n \log n)$ worst case for a skip-list based queue, making the TSLQueue more memory efficient than a skip-list based queue of the same size. We experimentally show, that TSLQueue significantly outperforms the best previous proposed skip-list based queues, with respect to throughput performance.

2.3 Contribution C

Adones Rukundo, Aras Atalar, Philippas Tsigas
Performance Analysis and Modelling of Concurrent Multi-access Data Structures

The major impediment to scaling concurrent data structures is memory contention when accessing shared data structure access-points, leading to thread serialisation and hindering parallelism. Aiming to address this challenge, a significant amount of work in the literature has proposed multi-access techniques that improve concurrent data structure parallelism. However, there is little work on analysing and modelling the execution behaviour of concurrent multi-access data structures especially in a shared memory setting. In this contribution, we analyse and model the general execution behaviour of concurrent multi-access data structures in the shared memory setting. We study and analyse the behaviour of the two popular random access patterns: shared (Remote) and exclusive (Local) access, and the behaviour of the two most commonly used atomic primitives for designing lock-free data structures: Compare and Swap, and, Fetch and Add. We model the concurrent multi-accesses by splitting the thread execution procedure into five logical sessions: i) side-work, ii) access-point search iii) access-point acquisition, iv) access-point data acquisition and v) access-point data operation. We evaluate our model on a set of concurrent data structure designs including a counter, a stack and a FIFO queue. The evaluation is carried out on two state of the art multi-core processors: Intel Xeon Phi CPU 7290 with 72 physical cores and Intel Xeon E5-2695 with 14 physical cores. Our model is able to predict the

throughput performance of the given concurrent data structures with 80% to 100% accuracy on both architectures.

CHAPTER 3

Concluding Remarks and Future Work

In this thesis, we have contributed research towards improving concurrent data structure access parallelism to increase data structure scalability. We have proposed new design frameworks that improve access parallelism of already existing concurrent data structure designs. Also, we have proposed new concurrent data structure designs with significant performance improvements. To give an insight into the interplay between hardware and concurrent data structure access parallelism, we have given a detailed analysis and modelled the performance scalability with varying parallelism.

In the first part of the thesis, we showed that semantics relaxation has the potential to monotonically trade relaxed semantics of concurrent data structures for achieving throughput performance within tight relaxation bounds. This was achieved through an efficient two-dimensional framework that is simple and easy to implement for different data structures. We demonstrated that, by deriving two-dimensional lock-free designs for stacks, FIFO queues and shared counters. Our experimental results showed that compared to previous solutions, our framework can be used to extend existing data structures with minimal modifications while achieving better performance in terms of throughput and accuracy.

We observed that the framework has several possible parameter configurations. As part of future work, we intend to further explore these configurations and propose possible optimal configurations for different system execution environments. Furthermore, we intend to look into how data structure semantic relaxation can be applied dynamically without having relaxation bounds as a fixed parameter. As an example, contention varies as threads request access to shared resources. There is a need for a mechanism that can detect the level of contention within the system and adjust the relaxation bounds accordingly to improve data accuracy (elastic relaxation). This would help maintain a meaningful data structure semantics scalability trade-off.

In the second part of the thesis, we introduced a new design approach for designing efficient priority queues. We have demonstrated the design with a linearizable lock-free priority queue implementation. Our implementation outperformed the previously proposed state-of-the-art skip list based priority queues. In the case of DeleteMin() we have achieved a performance improvement of up to more than 400% and up to more than 65% in the case of Insert(). Numerous optimisation techniques such as flat combining, elimination and back-off can be applied to further enhance the performance of TSLQueue.

As part of future work, we intend to apply numerous optimisation techniques such as balancing the tree search structure, flat combining, elimination and back-off to further enhance the performance of the TSLQueue. TSLQueue improved parallelism makes it a good candidate for semantic relaxation, we also intend to explore ways of implementing relaxed priority queue designs using the TSLQueue.

Subsequently, we analysed and modelled the performance of concurrent multi-accesses of lock-free data structures in multi-core/many-core shared memory systems. We considered disjoint memory accesses techniques that typically use two types of memory access patterns, locally and remotely. We considered two classes of atomic operations: Repeat until Condition (Compare and Swap) and Atomically Modify (Fetch and Add), which are the typical atomics used in the design of lock-free data structures. We then modelled the acquisition of a memory access-point, as a system of queuing networks with parallel servers, where each server corresponds to an access point. We also modelled memory latency in terms of cache location and data coherence status. Our evaluation results show that our model follows closely the actual execution behaviour without significant deviations independently of the num-

ber of access points or concurrent threads used.

As part of future work, we intend to explore ways of extending the model to data structures with a more complex search process such as that of hierarchical or 2D-framework data structures. Our preliminary results have shown that improved concurrent data structure parallelism has great potential towards reducing memory and energy consumption. We intend to explore this line further and study the impact of parallelism techniques such as semantic relaxation on energy and memory consumption. We consider memory and energy as some of the critical resources, especially for mobile computing.

References

[1] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, vol. 12, no. 2–3, pp. 72–109, Sep. 1994, ISSN: 0178-4617.

[2] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," *SIGARCH Comput. Archit. News*, vol. 11, no. 3, pp. 124–131, Jun. 1983, ISSN: 0163-5964.

[3] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 276–283, Jun. 1985, ISSN: 0163-5964.

[4] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 348–354, Jan. 1984, ISSN: 0163-5964.

[5] L. Rudolph and Z. Segall, "Dynamic decentralized cache schemes for mimd parallel processors," *SIGARCH Comput. Archit. News*, vol. 12, no. 3, pp. 340–347, Jan. 1984, ISSN: 0163-5964.

[6] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, 1990.

[7] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, ser. AFIPS '76, New York, New York: Association for Computing Machinery, 1976, pp. 749–753, ISBN: 9781450379175.

[8] W. C. Yen, D. W. L. Yen, and K.-S. Fu, "Data coherence problem in a multicache system," *IEEE Transactions on Computers*, vol. C-34, no. 1, pp. 56–65, 1985.

[9] Censier and Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, vol. C-27, no. 12, pp. 1112–1118, 1978.

[10] J. Archibald and J.-L. Baer, "Cache coherence protocols: Evaluation using a multiprocessor simulation model," *ACM Trans. Comput. Syst.*, vol. 4, no. 4, pp. 273–298, Sep. 1986, ISSN: 0734-2071.

[11] M.-C. Chiang and G. S. Sohi, "Evaluating design choices for shared bus multiprocessors in a throughput-oriented environment," *IEEE Trans. Comput.*, vol. 41, no. 3, pp. 297–317, Mar. 1992, ISSN: 0018-9340.

[12] S. Ramos and T. Hoefler, "Capability models for manycore memory systems: A case-study with xeon phi knl," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 297–306.

[13] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore smp systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, New York, New York: Association for Computing Machinery, 2009, pp. 413–422, ISBN: 9781605587981.

[14] D. Molka, D. Hackenberg, R. Schöne, and W. E. Nagel, "Cache coherence protocol and memory performance of the intel haswell-ep architecture," in *2015 44th International Conference on Parallel Processing*, 2015, pp. 739–748.

[15] J. Torrellas, H. Lam, and J. Hennessy, "False sharing and spatial locality in multiprocessor caches," *IEEE Transactions on Computers*, vol. 43, no. 6, pp. 651–663, 1994.

[16] T. Ta, D. Troendle, and B. Jang, "Chapter 3 - thread communication and synchronization on massively parallel gpus," in *Advances in GPU Research and Practice*, ser. Emerging Trends in Computer Science and Applied Computing, H. Sarbazi-Azad, Ed., Boston: Morgan Kaufmann, 2017, pp. 57–81, ISBN: 978-0-12-803738-6.

[17] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012, ISBN: 9780123973375.

[18] M. L. Scott, *Shared-Memory Synchronization*. Morgan and Claypool Publishers, 2013, ISBN: 160845956X.

[19] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *ACM Comput. Surv.*, vol. 15, no. 1, pp. 3–43, Mar. 1983, ISSN: 0360-0300.

[20] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, no. 2, pp. 125–143, 1977.

[21] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985, ISSN: 0020-0190.

[22] M. Michael and M. Scott, "Implementation of atomic primitives on distributed shared memory multiprocessors," in *Proceedings of 1995 1st IEEE Symposium on High Performance Computer Architecture*, 1995, pp. 222–231.

[23] D. Dolev, E. Gafni, and N. Shavit, "Toward a non-atomic era: L-exclusion as a test case," in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, ser. STOC '88, Chicago, Illinois, USA: Association for Computing Machinery, 1988, pp. 78–92, ISBN: 0897912640.

[24] F. Hoseini, A. Atalar, and P. Tsigas, "Modeling the performance of atomic primitives on modern architectures," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019, Kyoto, Japan: Association for Computing Machinery, 2019, ISBN: 9781450362955.

[25] J. H. Anderson and M. Moir, "Universal constructions for multi-object operations," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, Ottowa, Ontario, Canada: Association for Computing Machinery, 1995, pp. 184–193, ISBN: 0897917103.

[26] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01, Berlin, Heidelberg: Springer-Verlag, 2001, pp. 300–314, ISBN: 3540426051.

[27] A. Rukundo, A. Atalar, and P. Tsigas, "Monotonically relaxing concurrent data-structure semantics for increasing performance: An efficient 2d design framework," in *33rd International Symposium on Distributed Computing, DISC 2019, October 14-18, 2019, Budapest, Hungary*, J. Suomela, Ed., ser. LIPIcs, vol. 146, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 31:1–31:15.

[28] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, Sep. 1965, ISSN: 0001-0782.

[29] G. Taubenfeld, "The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms," in *Distributed Computing*, R. Guerraoui, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 56–70, ISBN: 978-3-540-30186-8.

[30] M. Blasgen, J. Gray, M. Mitoma, and T. Price, "The convoy phenomenon," *SIGOPS Oper. Syst. Rev.*, vol. 13, no. 2, pp. 20–25, Apr. 1979, ISSN: 0163-5980.

[31] B. W. Lampson and D. D. Redell, "Experience with processes and monitors in mesa," *Commun. ACM*, vol. 23, no. 2, pp. 105–117, Feb. 1980, ISSN: 0001-0782.

[32] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990, ISSN: 0164-0925.

[33] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991, ISSN: 0164-0925.

[34] M. Herlihy, "A methodology for implementing highly concurrent data structures," *SIGPLAN Not.*, vol. 25, no. 3, pp. 197–206, Feb. 1990, ISSN: 0362-1340.

[35] H. Attiya, N. Lynch, and N. Shavit, "Are wait-free algorithms fast?" *J. ACM*, vol. 41, no. 4, pp. 725–763, Jul. 1994, ISSN: 0004-5411.

[36] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit, "Obstruction-free algorithms can be practically wait-free," in *Distributed Computing*, P. Fraigniaud, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 78–92, ISBN: 978-3-540-32075-3.

[37] E. W. Dijkstra, "The structure of "THE"-multiprogramming system," *Commun. ACM*, vol. 11, no. 5, pp. 341–346, May 1968, ISSN: 0001-0782.

[38] E. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, 1965.

[39] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev, "Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated," *SIGPLAN Not.*, vol. 46, no. 1, pp. 487–498, Jan. 2011, ISSN: 0362-1340.

[40] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas, "A study of the behavior of synchronization methods in commonly used languages and systems," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1309–1320.

[41] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: ACM, 2013, pp. 33–48, ISBN: 978-1-4503-2388-8.

[42] D. Hendler, N. Shavit, and L. Yerushalmi, "A scalable lock-free stack algorithm," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04, Barcelona, Spain: Association for Computing Machinery, 2004, pp. 206–215, ISBN: 1581138407.

[43] N. Shavit and D. Touitou, "Elimination trees and the construction of pools and stacks: Preliminary version," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95, Santa Barbara, California, USA: Association for Computing Machinery, 1995, pp. 54–63, ISBN: 0897917170.

[44] N. Shavit and A. Zemach, "Combining funnels: A dynamic approach to software combining," *Journal of Parallel and Distributed Computing*, vol. 60, no. 11, pp. 1355–1387, 2000.

[45] G. Bar-Nissan, D. Hendler, and A. Suissa, "A dynamic elimination-combining stack algorithm," *CoRR*, vol. abs/1106.6304, 2011.

[46] N. Shavit, "Data structures in the multicore age," *Commun. ACM*, vol. 54, no. 3, pp. 76–84, Mar. 2011, ISSN: 0001-0782.

[47] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," *SIGPLAN Not.*, vol. 48, no. 1, pp. 317–328, Jan. 2013, ISSN: 0362-1340.

[48] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *International Conference on Principles of Distributed Systems*, Springer, 2010, pp. 395–410.

[49] D. Alistarh, T. Brown, J. Kopinsky, J. Z. Li, and G. Nadiradze, "Distributionally linearizable data structures," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18, Vienna, Austria: Association for Computing Machinery, 2018, pp. 133–142, ISBN: 9781450357999.

[50] A. Haas, M. Lippautz, T. A. Henzinger, *et al.*, "Distributed queues in shared memory: Multicore performance and scalability through quantitative relaxation," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13, Ischia, Italy, 2013, 17:1–17:9, ISBN: 978-1-4503-2053-5.

[51] E. Talmage and J. L. Welch, "Relaxed data types as consistency conditions," in *Stabilization, Safety, and Security of Distributed Systems: 19th International Symposium, SSS 2017, Boston, MA, USA, November 5–8, 2017, Proceedings*. Cham: Springer International Publishing, 2017, pp. 142–156, ISBN: 978-3-319-69084-1.

[52] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015, San Francisco, CA, USA, 2015, pp. 277–278, ISBN: 978-1-4503-3205-7.

[53] A. Haas, T. A. Henzinger, A. Holzer, *et al.*, "Local linearizability for concurrent container-type data structures," in *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*, 2016, 6:1–6:15.

[54] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, ACM, 2015, pp. 80–82.

[55] M. Williams, P. Sanders, and R. Dementiev, "Engineering multiqueues: Fast relaxed concurrent priority queues," in *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, P. Mutzel, R. Pagh, and G. Herman, Eds., ser. LIPIcs, vol. 204, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 81:1–81:17.

[56] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," in *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, Los Angeles, California, USA: Association for Computing Machinery, 1994, pp. 151–160, ISBN: 0897916549.

[57] D. Alistarh, J. Kopinsky, J. Li, and G. Nadiradze, "The power of choice in priority scheduling," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Washington, DC, USA: ACM, 2017, pp. 283–292, ISBN: 978-1-4503-4992-5.

[58] H. Rihani, P. Sanders, and R. Dementiev, "Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15, Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 80–82, ISBN: 9781450335881.

[59] A. Rukundo, A. Atalar, and P. Tsigas, "Performance analysis and modelling of concurrent multi-access data structures," in *SPAA '22: 34th ACM Symposium on Parallelism in Algorithms and Architectures, Philadelphia, PA, USA, July 11 - 14, 2022*, K. Agrawal and I. A. Lee, Eds., ACM, 2022, pp. 333–344.

[60] E. Gidron, I. Keidar, D. Perelman, and Y. Perez, "Salsa: Scalable and low synchronization numa-aware algorithm for producer-consumer pools," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, ACM, 2012, pp. 151–160.

[61] M. Wimmer and J. L. Träff, "An extended work-stealing framework for mixed-mode parallel applications," in *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings*, IEEE, 2011, pp. 1683–1690.

[62] M. Wimmer, D. Cederman, J. L. Träff, and P. Tsigas, "Work-stealing with configurable scheduling strategies," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, A. Nicolau, X. Shen, S. P. Amarasinghe, and R. W. Vuduc, Eds., ACM, 2013, pp. 315–316.

[63] I. B. Peng, R. Gioiosa, G. Kestor, P. Cicotti, E. Laure, and S. Markidis, "Exploring the performance benefit of hybrid memory system on hpc environments," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 683–692.

[64] H. Schweizer, M. Besta, and T. Hoefler, "Evaluating the cost of atomic operations on modern architectures," *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 445–456, 2015.