Universally Scalable Concurrent Data Structures

THÈSE N^{O} 7993 (2017)

PRÉSENTÉE LE 29 SEPTEMBRE 2017
À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE PROGRAMMATION DISTRIBUÉE
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Tudor Alexandru DAVID

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2017

“Because it’s there.”
— George Mallory, 1923, when asked why he wants to climb Mount Everest

To my family and to my friends. . .

Acknowledgements

Pursuing a PhD has been a tremendously enriching and enjoyable experience. Yet, looking back on the years of work which amounted to this dissertation, I am realizing that in many ways, this is the first difficult thing that I have ever done, and that I cannot imagine having been able to do it on my own. I therefore have a long list of “thank you”s. I have done my best to make it as complete as possible, but, since as is often the case in academia, writing this page was done under a tight deadline, I am certain I have omitted some names. For that, I apologize.

I would like to start by thanking my advisor, Prof. Rachid Guerraoui. I have learned a lot from Rachid, both as a researcher, and as a person. In particular, Rachid taught me how to truly get to the heart of the problem I’m working on, to always mind the difference between good research and good engineering, and how in an academic career, you can be very productive without having to sacrifice the things you like outside of work.

During my PhD, I had the opportunity of spending two summers in the research groups of VMware and Microsoft, where I started collaborations with Marcos Aguilera and Dahlia Malkhi (at VMware Research), and Aleksandar Dragojevic (at Microsoft Research). Each of them has taught me a lot and has shaped my perspective on research. The results of our productive collaborations are included in this dissertation. For all that, a big “thank you”!

I also want to thank the colleagues with which I collaborated during these years. I owe much of the quality of this dissertation to them as well. In particular, I want to thank Vasileios Trigonakis, with whom I had a very fruitful collaboration during the first couple of years of the PhD. I also want to thank Maysam Yabandeh, with whom I worked when I first joined the lab as a Master student, as well as Igor Zablotchi, with whom I collaborated on my last paper.

I would then like to acknowledge the members of the jury that accepted my PhD thesis: Dr. Marcos Aguilera, Prof. George Candea, Prof. Pascal Felber, and Prof. Willy Zwaenepoel. Many thanks for reading my dissertation, and for the insightful questions and comments. In particular, I am grateful to Marcos for traveling all the way from California to participate in my private defense.

I want to thank EPFL, VMware, the Swiss National Science Foundation, and the European Research Council for financially supporting my research.

Dealing with administrative tasks would would have been much more difficult without the help of Kristine Verhamme and France Faille, and my practical experiments less successful without the help of our system administrator, Fabien Salvi.

Also, many thanks to all my colleagues in the Distributed Programming Laboratory, which have contributed to a great working environment. I consider myself particularly lucky to be able to call several current and former members of LPD not just colleagues, but close friends. Without Vasilis, David, George, Adi, Matej, Mihai, Oana, these 5 years of PhD would not have been nearly as fun as they were.

Fortunately, during these years, I also got to (sometimes) go outside the research lab. I am thankful to Ioana, Cristina, Jean-Noel, Radu, Flaviu, Sonia, Tudor, Virgil, Razvan, Vlad for the good conversations, the drinks, the parties, the hikes, but especially for reminding me that at times it's worth putting work aside.

Last, but not least, I would like to thank my parents, Elena and Ioan, and my sister, Adina. This dissertation would not have happened without their unconditional love and support. I am deeply grateful.

Lausanne, September 7, 2017 Tudor David

Preface

This thesis was conducted in the Distributed Programming Laboratory at EPFL, under the supervision of Prof. Rachid Guerraoui, between 2012 and 2017.
The main results presented in this dissertation were first introduced in the following publications:

1. Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. *Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures*, 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Istanbul, Turkey, 2015.

2. Tudor David and Rachid Guerraoui. *Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free*, 28th Symposium on Parallelism in Algorithms and Architectures (SPAA), Monterey, CA, 2016.

3. Marcos K. Aguilera, Tudor David, and Rachid Guerraoui. *Locking Timestamps Versus Locking Objects.* Technical report (EPFL-REPORT-229425), 2017.

4. Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. *Log-free Concurrent Data Structures.* Under submission, 2017.

Besides the aforementioned publications, which constitute the bulk of this dissertation, I was involved in the following papers:

1. Tudor David, Rachid Guerraoui and Vasileios Trigonakis. *Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask*, Symposium on Operating Systems Principles (SOSP), Farmington, PA, 2013.

2. Tudor David, Rachid Guerraoui and Maysam Yabandeh. *Consensus Inside*, 15th International Middleware Conference (Middleware), Bordeaux, France, 2014. Best Paper Award.

Abstract

The increase in the number of cores in processors has been an important trend over the past decade. In order to be able to efficiently use such architectures, modern software must be scalable: performance should increase proportionally to the number of allotted cores.While some software is inherently parallel, with threads seldom having to coordinate, a large fraction of software systems are based on shared state, to which access must be coordinated. This shared state generally comes in the form of a concurrent data structure. It is thus essential for these concurrent data structures to be correct, fast and scalable, regardless of the scenario (i.e., different workloads, processors, memory units, programming abstractions). Nevertheless, few or no generic approaches exist that result in concurrent data structures which scale in a large spectrum of environments.

This dissertation introduces a set of generic methods that allows to build - irrespective of the deployment environment - fast and scalable concurrent data structures.

We start by identifying a set of sufficient conditions for concurrent search data structures to scale and perform well regardless of the workloads and processors they are running on. We introduce “asynchronized concurrency”, a paradigm consisting of four complementary programming patterns, which calls for the design of concurrent search data structures to resemble that of their sequential counterparts. Next, we show that there is virtually no practical situation in which one should seek a “theoretically wait-free” algorithm at the expense of a state-of-the-art blocking algorithm in the case of search data structures: blocking algorithms are simple, fast, and can be made "practically wait-free".

We then focus on the memory unit, and provide a method yielding fast concurrent data structures even when the memory is non-volatile, and structures must be recoverable in case of a transient failure. We start by introducing a generic technique that allows us to avoid doing expensive writes to non-volatile memory by using a fast software cache. We also study memory management, and propose a solution tailored to concurrent data structures that uses coarse-grained memory management in order to avoid logging. Moreover, we argue for the use of lock-free algorithms in this non-volatile context, and show how by optimizing them we can avoid expensive logging operations. Together, the techniques we propose enable us to avoid any form of logging in the common case, thus significantly improving concurrent data structure performance when using non-volatile RAM.

Finally, we go beyond basic interfaces, and look at scalable partitioned data structures implemented through a transactional interface. We present multiversion timestamp locking (MVTL), a new genre of multiversion concurrency control algorithms for serializable transactions. The

key idea behind MVTL is simple and novel: lock individual time points instead of locking objects or versions. We provide several MVTL-based algorithms, that address limitations of current concurrency-control schemes.
In short, by spanning workloads, processors, storage abstractions, and system sizes, this dissertation takes a step towards concurrent data structures that are *universally scalable*.

Key words: Concurrent data structures, multi-core, transactions, scalability, performance, non-volatile RAM;

Résumé

Les dix dernières années ont vu croître le nombre de coeurs au sein des processeurs. Afin de profiter au mieux de ces nouvelles architectures, les algorithmes actuels doivent être scalables : leur performance doit croître proportionnellement avec le nombre de coeurs.

Bien que certains algorithmes sont intrinsèquement parallèles (les différents fils d'exécution n'ayant presque pas besoin de se coordonner entre eux), la plupart des systèmes logiciels ont un état partagé, les accès auquel devant être coordonnés. Cet état partagé se présente communément sous la forme d'une structure de données partagée. Il est donc essentiel que ces structures de données soient, non seulement correctes, mais également rapides et scalables, et ce, quel que soit le scénario. Pourtant, il existe très peu, et dans certains cas il n'existe aucune, méthode générique permettant de produire des structures de données qui soient scalables sur un large éventail d'architectures.

Dans ce document, nous présentons un ensemble de telles méthodes génériques : des méthodes de conception de structures de données rapides et scalables, indépendemment de l'environnement où elles sont déployées.

D'abord, nous identifions les conditions nécessaires sous lesquelles des structures de données concurrentes peuvent être rendues scalables et efficaces indépendemment de la charge du système et de la plateforme d'exécution. Nous définissons la notion de "concurrence asynchronisé (ASCY)", un paradigme comprenant quatre motifs complémentaires de programmation de structures de recherche concurrente de données (Concurrent Search Data Structures, CSDS), et dont l'objectif est de rapprocher au mieux l'implémentation de la structure de donnée concurrente d'une implémentation séquentielle. Nous montrons ensuite qu'il n'existe aucune situation concrète où un algorithme wait-free en théorie surpasse les meilleurs algorithmes bloquants vis-à-vis des CSDS : les algorithmes bloquants sont simples, rapides, et wait-free "en pratique".

Nous nous concentrons ensuite sur la mémoire. Nous proposons une méthode donnant lieu à des structures de données rapides même quand la mémoire n'est pas volatile, et que les structures de données doivent être réparables en cas de fautes transitoires. Nous recommandons, arguments à l'appui, l'usage d'algorithmes lock-free, et montrons comment l'optimisation de ces derniers permet d'éviter des opérations coûteuse de logging. Nous étudions aussi la gestion de la mémoire, et proposons une solution adaptée aux structures de données qui évite une large partie des opérations de logging. Prises ensembles, les techniques que nous proposons permettent de conserver les performances des structures de données concurrentes lorsque la mémoire utilisée est une NVRAM.

Enfin, au-delà des structures de données en mémoire partagée, nous portons notre attention aux structures de données partitionnées implémentées à travers une interface de type transactionnelle. Nous proposons un système de verrou à base d'estampillage multi-versionnelle (multiversion timestamp locking, MVTL), un nouveau type d'algorithmes de contrôle de concurrence pour les transactions sérialisables. L'idée-clé derrière MVTL est simple et inédite : verrouiller des points temporels isolés au lieu de verrouiller des objets ou des versions. Nous proposons plusieurs algorithmes fondées sur MVTL qui outrepassent les limites des schémas de contrôle de concurrence actuels.
Par l'étude des charges de travail, des processeurs, des abstractions de stockage de données et de tailles de système, cette thèse établit un pas vers des structures de données concurrentes qui soient *universellement scalables*.

Mots clefs : structures de données concurrentes, multi-coeur, transactions, scalabilité, performance, NVRAM

Contents

List of Figures

List of Tables

1 Introduction

Data structures represent one of the main concerns of Computer Science. They are studied from the very first CS courses, and are at the core of virtually every modern software system. Linked lists, queues, stacks, search trees, hash tables, and skip lists are concepts that are familiar to every computer scientist.

Traditionally, we study sequential algorithms for such data structures: these algorithms assume a single thread of execution, and have little or no concern for storage. Data structures as means of organizing data are only useful if they are efficient for the user's goals. Thus, an important amount of time is dedicated to studying and optimizing the performance of their operations in terms of sequential complexity. However, the ultimate purpose of any computer science abstraction is to be useful in practice. In particular, this means that any practical algorithm for a data structure needs to be adapted to the hardware on which it is running.

If we look at the simplest blueprint of modern computers, that is, the von Neumann architecture [160], we note that there are two main components: the processor, and the memory unit. Therefore, any data structure deployed in practice needs to at the very least be adapted to these two components. The model of a single processing unit largely reflected the state of affairs in practice for more than half a century, until the advent of multi-cores in the early 2000s. Nowadays, a processor invariably has several cores, i.e., it is a machine that is able to run several threads of execution at the same time. For a data structure running on such a machine, it is important to be able to efficiently use all these concurrent threads at its disposal. Moreover, even more recently, memory units that are able to maintain their contents across transient failures have been introduced, potentially making application durability less costly. To take advantage of this capability, a data structure algorithm should store its state in a way that allows recovery, while maintaining as much of the performance of its volatile counterparts as possible.

Nevertheless, while a large amount of work has been dedicated to specialized and intricate data structure algorithms, there are few or no *generic* approaches that help us design data structures that are adapted to modern hardware (processors, storage techniques, or system structures) in such a way as to maximize performance. Hence, a developer trying to design a data structure for her system from scratch has little guidance on how the algorithm should look like in order to perform as she expects. One of the aims of this dissertation is to take a step towards solving this problem.

Additionally, when concurrency is an issue and more complex computations need to be performed on a data structure, whether in a multi-core or a distributed system, it is extremely convenient to provide the user with a mechanism that allows one to access the structure not only through its basic interface, but rather through an abstraction that groups several operations and executes them in an atomic, all-or-nothing manner. This abstraction is referred to as a transaction. Beyond its convenience for the user, a transactional system must of course provide performance by allowing the maximum number of concurrent transactions to proceed.

This thesis aims to provide generic approaches for fast and scalable data structures in all these environments and programming models. In the following, we provide a brief overview of the approaches we propose for each of them.

1.1 Fast and Scalable Concurrent Data Structures

We first concern ourselves with adapting data structures to modern processors. Traditionally, processors only had a single processing unit (core), to which sequential data structure algorithms were well suited. However, for more than a decade now, Dennard scaling [38], which allowed core frequency to increase from generation to generation, has stalled. With Moore's law [124], which predicts that the number of transistors per die doubles roughly every two years, still applying, computer architects have used these additional transistors that were becoming available with each CPU generation to place several cores on the same die. Such processors are known as *multi-cores*.

Software systems, which relied on continuous increases in frequency to become faster with each CPU generation essentially for free, now need to harness the power of all the cores on a processor in order to achieve this. In some cases this is trivial: execution can simply be split in several independent instances that never or only rarely need to communicate with each other. Such workloads are called "embarrassingly parallel". However, such cases are not very common: an important number of widely-used systems have some shared state at their core (e.g. [47, 59, 114, 122, 123]), state that is frequently accessed by a significant fraction of the execution threads in the system. Most often, this state is in the form of a data structure. We refer to data structures that must support access from multiple execution threads as *concurrent data structures*.

Of course, when multiple threads are accessing a concurrent data structure and potentially modifying its state at the same time, their access to the structure needs to be coordinated in such a way as to ensure correctness. Essentially, operations should appear to execute as if they were run in some sequential order, and the effects of each operation need to become visible at some instant between its invocation and its return, a property known as *linearizability* [75].

However, guaranteeing such correctness properties comes at a price: due to the need to coordinate between threads, total output is not always proportional to the number of threads concurrently running the data structure algorithm. The evolution of a concurrent data structure's performance as the number of cores dedicated to its execution (i.e., the number of concurrent execution threads accessing it) increases is called *scalability*.

Ideally, a concurrent data structure should exhibit good base performance, and *linear scalability*: performance that increases proportionally to the number of allotted cores. However, designing correct, fast, and scalable concurrent algorithms is a notoriously difficult and error-prone task. A large amount of work has been dedicated to the design of highly-tuned algorithms for such data structures [19,35,40,43,46,54,64,66,70,77,94,99,116,129,137,150,151]. Such algorithms might scale only under certain conditions and workloads, and when they do scale, it is not exactly clear what precisely it is that enables them to do so. Thus, no clear picture exists on how an algorithm that is fast and scalable regardless of the situation should look like. Even though sometimes, data structure implementations directly provided by a library are indeed sufficient, more often than not, when the goal is a high-performance system, concurrent data structures customized to the need of the particular system need to be designed and implemented from scratch. In Chapter 4 we take a step towards simplifying this task for an important class of data structures: search data structures. We identify and propose *Asynchronized concurrency*: a set of general patterns that when applied result in highly scalable implementations of concurrent search data structures. Moreover, these implementations are *portably scalable*: they scale under a wide variety of workloads and platforms. Intuitively, the idea is to design search data structures that resemble their sequential counterparts as much as possible, with stores only to cache lines semantically associated with updated nodes. Reads should not do any stores, nor should updates while traversing the data structure to reach the point where the change should be made. Only a small neighborhood of the data structure, containing the node to which the update actually refers to should be modified. We show that it is possible to apply all these patterns to a variety of existing state-of-the-art algorithms, both lock-based and lock-free, and that by applying them we improve the algorithms' performance. In addition, we showcase how new algorithms can be designed starting from these principles. Thus, Asynchronized concurrency offers the programmer both a way to optimize existing concurrent search data structures, and simplifies the task of designing new ones.

An important characteristic of concurrent algorithms is the theoretical *progress guarantee* that they provide. The progress guarantee tells us what we can expect from the behavior of individual threads under concurrency. An open question in the context of concurrent data structures is the extent to which the theoretical progress guarantee provided by the algorithm matters in practical situations. The stronger the progress guarantee, the more complicated an algorithm is to design. In addition, wait-free algorithms (i.e., the algorithms providing the strongest progress guarantee) usually suffer in terms of throughput in the common case. It has previously been shown that a large class of lock-free algorithms actually behave like wait-free algorithms in practice [3]. The question we raise is whether blocking implementations can also behave in a practically wait-free manner. One would expect a negative answer given the frequent use of mutual exclusion and the varied conditions under which these algorithms must run. As we show in Chapter 5, the answer is however *yes* for search data structures. To be able to obtain this answer, we first introduce a set of performance metrics capable of capturing wait-free behavior in practice: the absence of significant delays due to concurrency. We then look at state-of-the-art implementations of concurrent data structures under a large variety of workloads and conditions. The conclusion we draw is that blocking search data structures exhibit virtually no significantly delayed requests. We also give a theoretical intuition for the result, through an analogy with the birthday paradox. The most problematic case for blocking algorithms can be frequent interrupts, in particular context switches. We show how best-effort Hardware Transactional Memory (HTM) can be leveraged to allow blocking algorithms to maintain practically wait-free behavior in this case. By changing only the locking code to use HTM, and by taking advantage of the seeming inconvenience of hardware transactions aborting upon interrupts, threads virtually never hold locks while preempted. In summary, our work helps programmers to better determine when a blocking implementation is sufficient for their needs, thus potentially simplifying the task of implementing a concurrent data structure to a substantial degree.

1.2 Fast and Scalable Durable Data Structures

Our second concern is adapting data structures to current memory units. Traditionally main memory used DRAM, and was volatile: its contents would have been lost in case of a power failure. However, non-volatile RAM (NVRAM), an emerging technology, promises latencies and a programming model similar to DRAM, while also ensuring durability. One of the main challenges when working with NVRAM is ensuring the complete recoverability of a system in case of a restart, while not significantly degrading performance at runtime when compared to structures running on DRAM. Intuitively, the challenge comes from the fact that by default, we do not have control over the order in which data is evicted from the write-back caches, and thus this has to be handled explicitly trough costly write-back instructions. Often, in order to ensure the correctness of durable state, we have to make sure stores are reliably written to NVRAM before proceeding in our application. Such operations are extremely expensive. Logging, which was extensively used in previous work [23, 25, 29, 78, 88, 92, 96, 105, 113, 159], is particularly problematic from this point of view.

As a solution in the context of (concurrent) data structures, if these have to be durable, in Chapter 6 we propose using lock-free data structures. The theoretical property of lock-freedom states that at least one process in a system should be able to make progress regardless of the current state. In particular, this means that processes should never leave the system in an inconsistent state. This allows us to avoid expensive logging in the data structure algorithm. We show how data structures for NVRAM can be efficiently implemented, and propose generic techniques to optimize them further and to minimize the time spent waiting for stores to be persisted. In addition, we also look at the problem of memory management. Because of frequent logging operations, this is a particularly thorny issue from a performance standpoint when working with NVRAM. We propose a solution tailored to concurrent data structures, which by keeping track of memory at a coarser granularity avoids overheads at runtime. This does mean that upon a restart after a failure has occurred, some work will have to be performed to avoid persistent memory leaks. Nevertheless, by taking advantage of our knowledge of the concurrent data structure algorithms, we are able to keep this recovery time manageable.

Our work thus presents the first methodology for designing durable concurrent data structures that avoids logging in the common case, and therefore brings overall data structure performance closer to that of volatile memory counterparts.

1.3 Fast and Scalable Transactional Data Structures

We also look at the programming abstractions through which data structures are manipulated. Ideally, a user can express her computations using individual reads and writes. However, it is at times significantly more convenient to treat entire portions of code accessing a structure as a single atomic unit of sequential code. This abstraction is called a *transaction*. It is then the job of the underlying system to execute the transaction in an all-or-nothing manner, and in isolation from any concurrency (that is, all the operations in the transactions should appear to execute at the same point in time relative to operations of other transactions). The correct execution of such transactions is ensured through a concurrency control protocol. Aside from correctness, the performance of such transactional systems is of course essential. In general, concurrency control protocols are either pessimistic (they lock the required objects before doing any computations), or optimistic (they perform work speculatively, and then try to commit it). While pessimistic concurrency control protocols may limit parallelism, optimistic protocols may result in large numbers of aborts.

To mediate these downsides, in Chapter 7 we introduce a new paradigm for concurrency control: *Multiversion Timestamp Locking (MVTL)*. Unlike previous approaches, MVTL works not with objects or versions of objects, but at a much finer granularity - namely individual points of logical time. Along with the increased concurrency that comes with this finer-grain locking, MVTL transactions are able to explore several serialization points at the same time. We first prove the correctness of the generic MVTL framework, and then propose several MVTL-based algorithms, that circumvent some of the issues inherent in previous protocols. In particular, we provide an MVTL algorithm that, as we show, commits more workloads than previous related algorithms. We also propose algorithms that provide prioritized transactions, and avoid ghost and serial aborts. In addition, we provide an MVTL algorithm aimed at a distributed setting. While here we merely develop the theoretical foundations of MVTL, we believe it is in this environment that MVTL would shine in practice. These algorithms represent but a starting point: MVTL can be the basis of a large spectrum of new algorithms.

1.4 Contributions

The main aim of this thesis is to make the design of fast and scalable data structures a somewhat less onerous task for the programmer. To this end, we focus on methods that are generic in the sense that they can be applied to a large spectrum of data structures and concurrent objects, and we target environments that are both varied and relevant in today's technology landscape. In particular, we focus on modern processors and system structures, on next-generation memory units, as well as on programming models that go beyond individual operations. In this sense, we seek *universality*. More concretely, this dissertation makes the following intellectual contributions:

- Asynchronized concurrency: a set of general patterns that facilitate the design an implementation of portably scalable concurrent search data structures;
- A better understanding of the precise scenarios under which blocking concurrent data structures should be used;
- A generic methodology for the design of efficient lock-free concurrent data structures and memory management for the NVRAM environment;
- Multiversion Timestamp Locking, a family of concurrency control algorithms that address several shortcomings of previous approaches.

1.5 Roadmap

The rest of this dissertation is organized as follows:

- Part I provides some background (Chapter 2), and discusses published work related to the topics covered in this thesis (Chapter 3);
- Part II presents Asynchronized concurrency, showing how to design portably scalable CSDSs (Chapter 4), and argues that in the case of CSDSs, blocking algorithms may behave practically wait-free (Chapter 5);
- Part III shows how concurrent data structures for non-volatile RAM can be efficiently designed;
- Part IV discusses Multiversion Timestamp Locking, a new paradigm for concurrency control;
- Part V presents potential future work and concludes the dissertation.

Preliminaries Part I

2 Background

In this section, we present some background related to the topics which will be discussed in later chapters. We start with a discussion of modern architectures and the need for scalable concurrent software. We continue by discussing search data structures, and then go into more details into the design of concurrent algorithms for such structures. We present new fast non-volatile storage technologies, and we end with a discussion of concurrency control for transactions.

2.1 Modern multi-cores.

For more than a decade, Dennard scaling [38] has ground to a halt, thus preventing computer architects from further increasing core frequency. However, Moore's law [124] has still been in effect. As a result, computer architects have been using the additional transistors that were becoming available with each CPU generation to increase the number of cores placed on a single processor. This fundamentally changed the way software is designed. Before hitting the frequency wall, software was simply becoming faster with each CPU generation simply because of the increase in the clock frequency. Since then, software has mainly achieved better performance through scalability: the ability of efficiently using the available cores. A multi-core software system is said to be scalable if its output is proportional to the number of cores running that system. In particular, communication between cores must not hamper scalability.

The standard means of communication between the different cores is through shared main memory. As there are two orders of magnitude difference between the operating speed of a core and DRAM, traditionally several layers (generally 3) of caches are present between main memory and a core. The closer a cache is to the core, the smaller its capacity, and the lower its latency. Two levels of cache are generally private to each core, while the third, larger cache is shared among the cores of a processor. The granularity at which the memory system operates is the cache line, which on modern architectures is generally 64 bytes in size.

When multiple cores are working with the same cache line, correctness is ensured by the cache coherence mechanism, which ensures that everyone sees the same value for a particular cache line. Most modern processors use variants of the MESI cache coherence protocol to this effect. Briefly, a cache line can be in one out of several states:

- *Modified.* This is the only correct copy of the data. Every other copy is stale.
- *Exclusive.* This is the only copy of the data (except for main memory).
- *Shared.* This is one out of several correct copies of the data in the system.
- *Invalid.* The data held in this copy of the cache line is not usable.

When a core performs a load or store on a particular piece of data, the corresponding cache lines must be brought in the core's private caches in a state that allows the operation.

The time it takes a core to fetch a cache line depends on where the current valid copies are found. Typical values (in cycles) are 5 for the first-level cache, 10-15 for the second-level cache, 40 for the last-level cache, and more than 100 for RAM [36]. Modern servers often comprise more than one processor. If the latest copy of a piece data is currently in the memory of another processor, the latency to read or write it can be several hundred cycles.

Given the large latency differences depending on the state and location of data it is essential for scalable concurrent software keep data as much as possible in shared state, and avoid expensive data accesses whenever possible.

In general, modern architectures provide atomic Compare-and-Swap (CAS), as well as atomic Fetch-and-OP (where OP is an arithmetic or logical operation) instructions. These instructions usually operate at word-size granularity, and are the building blocks for higher-level software synchronization mechanisms and concurrent objects. As CAS has consensus number n [69], it is sufficient to implement any concurrent object.

Several newer architectures also provide best-effort hardware transactional memory(HTM). The user may specify a region of code which should be executed atomically, and the hardware then detects if any other core reads or writes the transaction's data while it is executing. Hardware transactions may however abort for several reasons, including data conflicts, capacity limits (the write-set of a transaction must be in the L1 cache of the core executing the transaction), interrupts (including context switches). Therefore, there is a non-negligible possibility of starvation. Thus, to prevent this, one must have to provide a non-transactional fall-back path, which is taken if the transaction fails a specified number of times.

2.2 Search data structures.

In this dissertation, while we will discuss a large variety of data structures, we place a particular emphasis on *search data structures*.

A search data structure consists of a set of elements and three main operations: *search*, *insert*, and *remove*. An element consists of a *key* and a *value*. The key uniquely identifies the element in the set. The value is often a pointer to a structure that contains the actual data.

The three main operations have the following semantics:

- *search(key)* looks for an element with the given key; if it is found, returns the value of the element, otherwise returns `NULL`.
- *insert(key, val)* attempts to insert a new element in the data structure; the insertion is successful iff there is no other element with the same key.
- *remove(key)* attempts to remove the element with the given key; it is successful iff such an element exists.

A common attribute of search data structures is that the updates (insertions and removals) comprise two distinct phases. First, they *parse* the structure until the update point is reached. Then the actual modification is attempted.

2.3 Concurrent search data structures.

We study the most basic and commonly-used CSDSs: *linked lists, hash tables, skip lists,* and *binary search trees (BSTs)*. We are interested in *linearizable* [75] implementations of the aforementioned data structures.

Linearizability significantly simplifies the task of the programmer for it requires every operation on the search to appear to execute in a single point in time, despite concurrency. To ensure linearizability, concurrent operations on the same data structure from different threads need to be *synchronized*: this is achieved using low-level operations provided by the underlying hardware.

It is common to classify linearizable implementations based on whether and how they make use of locks [74]. One can distinguish *fully lock-based, hybrid lock-based,* and *lock-free* [54] algorithms. Fully lock-based algorithms use locks to protect all three operations and are *blocking* [74], in the sense that a thread might have to wait for a lock to be released. Hybrid lock-based (henceforth called "lock-based") algorithms use locks to protect the actual updates to the structure. They are otherwise lock-free. For instance, a removal might parse the list (in a lock-free manner) until the target node is found, get the lock, and then do the actual deletion. Hybrid algorithms are also blocking. Finally, lock-free algorithms do not use locks and are *non-blocking* [54, 71]. They typically use the underlying atomic operations, such as compare-and-swap (CAS), provided by the hardware.

2.4 Memory reclamation.

The problem of memory reclamation is inherent when working with concurrent linked data structures in an unmanaged language, such as C. Essentially, when a node is removed from a data structure, it cannot be freed immediately, as other concurrent operations might still have references to it. Thus, a mechanism is needed in order to be able to safely determine when a certain data structure node can be freed. Ideally, this should be achieved without incurring significant time or space overheads. Additionally, memory should be freed as soon as possible. In practice, the most common solutions to this problem are hazard pointers [117], and epoch or quiescence based memory reclamation [54].

Briefly, when using hazard pointers, each thread holds a small number of pointers to nodes it may currently have references to (the thread's *hazard pointers*). Only a thread can write its own hazard pointers, but any thread can read them. When a node is unlinked and its memory should be reclaimed, one must first ensure that no hazard pointers concerning that particular node exist. An advantage of hazard pointers is the fact that the method provides lock-freedom. However, it is known to exhibit scalability issues, as the hazard pointers of all threads need to be collected.

In epoch-based memory reclamation, each thread is assigned an epoch number. In the most common implementation, threads increment their epoch numbers at the very beginning of an operation, as well as at the end. When a node is unlinked from the data structure, it is not immediately freed, but rather handed over to the memory reclamation mechanism, which also stores the epochs each thread was in at the moment the node was unlinked. When the epochs of all the threads that had operations in progress at the moment the node was unlinked have advanced, (thus guaranteeing they have finished any operations in which they might have had references to the unlinked node), the memory of the node can be safely reclaimed. While scalable, this method is not lock-free: a single thread that does not may progress in its execution can prevent memory from being freed.

In this work, we will use epoch-based memory reclamation, as in the case of the structures we consider, this technique tends to incur a smaller run time penalty.

2.5 Fast non-volatile memory.

Traditionally, storage has either been fast, but volatile (i.e., the data stored is lost in case of a power failure), such as is the case with DRAM, or non-volatile, but slow, such as is the case with flash storage for instance.

However, more recently, a new class of storage that promises both low latency and non-volatility is becoming available. Candidate technologies include Memristors [147], Phase Change Memory [100, 138], and 3D XPoint [120]. Latencies are expected to be somewhat larger than those of DRAM, with writes being somewhat more expensive than reads. Table 2.1

	L1	L2	LLC	DRAM	PCM	Memristor
Read	2	6	15	50	50-70	100
Write	2	6	15	50	150	100

Table 2.1 – Caches, DRAM, and NVRAM (projected) latencies (ns).

provides a comparison of expected PCM and Memristor latencies [143, 157, 170] with those of DRAM and caches.

As opposed to other forms of non-volatile storage, these new technologies will be byte-addressable, and from a programming model standpoint will be mappable to a region of virtual memory. Therefore, accessing it would then essentially be no different than accessing DRAM. In the rest of this dissertation, we refer to such fast, byte-addressable memory that maintains its contents across power failures as *non-volatile RAM (NVRAM)* or *persistent memory*.

From a programming perspective, one of the main difficulties when working with NVRAM stems from the fact that by default, we do not control the order in which cache lines are evicted from the write-back caches, and actually written to NVRAM. Thus, when performing several stores, even when we ensure that they appear in the proper order in the caches (for which we just need to mind the memory consistency model of our processor), it may still be the case that they appear in a different order from the point of view of the persistent memory. If a restart occurs before all the stores are persisted, we might end up with incorrect state (such as, for example, pointers to uninitialized memory). The only way to ensure that stores appear when we want them to and in the order we want them to in persistent memory is by using special instructions that force data to be written back to NVRAM.

On current Intel processors, one can ensure that a cache line is written to memory by using the `clflush` instruction. This instruction invalidates the cache line. In addition, such flushes are ordered with respect to one another: if we issue two flushes, the second one will only start executing once the first one has completed. On upcoming processors, Intel is introducing two new instructions targeted at non-volatile RAM [80, 82]. The first one is `clflushopt`. This instruction is not strongly ordered. Therefore, we can issue a `clflushopt` before the previous one has finished; thus, multiple such instructions can proceed in parallel. `clflushopt` is only ordered with respect to store fences. The second new instruction is `clwb`, which only does a write-back to memory, without invalidating the cache line from the cache hierarchy. Like `clflushopt`, it is only ordered with respect to store fences (or to instructions that have an implied store fence, such as, for example, Compare-and-Swap).

In the rest of this dissertation, when working with NVRAM, we will use the `clwb` instruction to ensure that cache lines are written to memory. An important observation is the fact that since these instructions are unordered with respect to one another, when the relative order in which they are written to persistent memory is not particularly important, batching several write-back instructions is beneficial for performance [81]. We refer to one or more write-back instructions followed by a store fence as a *sync* operation.

When working with durable memory, we assume a model transient failures may occur: a machine may fail at any point in time (due to, for example, a power failure), but can be expected to be restarted and resume normal operation.

We require, as is commonly the case in practice, that only the data stored in durable main memory is still available after a crash. The data that was in a processor's registers or in the write-back caches at the moment of the crash is not available after a restart. Nevertheless, our approach of using lock-free algorithms would be highly beneficial on an architecture that maintains enough residual energy to flush the register and the caches in case of a power failure as well.

Moreover, similar to related work [16], we assume that a region of non-volatile RAM can be mapped to the same region of virtual memory across restarts. Alternatively, if this is not the case, we can update persistent pointers at recovery time.

In the context of concurrent software, it is important to define correctness conditions in the face of restarts. In particular, we require a framework that allows us to reason about the state that is stored in NVRAM after a crash. For this purpose, we use the concept of *durable linearizability* introduced by Izraelevitz et al. [89]. Essentially, a durably linearizable implementation guarantees that the state of the data structure after a restart reflects a consistent operation subhistory that includes all the completed operations at the moment of the crash.

2.6 Concurrency control protocols.

A large number of applications express sections of code that need to be executed atomically as *transactions*. This is true in databases, in distributed data stores, as well as in concurrent software, such as software transactional memory (STM). Transactions represent a particularly convenient abstraction for application users, which do not need to concern themselves with concurrency and potential interleaving of operations. The user merely has to specify a portion of code that is to be executed in an atomic manner, and the underlying system ensures the transaction is correctly executed.

In terms of isolation, transactional systems may offer one out of several guarantees. Popular alternatives include *snapshot isolation*, *serializability*, and *strict serializability*. Snapshot isolation guarantees that the read set of a transaction reflects a consistent view of the objects in the system at a certain point in time. While systems providing snapshot isolation can be efficiently implemented, they are prone to anomalies, such as the write skew anomaly [15]. Serializability guarantees that the effect of the execution of transactions is as if they were executed one a time, in some particular order. Strict serializability requires that the serialization order of transactions reflect their execution order.

In the rest of this dissertation, we concern ourselves with providing serializable transactions. Where appropriate, we also discuss the necessary changes for strict serializability.

The way in which potential conflicts between concurrent transactions are handled is determined by the *concurrency control protocol*. Broadly speaking, there are two types of concurrency control protocols: pessimistic and optimistic protocols. Pessimistic protocols acquire locks on the items transactions before executing them. Thus, when the transaction is executed, it is certain to succeed. Such protocols are pessimistic in the sense that they assume that conflicts will occur, and thus acquire locks in the first phase in order to be sure of the success of the transaction. One example of such protocol is two-phase-locking (2PL) [15].

Optimistic concurrency control protocols on the other hand generally execute a transaction in a speculative manner, and only when actually trying to commit the results verify whether the transaction can be successfully committed, or if some other concurrent transaction has modified the concerned state in the meanwhile. These protocols are optimistic in the sense that they assume that conflicts will not generally happen, and thus most transactions will be successful.

In terms of performance, optimistic protocols are more appropriate when conflicts are rare, while pessimistic protocols shine in the face of frequent conflicts.

Concurrency control protocols may work with one or several versions, indexed by the (logical) time when they were created. The precise version a transaction accesses is determined by its rank in the serialization order. In general, working with several versions of the same object enables more concurrency, but incurs storage overheads. We discuss multiversion concurrency control in more detail in Chapter 7.

3 Related Work

In this chapter, we discuss published work that is related to the ideas introduced in this dissertation. We start by presenting work that focuses on various aspects of the design of concurrent data structures, as well as their use in systems (Section 3.1). We then discuss previous work focusing on the design of software for fast, persistent memory, as well as other software constructs that are similar to our link cache (Section 3.2). We finish by presenting work on concurrency control, both in the context of distributed transactions, as well as in the context of software transactional memory (Section 3.3).

3.1 Concurrent data structures

CSDS design. A large body of work has been dedicated to the design of efficient CSDSs [19,43, 46,54,66,74,77,112,115,116,129,137,148,153,155,162]. These efforts usually aim at optimizing a specific CSDS (e.g., BST) in terms of throughput, for a specific set of workloads and platforms. In contrast, ASCY is a paradigm that targets the scalability of various CSDSs across different platforms, for various workloads, and according to several performance metrics.

Memory reclamation is of key importance in CSDSs [18,41,44,72,117]. Various techniques have been proposed, such as *quiescent state* [40,64,65], *epochs* [54], *reference counters* [41,58,155], *pointers* [18,72,117], and, recently, hardware transactional memory [4,44]. ASCYLIB uses an epoch-based allocator that reduces the performance overheads.

Read-copy update (RCU) [112] is a concurrent programming technique that is heavily used in the Linux Kernel [9]. In short, RCU guarantees that readers always find a consistent view of the data structure. Writers perform atomic updates, after which, in the case of removals, they wait for all concurrent readers to finish in order to perform memory reclamation. Relativistic programming [153] is a technique that is related to RCU and maintains efficient reads even with large updates to the data structure (e.g., a resize).

Recently, a significant amount of work has been dedicated to improving the ease of use and concurrency of the RCU pattern. Arbel and Attiya [7] present an RCU-like search tree which allows concurrent updates. While out-performing classic RCU-based structures, the design is still shown to lag behind other state-of-the-art CSDSs that are closer to ASCY, in particular in write-intensive scenarios. Arbel et al. [8] propose predicate RCU, which reduces the waiting time writers have to wait for readers. Matveev et al. [110] introduce Read-Log-Update, a version of RCU that is enhanced with techniques inspired from software transactional memory, simplifying its usage, and allowing more concurrency with writers. Nevertheless, while improving on RCU, these approaches still do not match the performance and scalability of tailor-made, high-performance concurrent data structures. In general, RCU-like approaches purposefully optimize the performance of search operations at the possible expense of updates. Our ASCY$_1$ pattern is similar in the sense that it dictates that readers must be unaware of concurrency. However, our three remaining patterns achieve the benefits of sequential reads without sacrificing the performance of updates.

Hunt et al. [79] study the energy efficiency of lock-free and lock-based concurrent data structures (queues and linked lists). They find that lock-free algorithms tend to be more energy efficient than their lock-based counterparts. We observe that in the context of CSDSs, as long as the algorithms apply ASCY, there is no inherent difference between lock-free and lock-based algorithms. Essentially, this is because the ASCY patterns help reduce the number and the granularity of locks.

Gramoli [61] performs a study synchronization techniques. He observes that while generally offering good performance, the validate-lock-validate technique advocated by ASCY$_3$ in the context of blocking algorithms at times results in high performance variability.

More recently, Trigonakis and Guerraoui [63] generalize the method used in BST-TK, and propose Optik, a design pattern for CSDSs that uses trylocks based on version numbers, which allow the merging of locking and conflict detection. Based on ASCY, Antoniadis et al. [6] introduce sequential proximity, a theoretical approach that allows one to determine if a CSDS algorithm is close to its sequential counterpart, and thus scalable. Similar to what we proposed through out ASCY patterns, Gibson and Gramoli [57] theoretically show that excessive helping can be detrimental to performance.

Calciu et al. [22] propose Node Replication, a method that automatically transforms a sequential data structure algorithm into a concurrent one. Node Replication maintains several replicas of the data structure in a NUMA-aware fashion, and ensures consistency of the replicas though a shared log. Node Replication is shown to perform well in scenarios with very high contention, but in the case of search data structures, it is out-performed by state-of-the-art algorithms (such as those applying our ASCY patterns) in other scenarios. Moreover, Node Replication incurs significant space overhead, as the data structure is replicated in several times, and is blocking.

Scalability. Our work [36] has argued that scalability of synchronization is mainly a property of the hardware. In particular, synchronization primitives are shown to be inherently non-scalable on NUMA architectures due to expensive cache-line transfers. To bypass these problems, ASCY reduces the amount of synchronization on CSDSs, leading to designs that scale even in the presence of non-uniformity.

Clements et al. [28] show that as long as interfaces are commutative, a scalable implementation of an object must exists. Essentially, they make the case that if two operations commute, there must be an implementation where executing these operations results in no cache conflicts. Thus, from the point of view of the memory system, executing the two operations does not cause any scalability issues. Although basic CSDS interfaces commute, as defined by Clements et al., certain structures such as lists and trees do not allow for conflict-free implementations. We show, however, that even in these cases, scalable algorithms can be devised following ASCY.

Data structures in systems. Boyd-Wickizer et al. [17] performed a scalability study of the Linux kernel. They identify several bottlenecks, among which, one in the directory entry lookup operation (even though it is optimized using RCU). In general, many systems, such as Memcached [114], SILT [103], LevelDB [59], RocksDB [47], or Masstree [108] have a CSDS at their core. In several cases, these structures have been found to cause scalability problems. A well known instance is that of Memcached [17,49,131]. Fan et al. [49] increase the performance of Memcached by a factor of three, largely due to changes in the hash table algorithm. ASCY can be used to recognize possible optimizations in systems such as Memcached and develop scalable CSDS implementations.

Baumann et al. [12] argue that in principle, non-portable, hardware-specific optimizations in the OS kernel should be removed. They attribute the existence of such optimizations to "the basic structure of a shared-memory kernel with data structures protected by locks" and propose to rethink the OS structure. We show that, by applying ASCY, we reach portably-scalable implementations. We believe that ASCY can help alleviate the difficult problem of CSDSs in OSes.

Progress guarantees Gramoli [61] studies different classes of concurrent data structures, and observes no inherent difference in aggregate throughput between lock-free and lock-based search data structures. In this dissertation, we look at CSDSs through the perspective of a different criterion: practical wait-freedom.

We are not the first to ask whether algorithms with weaker progress guarantees do not actually provide wait-freedom, given some assumptions about the execution environment. We are however the first to look at blocking algorithms. In doing so, we also provide the first practical quantification of the effects of progress guarantees on CSDSs. Ellen et al. [53] show that under an unknown-bound synchronous model, obstruction-free algorithms can behave practically wait-free. Herlihy and Shavit [73] suggest that on realistic schedulers, lock-free algorithms behave in a wait-free manner. Alistarh et al. [3] prove that assuming a stochastic scheduler which determines the ordering of accesses to a memory location, lock-free algorithms behave practically wait-free. Our work suggests that in the case of CSDSs, given the fact that the probability of contention for any memory location is low, the characteristics of the scheduler governing the order of memory accesses might not be a determining factor.

Michael [118] also studies the practical trade-offs between different progress guarantees. However, his analysis only discusses non-blocking algorithms, and does not focus on CSDSs, but mostly on objects where a single point of contention is likely, such as counters and queues. As we show in this dissertation, the properties of state-of-the-art concurrent algorithms for search data structures are significantly different from other concurrent structures.

Rajwar and Goodman [139] propose a hardware mechanism for transactions and identify it as conductive to non-blocking behavior in lock-based programs. However, the transactional support they propose provides starvation freedom, which is not the case in practical implementations such as Intel TSX. In the context of blocking CSDSs, we show that HTM support is needed for wait-free behavior in a particular set of circumstances only. This observation is, we believe, interesting in its own right. We also show that wait-free behavior is possible even with best-effort transactional memory which may revert to locking, and explain why this conclusion does not extend to other concurrent structures beyond CSDSs.

3.2 Software for fast persistent memory.

Durable data structures. Several approaches have used transactions as means of interacting with NVRAM [23, 29, 39, 60, 88, 92, 96, 105, 113, 159]. Yet, the significant overhead associated with transactional logging, inherent to such methods, has been recently highlighted (e.g., [144]), and several attempts to alleviate the problem have been proposed. Izraelevitz et al. [88] introduce an approach in which if failures occur within a critical section, upon recovery the critical section can simply be run to completion instead of reverting to previous state. To achieve this, one simply needs to reliably keep track of the last store instruction performed by each thread. While efficient in a scenario where write-back caches can be assumed to be persistent as well, the approach essentially requires a write to the log for each store instruction in a critical section. Kolli et al. [96] focus on static transactions in lock-based applications, and attempt to minimize persist dependencies in order limit waiting time. The authors also show how the commit stage of transactions can be performed while not holding any locks. In the same vein, Kamino-Tx [113] uses a copy-on-write technique, and avoids logging in critical sections. DudeTM [105] optimizes redo logging by first executing the transaction and obtaining a redo log in volatile memory, then atomically flushing the redo log to persistent memory, and only then modifying the original data.

In this paper, we go beyond optimizations to logging: we provide a method that in the common case allows us to circumvent such logging altogether in the context of concurrent data structures.

Several efforts [23,25,78] have been dedicated to the generation of correct durable applications for NVRAM from existing code. These approaches generally assume lock-based code. Due to their general-purpose nature, they incur additional overheads, in particular due to logging, when compared to our method, which is specialized to concurrent data structures.

Several proposals for indexing trees for NVRAM have been made [26, 101, 133, 157, 168]. However, they either require logging in some form, or do not address potential memory leaks during new node creation. In addition, the techniques cannot be broadly generalized to other data structures. Nawab et al. [130] also identify lock-free algorithms as a good fit for NVRAM, but unlike this dissertation, consider a model where data in the caches is not lost in the case of a restart. Moreover, the authors do not consider associated issues, such as memory management. Friedman et al. [56] introduce lock-free algorithms for durable queues, but do not go beyond this data structure, or consider memory management.

Other Memcached adaptations for NVRAM have been proposed, but they use transactions extensively [29, 109, 127] or they do not guarantee all completed requests are durable [166], whereas *NV-Memcached* ensures all completed requests are durable and limits transactions to the slab allocator code by using our log-free hash table.

Memory management for fast persistent memory. The problem of general memory allocation and reclamation for NVRAM has also received a lot of attention. Generic persistent memory frameworks [23, 29, 159] handle allocation and reclamation as part of the transaction mechanisms they provide, and thus use logging to ensure correctness. nvm_malloc [143] provides an interface to allocate and free persistent memory, but because of the fine-grained accounting, incurs significant overheads for each allocation and deallocation. Makalu [16] and NVthreads [78] also keep track of allocator metadata at a coarser-grain level. However, they incur higher costs at recovery time, as they require a garbage collection pass over the entire memory. Unlike all these approaches, we propose a method that is highly tuned to concurrent data structures. Thus, we are able to minimize overheads both at run-time (by efficiently keeping track of active memory areas and not requiring inter-thread coordination), as well as at recovery time (by avoiding a full mark-and-sweep pass). Additionally, our memory management scheme builds upon basic memory allocators and deals with the issue of memory reclamation as well. Thus, our scheme can take advantage of an efficient memory allocator at its core.

Concurrent hash tables. The design of the buckets in our link cacheshares some similarities with the volatile metatdata hash table of RAMCloud [142] , the software cache of Li et al. [102], CLHT [35] and the hash index of MICA [104]. Like these designs, we attempt to align and pack multiple bucket entries in a single cache line. RAMCloud stores 8 key-value pairs in a cache line, while CLHT stores up to 6 key-value pairs in a cache line; MICA has 2 cache-line buckets, storing 15 elements. Like RAMCloud and MICA, we do not store the entire key, but rather just a partial hash. If false conflicts occur, we might have to flush the bucket when it would not have been strictly necessary, but this does not jeopardize correctness. With 32 buckets, our hash space is 2M elements, making false conflicts extremely unlikely. Unlike RAMCloud and CLHT, our cache is fixed size and does not support chaining of multiple cache-line size entries in a bucket. This is not problematic due to the best-effort nature of our cache. Like CLHT and MICA, but unlike RAMCloud, we store concurrency-control state on the same cache line as the bucket contents. Unlike CLHT and MICA, we use locks of multiple granularities: inserts can lock an individual entry in the bucket, while flushes obtain a global lock on the bucket. Thus, in case our HTM-based fast-path fails, unlike alternatives, we allow multiple writers to proceed concurrently. Our cache does not support individual remove operations, as the only way an element can be removed is when the entire bucket is flushed. In addition, our cache is optimized for use with hardware transactional memory.

Fundamentally, our cache is different from all these alternatives in the sense that it is best-effort - we prioritize common case performance over the certainty of being able to insert a link. This however does not jeopardize the correctness of our approach.

3.3 Concurrency control in software transactional memory and data stores.

The main novelty of this work is the idea of locking individual timestamps, leading to a genre of multiversion algorithms called MVTL. No other work proposes this idea, but because MVTL is a broad class, several existing algorithms become special cases of MVTL, leading to similarities in mechanism.

Multiversion concurrency control is an old idea [15] that has seen a resurgence in software transactional memory (STM) systems, several of which provide serializability [11, 21, 51, 90, 91, 128, 135, 136, 141]. Prior work in this space falls into three categories: (1) multiversion for read-only transactions, (2) conflict graph schemes, and (3) multiversion timestamp ordering algorithms. The first category [51, 135, 136, 141] are systems that use multiversion to benefit solely read-only transactions; update transactions rely on optimistic methods that, upon commit, validate the read-set and abort if any object has changed. While read-only transactions are important, these methods abort under simple concurrent update schedules, such as the following (where full multiversion schemes do not abort):

$$\begin{array}{llll} T_1: & R(X) & & W(Y) \\ T_2: & & W(X) & \end{array}$$

The second category [11, 91, 128] are multiversion STM systems that ensure serializability by detecting cycles in the *conflict graph*—a data structure that represents the conflicts across transactions—similarly to the MVSGT algorithm [164]. These algorithms have two drawbacks: they are complex and they incur significant computation overhead, as reported in some of these papers.

The third category [97] are systems that extend multiversion timestamp ordering. Specifically, Kumar et al. [97] explain how to provide opacity, which is stronger than serializability. However, the algorithm suffers from the same drawbacks of multiversion timestamp ordering that we address in Section 7.5. It should be possible to extend MVTL to provide opacity using the ideas of Kumar et al. [97], but this is future work.

Lomet et al. [106] introduce the multiversion timestamp range algorithm (MVTR). With MVTR, each transaction is assigned a range of timestamps, and this range shrinks as the transaction executes; at the end, MVTR commits if the range is non-empty. MVTR differs from MVTL because MVTR locks entire objects instead of timestamps. As a result, MVTR does not enjoy the full benefits of multiversion concurrency control, such as allowing two concurrent transactions to write the same object. Also, with MVTR one transaction manipulates the inner state of another transaction (e.g., by changing the range that another transaction uses), which synchronizes through a transaction scheduler or locks. Another context where ranges of timestamps have been used are elastic transactions [52], an STM technique aimed at search data structures. The system maintains a single version of objects, timestamped with the time they were last updated. Transactions keep track of a range of timestamps that determines if they can commit, based on their start time and the time when the accessed objects were last written to. A pessimistic two-phase locking protocol is used in order to commit.

Snapshot isolation [13] is both an isolation property and a protocol. The protocol uses multiversioning and timestamps, similarly to multiversion timestamp ordering, but it does not provide serializability. Other protocols that use multiversioning and timestamps provide even weaker notions than snapshot isolation [146].

Optimistic concurrency control [98] is another technique that can use multiversioning. Essentially, transactions do not acquire any locks when running. At commit time, the transaction verifies that the versions it has read still represent the latest data, and, if this is the case, applies updates. More recent work, such as the TicToc [169] has optimized the OCC protocol in order to adaptively serialize transactions based on the data they access. TicToc computes potential serialization points for the transaction before the validation and commit phases. Thus, a transaction for which the read and write sets have been inspected might later abort. In contrast, MVTL ensures that once a serialization point for the transaction has been found, the transaction is guaranteed to commit. Similarly, Faleiro et al. [48] propose Bohm, a multi-version protocol that pre-orders transactions before execution, and is in this sense more pessimistic than MVTL, which determines transaction ordering dynamically during execution. In addition, Bohm requires that the transaction be known ahead of time, and that the write-set be static. MVTL in contrast is able to operate with entirely dynamic transactions.

Our concept of policy is somewhat similar to that of a contention manager [62] in transactional memory, in the sense that it can be implemented in a large variety of ways, which may exhibit very different performance characteristics, but do not jeopardize safety. Nevertheless, the two concepts refer to different components of a transactional system.

Many practical systems providing distributed transactions only ensure snapshot isolation [45, 134], and do not allow concurrent writes for the same object. Other systems, such as Spanner [33], use two-phase locking in the case of read-write transactions, and thus provide very limited parallelism for such transactions. Spanner provides strict serializability (i.e., the serialization order of transactions conforms to their real-time execution order), and read-only transactions that acquire no locks and are guaranteed to succeed. MVTL also guarantees that read-only transactions never abort, and, given access to synchronized clocks, can also avoid any locking for such transactions and provide strict serializability. In summary, MVTL improves on the level of parallelism offered by such systems, while offering similar or stronger guarantees.

Part II

Scalable Concurrent Search Data Structures

4 Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures

In this chapter we introduce "*asynchronized concurrency (ASCY)*," a paradigm consisting of four complementary programming patterns. ASCY calls for the design of concurrent search data structures (CSDSs) to resemble that of their sequential counterparts. We argue that ASCY leads to implementations which are *portably scalable*: they scale across different types of hardware platforms, including single and multi-socket ones, for various classes of workloads, such as read-only and read-write, and according to different performance metrics, including throughput, latency, and energy. We substantiate our thesis through the most exhaustive evaluation of CSDSs to date, involving 6 platforms, 22 state-of-the-art CSDS algorithms, 10 re-engineered state-of-the-art CSDS algorithms following the ASCY patterns, and 2 new CSDS algorithms designed with ASCY in mind. We observe up to 30% improvements in throughput in the re-engineered algorithms, while our new algorithms out-perform the state-of-the-art alternatives.

4.1 Introduction

A *search data structure*, as discussed in Chpater 2, consists of a set of elements and an interface for accessing and manipulating these elements. The three main operations of this interface are a *search* operation and two update operations (one to *insert* and one to *delete* an element), as shown in Figure 4.1. Search data structures are said to be *concurrent* when they are shared by several processes. Concurrent search data structures (CSDSs) are commonplace in today's software systems. For instance, concurrent hash tables are crucial in the Linux kernel [111] and in Memcached [114], while skip lists are the backbone of key-value stores such as RocksDB [47]. As the tendency is to place more and more workloads in the main memory of multi-core machines, the need for CSDSs that effectively accommodate the sharing of data is increasing.

Nevertheless, devising CSDSs that scale and leverage the underlying number of cores is challenging [12, 17, 27, 49, 131]. Even the implementation of a specialized CSDS that would scale on a specific platform, with a specific performance metric in mind, is a daunting task. Optimizations that are considered effective on a given architecture might not be revealed

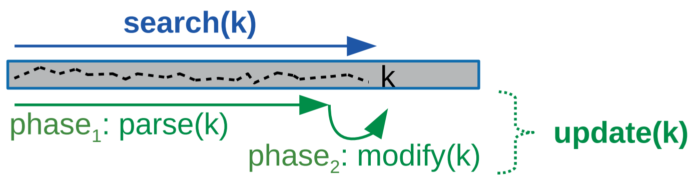


Figure 4.1 – Search data structure interface. Updates have two phases: a parse phase, followed by a modification phase.

as such on another [12, 36]. For example, NUMA-aware techniques provide no benefits on uniform architectures [36]. Similarly, if a CSDS is optimized for a specific type of workload, slightly different workloads can instantly cause a bottleneck. For instance, read-copy update (RCU) [112] is extensively used for designing CSDSs that are suitable for read-dominated workloads. However, it could be argued that this is achieved at the expense of scalability in the presence of updates.

The motivation of this work is to ask whether we can determine characteristics of CSDS algorithms that favor implementations which achieve what we call *portable scalability*, namely that *scale* across various platforms, workloads, and performance metrics. At first glance, this goal might look fuzzy for it raises a fundamental question: what scalability can we ideally expect from a given data structure, architecture, performance metric, and workload combination?

In fact, we can provide a practical estimation of an upper bound for a data structure's scalability, on a particular hardware and workload combination. We step on the observation that the coherence traffic induced by stores on shared data is the biggest impediment to the scalability of concurrent software. This is valid for practically any contemporary multi-core. Yet, some stores cannot be removed because they are inherent to the semantics of the data structure; typically those stores are employed by any standard sequential implementation of the same data structure (one that is not supposed to be shared by several processes). Assume however that we deploy, as is, such a sequential implementation on a multi-core and have it shared by multiple threads. Obviously, this deployment would result in incorrect (e.g., non-linearizable [75]) executions. The performance of these *asynchronized executions*, however, constitutes a reasonable indication of what can be ideally expected from a correct, *synchronized* implementation of the same structure.

We thus consider that a CSDS achieves *portable scalability* when its scalability closely matches that of asynchronized executions (a) across different types of hardware, including single and multi-sockets, (b) for various classes of workloads, such as read-dominated and read-write, and (c) according to different performance metrics. In the CSDS context, aspect (c) means that as we increase the number of threads, we want to remain as close as possible to the asynchronized execution in terms of *throughput* and *latency*, without sacrificing *energy* (i.e., without consuming more *power* than implementations that do not scale as well in terms of throughput or latency).

With this pragmatic objective in mind, we perform an exhaustive evaluation of CSDSs on six different processors: a 20-core and a 40-core Intel Xeon, a 48-core AMD Opteron, a 32-core Oracle SPARC T4-4, a 36-core Tilera TILE-Gx36, and a 4-core Intel Haswell. We measure four dimensions of scalability: throughput, latency, latency distribution, and power. We consider the state-of-the-art algorithms for linked lists, hash tables, skip lists, and BSTs. To the best of our knowledge, this is the most extensive CSDS evaluation to date. We find that for each data structure, there are CSDS algorithms whose performance is within 10% of the asynchronized versions. We observe that in general, the algorithms whose memory accesses to shared state best resemble those of a sequential – asynchronized – algorithm tend to achieve portable scalability. We further identify four patterns through which this resemblance to sequential implementations is achieved:

ASCY$_1$: The search operation should not involve any waiting, retries, or stores.

ASCY$_2$: The parse phase of an update operation should not perform any stores other than for cleaning-up purposes and should not involve any waiting, or retries.

ASCY$_3$: An update operation whose parse is unsuccessful (i.e., the element not found in case of a remove, the element already present in case of an insert) should not perform any stores, besides those used for cleaning-up in the parse phase.

ASCY$_4$: The number and region of memory stores in a successful update should be close to those of a standard sequential implementation.

None of these patterns is fundamentally counter-intuitive and each of them has already been identified as important in some form or another. We find that the existing algorithms that scale the best already apply some of these patterns. To our knowledge however, they have never been put in a coherent form and collectively applied and evaluated. We refer to these patterns as *asynchronized concurrency (ASCY)*, for together they indeed call for the design of concurrent algorithms to resemble that of their sequential counterparts in terms of access to shared state.

We apply ASCY to several existing state-of-the-art algorithms and obtain up to 30% improvements in throughput, accompanied by reduced latencies. Interestingly, ASCY not only leads to better throughput, but also results in CSDSs that consume less power (by 1.4% in average), hence further improving energy efficiency. We also present a hash table (CLHT) and a BST (BST-TK), two algorithms designed and implemented from scratch with ASCY in mind. CLHT (cache-line hash table) places each hash-table bucket on a single cache line and performs in-place updates so that operations complete with at most one cache-line transfer. BST-TK (BST Ticket) is a new concurrent BST that significantly reduces the number of acquired locks per update over existing algorithms. Both algorithms, by employing the ASCY patterns, outperform the state of the art.

Our evaluation also highlights a number of other interesting observations. We show, for instance, that the fact that an algorithm is lock-based or lock-free does not have a major effect on the scalability of CSDSs. Similarly, we show that the use of hardware transactional memory also makes little difference. We highlight however a number of hardware-related bottlenecks that should be taken into consideration by system designers.

In summary, the main contributions of the work presented in this chapter are:

- The analysis and comparison of a large number of state-of-the-art CSDS algorithms in a wide range of settings (i.e., platforms, workloads, and metrics), representing the most extensive evaluation to date. This evaluation helps identify characteristics of *portably scalable* algorithms and revisit some beliefs regarding CSDSs.
- *Asynchronized Concurrency*: A design paradigm which yields portably scalable CSDSs. When ASCY is applied, increasing throughput, reducing latency, and reducing power consumption go hand in hand.
- ASCYLIB: a CSDS library, including 34 highly optimized and portable implementations of linked lists, hash tables, skip lists, and BSTs, together with a companion memory allocator with garbage collection. ASCYLIB includes two novel CSDS algorithms designed from scratch, namely CLHT and BST-TK, and re-engineered versions of ten state-of-the-art CSDS algorithms. ASCYLIB is available at http://lpd.epfl.ch/site/ascylib.

Our patterns are not very precise guidelines and cannot be used to automatically generate the implementation of a CSDS from its sequential counterpart. They cannot be used to derive any theoretical lower bound either. Yet, as we show in this chapter, they provide very useful hints both for optimizing existing CSDSs and for designing new algorithms.

The rest of the chapter is organized as follows. We present ASCYLIB in Section 4.2. In Section 4.3, we provide our evaluation of CSDS algorithms. In Section 4.4 we identify and illustrate the benefits of ASCY. We describe in Section 4.5 the use of ASCY in the design of CLHT and BST-TK. We discuss the limitations of ASCY, and conclude the chapter in Section 4.6.

4.2 The ASCYLIB Library

ASCYLIB contains 32 fully/hybrid lock-based and lock-free CSDSs, as well as 5 sequential implementations.[1] These include existing state-of-the art designs, and optimized versions, which we adapt in order to enable one, or more, ASCY patterns. In addition, ASCYLIB contains two novel CSDS algorithms built from scratch based on ASCY. Some implementations in ASCYLIB were initially based on the Synchrobench benchmark suite [61].

Algorithms. Table 4.1 contains a short description of the existing algorithms we implement.[2] ASCYLIB further contains 10 re-engineered (using ASCY) state-of-the-art CSDS designs. In

[1] We use these as incorrect asynchronized CSDSs.

[2] The *urcu* and the *tbb* hash tables belong to the corresponding libraries and are not our own implementations.

	Name	Type	Short description
linked list	*async*	seq	A sequential linked list. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	coupling [74]	flb	All operations use hand-over-hand locking (grab next lock and release the previous) while parsing the list.
	pugh [137]	lb	Operations search/parse the list optimistically. Updates lock and then validate the target node. Removals employ pointer reversal so that a search/parse always finds a correct path.
	lazy [66]	lb	Nodes are deleted in two steps: marking and physical deletion. Searching/Parsing the list simply ignores marked nodes. Updates parse the list, grab the locks, validate the locked nodes, and perform the update.
	copy [132]	lb	Similar to Java's *CopyOnWriteArrayList*. Updates create new copies of the list and are protected by a global lock.
	harris [64]	lf	Nodes are deleted in two steps: mark with CAS and delete with a second CAS. Operations remove the logically deleted nodes while searching/parsing the list. If cleaning-up fails, searching/parsing is restarted.
	michael [116]	lf	A refactored implementation of *harris* for easier memory management.
hash table	*async*	seq	A sequential hash table. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	coupling [74]	flb	Uses one *coupling* list per bucket, with a single per-bucket lock.
	pugh [137]	lb	Uses one *pugh* list per bucket, with a single per-bucket lock.
	lazy [66]	lb	Uses one *lazy* list per bucket, with a single per-bucket lock.
	copy [132]	lb	Uses one *copy* list per bucket, with a single per-bucket lock.
	urcu [40]	lb	Part of the URCU (User-space RCU) (version 0.8) library. After each successful removal, it waits for all ongoing operations to complete before freeing the memory. Supports resizing.
	java [99]	lb	Similar to Java's *ConcurrentHashMap*. Protects the hash table with a fixed number of locks (we use 512 locks). Supports resizing.
	tbb [86]	flb	Part of Intel's Thread Building Blocks (version 4.2) library. Uses reader-writer locks. Supports resizing.
	harris [64]	lf	Uses one *harris-opt* list per bucket.
skip list	*async*	seq	A sequential skip list. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	pugh [137]	lb	Maintains several levels of *pugh* lists. Parses towards the target node without locking.
	herlihy [70]	lb	Update operations optimistically find the node to update and then acquire the locks at all levels, validate the nodes, and perform the update. Searches simply traverse the multiple levels of lists.
	fraser [54]	lf	Optimistically searches/parses the list and then does CAS at each level (for updates). A search/parse restarts if a marked element is met when switching levels. The same applies if a CAS fails.
bst	*async-int*	seq	A sequential internal BST. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	async-ext	seq	A sequential external BST. We use it as an *incorrect asynchronized* concurrent set for performance upper bounds.
	bronson [19]	lb	Partially external. A search/parse can block waiting for a concurrent update to complete.
	drachsler [43]	lb	Internal tree. Uses logical ordering to allow sequential read operations. Acquires ≥ 3 locks for removals.
	ellen [46]	lf	External tree. Updates help outstanding operations on the nodes that they intend to modify.
	howley [77]	lf	Internal tree. All three operations perform helping and might need to restart.
	natarajan [129]	lf	External tree. Minimizes the number of atomic operations and optimistically searches/parses the tree.

Table 4.1 – A short description of the existing CSDS algorithms we consider. "seq" stands for sequential, "flb" for fully lock-based, "lb" for (hybrid) lock-based, and "lf" for lock-free.

particular, we apply ASCY$_{1-2}$ on *harris* linked list and *fraser* skip list. We also apply ASCY$_3$ on *pugh*, *lazy*, and *copy* linked lists/hash tables, on *java* hash table, on *pugh* and *herlihy* skip lists, and on *drachsler* BST. Finally, we create a *urcu* hash-table variant that uses SSMEM (see below) instead of RCU for memory management and is closer to ASCY$_4$. In all our experiments, these optimizations result in better performance (see §4.3 and §4.4). We then use ASCY as the base to design two new CSDS algorithms, a lock-based and a lock-free variants of a hash table and a BST (see §4.5).

Memory management. We develop SSMEM, a memory allocator with epoch-based garbage collection (GC) [54]. SSMEM uses ideas similar to the RCU [112] mechanism: freed memory can only be reused once it is certain that no other thread holds a reference to this location. When some memory is freed, it does not become available until a GC pass decides that it is safe to be reused. The amount of garbage SSMEM allows before performing GC is configurable. Furthermore, SSMEM is non-blocking: it is based on per-thread counters that are incremented to indicate activity.

4.3 Evaluating the State of the Art CSDSs

In this section, we present a cross-platform evaluation of the state-of-the-art CSDS implementations and compare them with their asynchronized counterparts. We observe that regardless of the platform, the asynchronized executions perform the best. We further note that the algorithms with memory accesses to shared state that best resemble the asynchronized implementations are also the closest to these asynchronized upper bounds. Our evaluation also enables us to quantify the impact that various hardware features have on CSDS algorithms.

We start by describing the platforms and experimental settings used throughout this chapter. We consider four multi-processors (with multiple sockets) and a chip multi-processor (with one socket). We also briefly experiment with an Intel Haswell desktop processor with hardware transactional-memory support.

Opteron. The 48-core AMD Opteron contains four Opteron 6172 [31] multi-chip modules (MCMs). Each MCM has two 6-core dies. It operates at 2.1 GHz and has 64 KB, 512 KB, and 5 MB (per die) L1, L2, and LLC data caches respectively.

Xeon20. The 20-core Intel Xeon consists of two sockets of Xeon E5-2680 v2 Ivy-Bridge 10-core (20 hyper-threads). It runs at 2.8 GHz and includes 32 KB, 256 KB, and 25 MB (per die) L1, L2, and LLC, respectively.

Xeon40. The 40-core Intel Xeon consists of four sockets of Xeon E7-8867L Westmere-EX 10-core (20 hyper-threads). It clocks at 2.13 GHz and offers 32 KB L1, 256 KB L2, and 30 MB (per die) LLC, respectively.

Tilera. The Tilera TILE-Gx36 [149] is a 36-core chip multi-processor. It clocks at 1.2 GHz and has 32 KB, 256 KB, and 9 MB[3] L1, L2, and L3 data caches, respectively.

T4-4. The Oracle SPARC T4-4 is a four-socket multi-processor with 8 cores per socket and a total of 256 hardware threads (chip multi-threading). It operates at 2.85/3 GHz and has 16 KB, 256 KB, and 4 MB (per die) L1, L2, and LLC data caches, respectively.

Experimental settings. Each of our measurements represents the median value of 11 repetitions of 5 seconds each. We manually pin threads on cores in order to take advantage of the locality within sockets. Each operation is either a search, or an update, based on the update percentage we select. In general, we initialize the structure with a number of elements (N). The operations choose a key at random in the $[1 \dots 2N]$ range. This provides the guarantee that on average, half of the operations are successful and that the structure size remains close to N (the update percentage is split to half insertions and half removals). On all architectures, except Tilera, we set SSMEM to trigger GC when 512 memory locations have been freed. On Tilera, we set this value to 128 in order to optimize for the smaller TLBs of 32 entries. On the asynchronized implementations, we disable GC to avoid data corruption. Finally, we use 64-bit long keys and values. It is straightforward to replace both with larger structures.

Cross-Platform evaluation. We now look at the behavior of a large sample of the state-of-the-art CSDS algorithms, as presented in Table 4.1. Figures 4.2, 4.3, 4.4 and 4.5 show cross-platform results on various workloads, spanning low, average, and high degrees of contention for the different data structure types. The histograms plot the throughput and the scalability ratio compared to the single-threaded execution (on top of each bar) on 20 threads.

On average and low-contention levels, algorithms exhibit good scalability in terms of throughput: in the experiments with 20 threads, the average scalability of the best performing CSDS algorithm (per data structure) is 16.2 in the low contention case, whereas for the average and high-contention levels, this value is 14.1 and 9.8, respectively. While trends are generally valid across platforms, we do notice a certain variability: for each workload, the standard deviation of the average scalability values of different platforms is ~2. In some cases, scalability trends differ significantly between platforms. For instance, on the Opteron the hash tables do not scale beyond 32 threads (including *async*), due to bandwidth limitations. In short, the main hash table structure is initialized from a single memory node, thus all requests are directed to that node. This problem does not emerge on *java*, because of the fine-grained resizing (per one of the 512 regions) that spreads the hash table on multiple nodes.

In addition, we observe that there are various algorithms per data structure that are the "best". On linked lists (Figure 4.2), *pugh* is consistently competitive across workloads and platforms, but *lazy* is close in throughput. On hash tables (Figure 4.3), several algorithms perform close to each other (e.g., *pugh*, *lazy*, *copy*, *harris*). On skip lists (Figure 4.4), *herlihy* and *pugh* perform similarly. Finally, on BSTs (Figure 4.5), *natarajan* is generally the best. Overall, we see that, per data structure, both lock-based and lock-free algorithms are close in terms of performance.

[3]The 36 L2 caches are utilized as a distributed LLC.

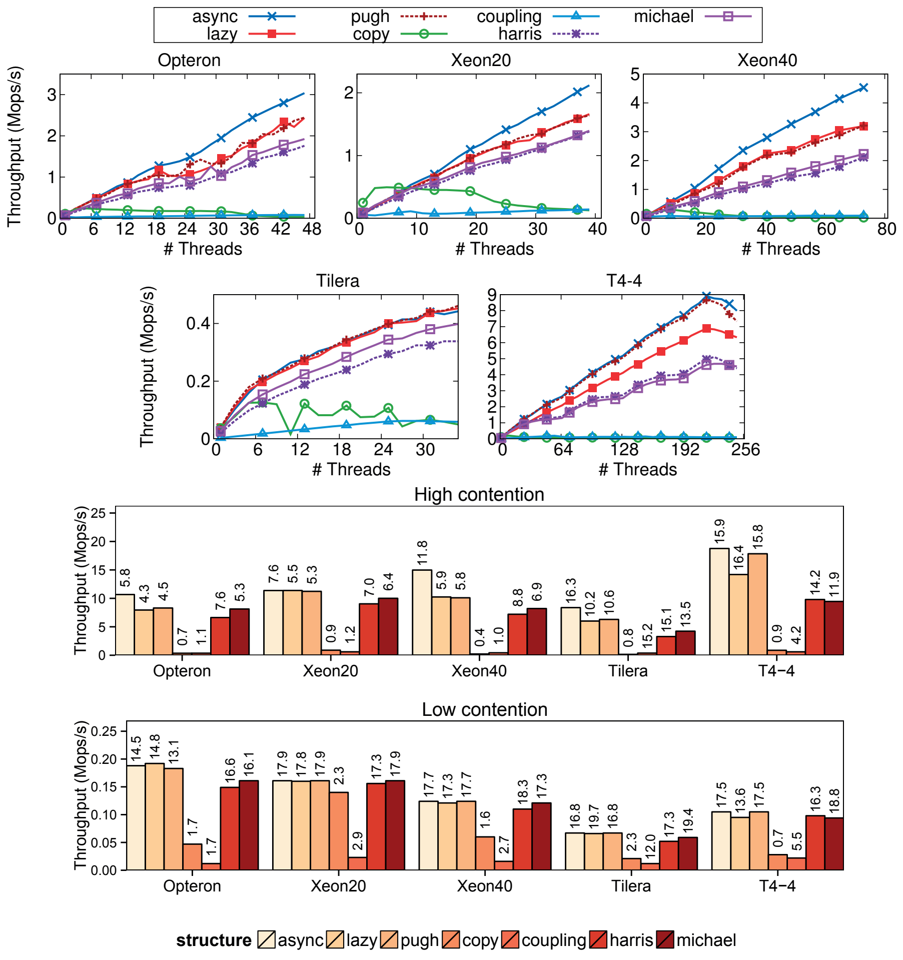


Figure 4.2 – Linked lists: cross-platform results on average (top graphs – 4096 elements, 10% updates), high (20 threads, 512 elements, 25% updates), and low contention (20 threads, 16384 elements, 10% updates).

Lock-freedom is more important when we employ more threads than hardware contexts (not shown in the graphs). In these deployments, lock-freedom provides better scalability than lock-based designs.

It is worth noting that the workloads we evaluate are uniform: the frequency and the distribution of updates are constant. We briefly experiment with non-uniform workloads (not shown in the graphs), such as those with update spikes and continuously increasing structure size. We notice that our observations are valid in these scenarios as well.

Dissecting asynchronized executions. In Figures 4.2, 4.3, 4.4, and 4.5, we also depict the behavior of the asynchronized implementations for each data structure. We notice that except for some corner cases, the asynchronized implementations outperform alternatives on all the platforms. The reason for the rare corner cases in which some concurrent implementations

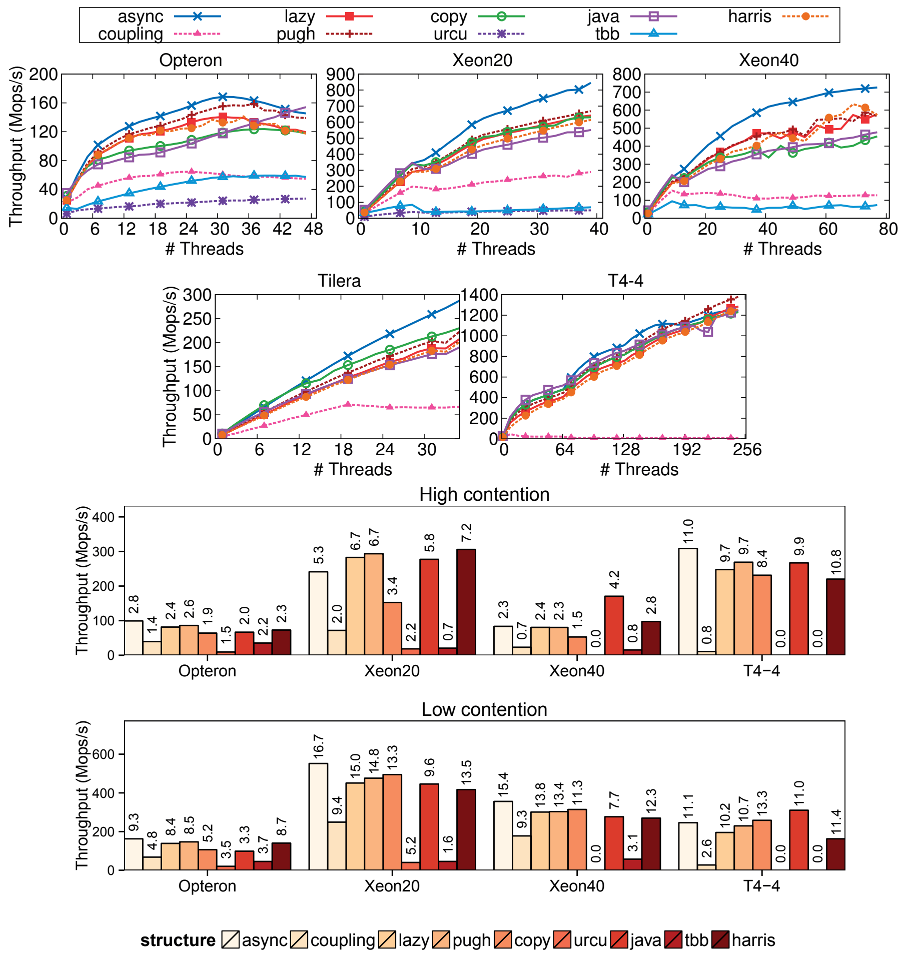


Figure 4.3 – Hash tables: cross-platform results on average (top graphs – 4096 elements, 10% updates), high (20 threads, 512 elements, 25% updates), and low contention (20 threads, 16384 elements, 10% updates).

can perform better than *async* is that asynchronized structures can become malformed in concurrent scenarios. For instance, an update operation on a skip list could update several pointer fields. We observe that these pointers are sometimes not properly set due to concurrency, leading to longer average path lengths.

For each data structure and regardless of the platform and workload, there is at least one concurrent algorithm that performs and scales close to the asynchronized implementation (*async*) of the data structure. On average, the best concurrent implementations are 10% slower than their asynchronized counterparts and exhibit similar scalability trends. The next section explores methods in which the portable scalability of these algorithms can be improved even further. Our empirical results thus confirm our intuition: asynchronized executions represent good approximations for the ideal behavior of CSDSs.

Intuitively, the asynchronized implementations perform and scale better than the concurrent

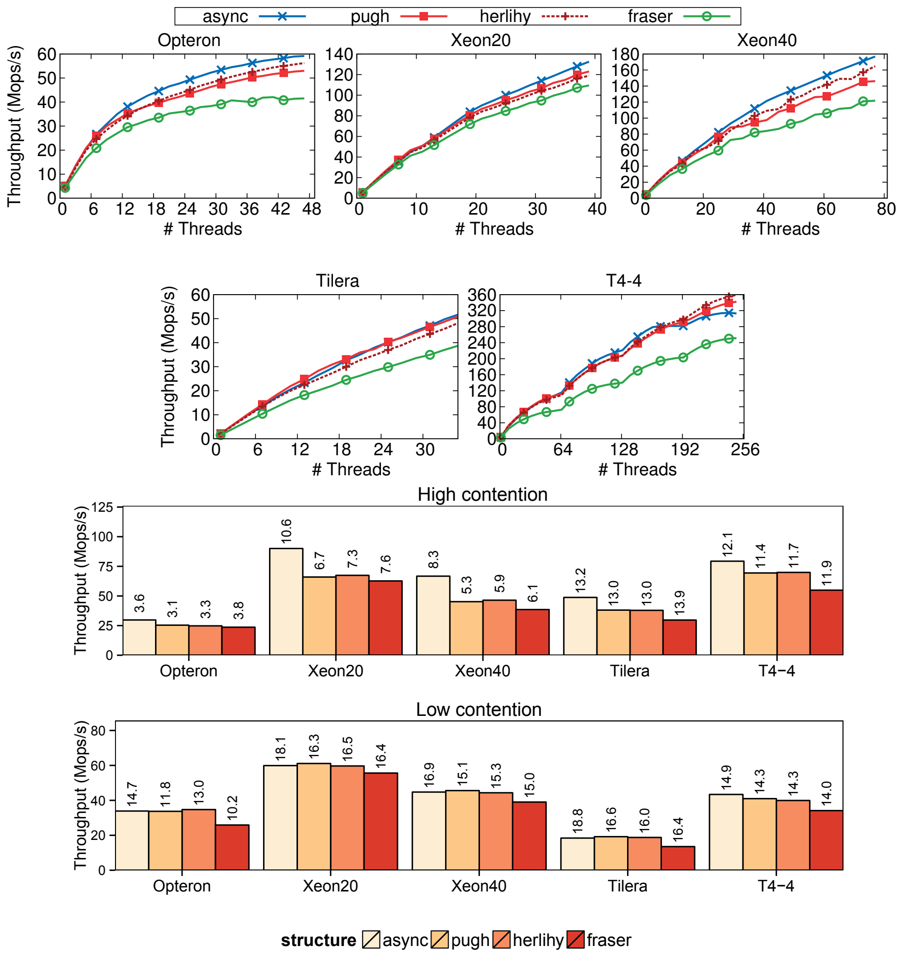


Figure 4.4 – Skip lists: cross-platform results on average (top graphs – 4096 elements, 10% updates), high (20 threads, 512 elements, 25% updates), and low contention (20 threads, 16384 elements, 10% updates).

alternatives because they trigger cache coherence less (i.e., fewer cache-line transfers): they modify only the data that is semantically necessary, as they do not employ synchronization, thus leading to a smaller number of memory stores. Cache coherence is the source of the most significant scalability bottleneck for concurrent algorithms on multi-cores, because the number of cache-line transfers tends to increase with the number of threads. Hence, it is essential for a CSDS algorithm to limit the amount of cache traffic it performs during each operation, which is directly linked to the number of stores on shared data. Stores cause cache-line invalidations, which in turn generate cache misses of future accesses.

We confirm this line of reasoning by showing a practical correspondence between the number of cache misses and the performance of an algorithm. In doing so, we use linked-list algorithms as an example. Figure 4.6 shows the number of cache misses per operation generated by various linked-list algorithms, as well as their scalability compared to single-thread throughput.

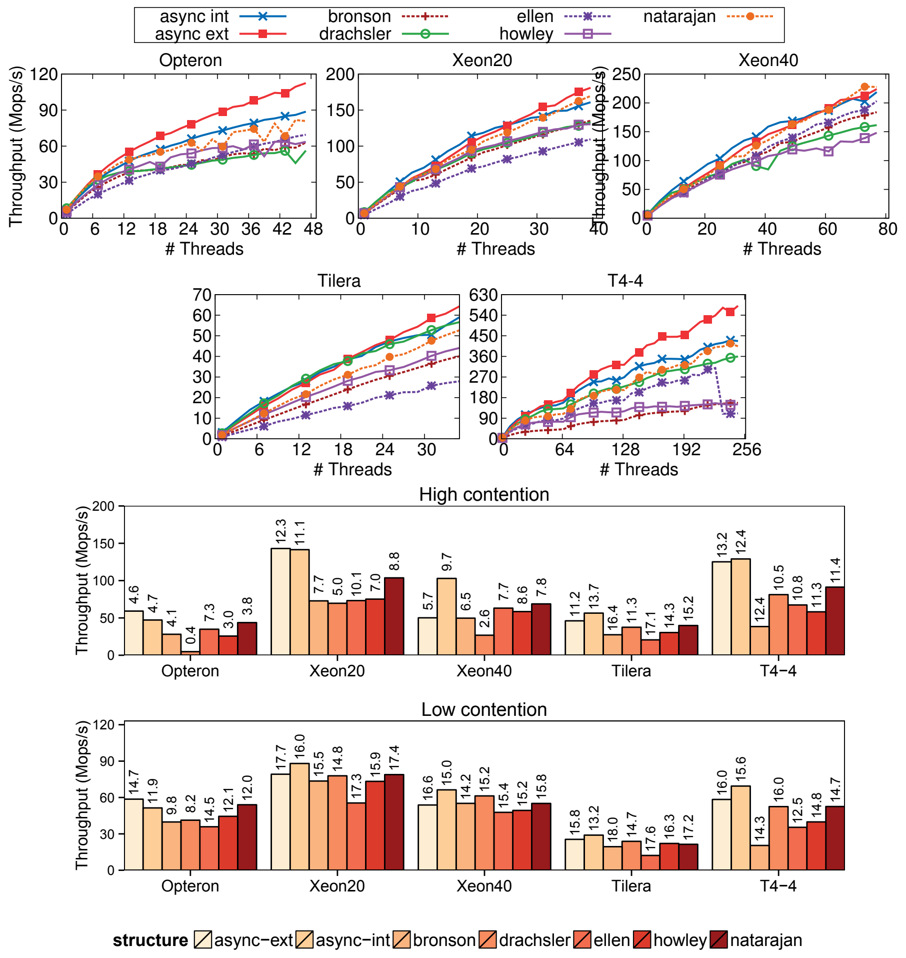


Figure 4.5 – BSTs: cross-platform results on average (top graphs – 4096 elements, 10% updates), high (20 threads, 512 elements, 25% updates), and low contention (20 threads, 16384 elements, 10% updates).

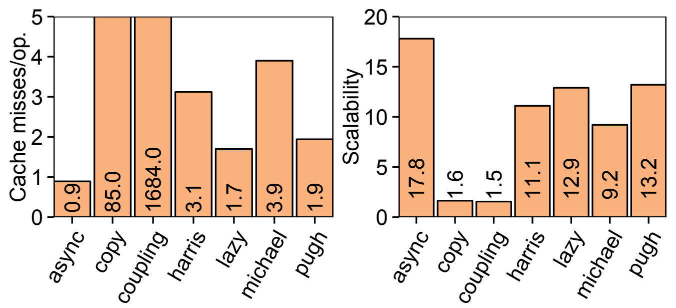


Figure 4.6 – Cache misses per operation and scalability for various linked-list algorithms.

We use a workload where the list has 4096 elements on average, 10% of the operations are updates, and 20 threads concurrently access the data structure. Clearly, the asynchronized execution has the fewest number of cache misses. It is also interesting to note that the number of cache misses per operation is directly correlated to the scalability and performance of the algorithms: the fewer cache misses an algorithm generates, the better it scales. This correlation pertains to the other data structures as well.

We also take a closer look at how the "best" concurrent algorithms access memory. We look at the memory access pattern of a CSDS algorithm by studying the number of loads and stores (often performed through read-modify-write instructions), as well as the branches in each operation and phase of the algorithm. We notice that the CSDS algorithms that tend to scale and perform the best also tend to have an average number of loads, stores, and branches closer to the asynchronized implementations than the alternatives. This is generally valid for all the operations and phases. Thus, we make the observation that the more an algorithm's memory access pattern resembles that of the asynchronized execution, the better it scales regardless of the platform and workload. In the next section, we look at ways in which this similarity can be achieved and validate this observation.

Hardware considerations. Our evaluation has revealed a number of other hardware-related observations.

We fine-tune several algorithms in ASCYLIB using Intel's TSX [84] hardware transactional memory (HTM), in order to assess whether HTM can be used to optimize CSDSs. Basically, in 60% of the scenarios, HTM improves throughput with up to 5%. In the remaining 40%, the results are either unaffected, or the throughput decreases (up to 5% as well). We use a 4-core desktop processor (8 hyper-threads) with the first iteration of Intel's TSX. As larger machines become available and the technology matures,[4] HTM might become even more helpful.

We also encounter a number of hardware-related bottlenecks, not specific to CSDSs. For example, the small TLBs on the Tilera (32 entries) require feeding SSMEM with small chunks of memory and keeping the amount of garbage low. Otherwise, if the threads use largely fragmented memory, the TLB misses become a bottleneck. Similarly, on Xeon40, large chunks of memory with a lot of garbage can lead to an excessive number of hardware prefetches that decrease performance up to an order of magnitude. On the Opteron, the interconnect bandwidth becomes a bottleneck when the structure is allocated on a single memory node. If a structure fits on one memory page, there is no straightforward solution to this problem, other than restructuring the data structure. System designers should be aware of the aforementioned issues as they can emerge in all types of software.

4.4 The ASCY Patterns

Having observed that CSDSs that resemble sequential algorithms scale and perform best, we now look at how this similarity can be achieved. We identify four patterns which are applicable to a broad class of CSDS algorithms and we show that when they are applied, the performance and scalability of CSDSs is systematically improved. These patterns collectively represent

[4]A recent announcement by Intel [85] suggests that this might take a while.

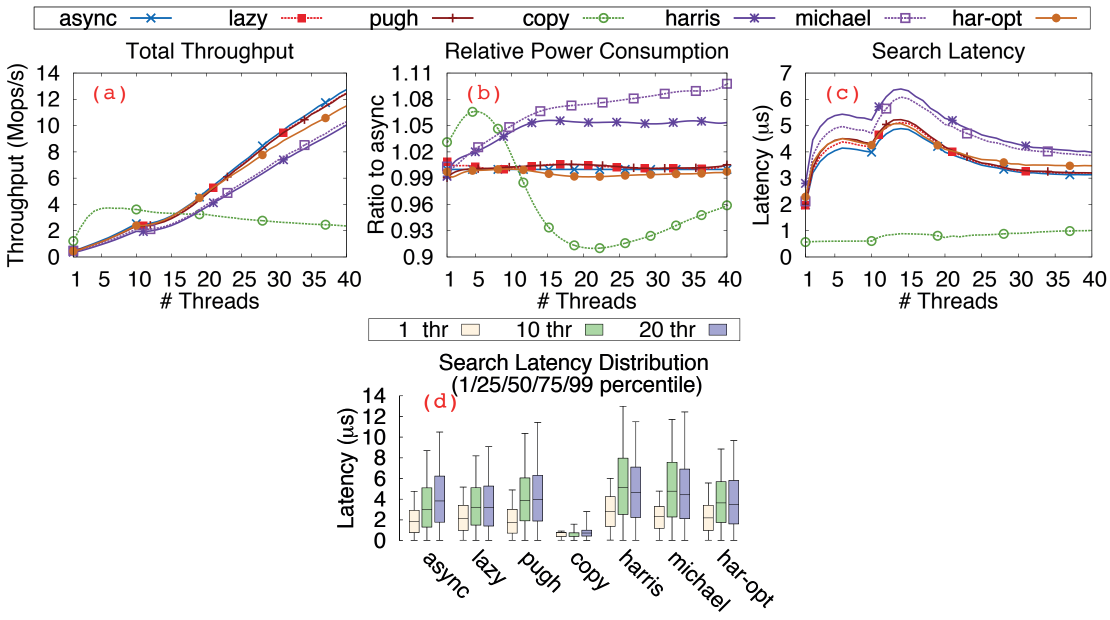


Figure 4.7 – Linked list with 1024 elements and 5% updates (2.5% successful).

asynchronized concurrency (ASCY).

For brevity, we select Xeon20 as the platform in our experiments. This is the most modern processor within our platform set. Also for brevity, we break down the results for each pattern using one of the four data structures. Note that we get similar results on any platform and data-structure combination: the patterns are globally beneficial.

ASCY$_1$. We first examine the search operation of CSDSs and use linked lists as a case study. Figure 4.7 depicts the behavior of the various linked lists of ASCYLIB on a search-dominated workload (only 2.5% successful updates).

Out of the existing algorithms, the *async*, *lazy* and *pugh* lists deliver the highest total throughput. Both *lazy* and *pugh* linked lists have a search that is identical to the sequential algorithm. Essentially, no stores, waiting, or restarting is involved. These algorithms perform within 10% of *async*. In comparison, the lock-free lists (*harris* and *michael*) diverge from the sequential code as they try to physically remove logically-deleted nodes using CAS. If a physical removal fails, the operation is restarted. Additionally, they also need to unmark[5] every pointer while traversing the list.

Overall, the results can be largely explained by the average search latencies (Figure 4.7(c)): the closer to the sequential an implementation is, the lower the latency is. On more than 20 threads, the search latencies decrease due to the effects of hyper-threading; the two hyper-threads of a core help each other by keeping the list nodes warm in the shared L1 cache.

[5]Clear the least significant bit that indicates a logically deleted element.

We thus identify ASCY$_1$ as a generic pattern: *The search operation should not involve any stores, waiting, or retries.*

We apply this pattern to the search operation of existing algorithms: in the case of linked lists, we apply it to the *harris* lock-free list by removing the physical removal of logically deleted elements from the search operation. We refer to the resulting algorithm as *harris-opt*.

If we now look at the three lock-free algorithms, namely *harris*, *michael*, and *harris-opt*, the effects of applying ASCY$_1$ become evident. In both *harris* and *michael*, the search tries to unlink logically deleted nodes and restarts if it fails, hence violating ASCY$_1$. In contrast, *harris-opt* ignores the deleted nodes while searching. The latency improvements due to ASCY$_1$ are approximately 10-30% as we can observe on the latency graphs (Figure 4.7(c) & (d)). Additionally, *harris-opt* has a tighter latency distribution (Figure 4.7(d)) than the other two. *harris-opt* provides more stable executions because it never restarts or invokes work that is unnecessary to the search. Furthermore, Figure 4.7(b) plots the power consumption relative to *async*. We observe that *harris* and *michael*, which do not follow ASCY$_1$, deliver lower performance while consuming more power than the rest.

Additionally, the behavior of the *copy* list is worth analyzing. It is apparent from the latency graphs that structuring the data as an array can bring tremendous benefits on serial data accesses. However, *copy* has two major limitations: (i) high memory overhead, as every update creates a new list copy, and (ii) synchronization of updates with a global lock, which easily becomes a bottleneck. In §4.5.1, we use the idea of array-based structures in the design of a hash table.

Finally, we apply ASCY$_1$ to the *fraser* skip list (not shown in the graphs) and observe performance improvements. In most cases, applying ASCY$_1$ means deferring cleaning-up and helping to the update methods. ASCY$_1$ also requires that updates always leave the data structure in a state that allows any existing node to be found. In addition, when removing nodes, their memory should not be freed while there is the possibility of an ongoing search accessing it. Memory reclaimation is handled by SSMEM in ASCYLIB.

ASCY$_2$. We now focus on the parse phase of the update operations. In a sequential algorithm, this phase is basically identical to the search operation. In CSDSs however, parsing might involve helping, cleaning-up the data structure, or re-starting the operation (e.g., due to a failed clean-up attempt). We study this phase more closely using skip-list algorithms.

Figure 4.8 depicts the behavior of the five skip lists in ASCYLIB on a workload with 10% successful updates. The best performing pre-existing algorithms are *pugh* and *herlihy*, which are within 22% of the asynchronized version. Looking closer at their parse phase, we note that the only stores that are performed are for cleaning-up purposes and that the parse is never restarted. In contrast, a parse in *fraser* might have to restart due to a failed clean-up attempt, or accessing a logically deleted node when changing levels.

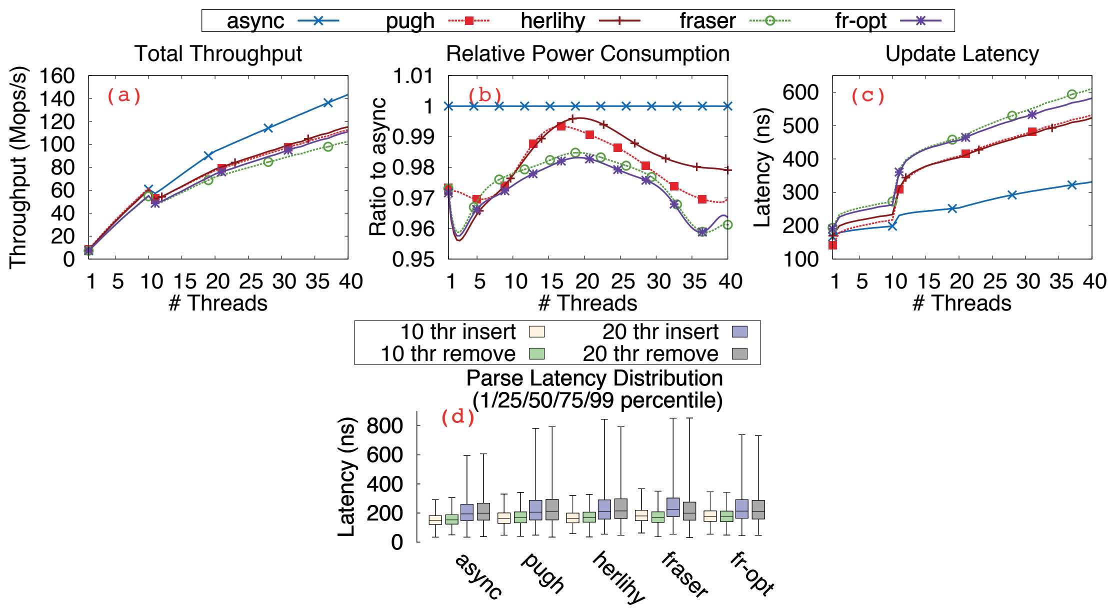


Figure 4.8 – Skip list with 1024 elements and 20% updates (10% successful).

Taking these results into account, we establish ASCY$_2$: *The parse phase of an update operation should not perform any stores other than for cleaning-up purposes and should not involve waiting, or retries.*

We apply this pattern in conjunction with ASCY$_1$ (based on [76]) to the *fraser* skip list and refer to the resulting algorithm as *fraser-opt*. *fraser-opt* delivers up to 8% better throughput than *fraser* and has a 5% lower average update latency (Figure 4.8(c)). Furthermore, Figure 4.8(d) plots the latency distribution of the parsing phase only. The behavior of both *fraser* and *fraser-opt* is similar. However, the latter eliminates the overhead of useless parses that have to be restarted. *fraser* performs 0.38%, 1.07%, and 1.82% more parses than updates on 10, 20, and 40 threads, respectively. *fraser-opt* has lower overheads: 0.03%, 0.09%, and 0.17%, respectively.

In terms of power consumption (Figure 4.8(b)), there are two main observations. First, ASCY not only improves the throughput of *fraser*, but also leads to an algorithm that consumes slightly less power than the initial design. Second, in the case of skip lists, the lock-based algorithms seem to consume more power than their lock-free counterparts.

Finally, we also apply ASCY$_2$ to the *harris* linked list. In practice, applying ASCY$_2$ means (i) not helping other threads while parsing the data structure and (ii) avoiding restarting if cleaning-up fails. Similar to ASCY$_1$, for ASCY$_2$ to work, concurrent updates should not render portions of the data structure unreachable.

ASCY$_3$. Another characteristic of sequential algorithms is that if an update operation cannot be completed (i.e., the parse does not find the node in case of a remove, or it finds the node in

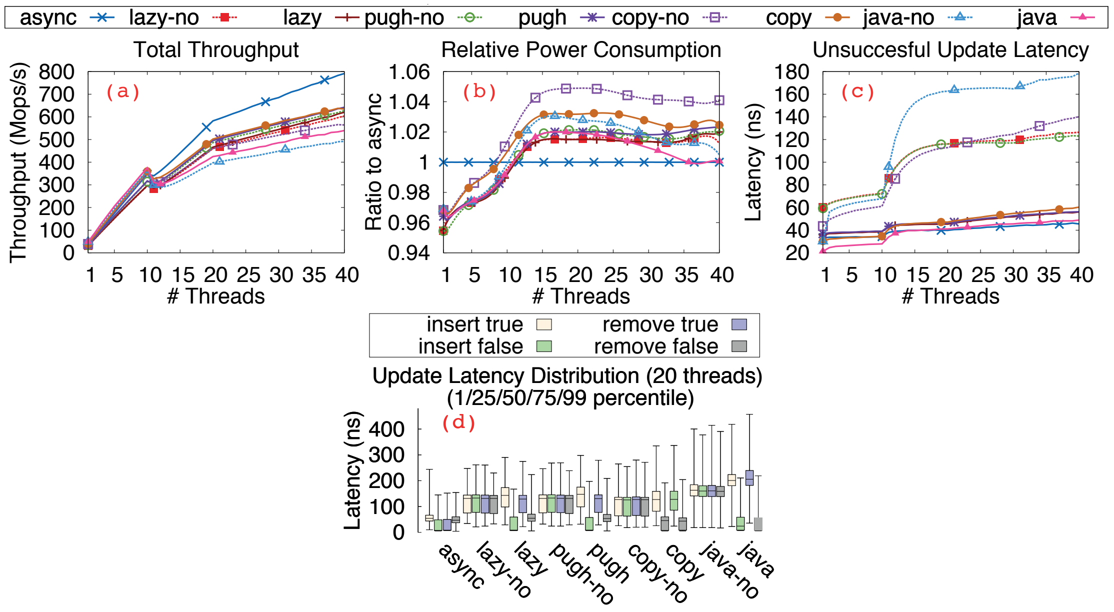


Figure 4.9 – Hash table with 8192 elements, 8192 (initial) buckets, and 10% updates (5% successful).

case of an insert), no additional stores are performed and the update method simply returns “false”. This is not always the case in existing CSDS algorithms.

We quantify the impact of this issue by looking at hash-table algorithms. Figure 4.9 depicts the performance of various hash tables with and without (suffixed with “*-no*”) the “read-only fail” of the sequential algorithms. In terms of throughput, algorithms doing no additional stores after unsuccessful parses perform up to 12.5% better than their counterparts doing stores. Additionally, they are within 19-30% of the *async* version. Although this change alters the behavior of only 5% of all operations in this workload, we observe throughput benefits up to 12.5% on *java*. This difference can be attributed to (i) the increased number of cache misses caused by unnecessary synchronization, and (ii) the increase of contention on locks. Basically, (i) manifests as a 2.8% increase in the cache-miss ratio on 40 threads, and (ii) as a 14% increased chance to find a lock occupied.

Given the apparent benefits, we model the behavior of the sequential algorithms through ASCY$_3$: *An update operation whose parse phase is unsuccessful should not perform any stores, besides those used for cleaning-up in the parse phase.*

Figure 4.9(c) details the benefits of ASCY$_3$ on the average latency of unsuccessful updates. It is clear that turning a failed update into a read-only operation yields a significant (1.5-4x) decrease in latencies. Nevertheless, depending on the algorithm, applying ASCY$_3$ can incur some overhead on successful updates, as we notice on the graph (d). For instance, enabling ASCY$_3$ on *java* requires an additional search to either decide that the update cannot succeed, or proceed to the actual update. In general ASCY$_3$ also reduces the power consumption of the CSDSs. This is achieved by decreasing the number of cache-line transfers.

Finally, we apply this pattern to multiple other existing algorithms: the *pugh*, *lazy*, and *copy* linked lists, the *pugh* and *herlihy* skip lists, and the *drachsler* BST. In many cases, applying ASCY$_3$ simply means checking the outcome of the parse and returning "false" without locking. The algorithms to which we apply ASCY$_3$ trivially maintain correctness, as unsuccessful updates can be seen as search operations.

ASCY$_4$. We now focus on the modification phase of the update operation. We use BSTs as an example for this scenario. Figure 4.10 includes the corresponding results.

Aside from the asynchronized algorithms, the best performing concurrent implementation is *natarajan*. We argue that the other four concurrent trees synchronize more than the minimum. Indeed, we measure the ratio of atomic operations to the number of successful updates on the same workload as Figure 4.10. *natarajan* uses two atomic operations per update on average, which is close to the asynchronized versions, whereas the other concurrent trees require more than three. In fact, *natarajan* is also the closest to *async* in terms of the number of stores and the number of affected cache lines. This major difference, together with the differences in the first three ASCY patterns, is reflected in the results, in terms of throughput, latency, and power consumption.

More precisely, *howley* employs helping even while searching or parsing the tree. *ellen* uses helping only on elements that the current operation wants to update. Helping is generally expensive, as it requires additional synchronization in order to be implemented. *bronson* is a complex algorithm that can block waiting for an update to complete. Finally, *drachsler* acquires a large number of locks (3.15 on average) for each successful update.

Figure 4.10(d) depicts the latency distribution of successful-only operations, isolating the effects of the modification phases. Clearly, the *natarajan* tree has lower latencies and a tighter distribution than the rest. Interestingly, *natarajan* consumes less power than two of the other four concurrent trees, similar power to *drachsler*, and more power than *howley*. *natarajan* simply performs at speeds different than the rest: on 40 threads, it issues 797M memory accesses per second with 13.4% cache-miss ratio, compared to the 578M/23.8% of *drachsler* and the 395M/18.7% of *howley*. Still, *drachsler* and *howley* consume 41% and 49% more energy per operation than *natarajan*, respectively.

We can thus identify ASCY$_4$: *The number and region of memory stores in a successful update should be close to those of a standard sequential implementation.*

Essentially, ASCY$_4$ means that updates should not block each other or write to the same memory addresses unless they operate on semantically related elements, such as adjacent nodes in the data structure.

Discussion. It is worth noting that while each pattern can be identified in some of the existing

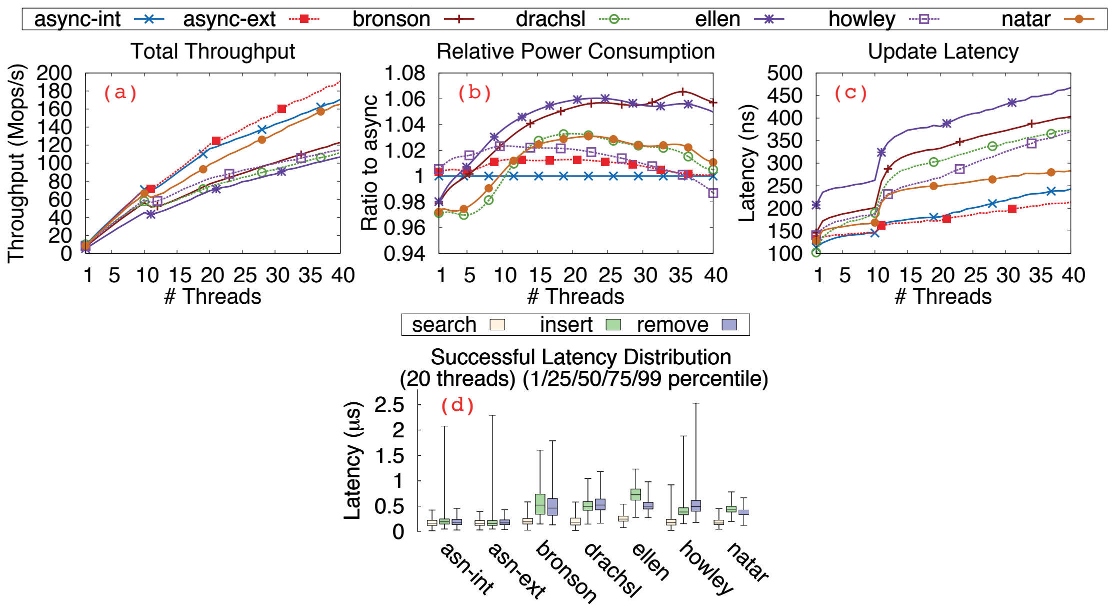


Figure 4.10 – BST with 2048 elements and 20% updates (10% successful).

algorithms, we have shown that CSDS algorithms can be improved even further when the ASCY patterns are applied collectively. Moreover, as seen in §4.3, for each data structure and platform combination, there is an algorithm that is reasonably close in performance to the asynchronized executions, in general a CSDS that resembles the sequential algorithm. In this section, by applying ASCY and bringing these algorithms even closer to their sequential counterparts, we have further improved their performance and scalability. We can therefore conclude that ASCY can help reach implementations that are portably scalable.

4.5 Designing with ASCY from Scratch

We illustrate the use of ASCY on the design of two new search data structure algorithms: (i) a hash table (CLHT), and (ii) a binary search tree (BST-TK). We only present the high-level ideas of the algorithms, and refer the reader to [34] and [152] for further details of these algorithms.

4.5.1 Cache-Line Hash Table (CLHT)

CLHT captures the basic idea behind ASCY: avoid cache-line transfers. To this end, CLHT uses cache-line-sized buckets and, of course, follows the four ASCY patterns. As a cache-line block is the granularity of the cache-coherence protocols, CLHT ensures that most operations are completed with *at most* one cache-line transfer.

CLHT uses the 8 words of a cache line as:

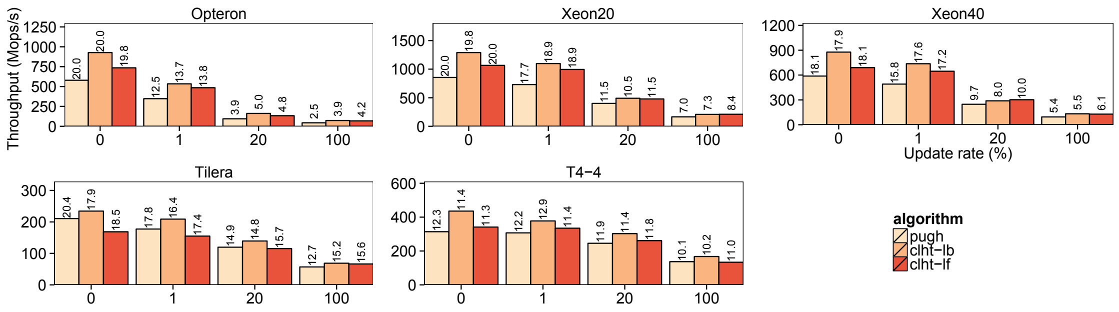


Figure 4.11 – CLHT with 4096 elements on 20 threads for various update rates.

`concurrency`	k_1	k_2	k_3	v_1	v_2	v_3	`next`

The first word is used for concurrency-control; the next six are the key/value pairs; the last is a pointer that can be used to link buckets. Updates synchronize based on the `concurrency` word and do in-place modifications of the key/value pairs of the bucket. To support in-place updates, the basic idea behind CLHT is that a search/parse does not simply traverse the keys, but obtains an *atomic snapshot* of each key/value pair[6]:

```
val_t val = bucket->val[i];
if (bucket->key[i] == key && bucket->val[i] == val)
    /* atomic snapshot of key/value */
```

CLHT comes in two versions, lock-based (CLHT-LB) and lock-free (CLHT-LF), which are further discussed in [152].

Evaluation. We compare CLHT to *pugh*, one of the best performing hash tables in §4.3. In contrast to the linked-based hash tables, CLHT performs in-place updates, thus avoiding memory allocation and garbage collection of hash-table nodes. Nevertheless, we use the SSMEM allocator for values.

Figure 4.11 includes the results. Noticeably, *clht-lb* and *clht-lf* outperform *pugh* by 23% and 13% on average, respectively. CLHT's design significantly reduces the number of cache-line transfers. For example, on the Opteron for 20% updates, *clht-lb* requires 4.06 cycles per instruction, *clht-lf* 4.24, and *pugh* operates with 6.57. Interestingly, *clht-lb* is consistently better than *clht-lf* on 20 threads. On more threads (e.g., 40), however, *clht-lf* often outperforms *clht-lb*.

4.5.2 BST Ticket (BST-TK)

BST-TK reduces the number of cache-line transfers by acquiring less locks than existing BSTs. Intuitively, on any lock-based BST an update operation parses to the node to modify and, if

[6]For an atomic snapshot to be possible, the memory allocator of the values must guarantee that the same address cannot appear twice during the lifespan of an operation. Additionally, the implementation has to handle possible compiler and CPU re-orderings (not shown in the pseudo-code)

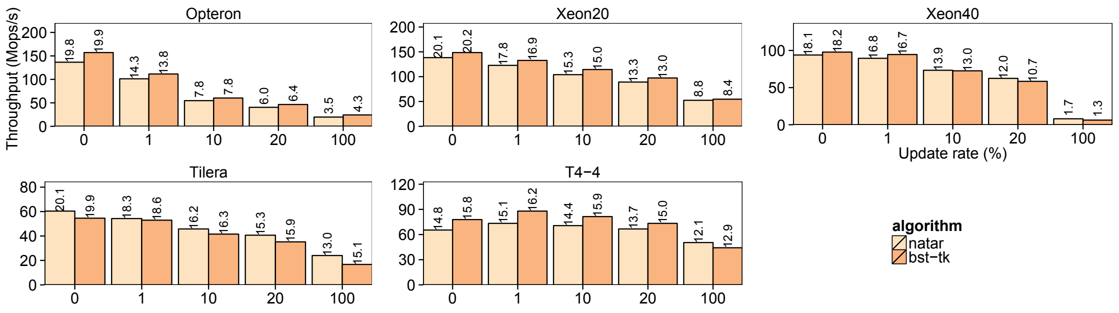


Figure 4.12 – BST-TK with 4096 elements on 20 threads for various update rates.

possible, acquires a number of locks and performs the update. This is precisely how BST-TK proceeds.

More specifically, BST-TK is an external tree, where every internal, router node is protected by a lock and contains a version number. The version numbers are used in order to be able to optimistically parse the tree and later detect concurrency. Overall, BST-TK acquires one lock for successful insertions and two locks for successful removals. The approach used to design BST-TK is further generalized through the Optik pattern [63].

Evaluation. We compare BST-TK to *natarajan*, the best performing BST in §4.3. Figure 4.12 depicts the results. In general, *bst-tk* behaves very similarly to *natarajan* (within 1% on average). It might have been expected that *bst-tk* would outperform the latter, because it uses less atomic operations per update. Although this is true, *bst-tk* has slightly increased parsing overhead compared to *natarajan* (0.045% vs. 0.032% with 20% updates on the Opteron). For simplicity, we did not implement certain optimizations that could prove beneficial under high contention. For instance, an insertion does not have to be restarted if the router node is locked by another insertion. Instead, it can be blocked and wait for the ongoing insertion to finish and then proceed almost normally.

4.6 Concluding Remarks

This chapter introduced *asynchronized concurrency (ASCY)*: a paradigm consisting of four complementary programming patterns to govern the design of portably scalable concurrent search data structures (CSDSs). We showed that ASCY can be used both to optimize existing algorithms and to assist in the design of new ones. In particular, using ASCY, we have optimized 10 state-of-the-art algorithms and designed 2 new algorithms from scratch, a hash table (CLHT) and a binary search tree (BST-TK). These are part of ASCYLIB, a new CSDS library that contains 34 highly-optimized cross-platform implementations of linked lists, hash tables, skip lists, and BSTs. ASCYLIB is available at http://lpd.epfl.ch/site/ascylib.

It is important to note that it is not always straightforward to apply some of the ASCY patterns. For instance, internal BSTs require either helping (e.g., *ellen*) or additional structures (e.g., *drachsler*) to implement ASCY$_{1-2}$. Similarly, in order to apply ASCY$_3$ on some lock-based hash tables, such as *java* and CLHT, we have to add a complete search operation before starting with the code of the update. As conveyed by our results, doing so is beneficial overall, because it reduces the coherence traffic. Enabling ASCY in these cases, however, results in overhead in successful updates.

Clearly, we expect ASCY to be applicable to other *search* data structures, such as prefix-trees, or B-trees. However, given that the ASCY patterns are based on the breakdown of operations to search and to parse-then-modify updates, some of the patterns might be meaningless for other abstractions such as queues and stacks. Still, we argue that the basic principle of ASCY (i.e., bring the concurrent-software design close to the asynchronized one) is generally beneficial.

Finally, it is important to note that we have focused this work on the basic CSDS interface, which is the common denominator for all search data structures. We did not consider data-structure-specific operations, such as *iterations, move*, or *max*. It is not clear whether it is easy, or possible, to implement these on top of ASCY-compliant CSDSs.

5 Blocking concurrent search data structures can be practically wait-free

In the previous chapter, we identified a set of patterns that yield portably scalable concurrent search data structures by studying coarse-grain metrics. In this chapter, we look at the progress of individual operations, and argue that there is virtually no practical situation in which one should seek a "theoretically wait-free" algorithm at the expense of a state-of-the-art blocking algorithm in the case of search data structures: blocking algorithms are simple, fast, and can be made "practically wait-free".

We draw this conclusion based on the most exhaustive study of blocking search data structures to date. We consider (a) different search data structures of different sizes, (b) numerous uniform and non-uniform workloads, representative of a wide range of practical scenarios, with different percentages of update operations, (c) with and without delayed threads, (d) on different hardware technologies, including processors providing HTM instructions.

We explain our claim that blocking search data structures are practically wait-free through an analogy with the birthday paradox, revealing that, in state-of-the-art algorithms implementing such data structures, the probability of conflicts is extremely small. When conflicts occur as a result of context switches and interrupts, we show that HTM-based locks enable blocking algorithms to cope with them.

5.1 Introduction

As we have seen in the previous chapter, with multi-core architectures being ubiquitous, concurrent data structures have become performance-critical components in many widely-used applications and software systems. In particular, *search data structures* are heavily used by numerous popular systems, such as Memcached [114], RocksDB [47], LevelDB [59], MySQL [126], MongoDB [123], MonetDB [122] and the Linux kernel [111]. We recall that search data structures are implementations of the *set* abstraction: they provide operations to search for a particular element, to insert, and to remove an element. The most common examples include *linked lists*, *skip lists*, *hash tables*, and *binary-search trees (BSTs)*.

However, despite the fact that a large body of work has been dedicated to concurrent search data structure (CSDS) algorithms [19, 35, 40, 43, 46, 54, 64, 66, 70, 77, 94, 99, 116, 129, 137, 150, 151], their design and implementation remains an onerous task. The difficulty lies in providing implementations that are both correct, i.e., linearizable [75], as well as efficient. Essentially, an ideal concurrent data structure (i) is easy to design, implement and reason about, (ii) provides high aggregate throughput and scalability, and (iii) ensures limited latency delays due to concurrency for all requests.

In theory, *wait-free* algorithms [69, 74] are the only ones that satisfy requirement (iii). These algorithms prevent known issues of locking (i.e., convoying, deadlocks, priority inversions), without the risk of starvation. In essence, a wait-free algorithm guarantees that every thread completes its operation in a finite number of its own steps, thus promising limited latency, even under high contention.

Nevertheless, despite the significant amount of research dedicated to the design of wait-free CSDSs [2, 50, 69, 94, 95, 150, 151], these algorithms exhibit low throughput, roughly half of that of a state-of-the-art *blocking* or *lock-free* search data structure (as we show experimentally in this chapter). Essentially, the reason for the difference in throughput between wait-free algorithms on the one hand, and lock-free and blocking algorithms one the other hand, is related to the amount of concurrency-related information that needs to be associated with the data. While in the case of lock-free and blocking algorithms, data and concurrency-related information can be efficiently manipulated in an atomic manner, this is not the case for wait-free algorithms. This results in more pointer chasing and thus slower traversals of the structures.

Lock-free algorithms have been shown to provide high aggregate throughput in the case of CSDSs, as we have seen in the previous chapter (and have been proven to ensure wait-freedom with high probability for a large class of schedulers [3]). However, they are difficult to design and implement correctly [61], with memory management making the task even more complex [30, 117].

Blocking algorithms are considered to be significantly easier to implement and use, mainly because data can only be modified in critical sections protected through mutual exclusion mechanisms. It is thus natural to ask whether it is possible to have blocking CSDS algorithms that provide high aggregate throughput, but also ensure that on realistic workloads, no individual request is significantly delayed due to contention, i.e., these algorithms are *practically wait-free*.

We believe this question can only be answered empirically. To this end, we conduct the most exhaustive study of blocking CSDSs to date. We deploy a wide variety of state-of-the-art CSDS algorithms, on the latest hardware technology. In particular, we enable locks to use Hardware Transactional Memory (HTM) in order to prevent threads from holding locks while not scheduled, which is a rather novel use-case for HTM. Our evaluation is extensive both in the metrics we collect, as well as in the scenarios we study: we consider numerous uniform and non-uniform workloads, representative of a wide range of practical scenarios. We also test different data structures with varying sizes and percentages of update operations, with and without delayed threads.

Our results indicate that state-of-the-art blocking implementations of search data structures behave practically wait-free in all realistic scenarios. Even in scenarios of extreme contention, requests experience acceptable delays with blocking CSDSs. The usage of HTM reveals effective in the face of frequent context switches and other interrupts.

The main conclusion we draw from our study is that there is virtually no practical situation in which one needs to seek an algorithm providing a strong theoretical progress guarantee at the expense of a state-of-the-art blocking algorithm in the case of search data structures: blocking algorithms provide good throughput and latency, even under high contention, and are much simpler than their non-blocking counterparts.

Our explanation for this fact is that state-of-the-art blocking CSDSs are specifically devised to minimize the probability of high contention for any memory address. This reason is different from the perspective taken by recent work [3], namely that if thread accesses are scheduled stochastically, lock-free algorithms behave in a wait-free manner even if there is high contention for a particular memory address. Our conclusion suggests instead that the main reason for non-wait-free CSDS algorithms, in the theoretical sense, to behave wait-free in the practical sense is the small probability of actual contention for a surprisingly wide spectrum of workloads. This follows from the fact that, in the case of state-of-the-art CSDSs, only short portions of the update operations may cause conflicts when occurring concurrently. State-of-the-art blocking algorithms simply go through the nodes and follow the next pointers, and only lock the area which needs to be modified. Indeed, the time needed to traverse the structure in order to reach the point where the operation needs to actually be performed generally dominates the total execution time. In addition, problematic scenarios, such as large delays that may occur due to context switches or interrupts can often be handled efficiently using HTM technologies available in commodity processors.

The probability of conflicts can be modeled using variations of the *birthday paradox*. We show that in common workloads, this probability is below 1%, with the probability of repeated conflicts for the same request being much smaller. Even in contended situations with a high number of conflicts, it is extremely rare that a wait-free algorithm outperforms state-of-the-art blocking algorithms.

Four remarks are, however, in order. (1) We do not claim that blocking search data structure algorithms behave in a wait-free manner under *every* conceivable scenario. We could create a scenario where (i) the data structure has a small number of nodes, (ii) these are accessed repeatedly, (iii) by a very large number of concurrent threads, and (iv) with a high update rate, and in which latency would indeed suffer.[1] (2) Our conclusion only applies to search data

[1]We do however claim this does simply not occur in the vast majority of practical situations. Practitioners, which often have some knowledge about their workloads and system requirements, can use our work to decide when blocking implementations are sufficient.

structures. Indeed, we show in the chapter that in the case of data structures which have the potential of inducing much more contention on a small number of memory addresses, such as queues and stacks, a blocking algorithm is not ideal. (3) We do not claim that non-blocking algorithms cannot offer the same performance as blocking ones. In fact, several lock-free algorithms are known to provide performance comparable to blocking algorithms. Rather, we show that state-of-the-art blocking algorithms, which are often much simpler to design and implement than their non-blocking counterparts, provide no disadvantage. (4) Our claims apply to data structures designed for volatile memory, such as DRAM. As we shall see in the next part of this dissertation, lock-free algorithms represent a better alternative in a durable setting.

The rest of the chapter is structured as follows. We discuss CSDSs in Section 5.2. We present our experimental methodology and settings in Section 5.3. In Section 5.4, we show that blocking CSDSs provide high throughput and scalability, while in Section 5.5 we perform an extensive evaluation of the degree to which blocking CSDS algorithms are practically wait-free. Section 5.6 discusses the analogy with the birthday paradox. We examine the extent to which our conclusions apply to structures other than CSDSs in Section 5.7. We conclude the chapter in Section 5.8.

5.2 Concurrent Search Data Structures: From Theory to Practice

In this section, we look at the metric types indicative of the performance of a concurrent algorithm: coarse-grained and fine-grained. Moreover, we clarify why theoretically wait-free algorithms fail to perform well in the former, and explain what we mean by practical wait-freedom in the context of blocking algorithms.

5.2.1 Performance and progress

The performance of concurrent algorithms is usually evaluated according to two main types of metrics:

- *coarse-grained performance metrics*: these capture the overall performance of the system, and usually include metrics such as throughput (the number of requests completed system-wide per unit of time), average latency (average duration of an operation), scalability (variation in throughput and latency as more threads are added to the system), and fairness (the difference in observed throughput and average latency between threads);

- *fine-grained performance metrics*: these metrics capture the behavior of individual requests; these include for example metrics such as latency distribution, variability, outliers, as well as other algorithm-specific metrics. Such metrics might be useful, for instance, to identify the quantity of particularly slow requests in a system.

An ideal algorithm would provide good results for both these metric types. However, optimizing for one of them can have a negative impact on the other. For example, attempting to maximize system throughput might result in an algorithm with a large variability in the operation latencies. Ensuring bounded latencies for all requests is likewise likely to limit throughput.

One of the main knobs used to adjust the relative importance of these metrics is the progress guarantee provided by the algorithm. The vast majority of published algorithms are either (i) *blocking*, (ii) *lock-free*, or (iii) *wait-free* [74].

- *Blocking* algorithms ensure mutually exclusive access to (parts of) the data structure using, for example, mechanisms such as locks. More broadly, threads have to explicitly release resources before others can use them, thus potentially preventing the progress of other threads indefinitely.
- *Lock-free* algorithms ensure that at least one thread in the system is able to make progress at any point in time.
- *Wait-free* algorithms ensure that every thread in the system will eventually complete its operation.

As we pointed out, blocking algorithms are generally simple. The downsides are the potential of several threads being indefinitely delayed by a slow thread holding a lock, deadlocks, livelocks, convoying or priority inversions. Lock-free algorithms present the risk of some threads starving. In theory, wait-freedom is the most desirable property for a concurrent algorithm. It ensures that no request is indefinitely delayed due to contention.

A significant amount of effort has been dedicated in recent years [50, 94, 95, 150, 151] to dispel the belief that wait-free algorithms provide low throughput in practice [53, 74]. In the context of search data structures however, such algorithms still lag significantly behind alternatives providing weaker progress guarantees in terms of system-wide throughput, as we will discuss below.

5.2.2 A closer look at wait-free CSDS algorithms

We recall some details regarding search data structures. A search data structure contains a set of elements, and allows access to each of them, regardless of their position in the data structure. These structures store elements of arbitrary sizes, indexed using keys, and are usually implemented as linked data structures. They have a simple base interface, consisting of three operations:

- *get(k)* returns the value associated with key k in case such an entry exists, or returns false in case the entry is not present;

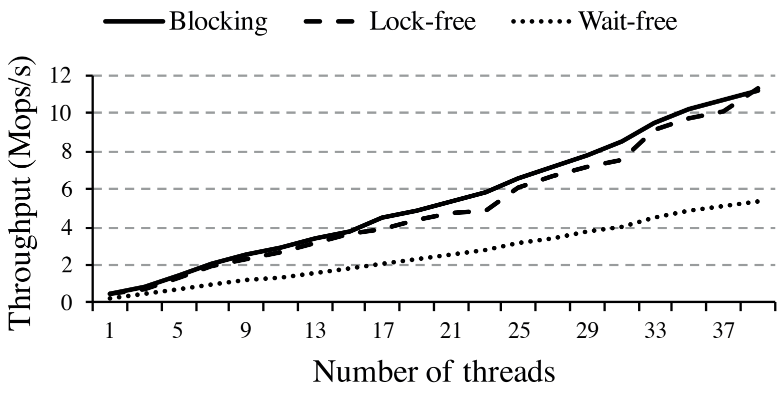


Figure 5.1 – The throughput of blocking, lock-free and wait-free linked lists of size 1024, with 10% of the operations being updates.

- *put(k,v)* inserts a key-value pair in case an entry for key *k* is not present, or returns false otherwise;
- *remove(k)* removes the entry corresponding to key *k* in case it exists in the data structure, or returns false otherwise.

Given the practical importance of these structures, a lot of effort has been dedicated to their efficient concurrent implementations.

In the following, we illustrate the current difference in throughput between a state-of-the-art wait-free algorithm [151], a state-of-the-art blocking algorithm [66] and a lock-free algorithm [64] for a linked list. We use a recent 20-core (40 hardware threads) Intel server. We depict the throughput of a linked list with 1024 elements and 10% updates (5% inserts, 5% removes) as we increase the number of threads from 1 to 40.

As conveyed in Figure 5.1, the throughput of the wait-free algorithm is around 50% of its blocking and lock-free counterparts. This trend pertains to other CSDSs as well (skip lists, BSTs and hash-tables[2]).

The main reason for the difference in throughput between wait-free and blocking CSDSs is that in a CSDS, the time needed to traverse the structure in order to reach the point where the operation needs to actually be performed generally dominates the total execution time. In essence, the smaller the number of memory locations an operation needs to read or write, the faster the operation. Due to their simpler nature, blocking algorithms have to chase a smaller number of pointers during an operation: the most efficient such algorithms simply have to go through the nodes and follow the next pointers, and only lock the area which needs to be modified.

[2]In the case of a hash table with average occupancy per bucket equal to 1, and hence on average no linked components, the wait-free algorithm is only around 33% slower. For the others, it is also 50%.

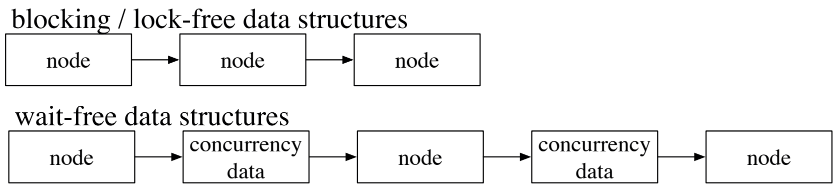

blocking / lock-free data structures
node
node
node
wait-free data structures
node
concurrency data
node
concurrency data
node

Figure 5.2 – CSDS traversals.

Linked lists	**Skip lists**
Lock-coupling list [74] Lazy linked list [66] Pugh linked list [137] Copy-on-write linked list [132]	Pugh skip list [137] Herlihy skip list [70]
Hash tables	**BSTs**
Lock-coupling hash table [74] Lazy hash table [66] Pugh hash table [137] Copy-on-write hash table [132] ConcurrentHashMap [99] Intel TBB [86] URCU hash table [40]	Practical binary tree [19] Logical ordering tree [43] BST-TK tree [35]

Table 5.1 – Blocking search data structure algorithms considered in our evaluation.

In the case of wait-free algorithms, the underlying reason for their inefficiency is a fundamental mismatch between current architectures and the complexity of wait-free algorithms. In non-blocking CSDS algorithms, the common approach is to associate some concurrency information with each *next* pointer of a node, and update this information and the pointer atomically. While the unused three least significant bits of a pointer (in a 64-bit architecture) are generally sufficient for this concurrency information in the case of lock-free algorithms, this is not the case for the most efficient wait-free algorithms (which might need, for example, version numbers associated with next pointers). In terms of practical implementations, this usually translates in additional objects being interposed between data structure nodes, resulting in slower traversals of the structure. This is illustrated in Figure 5.2. Thus, wait-free algorithms fail to provide at least one of the characteristics of an ideal concurrent algorithm we have previously identified: high aggregate throughput.

Of course, our experiment tells only half of the story: the main goal of wait-freedom is not high system-wide throughput, but rather the promise that every request will eventually return (fine-grain latency). In the rest of the chapter, we focus on showing that in addition to providing good coarse-grained performance metrics (Section 5.4), blocking CSDS algorithms also exhibit "wait-free behavior" for a very wide variety of workloads (Section 5.5).

5.2.3 Practical wait-freedom

We now look at how the theoretical guarantees of wait-free algorithms manifest in practice in the context of concurrent search data structures.

In theory, threads may crash. Wait-free algorithms ensure that despite the crash of a thread, the requests of other threads still get served. In practice, threads usually do not crash, and when they do, they do not crash independently: even when software bugs occur in a multi-threaded program, it is preferable to stop or restart the entire application rather than work with state that might have been corrupted.

Threads can however be temporarily delayed due to I/O, context switches, scheduling decisions or other interrupts. In practice, we expect wait-freedom to translate to a bound in the delay a request can suffer due to contention, i.e., all requests finishing before a certain deadline as long as the thread itself is scheduled (i.e., taking steps).

We thus argue that a data structure implementation which has a negligible percentage of requests that exhibit significant delays due to other concurrent threads in the system for a wide array of workloads is *practically wait-free*. In practice, most systems indeed provide a Service Level Agreement (SLA) tolerating small percentages (e.g. 0.1% or 1%) of slow requests. Hence, what we call practical wait-freedom is bounded delays due to contention, for all but a potentially infinitesimal percentage of requests, under all realistic workloads.

This characteristic of an implementation does not imply a particular theoretical guarantee provided by the algorithm. It simply identifies implementations whose execution (in terms of request delays due to contention and fairness) cannot be distinguished from that of an algorithm that actually provides the theoretical guarantee of wait-freedom.

In the context of blocking CSDSs, there are two possible causes for which a thread's operation can be delayed by other concurrent threads: *locks* and *restarts*. To quantify the extent to which a thread is delayed as a result of them, we can measure the time an operation waits in order to acquire locks, as well as the number of times an operation has to restart. These two fine-grained metrics are indicative of different sources of delays to which wait-free algorithms may serve as better alternatives:

- Large average waiting times to acquire locks can be for the most part linked with i) other threads suffering delays while holding locks or ii) a large number of threads attempting to acquire the same lock. In the following sections, we show that in the context of state-of-the-art blocking CSDS algorithms, the latter case occurs only extremely rarely.
- In contrast, restarts capture a pattern present in a large fraction of the state-of-the-art blocking CSDS algorithms: if at some point during an operation, a state which does not allow it to progress correctly is encountered, the operation is restarted. These possible inconsistencies triggering restarts are a result of the fine-grained locking and optimistic approaches used in state-of-the-art algorithms. In general, a large percentage

of requests having to be restarted repeatedly can be linked to high contention for a particular area of the data structure. The theoretical possibility of triggering restarts indefinitely might result in threads suffering from starvation.

We note that a more generic metric such as latency distribution is not appropriate in the case of all linked search data structures, given the fact that request latencies are often dominated by the time needed to reach the point of the data structure which needs to be accessed. Therefore, accesses to different parts of the data structure naturally have very different latencies. In addition, if context switches or other interrupts occur, threads may not even be taking steps, resulting in potentially unbounded latencies even for a theoretically wait-free algorithm.

In the following, we look at the extent to which blocking CSDSs provide the desired coarse-grained performance metrics, as well as the fine-grained performance metrics indicative of practical wait-freedom discussed above.

5.3 Experimental Setting

For our study, we evaluated a wide range of blocking CSDS algorithms, summarized in Table 5.1, and part of ASCYLIB [10]. We report on their behavior under a wide range of conditions, representative of a wide variety of workloads. We enhance the library with benchmarks allowing us to report on the metrics presented in this chapter. Roughly, we argue that the state-of-the-art algorithms have good throughput, scalability and behave practically wait-free regardless of the search data structure, thus representing ideal implementations for the vast majority of workloads.

For clarity, in the following experiments, we only highlight the behavior of the blocking algorithm exhibiting the best performance in our tests for each data structure, as our intention is to show that at least one practically wait-free algorithms exists for each structure type. Nevertheless, the conclusions we draw are in fact valid for multiple state-of-the-art algorithms for each data structure.

The best performing algorithms per data structure (which are shown in the following figures) are the lazy linked list, Herlihy's skip-list, the lazy linked list-based hash table (one lazy linked list per bucket, with per-bucket locks, with average load factor per bucket set to 1), and the BST-TK external binary search tree.

5.3.1 State-of-the art algorithms

Before we go into any details, it is important to recall the principles according to which the best performing blocking CSDS algorithms operate today [35]:

- Read operations do not perform any stores, and do not trigger any restarts;

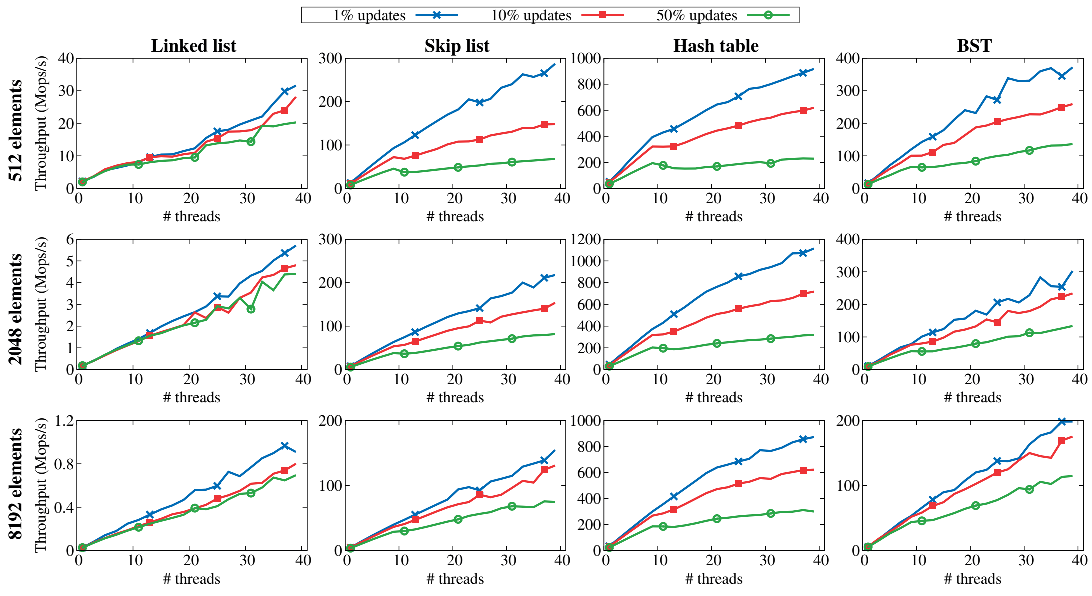


Figure 5.3 – Throughput scalability of blocking implementations.

- Update operations can be divided into a parse phase and a write phase: in the parse phase, the area of the data structure where the update needs to be applied is reached in a synchronization-free manner, while during the write phase, the actual updates are applied by writing to a small neighborhood of nodes in the data structure.
- Conflicts arise between two threads if they are executing their write phase concurrently, and the nodes accessed during these phases by the two threads intersect.
- Parses and reads are not obstructed by concurrent updates.

In short, in all these algorithms, the only blocking portions are short sequences of code in the update operations in which the actual modifications to the data structures are performed.

5.3.2 Implementation details

Our structures use either test-and-set locks or ticket locks in our experiments. We observe no benefits from using more complex locks, such as MCS locks, due the low degree of contention for any particular lock in these data structures. In addition, it has been shown that these simple locks often perform well in practice [36], even for moderately high degrees of contention. Our implementations use an epoch-based memory management scheme, similar in principle to RCU [112].

In order to test our implementations, we use an Intel machine having two Xeon E5-2680 v2 Ivy Bridge processors with 10 cores each (20 cores in total). In addition, each core supports two hardware threads. The cores run at 2.8 GHz and have caches of sizes 32KB (L1), 256 KB

(L2) and 25 MB (LLC per processor). The machine runs Ubuntu Linux 14.04 (kernel version 3.13.0). We bind threads to cores such that threads first use all physical cores before using both hardware contexts of the same core.

5.3.3 Methodology

We present results for different structure sizes and update ratios, which we believe to be representative of those used in practical systems. We consider structures consisting of 512, 2048 and 8192 key-value pairs, and percentages of update requests of 1%, 10%, and 50%. Half of the updates are inserts, and the other are removes. These parameters, are, in fact, similar to the ones we observe in real systems (e.g., in LevelDB [59], RocksDB [47], Memcached [114], MySQL [126], MongoDB [123], MonetDB [122]). We also address smaller structures and higher degrees of contention in a separate experiment.

We evaluate the extent to which blocking CSDSs provide the two main features of an ideal concurrent algorithm: (i) high system-wide performance, and (ii) practical wait-freedom. We start by showing that blocking CSDSs provide the desired coarse-grained performance metrics, after which we look at the fine-grained metrics indicative of wait-freedom, and study how they evolve as we modify the parameters of our workload and environment.

Unless otherwise specified, the distribution of accesses over the key space is uniform. Keys and values have 64 bits in size. Using larger values is straightforward: instead of the 64-bit value we would simply manipulate pointers to these larger values. Each of the worker threads in our benchmarks continuously issues requests.

In each of the workloads, we consider a key space twice as large as the structure size. Given that our workloads have an equal number of inserts and removes, this ensures that on average the data structure size remains close to the initial size throughout the experiment. We use runs of 5 seconds and report the average results of 11 runs.

In order to capture the system-wide performance and the degree to which the individual requests of these algorithms are delayed, we consider the following metrics: (a) the throughput as we increase the number of threads from 1 to 40 (the maximum number of hardware threads on our machine), (b) the average throughput per thread and the standard deviation of this quantity, (c) the average percentage of time spent waiting for locks by each thread and the standard deviation, and the (d) percentage of operations that have to restart at least once. We also look at the distribution of the values for the last two metrics among requests to identify any outliers. Where not specified, measurements are taken using 20 concurrent threads.

5.4 Coarse-Grained Metrics

In this section, we evaluate the first performance characteristic of an ideal CSDS: we verify the extent to which blocking CSDSs provide good aggregate throughput and scalability for various

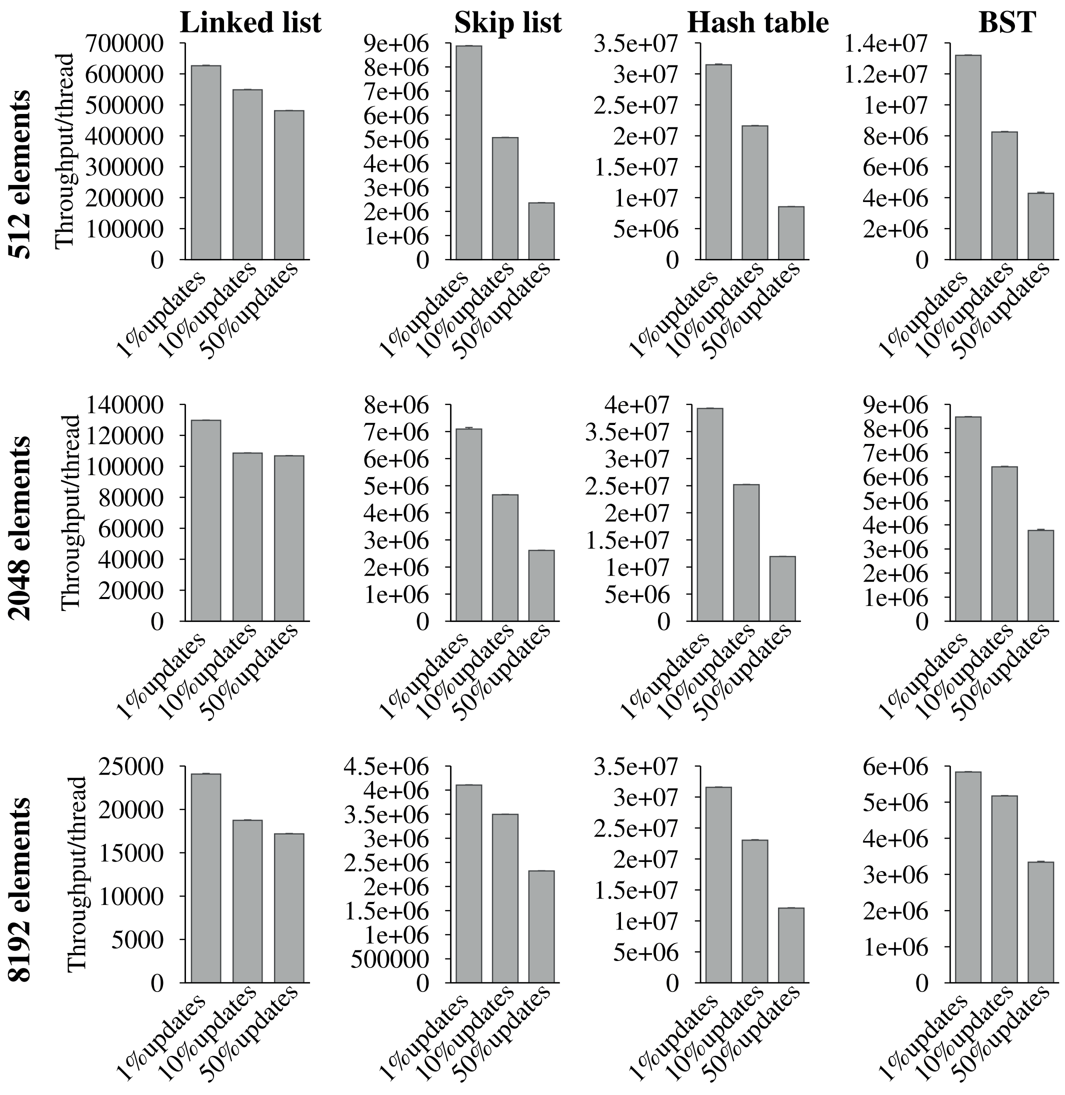


Figure 5.4 – Per-thread throughput (and standard deviation).

structure sizes and update ratios.

Figure 5.3 presents the evolution of the throughput as a function of the number of threads. We note that the structures exhibit no decrease in scalability as the number of concurrent threads increases. Of course, as we increase the percentage of updates more cache invalidations are generated, resulting in slightly larger latencies and overall lower throughput increases. However, the scalability trends remain the same. This is particularly noticeable in the case of the hash table, due to the much higher incurred throughput and the lower latencies of the requests. Once we have to use both sockets of the multi-core (for more than 10 threads), the scalability slope slightly reduces due to increased latencies to access data. This effect of the higher cache coherence latencies on the total throughput is inherent to each data structure, and cannot be bypassed regardless of the algorithm and its progress guarantees [35, 36].

In addition, in Figure 5.4, we also present the average throughput per thread and the standard deviation of this metric. We depict the standard deviation using error bars. On average, the standard deviation is 0.2% of the average per-thread throughput. This quantity is so small compared to the per-thread throughput, that it is not visible on the graphs. This is valid for all the data structures and for all the workloads. Given that threads continuously issue requests, we can also conclude that the average latency is identical among threads. The observation we extract from this experiment is thus that blocking CSDSs ensure a high degree of fairness.

The conclusion we can draw from these experiments is that the blocking nature of these algorithms is not an obstacle to high throughput or scalability, even as we modify the size of the structure or the percentage of update operations. In addition, all threads exhibit high performance: there is no skew between the throughputs of the threads.

5.5 Practical Wait-Freedom

While as we have shown in the previous experiments, blocking algorithms have good throughput scalability and are fair, in this section we look closer at the fine-grained performance metrics indicative of practical wait freedom. As presented in Section 5.2, these metrics are the amount of blocking and retries as we vary the data structure size and update ratio.

We study how these metrics evolve as we very the percentage of updates, decrease the size of the data structure, use non-uniform workloads, cause threads to become unresponsive, and induce frequent context switches.

5.5.1 Structure size and update ratio

In this experiment we collect the performance metrics as we vary the structure size and the update ratio. Figure 5.5 presents the percentage of time threads spend waiting for locks. The percentage is relative to the total execution time when threads are continuously issuing requests. We measure this amount of time by using ticket locks: once a thread has acquired its ticket, if it is not immediately its turn to be served, we measure the time until this event occurs. We note that this percentage is under 2% in all situations, with most values being significantly below this percentage. In the case of the 8192-element linked list, for example, no thread has to wait in order to acquire locks. For the 2048-element linked list, the standard deviation is large due to the fact that the waiting times are between 0 and a few hundred cycles: even one brief delay waiting for a lock makes a thread an outlier. In the case of the BST, the tested algorithm uses trylocks, and restarts the operation in case the locking attempt fails. Therefore, the time spent waiting for locks is zero, but this is compensated by the slightly higher percentage of operations that are restarted.

Similarly, as shown in Figure 5.6, the percentage of operations delayed due to restarts is significantly smaller than 1% in all situations. This value is 0 in the case of the hash table: each

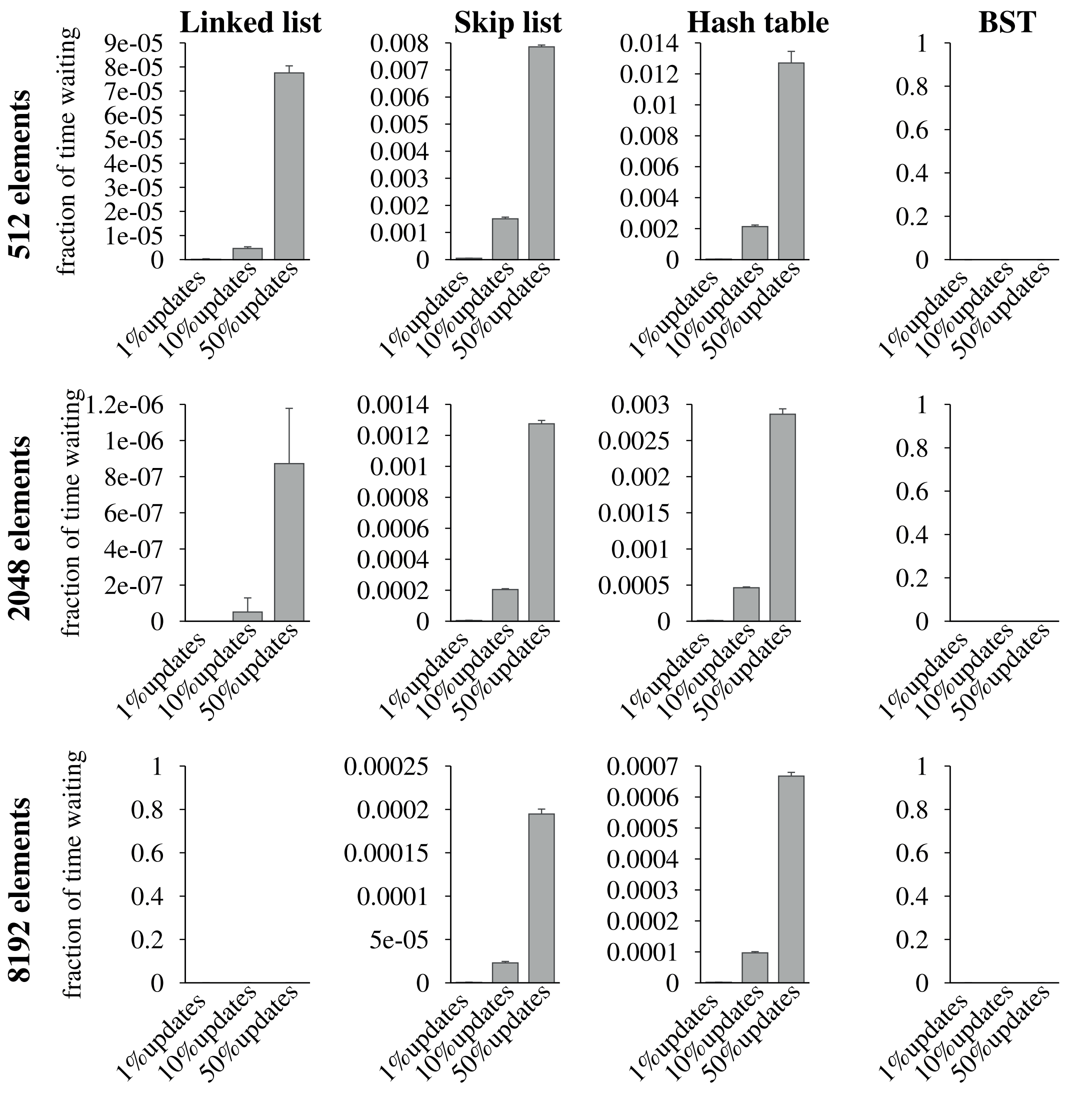


Figure 5.5 – Fraction of time threads spend waiting for locks (and standard deviation).

bucket is protected by a lock, so once the operations have acquired the lock they never restart.

We also run an experiment in which we look at the distribution of the two sources of delays (the number of restarts and the time spent blocked) on a per-request basis. We want to identify any outliers: requests significantly delayed due to concurrency, which would violate practical wait-freedom. We consider a workload using a linked list of 512 elements, 40 threads and 10% updates. Only 0.01% of the requests had to wait for locks, with no requests waiting for more than 6μs. In addition, out of the 26 million operations, 2900 had to restart once, 9 had to restart 2 times, and none had to restart more than that.

The conclusion we can draw from these experiments is that blocking CSDS algorithms cause negligible request delays due to concurrency, thus allowing threads to complete their requests

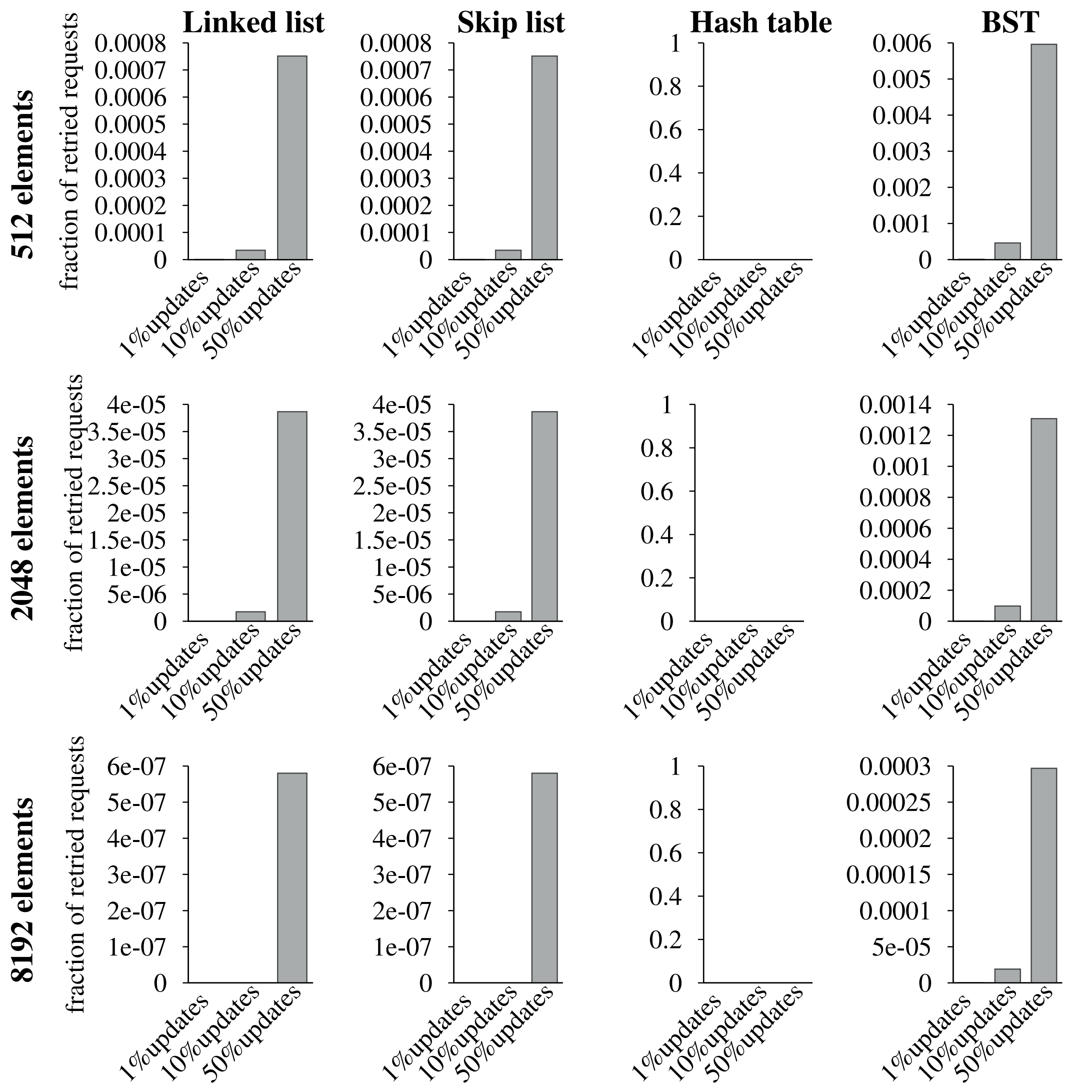


Figure 5.6 – Fraction of requests that are restarted.

in a finite number of their own steps. When these small delays do occur, there are no requests that are affected significantly more than others.

These metrics also allow us to confirm the fact that our conclusion applies to the state-of-the-art search data structure algorithms, and not to more naive implementations, which have more frequent or longer critical sections. For example, we consider a lock-coupling linked list [74]. This algorithm, while using fine-grained locks, acquires locks as the structure is traversed. We measure the percentage of time threads spend waiting for locks for this algorithm. With 20 concurrent threads and just 1% updates, threads spend around 10% of their time waiting for locks, regardless of the structure size. Therefore, we do not claim such algorithms are practically wait-free.

5.5.2 Non-uniform workloads

We now look at non-uniform workloads, in which some keys are more popular than others. We use a Zipfian distribution of requests over the key space with $s = 0.8$. Zipfian distributions are known to model a large percentage of real workloads [32]. We show results for each of the data structures, using a workload with 20 threads, 2048 data structure size and 10% updates. We provide numbers for the time threads have to wait for locks, and the average number of retries that have to be performed. The results of this experiment are depicted in Figure 5.7. While the values observed in this experiment are slightly higher than for uniform workloads, the delays remain very low: threads spent at most 1% of their time waiting for locks, and only restart at most 0.30% of their operations regardless of the data structure.

Thus, we can conclude that blocking CSDS algorithms behave practically wait-free on such non-uniform workloads as well.

5.5.3 High contention

We present the following experiment: we consider a scenario in which 40 threads concurrently access a data structure, with 25% of the operations being updates. We start with a structure having 512 elements and in subsequent runs reduce its size down to an extreme-contention configuration with a structure consisting of 16 elements on average out of 32 possible keys. We report the percentage of time threads spend waiting for locks, as well as the percentage of update operations that restart at least once, and more than three times (reads do not restart). The percentage of updates restarted at least three times provides a measure of the number of such requests that are significantly delayed. As before, the hash table does not restart (as we use per-bucket locks), while the BST does not wait for locks.

Figure 5.8 presents the results of this experiment. In the case of the linked list, in the most extreme contention configuration, with only 16 elements in the structure, threads spend about 30% of their time waiting for locks, with 20% of the operations restarting at least once, and 1.8%

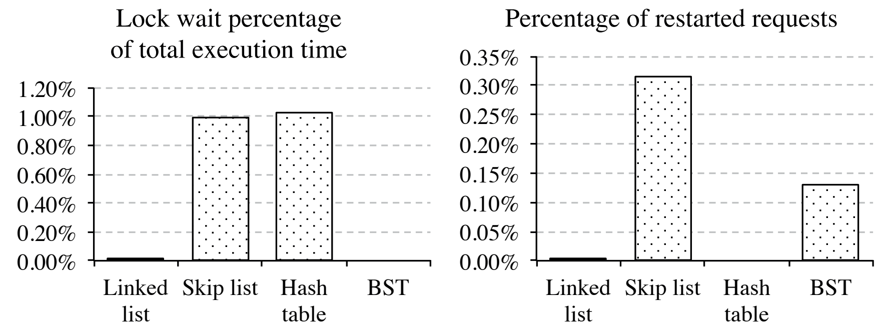


Figure 5.7 – Percentage of time spent waiting for locks and percentage of requests restarted for blocking search data structures on a workload with a Zipfian request distribution.

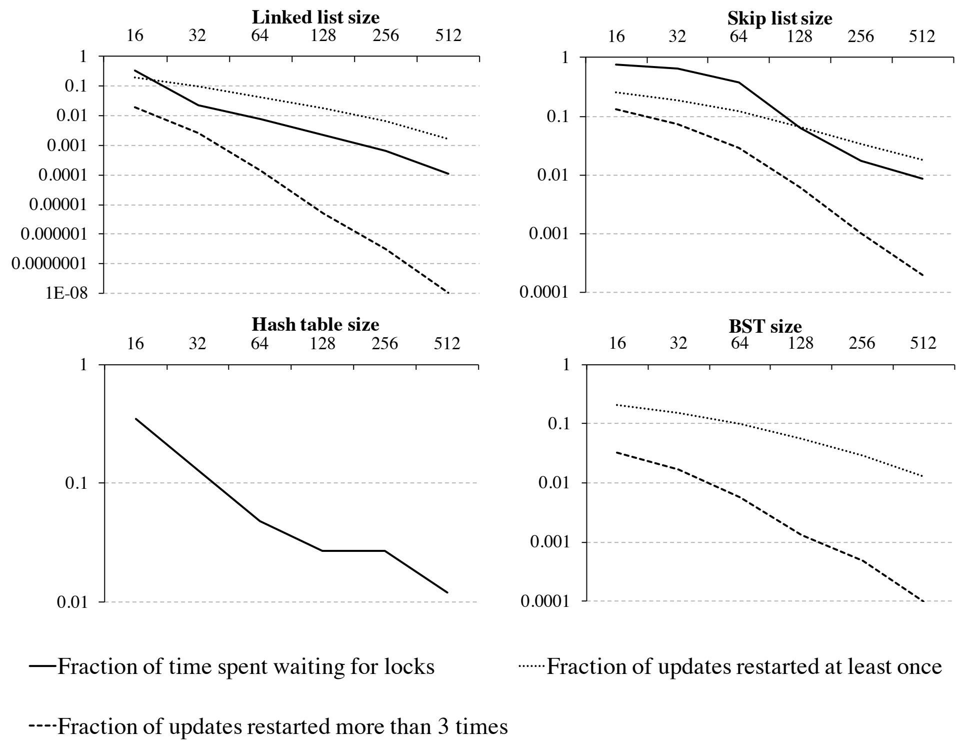


Figure 5.8 – Delayed requests and time spent waiting for locks as a function of structure size.

of the operations restarting more than three times. Arguably, these numbers stretch the limits of what could be considered as practical wait-freedom, and in this particular configuration non-blocking algorithms may represent a better alternative. However, a data structure of size 32 already spends only around 1% of the time waiting for locks, restarts only 0.6% of its operations and repeatedly restarts only 0.02% of them, thus warranting the claim of practical wait-freedom. The values of these metrics continue to decrease steeply with the increasing data-structure size. With a linked list of 512 elements, practically no requests are significantly delayed or restarted more than three times.

Other data structures behave similarly: while for the very smallest structures our metrics can be non-negligible, they decrease exponentially as we increase data structure size. In the case of the hash table, since there are no restarts and we use per-bucket locks, the time spent waiting for very small hash tables is somewhat larger than for the other structures. This can be addressed by using finer-grained locks: i.e, use per-node locks instead of per-bucket locks. Nevertheless, since the hash tables used in practice are usually large in size, we use per-bucket locks throughout the experiments in this chapter.

The conclusion we can draw from this experiment is that there are indeed some extreme cases in which blocking data structures exhibit a non-negligible percentage of delays. However, these cases require extremely small data structures, a very high degree of concurrency and relatively high update ratios. We argue that such particular situations are rare in practice in the context of CSDSs. In fact, we have not observed such highly contested structures in popular practical systems which use CSDSs. In the majority of cases, even under high contention, blocking CSDSs still behave practically wait-free.

5.5.4 Unresponsive threads

We now look at a scenario in which threads become slow. In the first instance, we cause a thread to be delayed for a random interval between 1000 and 100000 ns every 10 updates, while holding locks. This range of delays captures many of the events that might occur in practice, such as accessing data from memory, SSD, or sending packets over the network. In essence, given that threads usually hold locks for short intervals only, this experiment looks at a worst-case scenario, where delays only happen while the locks are held. We present results for a workload having 20 concurrent threads, using data structures consisting of 2048 key-value pairs and 10% update operations. In Figure 5.9, we provide results for the time spent waiting for locks and the number of restarted requests. Given that BST-TK uses trylocks [63], the waiting time for locks is normally 0. However, to better capture the effects of a slow thread on this algorithm, we depict the average time spent by threads on retries due to trylock failures. We note that threads spend at most 1% of their execution time waiting for locks. In fact, aside from the hash table, the percentages for the other data structures are significantly lower. We can lower this percentage for the hash table as well by replacing the per-bucket locks with finer grained locks for the linked lists representing each bucket. The percentage of restarted requests is also small: at most 0.015% for the skip-list. Thus, even under such workloads with temporary delays, the practically wait-free behavior of blocking CSDSs is not affected.

There is of course a limit up to which threads can be delayed before the system behavior is affected and practical wait-freedom jeopardized. In essence, an unresponsive thread might be problematic if (i) the delay happens while a thread is holding a lock and (ii) the thread is unresponsive for a large period of time. We find that such a scenario is possible, for example, in workloads where the number of threads significantly outnumbers the available cores (multi-

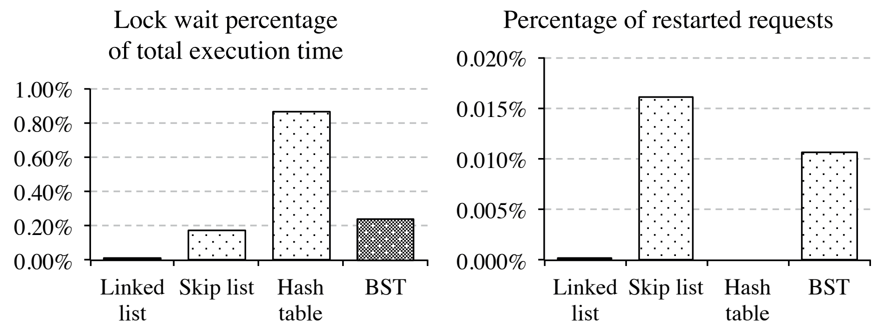


Figure 5.9 – Percentage of time spent waiting for locks and percentage of requests restarted for blocking search data structures when a thread repeatedly suffers delays.

programming). In an example using 4 threads per hardware context (160 threads in total), we find that around 3300 context switches occur every second. Roughly, that means that every thread executes for 12 ms, after which it is swapped out for 37 ms. Although critical sections in our algorithms are very short, a few of these context switches are bound to happen while locks are held. In such scenarios, while algorithms' performance is affected regardless of their progress guarantee, we find this to be more obvious in the case of blocking algorithms, particularly under workloads with higher update ratios.

To address this issue, we propose using hardware features recently introduced on modern architectures, such as Intel's Transactional Synchronization Extensions (TSX) [84], which allow us to elide locks. One characteristic of Intel's TSX implementations is the fact that hardware transactions are aborted when interrupts occur. While this is often regarded as a limitation [121, 163], we use the abort-on-interrupt characteristic of TSX to our advantage, as it enables us to maintain the practically wait-free behavior of CSDS algorithms even in the case of frequent context switches, I/O, or other interrupts: even if a thread is within a critical section when the interrupt occurs, the hardware transaction is aborted and locks are not held when threads are not scheduled. This technique does not change the blocking nature of the algorithms: since Intel TSX is best-effort only, we need to provide a fall-back path which uses the actual locks for the situation when a speculative execution of a critical section is repeatedly aborted. The effectiveness of this approach is thus contingent on hardware transactions not aborting extremely frequently.

As we have observed in our evaluation, and as we will further show in Section 5.6, the probability of contention between two threads is small. Therefore, only a very limited percentage of updates will be repeatedly aborted due to data conflicts, even when contention is high. In addition, the short critical sections in the write phase do not trigger interrupts in the common case. While it is possible for page faults to occur within a critical section, we argue that this is a fairly rare event, and, given that by using TSX we can re-try the optimistic transactional approach multiple times before reverting back to the pessimistic path, this does not jeopardize wait-freedom. Additionally, TSX also provides us with a degree of tolerance to failed threads. Although in general one would stop a concurrent system rather than work with corrupted state, using TSX-enabled locks could also prevent failed threads from leaving the system in an inconsistent state, or holding locks after crashing.

We validate our hypothesis using a 3 GHz 4-core (8 hardware threads) Intel Haswell Core i7-4770 machine. The processor has 32 KB L1, 256 KB L2, and 8 MB L3 caches. Our algorithms remain unchanged: we only add TSX instructions to the acquire and release methods of the locks. Except for BST-TK, which uses ticket trylocks, all the other algorithms use test-and-set locks. We show results for a workload using 32 concurrent threads (8/physical core), with data structures consisting of 1024 key-value pairs and workloads with varying update ratios. Given the frequent context switches, measuring the amount of time threads spend waiting to acquire locks is not an appropriate metric here. Instead, we measure the number of operations that fail to elide the locks and fall back to normal lock acquisition: it is only these operations that

Update ratio	Linked list	Skip list	Hash table	BST
20	0.001	0.011	0.001	0.000
50	0.001	0.012	0.001	0.000
100	0.001	0.014	0.002	0.001

Table 5.2 – Fraction of critical sections falling back to acquiring locks using 8 threads/physical core (32 threads in total) with data structures of size 1024.

have the potential of delaying other threads for longer amounts of time. We report this as a fraction of the total number of lock acquisition calls. The results are shown in Table 5.2. We note that in most cases the percentage of operations that acquire the locks is well below 1%. This number is slightly larger in the case of the skip-list, which needs to take multiple locks per update. The conclusion we can draw from this experiment is that the probability of a thread being de-scheduled while holding a lock is extremely small: an interrupt would have to occur while a thread is (i) in the write phase of an update operation and (ii) after the thread failed repeatedly to elide the locks. We theoretically estimate this probability in Section 5.6.

We also measure the increase in throughput we obtain in these scenarios when using TSX. Table 5.3 shows the ratio between the throughput of the TSX-enhanced versions of the algorithms and the default implementations. We note important improvements in all data structures. The benefits of using TSX are particularly impressive in the case of the skip list. This is due to the lager number of locks per update operation, which increases the potential of a thread being de-scheduled while holding a lock.[3]

The behavior revealed in the previous experiments allows us to conclude that TSX indeed enables us to maintain practically wait-free behavior even in the face of frequent context switches, which is a scenario particularly well-known for causing issues in the context of concurrent systems.

Depending on the level of privilege of the code, an alternative to our solution could be to make the critical sections non-preemptable. However, even this might be problematic in the context of virtualization: scheduling multiple virtual machines in the presence of locks is a well-known challenge [55, 93, 145, 154]. Essentially, the guest OS does not have control over the scheduling decisions of the hypervisor. Our proposed approach would maintain the practical wait-freedom of CSDS algorithms even in such a situation, and may prove beneficial for other applications in virtualized environments as well.

In this section, we have shown that blocking algorithms behave practically wait-free on structures of different sizes, with different percentages of updates, under non-uniform workloads, under extreme contention and with frequently unresponsive threads. While it is extremely

[3]We notice no important difference between the default and TSX-enabled implementations in other experiments used in this chapter.

Update ratio	Linked list	Skip list	Hash table	BST
20	1.11	10.6	2.46	2.21
50	1.23	20.21	3.06	2.65
100	2.26	53.28	2.75	2.56

Table 5.3 – Throughput improvements of TSX-enabled versions vs. default implementations using 8 threads/physical core (32 threads in total) with data structures of size 1024.

difficult for an evaluation to address every possible scenario, we argue that these experiments cover most situations that arise in practical systems using CSDSs. Thus, given the behavior of blocking CSDSs observed in this section, we conclude that in practice, for the vast majority of workloads, blocking CSDSs behave practically wait-free.

5.6 The Birthday Paradox

In essence, the explanation for the wait-free behavior of blocking CSDSs is that the probability of threads being delayed due to contention is very small. We say that a *conflict* occurs when a thread is blocked or restarted by another thread. In state-of-the-art CSDS algorithms, a thread can generally encounter a conflict only during the write phase of an update operation. We provide an estimation of this conflict probability using an analogy with the *birthday paradox*.

Assuming threads that continuously issue requests, we first estimate the fraction of time a thread spends doing update operations (in a concurrent execution, including the acquisition and release of locks):

$$f_u = \frac{u \times dur_u}{u \times dur_u + (1-u) \times dur_r} \tag{5.1}$$

Here u is the update ratio (the fraction of operations that are updates), dur_u is the average duration of an update, and dur_r is the average duration of a read.

We then estimate the fraction of time a thread spends in its write phase, where d_w is the average duration of the write phase, and d_p is the average duration of the parse phase (recall that in state-of-the-art algorithms updates have a parse phase and an update phase):

$$f_w = f_u \times \frac{d_w}{d_w + d_p} \tag{5.2}$$

It is then a matter of computing the probability of conflict between threads concurrently executing their write phase. This is highly specific to the data structure, the implementation, and the request distribution. In general however, it can be reduced to variations of the birthday paradox: the probability of randomly chosen variables being similar enough to each other to cause conflicts. We denote by $B_s(k,n)$ the probability that if k threads are concurrently executing their write phases on a structure s of size n, at least one conflict will arise.

The probability of conflict in a system with t threads is thus:

$$p_{conflict} = \sum_{k=1}^{t} \binom{t}{k} {f_w}^k (1 - f_w)^{t-k} B_s(k, n) \tag{5.3}$$

We now provide numeric examples for some of the structures and workloads employed in this chapter.

5.6.1 Hash table

Our hash table implementation has the particularity of using one lock per bucket. Hence, in the above equations, d_p is 0: the lock is acquired immediately after the update starts. In the case of the hash table, since conflicts only appear if two threads want to write to the same bucket concurrently, the problem reduces to the classical version of the birthday paradox. We therefore have:

$$B_{ht}(k, n) = 1 - \frac{\prod_{i=1}^{k-1}(n-i)}{n^{k-1}} \tag{5.4}$$

We first assume a uniform workload for simplicity (see below for a non-uniform case). In this scenario, an update operation takes approximatively twice as long as a read operation. We assume a scenario with 1024 buckets and 20 threads, with 10% of the operations being updates. f_u is then 0.18, f_w has the same value (given that d_p is 0), and thus, using the formulas above, we obtain $p_{conflict} = 0.0058$. Therefore, the probability of any thread being delayed due to concurrency at any point in time is about 0.58%, a small enough percent to lead to wait-free behavior.

5.6.2 Linked list

In the case of other data structures, this computation is slightly more complex. For example, in our linked list, in which a remove operation has to lock two consecutive nodes, we can use the solution to the "almost birthday paradox" [1] to provide an upper bound to the probability of conflict. In this case, we have:

$$B_{ll}(k, n) = 1 - \frac{(n-k-1)!}{(n-2k)!\,n^{k-1}} \tag{5.5}$$

We provide a numerical example, considering a slightly higher contention example, with a list of 512 elements, 40 concurrent threads and 20% updates. An important distinction with the hash table is that the parse phase dominates the latency of an update operation. In fact, on average, the write phase takes only around 10% of the parse phase duration. That also means that updates are only one tenth more expensive than reads. Applying the presented formulas, we obtain $f_w \approx 0.0215$. We apply the formula for computing the probability of conflict in the case of the almost birthday paradox and obtain $p_{conflict} = 0.0021$. In the case of the linked list, the probability of conflict is thus 0.21%, again a percentage permitting wait-free executions.

5.6.3 Uniform vs. non-uniform workloads

In the above, we have considered uniform workloads. Of course, one could also repeat the computations considering non-uniform accesses. In that case, a variation of the birthday paradox using non-uniform probability distributions would have to be employed. For instance, using a Poisson approximation to model the probability of conflicts in such scenarios, we have:

$$B_{non-uniform}(k,n) = 1 - e^{-\binom{k}{2}\sum_{i=1}^{n} p_i^2} \tag{5.6}$$

Here, p_i denotes the probability of element i in the structure to be accessed. Using our linked list example from above and a Zipfian workload with $s = 0.8$, the probability of conflict is 0.47%. This value is slightly larger than in the uniform case, but still well below 1%, allowing practically wait-free executions.

These examples do not take into account slow threads, assuming similar speeds for all participants. One could model a slow thread, by assuming that while a thread t_{slow} writes to a set of memory locations, the other threads do multiple operations, writing to multiple such sets of locations. We give below an intuition for the probability of conflict when using TSX to avoid threads holding locks being interrupted.

5.6.4 TSX-based algorithms

In the algorithm versions in which the critical sections are wrapped with TSX instructions, speculative execution can be attempted several times. We assume each transactional region is tried five times before falling back to actually taking the locks. In these versions of the algorithms, conflicts occur if data that is being written during the critical section is read or written by another thread, or if data read during the critical section is written by another thread. Hence, while as before, only threads which are in the write phase may be the victims of a conflict, in this case all other threads may be causing the conflict, even those in the process of reading data. We can continue to use Equation 5.2 in order to estimate the probability of a conflict. However, given that threads during their read or parse phases also need to be taken into account, the computation of the term $B_s(k,n)$ will be different, as we will illustrate in the following. With five re-tries before reverting to locking, we can estimate the probability of a thread reverting to locking as $p_{lock} = p_{conflict}^5$.

We use the same hash table and linked list examples as above. In the case of the hash table, by taking reads into account when computing the probability of conflict with k current writers on a hash table of size n, we have:

$$B_{ht-tsx}(k,n) = 1 - \frac{(n-k)^{t-k}\prod_{i=1}^{k-1}(n-i)}{n^{t-1}} \tag{5.7}$$

With the same numeric values as before, $p_{lock} = 0.0005\%$.

In the case of the TSX-enhanced linked list, we have:

$$B_{ll-tsx}(k,n) = 1 - \frac{(n-k-1)!}{(n-2k)!n^{k-1}}\left(\frac{(n-2k)(n-2k-1)}{n(n-k-1)}\right)^{t-k} \tag{5.8}$$

For our linked list example, this results in $p_{lock} = 0.001\%$ (in the case of this fairly contended linked list, the probability of at least a re-try of the transactional region is however non-negligible: 16%).

It is important to note the significance of the low probabilities of conflict computed throughout this section: it simply means that there is a 1% chance that *some* thread in the system is delayed at any point in time. This delay, as shown by our practical evaluation, is likely to be insignificant for any particular thread. In addition, after such small delays the thread quickly returns to its steady state, with no changes to the parameters considered in this section.

5.7 Beyond Search Data Structures

In the previous sections, we focused our attention on search data structures. One of the main reasons we are able to obtain a practically wait-free behavior for a large spectrum of workloads in the case of state-of-the-art CSDSs is the low probability of conflicts between concurrent operations: accesses are distributed over all the nodes contained in the structure. In this section, we briefly present the limits of our conclusion when applied to concurrent objects other than search data structures.

5.7.1 Intuition

Intuitively, in the case of data structures such as queues, stacks, priority queues, and counters, the accesses are not distributed among multiple memory addresses, but rather concentrated on a small number of "hotspots". In this case, blocking algorithms essentially serialize operations by only allowing one (or a small number) to execute at a time. Thus, each thread spends a significant amount of time waiting for its turn. HTM-based techniques, which mitigate delays due to interrupts for search data structures, are also not applicable for such data structures: virtually all hardware transactions would repeatedly abort due to data conflicts. In a search data structure, it is only possible to encounter similar scenarios if (i) only a handful of nodes in the CSDS are accessed (ii) continuously by a large number of threads (iii) with a large fraction of the accesses being updates.

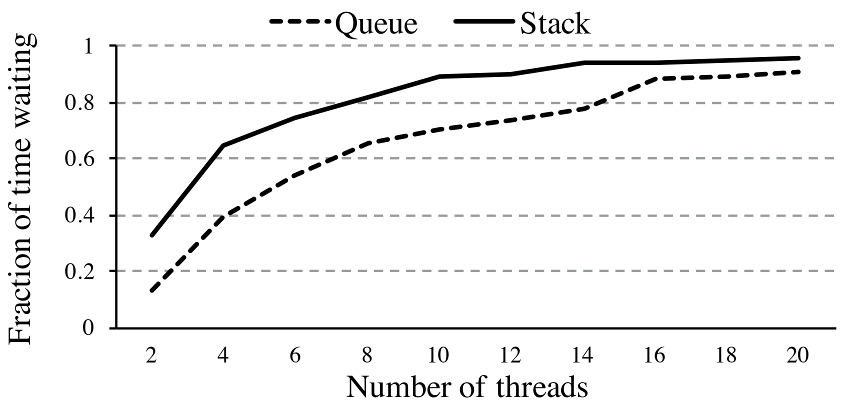


Figure 5.10 – Percentage of time threads spend waiting in blocking queue and stack implementations.

5.7.2 Experimentation

We quantify the extent to which these characteristics hinder practical wait-freedom using standard lock-based algorithms for a queue and a stack. Figure 5.10 shows the fraction of time threads spend waiting for locks in the case of these data structures. We use 20 concurrent threads, with 50% of the operations being inserts (enqueue/push), and 50% removes (dequeue/pop). The structures contain 1024 nodes. We note that the fraction of time threads spend waiting quickly approaches 1 as we increase the number of threads. This behavior is clearly not compliant with practical wait-freedom.

In the case of such data structures therefore, one can expect to obtain better performance by using lock-free or wait-free algorithms, to the design of which much work has been dedicated [50, 68, 87, 95, 119, 148, 156]. Another technique which has proved effective for such data structures is flat combining [67]. A further avenue which has been pursued in recent years in order to reduce the contention for any particular node in such structures is to relax the semantics of the data structure operations [5, 42, 165].

5.8 Concluding Remarks

The main conclusion we can draw from this work is that, practically, one can achieve the behavior of wait-free CSDS algorithms in terms of individual thread progress with blocking implementations. The nature of search data structures is such that there is no single contention point in the structure, rendering locks less problematic than they might be for structures such as concurrent queues, stacks, and counters. Our conclusions only concern CSDSs: we do not claim blocking implementations of objects such as queues and stacks are practically wait-free. It is also important to note that we do not claim that *every* blocking CSDS algorithm is practically wait-free. Rather, we considered state-of-the-art blocking algorithms, which generally have synchronization-free reads and writes with minimal and extremely fine-grained synchronization. We find such practically wait-free algorithms for each data structure we study. In addition to showing that these algorithms are practically wait-free, this work also represents the first detailed quantification of the effects of a progress guarantee on the behavior of practical CSDSs. Moreover, we have shown how new technologies such as Intel TSX can be leveraged to provide the desired performance characteristics of CSDSs. We also note that while the experiments presented in this chapter were run on an Intel Xeon server, we have verified our conclusions on other architectures as well, including servers from AMD and Oracle.

Durable Concurrent Data Structures Part III

6 Log-Free Concurrent Data Structures

Non-volatile RAM (NVRAM) makes it possible for data structures to tolerate transient failures, assuming however programmers have designed these structures in a way to preserve their consistency upon recovery. Previous approaches, typically transactional, inherently used logging, resulting in implementations that are significantly slower than their DRAM counterparts. In this paper, we introduce a set of techniques that, in the common case, remove the need for logging (and costly durable store instructions) both in the data structure algorithm as well as in the associated memory management scheme. Together, these generic techniques enable us to design what we call *log-free concurrent data structures*, which, as we illustrate on linked lists, hash tables, skip lists, and BSTs, can provide several-fold performance improvements over previous, transaction-based implementations, with overheads of the order of milliseconds for recovery after a failure. We also highlight how our techniques can be integrated into practical systems, by introducing a durable version of Memcached that is able to maintain the performance of its volatile counterpart.

6.1 Introduction

Fast, non-volatile memory technologies have been intensively studied over the past years, with various alternatives such as Memristors [147], Phase Change Memory [100, 138], and 3D XPoint [120] being proposed. Nevertheless, they are only now starting to become commercially available. Such memories, referred to as *non-volatile RAM (NVRAM)*, promise byte-addressability and latencies that are comparable to DRAM, yet also non-volatility and higher density than DRAM.

From a programmer's perspective, NVRAM can be read and written using load and store instructions, identically to DRAM. However, a significant fraction of software needs to be redesigned. Unlike DRAM on the one hand, in order to take advantage of NVRAM's non-volatility, the stored data needs to be in a state that allows the resumption of execution after a transient failure (e.g., a power failure). Unlike block-based durable storage on the other hand, the granularity at which data is read and written is much finer, and the latencies much smaller.

Thus, strategies that might have yielded the best performance in case of block-based storage might not be appropriate when working with NVRAM.

In this chapter, we focus on the design and implementation of concurrent data structures for NVRAM. Such structures are central to many software systems [47, 59, 114, 122, 123, 126]. Ideally, in the NVRAM environment, one would like concurrent data structures which can be recovered in case of a transient failure, and whose state would reflect all completed operations up to the failure, yet whose performance and scalability resemble those of their counterparts designed for DRAM.

However, this task is complicated by the fact that neither data stored in registers, nor caches, are durable in the face of transient failures. Moreover, by default, the program does not control the order in which cache lines are evicted and written to NVRAM. Explicit instructions, which we refer to as *sync* operations, must be used to ensure that a store is written through to NVRAM at the desired point. Finally, if the user expresses their updates as transactions or critical sections containing several stores, some form of logging is necessary for the eventuality of a failure in the middle of a transaction. This log needs to be reliably written before the transaction is executed. In all these scenarios, one must wait for stores to be written to NVRAM before proceeding, which is particularly expensive: whereas when using DRAM, one would at most wait for data to be written to the L1 cache, now one has to wait for data to be written all the way to NVRAM.

Previous approaches [23, 25, 29, 78, 88, 92, 96, 105, 113, 159] to implementing data structures for NVRAM relied mainly on transactions (either explicitly, or implicitly derived from critical sections), and focused on minimizing the associated logging overhead. The motivation of our work is to remove the need for logging in the data structure altogether, without incurring additional *sync* operations. We achieve this (in the *common case*, when we can take advantage of locality in memory allocation and reclamation) by using three techniques: (1) we reduce *sync* operations in our algorithms by using a *link cache*, (2) we remove logging in the data structures by using *lock-free* algorithms, and (3) we avoid logging associated with memory management by focusing on *coarse-grained* memory tracking. We briefly discuss these three techniques below.

The link cache is essentially an extremely fast, best-effort concurrent hash table stored in volatile memory, which contains data structure links that have not yet been durably written. When modifying the data structure, instead of ensuring updated links are durably written, we add them to the link cache. This enables us to avoid writing them to NVRAM one at a time. When the durable write of one of them is necessary for correctness, we batch the write-backs of all the links stored in our cache, which is significantly more efficient than waiting for writes to complete one at a time.

Logging in a concurrent data structure can be avoided using lock-free algorithms: intuitively, a lock-free algorithm must never bring a data structure in a state that prevents other threads from continuing normal execution. Thus, as long as the order of stores in the algorithm is reflected in NVRAM, no further logging is required. We also discuss how the store ordering requirement can be relaxed without jeopardizing correctness.

Finally, memory allocation and reclamation is also a central concern for concurrent data structures. Inserting and removing objects in a data structure consists of two main steps: (i) allocating (or deallocating) memory for the node, and (ii) adding (or removing) a pointer to the node in the data structure. When working with NVRAM, a crash between these two steps would result in a persistent memory leak or use-after-free problems. The traditional approach for avoiding such issues is again some form of logging. To avoid this, we propose *NV-epochs*, an epoch-based memory reclamation scheme for durable and concurrent data structures. NV-epochs groups memory nodes into memory areas, and reliably and persistently keeps track of the active (recently used) memory areas instead of individual allocations. This bookkeeping of active memory areas can be seen as the only form of logging in our approach. However, in the common case, due to locality in allocation and reclamation, the memory area an operation accesses will already be marked as active, and thus we do not have to wait for any additional store for memory leak prevention[1]. When recovering after a failure, we simply need to traverse the memory areas that were active at the moment of the crash and detect which objects belonging to these areas are still linked in the data structures. This is significantly faster than generic mark-and-sweep garbage collection for instance [16].

Each of these techniques is of independent interest, and can be used individually while maintaining its associated benefits. Together, these three techniques result in what we call *log-free durable concurrent data structures*, namely, durable concurrent data structures that, in the common case described above, require no form of logging. As we show in the chapter, these data structures provide up to an order of magnitude faster updates when compared to a traditional, log-based approach, in a single-threaded, as well as in a concurrent environment. Moreover, we achieve these benefits while maintaining low recovery times in case of restarts: even for gigabyte-sized structures, the time required to recover the structure is of only a few milliseconds. In terms of correctness, our implementations guarantee durable linearizability [89]. Briefly, all the operations that were completed before a crash are reflected after recovery.

We also highlight the practicality of our techniques by developing *NV-Memcached*, a persistent version of Memcached [114] that is based on a lock-free, durable hash table. NV-Memcached performs similarly to the volatile memory version of Memcached (more details in Section 6.5.6).

Our approach is however not a silver bullet. While it is extremely efficient for small and medium-sized data structures, its benefits are less apparent for very large data structures. And, as we discuss in this chapter, in a scenario with a large number of concurrent updaters, our link cache may limit scalability and might need to be turned off.

[1] For small and medium sized data structures, as we show, this covers more than 99% of memory operations.

To summarize, the contributions presented in this chapter are:

1. The *link cache*: a component that mostly eliminates sync operations in durable data structures;
2. *NV-epochs*: a coarse-grained durable memory management scheme that requires no logging in the common case;
3. A methodology for building *log-free* durable data structures starting from algorithms designed for DRAM;
4. *NV-Memcached*, a durable version of Memcached based on our techniques;

The rest of the chapter is organized as follows. We discuss our link cache in Section 6.2, and memory management in Section 6.3. We describe our log-free durable structures in Section 6.4. Experimental results are provided in Section 6.5, and Section 6.6 concludes the chapter.

6.2 Avoiding frequent write-backs: the link cache

We now introduce our first technique, aimed at minimizing the number of *sync* operations in durable data structures.

6.2.1 Link cache overview

In linked data structures, a new node becomes visible when a link to it from an existing node is atomically inserted. Once this happens, other operations can see that the node is present. Furthermore, in many algorithms, a node becomes logically deleted when a mark is atomically inserted on a link to signal deletion. After this, all operations enquiring about the state of this node will consider the node as no longer in the structure. A node becomes unreachable when the last link to it from another node in the data structure is atomically removed.

All these operations change the fundamental state of the node, and determine the return value of other operations which depend on the particular node. In the context of NVRAM, in order to ensure durable linearizability, it is therefore essential for each operation to ensure that the data its return value directly depends on is durably written before the operation returns. Otherwise, a scenario in which the operation returns the outcome to the user, the system restarts, and the state no longer reflects the user's operation is possible.

In order to deal with this issue, the most straight-forward approach is what we call the *link-and-persist* operation. Essentially, when performing an operation that changes the state of a node, a link is atomically updated normally, but contains a mark to signal the there is no guarantee its state is persisted. The updating operation then persists the newly modified link, and once the link is guaranteed to be persisted it atomically removes the mark. If another operation

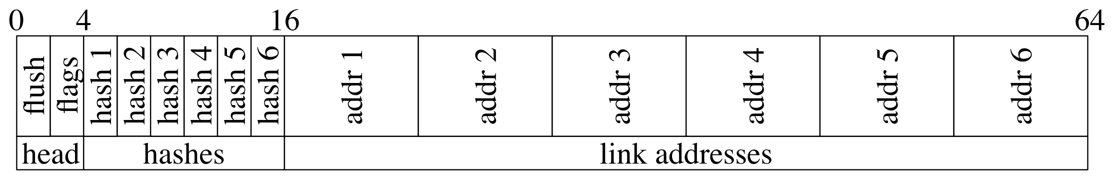


Figure 6.1 – A bucket in the link cache.

whose result depends on the marked link occurs before the updating thread can persist it and remove the mark, the second operation can try to do these steps itself. This method involves no blocking, and is thus suitable for all concurrent algorithms classes, including lock-free and wait-free algorithms [69].

However, as previously discussed in Section 2.5, batching multiple cache line write-backs is significantly more efficient than persisting them one at a time. Therefore, we propose the following scheme: do not wait for links to be immediately persisted when doing an update, but place them in a fast, volatile memory cache, and write back all the links in this fast cache when an operation that directly depends on one of them occurs. We call this auxiliary structure a *link cache*. Of course, this means that clients which have inserted links into the link cache will only be notified of the successful completion of their operations once the link cache is flushed to NVRAM. The changes of a link and the insertion of a corresponding entry in the link cache must occur atomically (can be achieved in a non-blocking manner by using hardware transactional memory, or by marking the pointers to be inserted in the link cache while the operation is ongoing). If a restart happens, the modified link currently in the link cache might be lost. However, this is not problematic: the fact that these link addresses were in the cache at the moment of restart means that no operation that directly depends on them completed, and thus its outcome may or may not be visible. We thus maintain the durable linearizability property. In addition, an atomic update of an ongoing operation not being durably recorded does not leave the data structure in an incorrect state after a restart. Where ordering of durable updates is necessary, we enforce it in the data structure algorithm (see Section 6.4.2). The link cache is practical as long as inserting an entry in the cache is faster than waiting for a cache line to be written back to NVRAM.

6.2.2 Link cache implementation

We now go into details regarding the implementation of the link cache. It is important to note that the link cache does not have to reside in NVRAM, as we do not rely on its content after a restart. Our main aims were small memory footprint for the cache, non-blocking operation, and fast insertions.

With these requirements in mind, we chose to simplify the design, by making insertions to the cache best effort. The cache is only useful if it can improve the time updates spend waiting. Therefore, if an update attempts to insert an entry in the cache, but does not succeed on the first try, it gives up and persists the modified link itself instead of waiting. Thus, the link cache has constant worst case performance.

Our hash table has a configurable (but fixed throughout the execution) number of buckets. Each bucket spans exactly one cache line, and can store up to 6 links. Links concerning a particular key map to one and only one bucket. Figure 6.1 details the contents of a bucket. The first two bytes are used to signal whether the bucket is currently being flushed. The next two bytes are used to store the current state of each of the entries in the bucket. An entry can be free, pending or busy. We next store the 6 keys associated with the links in the bucket. In order to be able to fit 6 entries in a single cache line, instead of storing the entire key, we only store a 2-byte hash for each of the keys. While this might result in false collisions, they are extremely unlikely. With 32 buckets, we essentially have a hash space with 2M elements. Even if false collisions do occur, this is not problematic: we would simply be triggering a flush of the links in the cache when this might not have been strictly necessary. The hashes therefore require 12 bytes in each bucket. The remaining 48 bytes in the cache line are used to store the addresses of the 6 links.

The interface of the link cache has three operations, which we discuss in the following.

Try link and add. If there is space in the link cache, this operation atomically modifies the link in the data structure and inserts an entry in the link cache. The operation first tries to reserve an entry in the link cache. To this effect, it tries to atomically change the state of an entry from free to pending. If no free entry exists, or the attempt to reserve an entry is not successful, the caller is notified that the operation did not succeed and that it should persist the link itself. Once an entry is reserved, we set the corresponding key and link address in the link cache. Next, we try to update the link in the data structure. We insert the new link, but use a bit to mark the fact that for now, this link has been neither persisted, nor has its addition to the link cache been marked as completed. If the link update fails, we set the state of the link cache entry to free and return failure to the caller. We next set the state of the entry in the link cache to busy (to mark the fact that we have added the key and link address, and that the link address in fact contains the value that we want to persist). Finally, we remove the mark from the link in the data structure.

The fact that this operation is best effort, and the fact that we do several atomic updates (link marking, transitioning between multiple states) just in order to be able to handle concurrent readers make this operation an ideal candidate for the use of hardware transactional memory (HTM). In fact, we first try to execute a fast-path HTM-based operation before reverting to the code presented above. In the HTM path, we do not need to insert a marked link into the data structure, and we can avoid going through the pending state in the link cache.

Flush. This operation writes all the finalized entries in a bucket to NVRAM. The operation first atomically sets a flag to signal that it is in the process of flushing a bucket. The flush

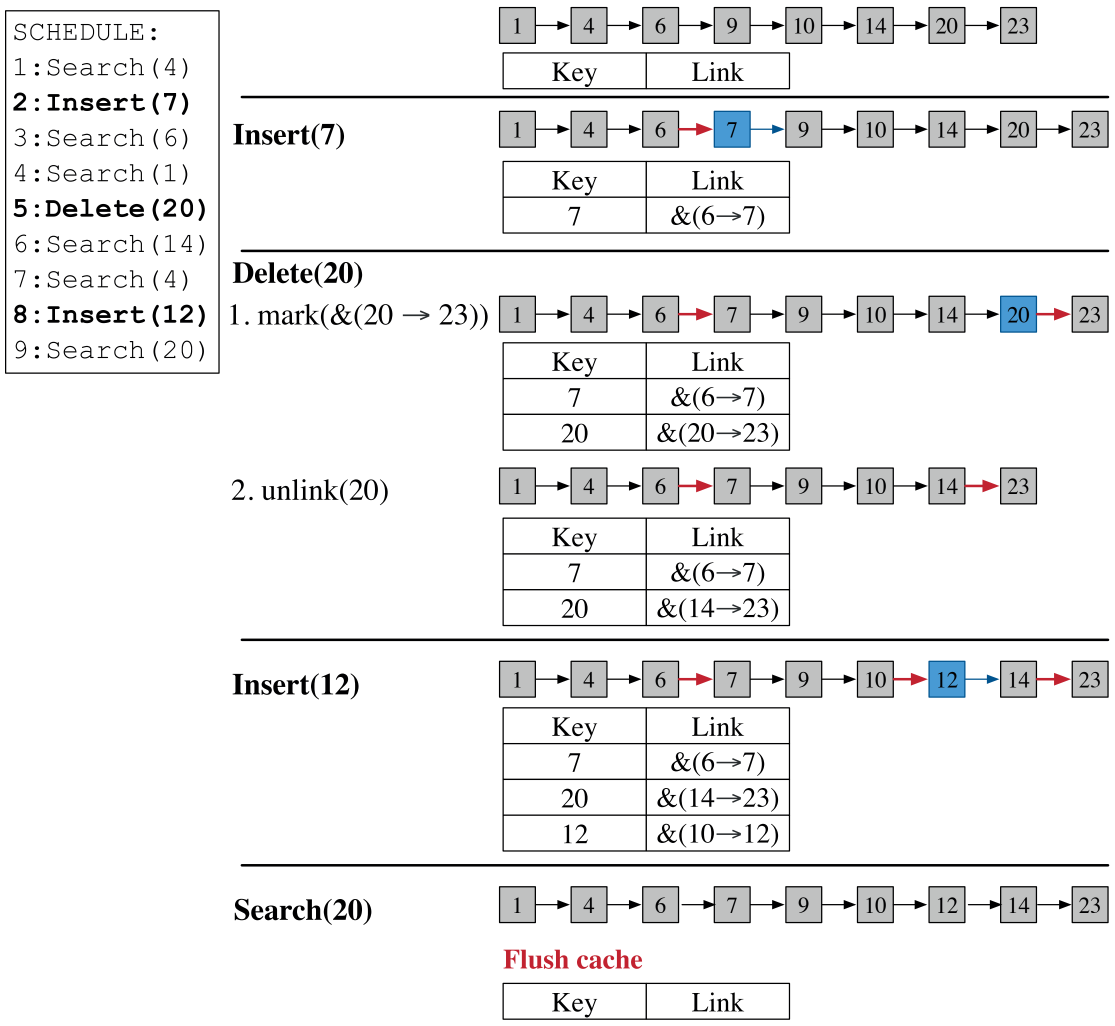


Figure 6.2 – Example of how the link cache is constructed.

operation then issues write-backs for the link addresses in the busy entries in the bucket one by one (without waiting for the write-backs to complete) and sets the state of these entries to free. Next, the operation checks if any of the entries we have not written back have become busy (completed) in the meanwhile, and if yes, issues write-backs for them as well. This is repeated until no new busy entries appear. The thread then waits for the write-backs to complete by issuing a fence, resets the flushing flag, and returns.

Scan. The scan operation is given a key and searches if any link corresponding to the key is in the cache link. If such an entry is found in busy state (i.e., the insertion of the link was finalized), a flush is triggered. If an entry is found but is in pending state, the operation checks whether the new pointer has been inserted into the data structure. If this is the case, the current operation's linearization point should be after that of the operation currently inserting into the cache link, and therefore the current operation triggers a write-back of the new value of the link. Otherwise, the current operation's linearization point is before that of the update, and no further action needs to be taken. In order to guarantee durable linearizability, every data structure operation needs to call the scan method for its key, as well as for its predecessor in the structure in case of updates. However, this is as fast as reading two cache lines.

6.2.3 Illustration of the link cache's effectiveness

We illustrate the effectiveness of the approach through an example. We consider a lock-free linked list that uses the algorithm proposed by Tim Harris [64]. In this algorithm, in the case of inserts, once a node is properly allocated and initialized, we simply have to set the *next* pointer of its predecessor to point to it. In the case of a delete, we must first atomically mark the *next* pointer of the node to be deleted to signal logical deletion, after which the *next* pointer of its predecessor is set such that it bypasses the node to be deleted.

The schedule of operations in our example, as well as the way the link cache is constructed are presented in Figure 6.2. We assume an initially empty link cache, and we only depict the effects of operations that change the state of the data structure or the link cache. Normally, updates would have to wait for one link to be persisted in the case of the insert operations, and two links in the case of the delete operation (one for marking and one for deletion). However, in this example, by using the link cache, we have replaced writing back 4 cache lines one at a time by a single batch of 3 cache line write-backs.

6.3 Memory Management with NV-Epochs

We now address another issue that is unavoidable whenever inserting or removing nodes in a concurrent data structure: memory management.

6.3.1 Overview

Two separate steps need to be performed both when inserting and removing a node: in case of an insertion, memory for the new node first needs to be allocated and initialized, after which the node has to be linked into the data structure; in case of a deletion, the node is first unlinked from the data structure, and later, when we are sure no references to it exist, its memory is freed. If a restart occurs in between these two main steps both in case of an insertion and a removal, a persistent memory leak would occur: we would have allocated data that is not linked anywhere in our data structure.

The typical way of addressing the issue in the context of NVRAM is to use some form of logging: before allocating and linking, we log our intention, as we do before unlinking and freeing memory. Once the operation has (durably) completed, the log entry can be removed. However, this entails an extra write to NVRAM per update. In addition, this write needs to complete before we can proceed with the update, thus producing a non-negligible increase in the latency of updates.

In order to avoid waiting for the durable log to be written at each allocation or deallocation, we propose keeping track of active memory areas instead of keeping track of individual allocations/deallocations. Intuitively, when allocating, threads often reserve larger contiguous memory areas from which they allocate. Therefore, consecutive allocations tend to belong to the same memory area. In addition, memory reclamation schemes keep track of which objects have been unlinked, and periodically free those for which it is guaranteed that no references are held. This reclamation step is only run periodically (either at fixed time intervals, or, more commonly, when a certain number of unlinked objects have been collected) for performance considerations. Therefore, we tend to free multiple nodes at the same time. Out of these, it is usually the case that several of them map to the same memory area. Thus, there is a certain degree of locality in deallocation as well. Hence, if instead of logging every node we unlink from the data structure, we only keep track of the memory areas from which the unlinked nodes come from, we can expect significant savings in term of write-backs to NVRAM: in the common case, the memory area will already be marked as active.

While providing us with important time savings at run time, this method does defer some work for when we need to recover. In particular, we need to go over the allocated memory addresses in the active pages and check if they indeed represent nodes that are linked into the data structure. To be able to do this, we also make the assumption that we can dedicate memory pages to only store data structure nodes. To achieve this, we use an allocator specifically for such nodes.

We first briefly describe the principles of the memory reclamation scheme that we employ, after which we go into more details into how we keep track of the active memory page set, and how we recover in case a restart occurs.

6.3.2 Epoch-based memory reclamation

Epoch-based memory reclamation [54] is based on the following principle: if an object is unlinked, then no references to the object are held after the operations concurrent with the unlink have finished.

One method of using this principle in practice (and which we use in our reclamation scheme) is to provide each thread with a local counter, keeping track of the *epoch* the current thread finds itself in. The epoch of a thread is incremented when the thread starts an operation, and when it completes it. Thus, if the current epoch number of a thread is odd, the thread is currently active. We collect multiple objects, and free them when the vector formed by the current epochs of the threads is larger than the one when any of the objects were unlinked (only the epochs of threads that were active at the moment of unlinking need to be larger). We refer to the set of unlinked nodes which we attempt to free together as a *generation*.

6.3.3 Interface with NVRAM allocators

Memory allocators usually reserve a large contiguous memory address space, which they then recursively split into smaller chunks. The chunk from which an object is allocated then depends on the object's size. These chunks of contiguous memory are generally referred to as allocator *pages*. Since smaller pages from which objects such as data structure nodes are directly allocated are part of larger pages, we can configure the granularity of the pages which we keep track of. High-performance concurrent allocators usually partition the memory space for allocations among threads, such that there is minimal communication necessary between them: pages are assigned to individual threads.

Existing persistent allocators provide the capability of atomically allocating and linking (or unlinking and deallocating) objects, which, as discussed, is generally achieved through some form of logging. We do not require this capability: we only require that the persistent allocator is able to correctly maintain its durable metadata when allocating or deallocating. Moreover, in our case, the last write-back (which marks the memory as allocated or free in a thread's local allocator metadata, and is usually the only write-back the allocator issues) does not have to be completed before proceeding: in the case of an allocation, the data structure algorithm will have to wait for the write-backs to complete after the memory is initialized, while in the case of deallocations the memory reclamation scheme waits for all the deallocations it issues at once to be completed. Thus, in most cases, when the allocator only does one store to thread-local data, we do not have to issue a sync operation for the allocator metadata. Based on its metadata and our active page tables, the allocator can recover its state in case of a restart. An existing persistent allocator, such as for instance, `nvm_malloc` [143], can be used for our purposes, with the small changes we mentioned. We also require the allocator to provide a method that returns the next node address to be allocated. As allocators generally assign larger chunks of memory to individual threads, and threads do not "steal" memory from one another, adding this method is trivial.

6.3.4 Maintaining the set of active NVRAM memory areas

In our approach, each thread keeps a set of active memory pages. For each memory page, we also store some metadata determining when the page can be considered as no longer active and can thus be removed from the set. This metadata consists of (i) the largest epoch at which this thread has unlinked memory belonging to the page from the data structure, and (ii) the largest epoch at which this thread has allocated memory belonging to this page. The addresses of the memory pages need to be stored persistently (meaning that when we insert a new page, we have to wait for the write-back of the address of the page to complete before continuing), while the metadata is only needed for removing table entries, and is not needed in case of a restart.

We attempt to trim a thread's active page table when it exceeds a certain size. For this purpose, the metadata associated with each page is used as follows. A page from which unlinks have happened is active until the epoch-based memory reclamation scheme is guaranteed to have freed all unlinked nodes. This can be verified by having the reclamation scheme keep track of the epoch vector of the most recent generation of objects that were collected. A page from which allocations have happened is active until the insert operation has finished, i.e., while the current epoch of the thread is equal to the last epoch at which a node allocation from this page took place. When using a link cache, we also have to ensure that it contains no entries pertaining to the page under consideration. For this reason, the operation that attempts to trim the active page table issues a link cache flush as well. If all the unlinked nodes have been freed, and all the allocated nodes have been linked, the page can be removed from the table.

We use a separate persistent allocator for the active memory page table. Allocations for the table happen very infrequently (we preallocate a number of entries for each thread at start-up, and allocate multiple entries at once when more space is needed; in addition, tables usually do not grow beyond a certain size, and thus no allocations are needed from a certain point). We require that this second allocator provide the interface previous work on NVRAM memory allocators does [29, 83, 125, 143, 159]. In this instance, we used the allocator provided with `nvml` [83].

6.3.5 Recovery after transient failures

On recovery, we must make sure that there are no nodes that are not linked in the data structure but are allocated.

There are two ways of verifying this. The efficiency of each of these methods depends on the size of the data structure, the complexity of the search method, and the size of the memory space that needs to be verified. In both cases, we assume a well-formed data structure. That is, the recovery procedure should first ensure that the data structure is brought to a consistent state before attempting to remove memory leaks. This step is not necessary for any of the data structures we developed.

The first approach is to go over all the node addresses in the active memory pages at the moment of the crash, and, if they are allocated, perform a search in the data structure for the key the allocated address contains. If the search (i) returns a result and (ii) the address of the returned node is the same as the address we were considering, we leave the node as allocated. Otherwise, we free the node. Condition (ii) is necessary because we might have an allocated but uninitialized node. Therefore, a node with the key that we retrieve from that uninitialized memory might indeed exist in the data structure, but it might not be pointing to this uninitialized memory.

The second approach is to traverse the structure only once, and for each node check if its address belongs to the set of active pages. If this is the case, store the address of the node in a volatile memory buffer. Next, go over all the allocated node addresses in the active memory pages, and check if they are in the volatile memory buffer. If they are not, it means they are not linked in the data structure and can be deallocated.

Both of these approaches can be parallelized in order to decrease the time spent on recovery.

We note that in our implementation, it cannot be the case that a node is linked into the data structure, but not marked as allocated. This is because before linking a node in the data structure, we issue a store fence that ensures that the contents of the node, as well as the allocator metadata (for which we issue write-backs, but do not wait for last one to complete when calling the allocation method) are durably written.

6.4 Log-Free Durable Data Structures

In this section, we present our general approach to designing concurrent and durable data structures. We first argue that such data structures should be lock-free, and illustrate the steps we take in order to obtain correct lock-free data structure implementations for NVRAM. We focus on implementations of linked lists, skip lists, hash tables, and search trees, which are commonly used in practice [47, 59, 114, 122, 123, 126]. Nevertheless, our techniques apply to other data structures as well. Besides the optimizations presented in this section, the data structures use the previously introduced techniques, namely the link cache and NV-epochs.

6.4.1 The Case for Lock-free Algorithms

As discussed, previous work has taken the approach of using transactions and logging (normally either write-ahead logging or copy-on-write) to ensure the correctness of durable data structures. The log must be durably written before the updates it refers to, thus introducing frequent waiting of hundreds of cycles. Thus, when moving from volatile to durable data structures, one can expect a significant drop in performance. This is particularly problematic for small and medium-sized concurrent data structures, which would normally read and write most of the data from the write-back caches.

In order to mitigate the problem at the level of the data structure algorithms, we leverage lock-free algorithms. A concurrent algorithm is said to be *lock-free* if it can guarantee that at any point in time, at least one thread that is trying to take steps is able to make progress. As previous work has shown [35], lock-free algorithms tend to scale and perform extremely well in practice.

We argue that lock-free algorithms are even more appealing in a system using NVRAM. A corollary of the theoretical definition of lock-freedom is that in a system with n threads, if $n-1$ threads stop executing at any point in their operation, the one remaining thread must be able to continue making progress. Otherwise, a thread stopping execution at a problematic stage could prevent the progress of all other threads, even when they are trying to make progress, thus breaking the lock-free property. In other words, no thread can at any point in its execution leave the data structure in a state that is inconsistent and from which other threads cannot continue their operation themselves. In particular, in the state-of-the-art lock-free algorithms, there is an atomic step (usually performed through an atomic compare-and-swap) which makes an update visible: enough information is introduced in the data structure through this step for any other thread to be able to complete the update. Once this atomic update is durably written, upon a restart, the update can be completed by some thread, and the data structure is thus in a consistent state. If the update is not persisted, it is as if the update had not occurred at all, and thus the data structure is of course in a consistent state. Thus, the moment when this essential update is durably written is the linearization point [75] of updates.

In the case of NVRAM, we can thus guarantee that as long as the stores of the threads are persisted in the order in which they are issued (we show how this can be relaxed), regardless of where a crash occurs, upon a restart the data structure is in a consistent state that allows the execution to resume. Therefore, we remove the need for logging for the data structure itself.

6.4.2 Durable Data Structure Implementations

We have implemented durable data structures for linked lists, hash tables, skip lists, and BSTs. These structures model the set abstraction, and have methods to insert, remove, and search of elements identified through a unique key. We consider one implementation per data structure type, starting from the concurrent algorithm that has been shown to provide the best performance and scalability [35]. Our linked list is based on Harris' algorithm [64], the hash table uses one Harris linked list per bucket, the skip list uses Herlihy and Shavit's lock-free algorithm [74], while the BST uses the algorithm proposed by Natarajan and Mittal [129]. Other algorithms and data structures can be similarly modified.

We illustrate our approach on the skip list, as most of the issues that appear in the context of other data structures appear in the case of the skip list as well.

Illustration: Log-free Durable Skip list. We start from a version of Herlihy and Shavit's algorithm that uses the optimizations proposed by Asynchronized Concurrency [35]. In a volatile memory environment, searches in this algorithm are wait-free and perform no stores. Insert operations link a new node in the data structure starting from the bottom level, and progressively link the node in its higher level lists. Remove operations first mark a node's next pointers to signal logical deletion (starting from the node's top level next pointer, and going down to the bottom level), after which the node is physically unlinked from the skip list one level at a time (again, starting from its top level). We note that nodes in a skip-list may span multiple cache lines.

One particularity of this algorithm is the very fact that it does not guarantee that the skip list is well formed. Due to concurrency, it might be the case that a node is present in some higher-level list, but not in (some of) the levels below. It is also possible that a node that was unlinked from all the skip-list levels might reappear (in a state that is marked for deletion) in one of the higher skip list levels. However, since the membership of a node in a skip list is determined by its presence or absence in the bottom-level list, this does not affect correctness. The higher-level lists are used simply for performance reasons.

In a sense, this makes this algorithm particularly appealing for NVRAM: the algorithm itself can work with a data structure that is not completely well-formed, so we can take advantage of this and not ensure that every update to the higher levels of the skip list is persisted, since we know that at recovery we can continue operation even if certain links are missing. Thus, for the higher-level links, we do not issue write-backs and wait for them to be persisted one at a time. We only issue these write-backs at the end of the operation, but we do not wait for them to be completed before proceeding.

In another sense, this causes difficulties at run-time: the fact that at the end of a delete operation we cannot guarantee the fact that a node is no longer reachable in the data structure makes memory reclamation problematic. To address this, we identify the scenarios under which a node may still be traversed even after its delete operation has completed. We modify the original algorithm such that once all the updates that were concurrent with a delete operation finish, the deleted node is guaranteed to no longer reachable in the data structure (even though at the end of the delete operation itself it may still be). This is sufficient for our purposes, as this is the same mechanism our memory reclamation scheme uses. We believe these optimizations may be useful when using the algorithm in a volatile memory setting as well.

Whenever we insert or remove a node from the bottom-level list, in order to ensure durable linearizability, we can either use the previously described *link-and-persist* technique, or our link cache. Before performing an update, in order to provide durable linearizability, we must ensure previous updates concerning the node's key, as well as nodes directly related to the update have been durably written. We discuss these aspects in the following.

Correctness. Our data structures are linearizable, since we start from linearizable algorithms and add only flushes or link cache operations, which do not impact linearization points. We also ensure two additional properties [56]. First, each update operation ensures that its changes are durable before returning (when using the link cache, this happens after a cache flush). Second, each operation O ensures that all operations O depends on (that involve some of the same nodes and were already linearized) are also durably linearized before O makes changes. Together, these two properties ensure durable linearizability, because they ensure that after a restart, the data structure reflects a consistent cut [89] of the history including all operations that completed before the crash and potentially some operations that were ongoing when the crash occurred.

The first property is easily ensured by durably writing any new edges or nodes introduced by an operation. Making sure an edge e is durably written just means checking if e is marked (or in the link cache), and issuing write-backs only if e is not yet durable. The second property is achieved because operations ensure that (1) before an edge is modified, the edge is durably written and (2) incoming and outgoing edges (*adjacent edges*) of nodes involved in the operation are durable before proceeding.

We detail point (2) on a linked list (similar considerations apply for the other linked data structures). For a successful search, we make sure adjacent edges to the returned node are durably written before returning. For a failed search, we make sure the node is durably unreachable before returning (e.g., in the case where a node is marked but not yet durably unlinked). For the parse phase of a modify operation (insert or delete), we take the same steps as for a search. For an insert, we also ensure that adjacent edges to the predecessor are durable before linking the new node. For a delete, we ensure that adjacent edges to the target node T and to T's predecessor are durable before unlinking the target node. In all cases, if an edge e has changed between the time e is read and the time we try to durably write e, then the operation that changed e made sure e was durable.

6.5 Evaluation

We now study the impact our proposed techniques have on practical data structures. We look at the overall performance improvements, as well as at the behavior of components such as the link cache, and the active page tables.

6.5.1 Experimental setup

We run our experiments on an Intel Xeon machine that has four E7-4830 v3 12-core sockets. The cores operate at 2.1-2.7 GHz, while cache sizes are 32KB (L1), 256KB (L2), and 30MB (LLC, per die). We work with key-value pairs, both of which are 64 bytes in size. Larger values can be accommodated by using a pointer instead of the 64-byte value. Experimentally obtained values are the median of 5 repetitions.

As neither NVRAM with latencies comparable to DRAM, nor processors providing the `clwb` instruction are available yet, we write data to DRAM, simulate the `clwb` instruction, and inject software created delays, similar to previous work [24, 29, 105, 158, 159]. Intel reports issuing several flushes with `clflushopt` can be up to an order of magnitude faster than flushing them one at a time using `clflush` [81]. We assume similar performance characteristics for the `clwb` instruction. Moreover, we assume an NVRAM write latency of 125ns, which is an average of the projected values.

6.5.2 Data structure performance

We first look at the run time behavior of our data structures. We focus on updates, as it is these operations that must be durably recorded in NVRAM. We compare our implementations with alternatives that use lock-based critical sections (and thus use logging). We find that

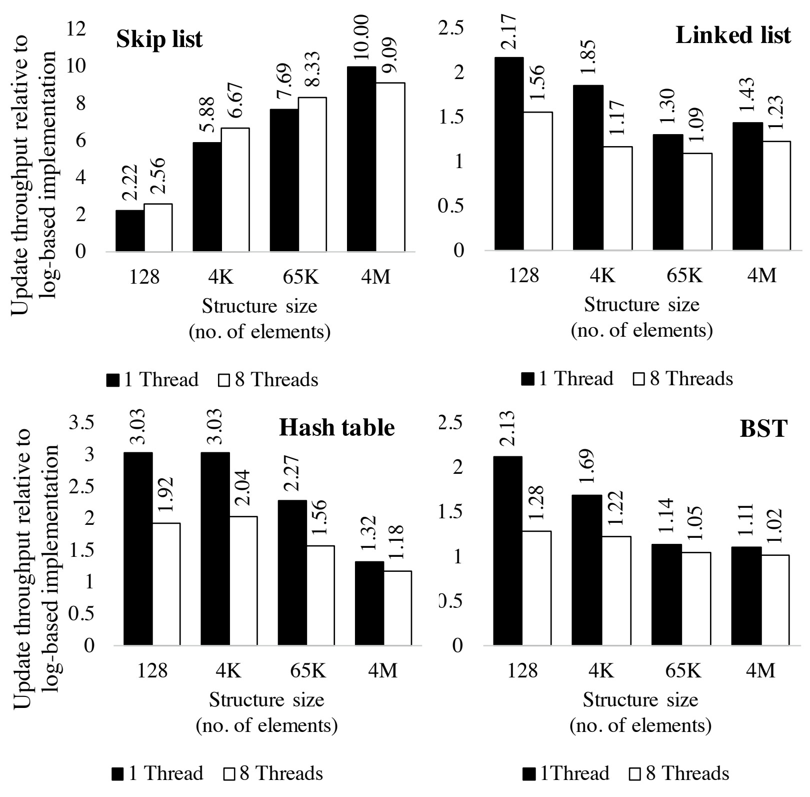


Figure 6.3 – Update throughput improvements compared to redo log based implementations.

in the context of such data structures, an approach that uses redo logging provides good performance in addition to ensuring durable linearizability. We use the algorithms that we find perform best for each data structure: the lazy linked list [66], Herlihy's lock-based skip list [70], bst-tk [35], and a hash table with a lazy linked list per bucket.

In Figure 6.3, we show the increase in the number of updates per second obtained by using our structures relative to log-based implementations. We use a workload where 50% of the operations are inserts of random keys, while 50% are removes of random keys, and show results for 1 and 8 concurrent updating threads. We show relative improvements, as the precise latencies are dependent on the assumptions made about `clwb` instruction performance, as well as NVRAM store latencies.

Our method yields important benefits regardless of the data structure type. In particular, for the skip list, where in a log-based implementation a logarithmic number of locks are held while a logarithmic number of updates must be logged, our approach results in an order of magnitude increase in performance. We note that for small and medium sized data structures, we obtain significant improvements by applying our techniques. For large structures however, our improvements become less impressive. There are two main reasons for this. The first is that as the structure size increases, the latency of an update becomes dominated by the time needed to reach the point in the data structure where the modification needs to be

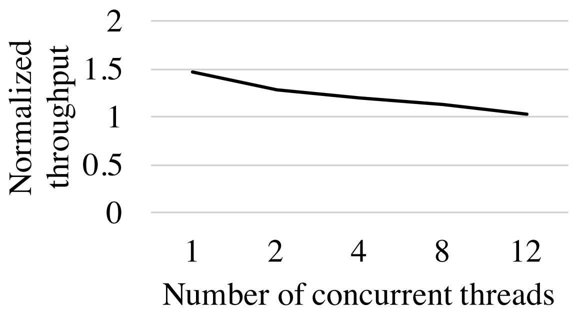


Figure 6.4 – Throughput when using the link cache normalized to throughput when not using link cache, for a 1024-element hash table, with all operations being updates.

made, both due to the need to traverse more pointers, and because when the structure does not fit in the caches anymore, reads become more expensive. In the case of the linked list in this experiment, it is in fact the only factor that is responsible for the decrease in latency improvement. The second reason has to do with a decrease in the efficiency of our active page tables for deallocations as the structures become large. We discuss why this is, as well as ways of alleviating the issue in Section 6.5.4. In addition, as the number of concurrent updating threads increases, the link cache becomes somewhat less efficient, as we discuss in the following. Thus, for high degrees of concurrency, we can turn the link cache off.

To summarize, it is important to note that while the precise magnitude of the improvements of our approach may depend on the characteristics of the NVRAM technology being used, this experiment has shown that our approach is beneficial for all the situations we have considered.

6.5.3 Link cache efficiency

The link cache technique is an optimization that can be turned on or off in an algorithm. In this experiment, we identify the scenarios under which it is beneficial.

The main potential barrier to link cache performance is scalability, given that it is a structure occupying a small number of cache lines, which is repeatedly accessed by all threads.

We evaluate an algorithm with and without the link cache, and measure its scalability. In this experiment, we use a hash table as a base data structure, due to the small latencies of updates, which allows us to stress the link cache. Each thread continuously issues insert and remove operations (thus, no read operations). The link cache occupies 32 cache lines.

Figure 6.4 presents the throughput of the hash table version using the link cache, normalized to the throughput of the hash table version not using the link cache. While with one thread, the link cache improves throughput by ~50%, as the contention increases, the link cache becomes less effective, until, with 12 concurrent threads, we observe no benefit.

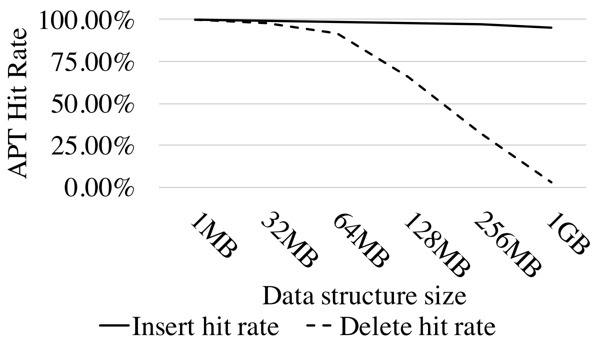


Figure 6.5 – Active page table hit rates.

Thus, this experiments highlights two main points (1) the link cache can be extremely beneficial up to moderate degrees of concurrency; however, (2) as its cache lines become contested, its benefits become less apparent.

We note that the link cache is the only component we add which requires inter-thread coordination, and is thus the only potential impediment to scalability.

6.5.4 Active page table efficiency

We now look at the efficiency of our active page table mechanism. The active page table is only efficient if it saves *sync* operations: that is, if an important fraction of updates do not have to write active page table entries.

In this experiment, we consider 4KB memory pages, and we try to trim an active page table when it exceeds 16 elements.

We run a data structure algorithm, and measure the fraction of allocations and deallocations that do not need to add an entry to the active page tables (that is, the fraction of hits in the active page table). Results are shown in Figure 6.5. In this experiment, we have used a skip list. Results are similar for other data structures, as the important factor is the data structure size, rather than its type.

We note that the hit rate is close to 100% for allocations, regardless of data structure size. In case of deallocations, the hit rate starts decreasing after the structure exceeds 64 MB (that is for data structures of more than 1M nodes; or more than 0.5M in the case of skip lists). This is because as the amount of used memory increases, there is less locality in memory reclamation steps. However, fast memory allocation and deallocation is particularly important for small data structures that fit in the write-back caches, and which have small access latencies. In such situations, our approach is effective for both types of operations.

The parameters of our system can be adjusted to deal with deallocations in larger data structures as well. The granularity at which we keep track of active memory areas is adjustable. By using larger size pages, we can improve the hit rate. This coarser granularity would however result in a somewhat larger recovery time after a restart. Additionally, we can also maintain larger active page tables (in terms of the number of entries). However, this would make active page table operations slightly slower at run time. Moreover, the number of nodes that our memory management scheme stores in a generation (and thus frees as the same time) can be increased.

To summarize, our memory management scheme is useful regardless of the size of the data structure, but is particularly efficient for small and medium-sized data structures.

6.5.5 Recovery

We now measure the time it takes to recover a data structure. We stop execution of the concurrent algorithm at a certain point, and then traverse its active pages and check for memory leaks. Prior to recovering, we ensure the structure's data is not in the write-back caches. We show recovery times for the various data structures as a function of their size in Figure 6.6.

For hash tables, BSTs, and skip-lists, which have fast search methods, recovery is extremely efficient: even in structures with 4M elements, we can recover in less than 5ms. Recovery time for such structures is two orders of magnitude lower than doing a full mark-and-sweep pass in this environment [16]. In the case of the link list, which has a linear search method, in order to avoid repeated passes over the entire structure, we employ a strategy similar to mark-and-sweep. Recovery time in this case is somewhat slower: a linked list with 64K elements can be recovered in 16ms. For all structures, recovery time increases with data structure size. Small structures tend to have a smaller number of active pages at any point in time. In addition, the search operations must traverse more pointers for larger structures, and data is less frequently present in the higher-level caches. We believe the recovery times we observed in this section are acceptable in case of a reboot.

6.5.6 NV-Memcached

We now show how our techniques can be applied in a larger context by developing an object caching system for durable memory: *NV-Memcached*. The main idea behind *NV-Memcached* is to make Memcached persistent by replacing its core data structures—the hash table and the slab allocator—with persistent versions. Straightforward as this idea may seem, it does entail interesting technical challenges.

First, Memcached uses a lock-protected sequential hash table; thus replacing it with our persistent non-blocking hash table would negate the latter's lock-freedom. We solve this challenge by basing *NV-Memcached* on *memcached-clht* [107], a version of Memcached that avoids protecting the hash table with locks by employing a concurrent hash table—CLHT [35], and replacing CLHT with our persistent hash table.

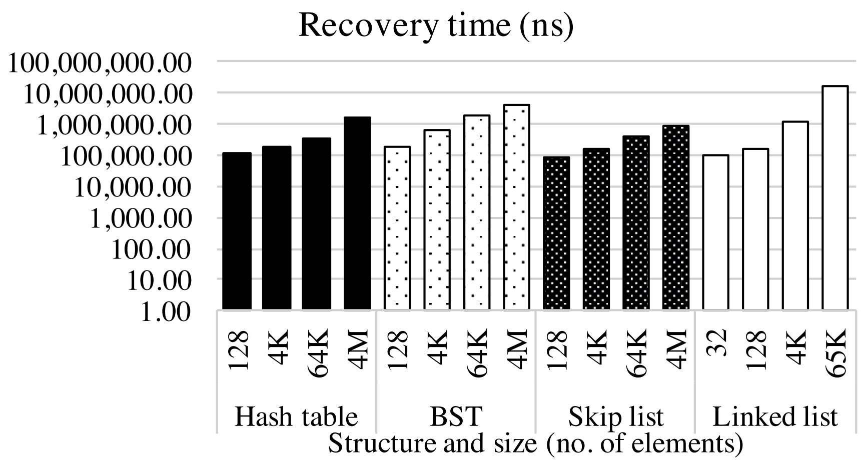


Figure 6.6 – Data structure recovery times.

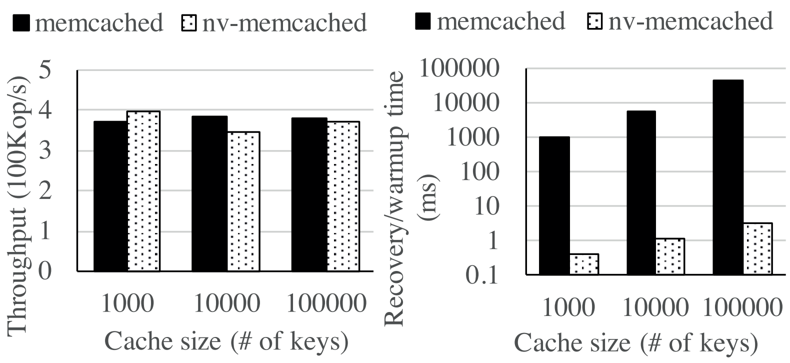


Figure 6.7 – Performance and warm-up time comparison of Memcached and NV-Memcached.

The second challenge is related to the recovery of items. With a naive implementation of a persistent slab allocator, it is possible for memory leaks to occur after a restart. An item can be allocated, but not yet linked in the hash table, or an item can be unlinked from the hash table but not yet marked as free in the allocator. We address this issue with a similar approach to our active page technique (Section 6.3): we keep track of active slabs. During recovery, we iterate over each thread's active slab table and free any memory which is marked as allocated but not yet or no longer reachable from the hash table.

We compare the performance of *NV-Memcached* to that of Memcached using `memtier-benchmark` [140]. The benchmark runs for a predetermined amount of time, issuing a mix of `get` and `set` operations using keys drawn uniformly at random from a given key range. The key range and the ratio of `get` to `set` operations are configurable parameters. Before each experiment, we warm up the cache by inserting items covering half of the key range. Both the server and the client are run with the default number of threads (4). The results are averaged over 5 runs for each configuration.

The first experiment compares the throughput of *NV-Memcached* and Memcached for three different key range sizes, under a 10:1 `set` to `get` ratio. As we can see in Figure 6.7, there is no notable performance drop between *NV-Memcached* and Memcached. While this is partly due to the fact that *NV-Memcached* uses a faster, more scalable concurrent hash table, the comparable performance also shows that our techniques remain practical when applied to real-world applications.

The second experiment compares, for three different key range sizes, the warm-up time of Memcached (the time it takes to populate the cache with half of the key range) to the recovery time of *NV-Memcached* (the time it takes to recover after a restart). Figure 6.7 shows that populating a (volatile) Memcached instance with items can take up to four orders of magnitude more time than recovering a *NV-Memcached* instance of the same size. This justifies the practicality of a non-volatile memory caching service—recovering such a service after a machine restart takes just a fraction of the time necessary for its volatile counterpart to get re-populated (and thus be useful again).

6.6 Concluding Remarks

In this chapter, we introduced an approach yielding fast and durable concurrent data structures. By using lock-free algorithms, we avoid excessive writes to NVRAM. By using a link cache, we mostly eliminate waiting for the completion of write-backs, regardless of the data structure. By keeping track of memory allocations and deallocations at a coarser grained granularity, we avoid logging associated with these operations at run time, at the cost of modest increases in recovery time.

Yet, we do not claim our approach is a silver bullet. While it is extremely effective for small and medium-sized concurrent data structures, which are common in practical situations, our method is somewhat less effective for large data structures. Moreover, our approach is applicable to concurrent objects for which efficient lock-free algorithms exist. While this is the case for most common data structures, our approach is not as generic as a complete transactional system.

In the evaluation of this work, we have simulated an NVRAM-based platform by using DRAM, as well as the existence of cache line write-back instructions, which will be available on upcoming Intel architectures. Where necessary, we have injected extra latency. We plan to deploy and test our approach on a platform with NVRAM and the new write-back instructions enabled once these become widely available.

Scalable Concurrency Control Part IV

7 Locking timestamps versus locking versions

In this chapter, we shift our attention to more complex programming abstractions, i.e., transactions, and present *multiversion timestamp locking (MVTL)*, a new genre of multiversion concurrency control algorithms for serializable transactions. The key idea behind MVTL is simple and novel: lock individual time points instead of locking objects or versions. After showing what a generic MVTL algorithm looks like, we demonstrate MVTL's expressiveness: we present several simple MVTL algorithms that address limitations of current multiversion schemes, by committing transactions that previous schemes would abort, by avoiding the problem of serial aborts and ghost aborts, and by offering a way to prioritize transactions that should not be aborted.

7.1 Introduction

The serializable transaction abstraction is a powerful paradigm available in many computing systems, such as transactional memory, database systems, and key-value stores. To ensure serializability, transactions require a scheme for concurrency control to handle any negative consequences of transaction interleaving.

The literature on concurrency control is rich [15, 164], and a particularly appealing class of algorithms is called *multiversion concurrency control* [14]. Briefly, these algorithms keep a *history* of each object with timestamped versions. This history gives the system a *choice* of which version to use when an object is accessed. This choice permits more transactions to execute concurrently without blocking or aborting. For example, in some multiversion algorithms [15, 33], read-only transactions can execute without ever blocking or aborting, and update transactions can concurrently update the same object. Enabling more concurrency has become particularly important with the proliferation of multi-core and large-scale systems. Multiversion algorithms have wide application: they are used often in database systems both commercial and academic [48, 106, 164, 167], and more recent work has applied them to key-value stores and transactional memory (e.g., [51, 90, 91, 97, 135, 136]). In this work, we do not restrict ourselves to any particular application, but rather study multiversion algorithms in their broadest and most conceptual scope.

There are three main genres of multiversion algorithms: *lock based*, *timestamp ordering*, and *serialization graph based* [15]. Lock-based algorithms (e.g., MV2PL [15]) acquire locks to avoid the ill-effects of concurrency; these algorithms are very simple. Timestamp ordering algorithms (e.g., MVTO [15]) assign a timestamp to each transaction, and then serialize transactions by timestamp; these algorithms permit read-only transactions to execute without ever aborting. Serialization graph algorithms (e.g., MVSGT [164]) detect cycles in the serialization graph to prevent a violation of serializability; these algorithms permit higher levels of concurrency than the alternatives.

Despite their many benefits, all types of multiversion algorithms have limitations. Lock-based algorithms significantly limit the degree of concurrency. Timestamp ordering algorithms are susceptible to aborts, including *serial aborts*—aborts in serial executions—and *ghost aborts*—aborts caused by a conflict with a transaction that already aborted. Serialization graph algorithms are complex and incur significant computation overheads [11, 91, 128].

In this chapter, we introduce a new genre of multiversion algorithms, called *multiversion timestamp locking*, or MVTL. MVTL is based on a simple novel idea: use locks as in lock-based algorithms, but lock individual timestamps of objects, rather than entire objects at a time. A transaction is allowed to commit if it can find at least one timestamp that it managed to lock across all its objects. Intuitively, MVTL excels because it uses locks with very fine granularity: not only individual objects have separate locks, but individual timestamps within objects have their own locks. Locking at fine granularity increases parallelism and decreases blocking and aborting, as the system can explore many serialization points for each transaction.

Conceptually, MVTL keeps a lock state for each object and each timestamp, which amounts to an infinitely large lock state. However, pragmatically we can reduce the lock state significantly using interval compression, so that each object holds just a few lock intervals, and this state can be subsequently discarded when the associated versions are purged.

To precisely define MVTL, we give a generic algorithm (Section 7.4) that has several nondeterministic choices, such as what timestamps each operation tries to lock, and how locks are acquired (wait or give up on blocked locks). We prove that the generic algorithm is correct irrespective of those choices: they do not affect safety. They may however be crucial for performance.

We then propose several specific algorithms that specialize the generic MVTL algorithm by fixing these nondeterministic choices to obtain different benefits (Section 7.5). These algorithms are simple and address some important drawbacks of existing multiversion algorithms, such as serial aborts, ghost aborts, the lack of a priority scheme for transactions, and more. We also show that pessimistic and timestamp ordering algorithms can be seen as special cases of MVTL. Thus, in a precise sense, MVTL unifies these algorithms.

For simplicity of presentation, we focus most of this chapter on a shared-memory version of MVTL. However, we believe MVTL is particularly relevant in distributed message-passing systems, where MVTL can achieve a high degree of communication efficiency. Thus, we also show how to extend MVTL to a distributed setting (Section 7.6).

Next, we discuss some pragmatic considerations around MVTL, such as how to compress the lock state (Section 7.7). We separate out these considerations because they are orthogonal to the concepts underlying the MVTL algorithm. However, they are important to applying MVTL in practice.

To summarize, the contributions of this work are as follows:

- We propose a new genre of multiversion algorithms for transactions, called multiversion timestamp locking (MVTL), which is based on the idea of locking timestamps.
- We give several MVTL algorithms, which address various limitations of current multiversion algorithms.
- We show that MVTL generalizes both multiversion timestamp ordering and pessimistic multi-version algorithms.
- We discuss practical considerations for implementing MVTL, including techniques to significantly reduce the space of lock state.
- We explain how MVTL can be applied to a distributed system setting.

Our main contribution is conceptual in nature. We believe that locking individual timestamps introduces a new way to approach multi-version algorithms. The specific MVTL algorithms we present are simple and just scratch the surface; we think the investigation of additional MVTL algorithms is an exciting direction for future work. Also interesting is to carry out an experimental study to measure the performance of MVTL algorithms in a transactional system, such as software transactional memory, transactional key-value storage systems, transaction object systems, or database systems. While the fundamental MVTL algorithms we present are independent of the type of transactional system, the details of how these algorithms are implemented are system specific and deserve further study.

7.2 Model

We consider a standard model for a multi-threaded concurrent system [74]. The system has processes that communicate via atomic shared memory. The system is asynchronous: there are no bounds on the relative speed of processes. We assume the existence of a discrete global clock with domain $\mathscr{T} = \{0, 1, \ldots\}$, and processes may or may not have access to the global clock. More precisely, processes may have local clocks that match the global clock ("synchronized clocks") or that are within a known bound ϵ of the global clock ("ϵ-synchronized clocks").

We are interested in algorithms that implement a transactional storage system. Such a system maintains a set of objects and allows processes to manipulate the objects using transactions.

Each object has a unique key (identifier) and, by abuse of language, we refer to the object and its key interchangeably. The system supports four operations with their usual semantics: BEGIN(tx) starts a transaction tx, COMMIT(tx) tries to commit tx and returns a success indication, READ(tx, k) reads key k within tx, and WRITE(tx, k, v) writes v to k within tx. Transactions are dynamic: their read and write operations can depend on the results of prior operations issued within the transaction. Algorithm 1 shows an example.

Algorithm 1 Example of a transaction

```
1: BEGIN($tx$)
2: $v \leftarrow$ READ($tx, A$)
3: **if** $v \leq 100$ **then** WRITE($tx, B, 50$)
4: $result \leftarrow$ COMMIT($tx$)
```

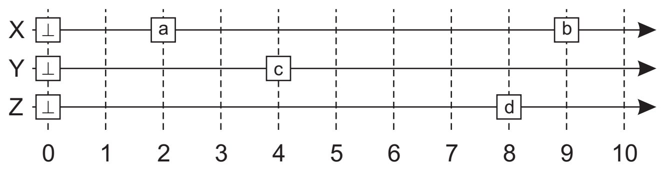
The correctness condition for transactional storage we consider is *multiversion view serializability*, a form of serializability well-suited for multiversion algorithms. Roughly speaking, this condition requires every multiversion schedule of the algorithm to be equivalent to a serial monoversion schedule [15, 164].

Some of our results refer to a *workload*, which is a sequence of operations (indexed by the transaction they belong to) that are input to the system, where each operation is *read*(k), *write*(k, v), or *tryCommit*. The purpose of this definition is to observe how different protocols react under the same operations, to understand which one aborts or delays transactions more.

7.3 Overview

In this section, we first recall multiversion concurrency control algorithms, after which we introduce our new concept of *timestamp locking* and explain how it enables us to address the weaknesses of existing multiversion algorithms.

Multiversion concurrency control and the MVTO+ algorithm. The basic idea of multiversion timestamp ordering is to assign a timestamp to each transaction and then use the timestamp to determine (a) what version the transaction reads from, (b) what version it writes to, and (c) the serialization order of transactions. This idea can lead to several slightly different algorithms. To focus the discussion, here we present a concrete algorithm denoted MVTO+, which is identical to the MVTO algorithm in [15] but with an improvement: it avoids cascading aborts by not reading uncommitted data. For each object, MVTO+ keeps many versions and a timestamp for each version. It is useful to think of each object as an evolving timeline with values. Each transaction tx has a unique timestamp t, which determines the version of objects that tx reads and writes. Specifically, when tx reads an object, it obtains the version of the object with the largest timestamp before t. When tx writes an object, tx does not immediately produce a new version but instead it stores the written value in a temporary area for the transaction. Upon commit, tx takes each written value in this temporary area and produces a new version with timestamp t.



For example, the figure above depicts three objects X, Y, and Z. Each object has an initial version denoted $\perp$. In addition, X has two other versions with data a and b and timestamps 2 and 9 respectively; Y has data c with timestamp 4; and Z has data d with timestamp 8. Now suppose a transaction *tx* is assigned a timestamp 6. If *tx* reads X, it obtains a—the largest version with a timestamp before 6. Similarly, if *tx* reads Y, it obtains c. If *tx* writes e to Z and commits, then Z gets a new version with data e and timestamp 6.

Ultimately, transactions are serialized by the order of their timestamps. A key implication is that, after *tx* reads X and obtains a, another transaction should not produce a version of X with a timestamp between 2 and 6. To prevent this behavior, MVTO+ keeps a *read-timestamp* for each version: this is the largest timestamp with which the version was read by a transaction. In the example, after *tx* reads X and obtains a, the read-timestamp of a becomes 6 (if it was not already larger than 6).

Timestamp locking. We look at MVTO+ slightly differently, using our new notion of *timestamp locking*. This notion allows us to generalize MVTO+ into our new MVTL algorithm. Rather than read-timestamps, we can think that each object has several locks, one for each timestamp. When *tx* reads X, rather than updating the read-timestamp of a to 6, we can think that *tx* obtains a read-lock on each timestamp between 3 and 6. When another transaction wishes to write a version with timestamp, say 5, it must obtain the write-lock on that timestamp. But the read-locks by *tx* prevent this from happening, as required by MVTO+. We can now see the read-timestamp of a as simply a compact representation of the fact that there are read-locks between 3 and 6.

Thinking about timestamp locks has several advantages over read-timestamps. First, with read-timestamps, it is not clear what should happen if *tx* aborts: should the read-timestamp of a be updated to its previous value? But what is the previous value if several other transactions read a concurrently? This is a hard question, and MVTO+ avoids it altogether by taking an unnecessarily conservative approach: when *tx* aborts, it leaves the read-timestamp of a at 6. We show that this choice leads to ghost aborts. In contrast, timestamp locks provide a better alternative: if *tx* aborts, its read-locks are removed but the read-locks of other transactions remain.

Second, with timestamp locks, there is no reason that a transaction should be restricted to obtaining write-locks on just one timestamp, or obtaining read-locks on a range that ends with the transaction's timestamp. Permitting more choices allows the system to avoid serial aborts, as we explain later.

These advantages are captured by our MVTL algorithm, which we now briefly summarize. With MVTL, when a transaction wishes to read an object, it selects a version of the object to read and obtains read-locks on one or more timestamps adjacent to and immediately following that version. To write an object, the transaction obtains write-locks on one or more timestamps anywhere. To commit, the transaction must find a single common timestamp that is read-locked or write-locked across all objects read or written by the transaction, respectively. If such a timestamp exists, the transaction commits; otherwise, it aborts.

The exact timestamps that are locked by reads and writes depend on a *locking policy*. The algorithm remains correct for any locking policy, but a poorly chosen policy causes many aborts because there is no common locked timestamp. We present some simple but interesting algorithms using various locking policies, each with its own advantages.

7.4 Generic MVTL Algorithm

We now present our generic MVTL algorithm in detail. We start with some basic concepts (§7.4.1), explain a simple lock extension we use (§7.4.2), and cover the main algorithm (§7.4.3).

For simplicity of presentation, we focus on the centralized algorithm, and defer discussion of the distributed version to Section 7.6. While more complex, the distributed version is even more practically appealing from a communication efficiency perspective. Some practical considerations for implementing MVTL, including how locks and data can be compacted, are discussed in Section 7.7.

7.4.1 Preamble

In multiversion algorithms, the system keeps many data versions for the same key, in an array $\mathit{Values}[k,t]$ where k is a key and t is a timestamp. It is often useful for a process to be able to pick distinct timestamps from another process; we do this by adding a process id to each timestamp; thus, each timestamp consists of a pair (v, p) ordered lexicographically, where v is a real number. There is a smallest timestamp, which we call 0, and a special value, which we call $\perp$, such that initially $\mathit{Values}[k,0] = \perp$ for every key k.

7.4.2 Freezable locks

The MVTL algorithm deals with *write-once objects*—objects initially set to $\perp$ that may change their state at most once. We define a simple variation of readers-writer locks, which we call freezable locks, which are appropriate for such objects and we use them in MVTL. A freezable lock is similar to a readers-writer lock, except that a lock holder can freeze the lock to indicate that it will never release it. Freezing is useful because it tells other processes that they should not wait to acquire the lock; we use this feature in several specialized MVTL algorithms. If a lock holder does not freeze a lock, it is expected to release it eventually.

We apply freezable locks to write-once objects as follows. A process acquires the lock in write mode if it intends to write the object. The process may ultimately fail to write if the transaction aborts, in which case it releases the lock; but if the transaction commits, the process freezes its lock to ensure other processes will not try to write the object again. Similarly, a process acquires the lock in read mode to read the object and it freezes the lock in case of a commit; if the object was not written (its state is ⊥), this prevents other processes from writing to it, sealing its fate.

7.4.3 Algorithm

Algorithm 2 The generic MVTL algorithm (part 1/2): main code

```
 1: function BEGIN($tx$)
 2:     $tx.readset \leftarrow \emptyset$; $tx.writeset \leftarrow \emptyset$; $tx.committs \leftarrow \perp$
 3: function WRITE($tx, k, v$)                                   ▷ write $v$ to $k$ in transaction $tx$
 4:     WRITE-LOCKS($tx, k$)                                  ▷ write lock some subset of timestamps
 5:     add $(k, v)$ to $tx.writeset$                          ▷ remember key and value we wrote
 6: function READ($tx, k$)                                          ▷ read $k$ in transaction $tx$
 7:     $tr \leftarrow$ READ-LOCKS($tx, k$)      ▷ read lock some interval $[tr+1, \ldots]$ with $Values[k, tr] \neq \perp$
 8:     if $tr = \perp$ then return $\perp$                                          ▷ read failed
 9:     add $(k, tr)$ to $tx.readset$                          ▷ remember key and version we read
10:     return $Values[k, tr]$                                         ▷ return committed value
11: function COMMIT($tx$)                                         ▷ try to commit transaction $tx$
12:     COMMIT-LOCKS($tx$)                                      ▷ locks to acquire at commit time
13:     $T \leftarrow \{t : \forall k \in tx.readset.keys, tx$ has a lock on $(k, t)$ and   ▷ try to find a locked timestamp for $tx$
            $\forall k \in tx.writeset.keys, tx$ has a write-lock on $(k, t)\}$
14:     if $T = \emptyset$ then mark $tx$ as aborted
15:     else
16:         $tx.committs \leftarrow$ COMMIT-TS($T$)                          ▷ pick some timestamp in $T$
17:         for $(k, v) \in tx.writeset$ do
18:             freeze write-lock for $tx$ on $(k, tx.committs)$                          ▷ freeze locks
19:             $Values[k, tx.committs] \leftarrow v$                             ▷ expose committed value
20:         mark $tx$ as committed
21:     if COMMIT-GC($tx$) then GC($tx$)                                          ▷ invoke gc or not
22: function GC($tx$)                                  ▷ garbage collect locks of $tx$ after it ended
23:     if $tx$ committed then
24:         for $(k, tr) \in tx.readset$ do
25:             freeze read-locks for $tx$ on $[tr+1, tx.committs]$
26:     release all unfrozen read- and write-locks for $tx$
```

Algorithm 2 shows the main code of the generic MVTL algorithm. For clarity, we assume that the code in lines 17–19 is executed atomically, but we later remove this assumption (Section 7.7). To write a value into key k, a transaction obtains zero or more write-locks on timestamps for that key (function WRITE-LOCKS in line 4). Intuitively, a write-lock on a timestamp t for key k allows the transaction to commit with timestamp t as far as accesses to k are concerned. After getting the locks, the transaction remembers the key and value; the write is not visible to other transactions until the transaction commits.

To read a key, a transaction gets zero or more read-locks on timestamps for that key (function READ-LOCKS in line 7), with the requirement that these timestamps form a contiguous interval that starts immediately after the version that the read returns. For instance, if $[tr+1, te]$ denotes the read-locked timestamps, then the read must return the value committed with timestamp *tr*. This requirement is necessary for serializability: intuitively, the read locks permit the transaction to commit with any timestamp $t \in [tr+1, te]$ after having read v, by preventing other transactions from writing a different value with a timestamp between *tr* and *te*. After locking, the transaction remembers k and *tr*; knowledge of k is necessary to commit, and knowledge of both k and *tr* is needed to garbage collect the locks of the transaction.

To commit, a transaction gets zero or more additional locks (function COMMIT-LOCKS in line 12) and tries to find a commit timestamp t that is write-locked for every k in the write-set, and that is read- or write-locked for every k in the read-set. (A key in the read-set may be write-locked because the transaction read the key and then wrote it.) If there are many such timestamps, the transaction picks one (function COMMIT-TS in line 16). The transaction then freezes write-locks on that timestamp and records the written values so that they can be seen by other transactions. As an optional step (as determined by calling COMMIT-GC in line 21), the transaction may garbage collect the locks it holds. Doing so freezes the read locks between the version read and the commit timestamp, and releases all other locks. If the algorithm skips garbage collection on commit, garbage collection can be invoked any time later in the background; this is not shown in the code.

Algorithm 3 The generic MVTL algorithm (part 2/2): policy

```
1: function WRITE-LOCKS($tx, k$)
2:     acquire write-locks for $tx$ on $(k, T)$ for some set $T$
3: function READ-LOCKS($tx, k$)                                  ▷ returns a timestamp or ⊥
4:     acquire read-locks for $tx$ on $(k, T)$ for some $T = [tr+1, \ldots]$ where $Values[k, tr] \neq \bot$
5:     either return $tr$ or return ⊥
6: function COMMIT-LOCKS($tx$)
7:     acquire read- or write-locks for $tx$ on some keys and timestamps
8: function COMMIT-TS(T) return some $t \in T$
9: function COMMIT-GC($tx$) either return true or return false
```

The algorithm depends on a policy of what locks to acquire, how to pick one of many possible commit timestamps, and whether to garbage collect during commit; these choices can depend on the transaction and other considerations. The choices are determined by the functions that we mentioned above: WRITE-LOCKS, READ-LOCKS, COMMIT-LOCKS, COMMIT-TS, and COMMIT-GC. The generic MVTL algorithm uses a generic policy that makes these choices nondeterministically (Algorithm 3). For example, to obtain write locks, the generic policy nondeterministically picks a set T of timestamps to lock. We note that in the READ-LOCKS function, the policy always locks an interval of timestamps starting immediately after a committed version, whose value is returned. Given that read locks are applied for this entire interval, no other write lock or version can exist within this interval. We also note that, as explained in Section 7.4.1, in the case of write locks, we can choose timestamps that are unique to the process performing the transaction.

We prove that the generic MVTL algorithm is correct with its nondeterministic choices. Naturally, this correctness carries over to any specialization that fixes the nondeterministic choices in any way. These specializations lead to different algorithms (Section 7.5) that achieve different benefits.

Some policies of the generic algorithm may cause deadlocks, where a process waits forever to acquire a lock. In such cases, standard techniques for deadlock detection can be used to abort the required transactions (e.g., cycle detection in the wait-for graph, timeout, etc).

Theorem 1. *The generic MVTL algorithm (Algorithms 2 and 3) ensures serializability.*

We now provide a detailed proof. Essentially, we show that any schedule of our protocol is view equivalent to a serial one-version schedule where transactions are serialized in the order of their commit timestamps. We prove this by showing that the multiversion serialization graph [15] resulting from our protocol is acyclic.

Proof. We denote by $T.committs$ the timestamp at which transaction T is serialized and commits (aborted transactions do not have a serialization timestamp). Each transaction has a unique serialization timestamp, as explained in Section 7.4.1. If a transaction T commits at a timestamp $T.committs$, then it holds write locks at $T.committs$ for all the data in its write set, and read locks from the largest timestamp smaller than $T.committs$ containing a committed value to $T.committs$ for all the data in its read set (Algorithm 2, line 13). We denote by $r_i[x_j]$ the fact that transaction T_i has read a version of object x written by transaction T_j (i.e., the read operation has returned $Values[x, T_j.committs]$). In addition, we denote by $w_k[x_k]$ the fact that transaction T_k has written a new version of object x (i.e., it has written a value to $Values[x, T_k.committs]$).

We assume the serialization order is given by the commit timestamp of the transaction. That is, if transaction T_1 creates version v_1 of object o, and transaction T_2 creates version v_2 of object o, we say $v_1 \ll v_2$ iff $T_1.committs < T_2.committs$.

Let H be a multiversion history over a set of transactions $\{T_0, \dots, T_n\}$, and $C(H)$ the committed projection of this history. The committed projection of an operation history retains only the operations that belong to committed transactions. A multiversion serialization graph (MVSG) has the transactions $\{T_0, \dots, T_n\} \in C(H)$ as vertices and edges (1) from T_i to T_j if T_j reads from T_i, and (2) for $r_k[x_j]$ and $w_i[x_i] \in C(H)$, if $x_i \ll x_j$, then the graph has an edge from T_i to T_j, otherwise it has an edge from T_k to T_i.

It has been shown [15] that if the multiversion serialization graph is acyclic, then a multiversion history is *one copy serializable*, that is, equivalent to a serial one version history.

Similarly to the proof of the original multiversion timestamp order algorithm, we show the MVSG resulting from MVTL is acyclic by showing that if an edge between T_i and T_j exists in the graph, $T_i.committs < T_j.committs$. We consider the types of edges that can appear in a multiversion serialization graph. The first type of edges are *reads-from edges*. In this case, transaction T_j reads a version written by transaction T_i. Function READ-LOCKS acquires locks for timestamps starting immediately after the timestamp containing the version whose value is returned (and, since it read-locks an interval of timestamps, does not lock timestamps equal or larger to later versions). Hence, the read can only be serialized at a timestamp higher than that at which the read version was created. Thus, $T_i.committs \leq T_j.committs$. The second type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_i \ll x_j$. In this case, an edge from T_i to T_j exists in the graph. By definition of $\ll$, $x_i \ll x_j$ iff $T_i.committs < T_j.committs$. Finally, the third type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_j \ll x_i$. In this case, an edge from T_k to T_i is created (this assumes $k \neq i$). Since $x_j \ll x_i$, we know that $T_j.committs < T_i.committs$. Given that T_k has performed a read of version x_j, T_k has necessarily applied read locks for each timestamp from $T_j.committs + 1$ to $T_k.committs$. A read lock can only be acquired if no write lock from another transaction is present. Similarly, a write lock on a timestamp cannot be acquired if a read lock from another transaction is present. Thus, $w_i[x_i]$ could not have occurred in the interval $[T_j.committs + 1, T_k.committs]$. And since we know $T_j.committs < T_i.committs$, $w_i[x_i]$ must have necessarily occurred after the interval. Thus, $T_k.committs < T_i.committs$. Given that all the edges in the graph are from transactions with lower serialization timestamps to transactions with higher serialization timestamps, a cycle cannot exist. Thus, H is one-copy serializable. □

7.5 Simple MVTL Algorithms

We now give several simple algorithms that are special cases of the generic MVTL algorithm, each with a different benefit. To specify these algorithms, we specialize the generic policy of MVTL (Algorithm 3).

7.5.1 The preferential algorithm

Roughly speaking, the preferential algorithm, denoted MVTL-Pref, works with multiple timestamps for each transaction, where one of the timestamps is preferential. The algorithm tries to commit a transaction using its preferential timestamp, but if doing so would abort, it tries one of the other timestamps. To ensure viability of the other timestamps, the algorithm locks them as necessary during the execution.

More precisely, MVTL-Pref is parameterized by a function $A(t)$ that takes the transaction's preferential timestamp and returns a non-empty set of alternative timestamps different from t. $A(t)$ is a choice of the user of the algorithm. For example, $A(t) = \{t{-}10, t{+}10\}$ indicates that $t{-}10$ and $t{+}10$ are the alternative timestamps for a transaction with preferential timestamp t. The preferential timestamp itself comes from a clock, as in other timestamp-based protocols.

We assume that clock timestamps are unique (e.g., by appending the process id to each timestamp t) and that $A(t)$ also produces unique timestamps (e.g., by using the process id in t for each timestamp in $A(t)$).

When executing a read on a key k, the algorithm determines a version to return based on the preferential timestamp, and then read-locks contiguous timestamps of k to cover as many alternative timestamps as possible. When executing a write to key k, the algorithm obtains no locks; rather, locks are acquired at commit time, as follows. If the algorithm cannot obtain a write-lock for the preferential timestamp for each written key, it tries one of the alternative timestamps. If it manages to obtain read- and write-locks for all read and written objects at one of the timestamps, the transaction commits; otherwise it aborts.

We can show that if we choose the alternative timestamps $A(t)$ to be smaller than the preferential timestamps t, then MVTL-Pref aborts strictly fewer workloads compared to MVTO+. More precisely:

Theorem 2. *Suppose that $\forall t' \in A(t), t' < t$. (a) If a workload W produces no abort under MVTO+, then W produces no abort under MVTL-Pref. (b) There are infinitely many workloads that produce no aborts under MVTL-Pref but produce aborts under MVTO+.*

The MVTL-Pref algorithm is given in Algorithm 4. Each transaction is assigned a *preferential timestamp* and one or more alternative timestamps. The system tries to commit the transaction using first the preferential timestamp, but if that would abort the transaction, it tries the alternative timestamps. Transactions are serialized in the order of their commit timestamps.

More precisely, the algorithm is parameterized by a function $A(t)$ that takes the transaction's preferential timestamp and returns a non-empty *set* of alternative timestamps different from t. For example, $A(t) = \{t-10, t+10\}$ indicates that $t-10$ and $t+10$ are the alternative timestamps for a transaction with preferential timestamp t. The preferential timestamp itself comes from a clock, as in other timestamp-based protocols. Similarly, we assume that processes obtain unique timestamps (e.g., by appending the process id to each timestamp t) and that $A(t)$ also produces unique timestamps (e.g., by using the process id in t in each timestamp in $A(t)$).

When reading, the system acquires read-locks for a set that includes the preferential timestamp and as many other timestamps as possible. When committing, the system tries to write-lock on all objects in the write set and the preferential timestamp; if that is not possible, it tries each of the alternative timestamps.

We provide a more precise definition of the concept of a workload:

Definition 1. *A* workload *is a set of n transaction inputs, where each transaction input is a finite sequence of operation-timestamp pairs with increasing timestamps and an operation is either read(k), write(k, v) or tryCommit.*

We now show that under certain conditions on $A(t)$, MVTL-Pref is strictly better than MVTO+, in the sense that (a) if MVTO+ does not abort under a workload, then MVTL-Pref does not

Algorithm 4 The MVTL-Pref algorithm

```
1: function INITIALIZATION($tx$)
2:     $tx.PrefTS \leftarrow clock()$
3:     $tx.PossTS \leftarrow \{tx.PrefTS\} \cup A(tx.PrefTS)$                ▷ possible timestamps for $tx$
4: function WRITE-LOCKS($tx, k$) return                                      ▷ lock write-set only on commit
5: function READ-LOCKS($tx, k$)
6:     repeat
7:         $tr \leftarrow \max\{t : t < tx.PrefTS$ and $Values[k, t] \neq \perp\}$      ▷ Candidate value to read
8:         $tmax \leftarrow \max\{t \in tx.PossTS$ : no timestamps in$[tr{+}1, tmax]$ are write frozen$\}$
9:         for $t \leftarrow tr{+}1$ to $tmax$ do                  ▷ read-lock interval $[tr{+}1, tx.TS]$ if possible
10:            try to acquire read-lock for $tx$ on $(k, t)$, waiting
                   if timestamp is write-locked but not frozen
11:            if found frozen write-lock then release read-locks acquired above; break ▷ exit the "for" loop
12:    until found no frozen locks in the for loop
13:    $tx.PossTS \leftarrow tx.PossTS \cap [tr, tmax]$                       ▷ update possible timestamps
14:    return $tr$
15: function COMMIT-LOCKS($tx$)
16:    for $t \in tx.PossTS$ do   ▷ Find a good timestamp. Loop order: first $tx.TS$ then arbitrary for $PossTS$
17:        $gotlocks \leftarrow$ true
18:        for $(k, tr) \in tx.writeset$ do
19:            try to write-lock for $tx$ on $(k, t)$, without waiting if a timestamp is read-locked
20:            if write-lock not acquired then
21:                $gotlocks \leftarrow$ false                              ▷ this timestamp will not work
22:                release all write locks for $tx$
23:                break                                               ▷ exit inner "for" loop
24:        if $gotlocks$ then break ▷ found a timestamp for which we can get write locks; exit outer "for" loop
25:    if $gotlocks$ then $tx.TS \leftarrow t$                             ▷ found good timestamp
26:    else $tx.TS \leftarrow \perp$                                          ▷ no good timestamps
27: function COMMIT-TS(T) return $tx.TS$
28: function COMMIT-GC($tx$) return $false$
```

abort either, and (b) there are infinitely many workloads where MVTO+ aborts but MVTL does not. These results hold assuming that $A(t)$ contain only timestamps smaller than t, that is, the alternative timestamps are smaller than the preferential one.

Proof sketch. (a) Consider a workload W that does not abort under MVTO+. We prove that, for each transaction T in W, the execution of T under MVTO+ and MVTL-Pref will read- and write-lock exactly the same timestamps. The intuition here is that MVTL-Pref will choose the same timestamps as MVTO+ under workload W, because W does not cause any aborts. More precisely, we can show that (i) whenever a read occurs, both MVTO+ and MVTL-Pref pick the same value to return for the read (the first non-$\perp$ value with a timestamp smaller than the preferential timestamp); because the preferential timestamp is higher than any of the timestamps in $A(t)$, the MVTL-Pref picks the preferential timestamp as *tmax* and therefore locks the same range as MVTO+. Moreover (ii), whenever a commit occurs, both MVTO+ and MVTL-Pref pick the same timestamp to lock. This is because MVTL-Pref picks the preferential timestamp, given that MVTO+ does not abort. From (i) and (ii), it is possible to show that MVTL-Pref executes in exactly the same way as MVTO+ under W. Therefore, MVTL-Pref does not abort any transactions under W.

(b) Pick three timestamps $t_1 < t_2 < t_3$ such that $\max A(t_2) < t_1$. These will be the timestamps for transactions T_1, T_2, T_3. Consider the following workload: $W_1(Y)\,C_1\,R_2(X)\,R_3(Y)\,C_3\,W_2(Y)\,C_2$. Under MVTO+, this workload aborts T_2 since the timestamp at which T_2 wants to write Y is between t_1 and t_3. However, under MVTL-Pref, T_2 commits because MVTL-Pref can pick the alternative timestamp $\max A(t_2)$ with which to commit T_2. It is easy to generalize this example to several transactions, and thus obtain infinitely many workloads where MVTO+ causes an abort but MVTL-Pref does not. □

7.5.2 The prioritizer algorithm

Multiversion timestamp ordering provides no way for critical transactions to be prioritized over normal transactions. We explain how MVTL can do that, by using a policy that gives more locks to critical transactions. There are many ways to do that, but the simplest one is as follows. Normal transactions obtain their locks as in multiversion timestamp ordering using synchronized clocks, while critical transactions try to acquire all locks as in pessimistic concurrency control except that critical transactions do not block waiting for any of its locks. Both types of transactions garbage collect on commit.

Under this policy, we can show the following:

Theorem 3. *In the MVTL-Prio algorithm, transactions labeled critical are never aborted by transactions labeled normal.*

Given that high-priority transactions behave similarly to pessimistic concurrency control, they can cause deadlocks. However, as we show below, transactions with normal priority behave identically to those in MVTO+, and thus never cause deadlocks.

The MVTL-Prio algorithm is given in Algorithm 5. Operations from transactions with priority try to lock timestamps up to $+\infty$: writes attempt to lock all timestamps, while reads lock from the latest observed write onwards; the transaction commits at the lowest timestamp that was locked for all its data items. In contrast, transactions with no priority behave identical to the MVTO+ algorithm: they read the clock at the beginning and try to serialize all operations at that point (thus only acquiring locks for timestamps lower than or equal to the clock value at the beginning of the transaction).

Proof sketch. Assume $maxts$ is the maximum serialization timestamp of all completed or executing transactions with no priority. For any objects, transactions without priority will not prevent a transaction with priority from locking the interval $[maxts, +\infty]$, and thus committing at a timestamp at most $maxts$. Thus, transactions without priority cannot cause

Algorithm 5 The MVTL-Prio algorithm

```
1: function INITIALIZATION($tx$)
2:     if $tx.priority = false$ then $tx.TS \leftarrow clock()$
3: function WRITE-LOCKS($tx, k$)
4:     if $tx.priority = true$ then
5:         for $t = +\infty$ downto 0 do                    ▷ write-lock all the possible timestamps
6:             try to acquire write-lock for $tx$ on $(k, t)$, waiting
               if a timestamp is read- or write-locked but not frozen
7: function READ-LOCKS($tx, k$)
8:     if $tx.priority = true$ then
9:         repeat
10:            $tr \leftarrow \max\{t : t < tx.TS$ and $Values[k, t] \neq \bot\}$
11:            for $t = +\infty$ downto $tr+1$ do           ▷ read-lock interval $[tr+1, +\infty]$ if possible
12:                try to acquire read-lock for $tx$ on $(k, t)$, waiting
                    if timestamp is write-locked but not frozen
13:                if found frozen write-lock then release read-locks acquired above; break     ▷ exit the
    “for” loop
14:        until found no frozen locks in the for loop
15:    else
16:        repeat
17:            $tr \leftarrow \max\{t : t < tx.TS$ and $Values[k, t] \neq \bot\}$
18:            for $t = tr+1$ to $tx.TS$ do                 ▷ read-lock interval $[tr+1, tx.TS]$ if possible
19:                try to acquire read-lock for $tx$ on $(k, t)$, waiting
                    if timestamp is write-locked but not frozen
20:                if found frozen write-lock then release read-locks acquired above; break     ▷ exit the
    “for” loop
21:        until found no frozen locks in the for loop
22:    return $tr$
23: function COMMIT-LOCKS($tx$)
24:    if $tx.priority = false$ then
25:        for $(k, tr) \in tx.writeset$ do
26:            try to write-lock for $tx$ on $(k, tx.TS)$, without waiting if a timestamp is read-locked
27:            if write-lock not acquired then
28:                $tx.TS = \emptyset$ and release all write locks for $tx$;
29:                return ;
30: function COMMIT-TS(T)
31:    if $tx.priority = true$ then
32:        return min $T$
33:    else
34:        return $tx.TS$
35: function COMMIT-GC($tx$)
36:    if $tx.priority = true$ then
37:        return $true$
38:    else
39:        return $false$
```

a transaction with priority to abort. □

7.5.3 The ϵ-clock algorithm

Multiversion timestamp ordering uses clocks to obtain its timestamps, but if clocks are not synchronized or monotonic[1], it is susceptible to *serial aborts*—aborts that occur in an execution that is completely serial. This is a concern in modern multi-core machines that do not guarantee that clocks across cores are perfectly synchronized. For example, T_2 gets timestamp 2, reads an object X, and commits. Afterwards, T_1 gets a smaller timestamp 1, writes X, and tries to commit. This will cause T_1 to abort since the read-timestamp of X at version 0 is 2. This is the schedule:

$$\begin{array}{lllll} T_2: & R(X) & C & & \\ T_1: & & & W(X) & A \end{array}$$

Here, time flows to the right and each line shows the operations of a transaction. R, W, C, and A indicate a read, write, commit, and abort; and X is the key. Thus, this schedule has two transactions T_1 and T_2, where T_2 reads X and commits, and then T_1 writes X and aborts.

The MVTL-ϵ-clock algorithm, which we now introduce, avoids serial aborts when used with ϵ-synchronized clocks. Briefly, when it starts, a transaction reads the clock, obtains a time t, and for each read and write tries to lock the interval $[t-\epsilon, t+\epsilon]$. At the end, it commits at the smallest common timestamp it locked for every accessed object. Before completing the commit, the transaction runs garbage collection. In a sequential execution, it is possible to show that *tx* picks a commit timestamp that is at most t, and thus it releases the lock on higher timestamps. As a result, the next transaction in the sequence will always have its own real time in the intersection of locked time points, and therefore does not abort.

The MVTL-ϵ-clock algorithm is shown in Algorithm 6. It assumes that clocks are ϵ-synchronized and ensures that transactions never abort in serial executions.

Upon start, a transaction *tx* reads the clock, obtains a time t, and sets a local variable *tx.TS* to the interval $[t-\epsilon, t+\epsilon]$. This set has the timestamps that *tx* tries to lock as it executes. To write k, *tx* obtains a write-lock on as many timestamps in *tx.TS* as possible, waiting if any of the timestamps is read- or write-locked (but not frozen) by another transaction; if *tx* already holds a read-lock on a timestamp, it waits until it can upgrade it to a write-lock. Next, if *Tw* denotes the locks that *tx* actually manages to acquire, *tx* sets *tx.TS* to *Tw*.

To read k, *tx* selects the largest timestamp m in *tx.TS*, finds the largest timestamp $tr < m$ under which k has been written, and then tries to acquire a read-lock on $[tr+1, m]$ (if *tx* already has a write-lock then it does not need to acquire a read-lock), waiting if a timestamp is write-locked (but not frozen) by another transaction. *tx* may find a frozen write-lock if some other

[1]A monotonic clock is one that ensures that it returns a higher timestamp if it is queried later in time. Monotonic clocks and time-synchronized clocks are equivalent insofar this discussion is concerned.

Algorithm 6 The MVTL-ϵ-clock algorithm

```
1:  function INITIALIZATION($tx$)
2:      $now \leftarrow clock()$
3:      $tx.TS \leftarrow [now - \epsilon, now + \epsilon]$
4:  function WRITE-LOCKS($tx, k$)
5:      try to write-locks for $tx$ on $(k, tx.TS)$, waiting
            if a timestamp is read- or write-locked but not frozen
6:      $tx.TS \leftarrow$ write-locks that $tx$ could acquire
7:  function READ-LOCKS($tx, k$)
8:      if $tx.TS = \emptyset$ then return $\perp$
9:      $m \leftarrow \max tx.TS$
10:     repeat
11:         $tr \leftarrow \max\{t : t < m$ and $Values[k, t] \neq \perp\}$
12:         for $t = tr{+}1$ to $m$ do            ▷ read-lock interval $[tr{+}1, m]$ if possible
13:             try to acquire read-lock for $tx$ on $(k, t)$, waiting
                    if timestamp is write-locked but not frozen
14:             if found frozen write-lock then release read-locks acquired above; break  ▷ exit the "for"
    loop
15:     until found no frozen locks in the for loop
16:     $tx.TS \leftarrow tx.TS \cap [tr{+}1, m]$
17:     return $tr$
18: function COMMIT-LOCKS($tx$) return
19: function COMMIT-TS(T) return $\min T$
20: function COMMIT-GC($tx$) return $true$
```

transaction commits after *tx* picked *tr*; In that case, *tx* picks *tr* again and retries. Then *tx* updates *tx.TS* to contain the locked timestamps.

To commit, *tx* picks the smallest locked timestamp and runs garbage collection before completing the commit.

Note that initially *tx.TS* contains the correct real-time *treal* when *tx* started. In a sequential execution, we show that *tx* picks a commit timestamp that is at most *treal*, and thus it releases the lock on higher timestamps. As a result, the next transaction in the sequence will always have its own real time in its *tx.TS*, so that does not abort.

We now show that MVTL-ϵ-clock is not susceptible to serial aborts, which we define precisely as follows:

- *(Serial abort)* An algorithm is susceptible to serial aborts if it has a serial schedule that aborts some transaction.

Theorem 4. *The MVTL-ϵ-clock algorithm is not susceptible to serial aborts when clocks are ϵ-synchronized.*

Proof sketch. According to the ϵ-clock assumption, the local clock the transaction sees can diverge from the real time by at most ϵ. The first step a transaction takes when it starts is to read its local clock t. Assume t_{real_start} is the real time when local clock value t is read. Given that T starts with the interval $[t-\epsilon, t+\epsilon]$, it is guaranteed that $t_{real\_start} \in [t-\epsilon, t+\epsilon]$. At commit time, according to the ϵ-clock algorithm, a transaction commits with the smallest timestamp in its interval it was able to lock for all data items.

We show that if all transactions execute serially, each transaction will be able to commit, and that its commit point will not be larger than the real time at the beginning of the transaction. We prove this by induction:

Base case. Assume T_1 is the first transaction that executes serially in the system. The first point in its assigned interval $(t-\epsilon)$ will be at most equal to the real time at the start of the transaction. Given that no conflicting data exists in the system, this first transaction will be able to commit at this smallest timestamp in the interval.

Inductive step. Assume $n-1$ transactions have executed serially, and have each committed at a timestamp that was at most equal to the real time at the respective start of the transaction. We now show the n-th serial transaction will also commit with a timestamp at most equal to the real time at which it started.

Given that transactions execute serially, we know that the n-th transaction begins only after the previous one has completed. According to the algorithm, a transaction completes only after it performs garbage collection. Therefore, assuming the transactions committed with timestamps at most equal to the real time when they started, after the first $n-1$ transactions commit, no lock is held for timestamps higher than the real time the $n-1$-th transaction started. As the transactions execute serially, the real time the n-th transaction starts is larger than the real time any of the previous transactions started, and thus higher than any lock held in the system (therefore, no conflict can arise for a serial transaction that tries to commit at this timestamp). As the interval assigned to transaction n is guaranteed to contain the real time at the transaction's start, the n-th transaction will be able to commit with a timestamp at most equal to the real time when it started.

□

If concurrent transactions start less than $2 * \epsilon$ time apart in real time, since operations always wait if timestamps are locked but not frozen, they may have to wait for each other's operations to complete. Therefore our algorithm intuitively behaves similarly to pessimistic concurrency control for these transactions. Thus, the trade-off with this algorithm is that deadlocks are possible, and the system requires a deadlock detection mechanism.

7.5.4 Existing algorithms as special cases

We now show that MVTL generalizes two popular transactional algorithms, MVTO+ and pessimistic concurrency control. More precisely, we give two algorithms MVTL-TO and

MVTL-Pessimistic, which specialize MVTL and behave exactly like MVTO+ and pessimistic concurrency control, respectively

MVTO+ as a special case of MVTL

In MVTL-TO, each transaction obtains a timestamp t from a clock when the transaction starts. Writes do not lock anything, reads try to lock $[tr+1, t]$ (waiting for unfrozen locks) where tr is the largest timestamp before t for which $Values[k, tr] \neq \perp$, and commits lock t for each object in the transaction's write-set. Garbage collection is not invoked on commit.

The MVTL-TO algorithm is given in Algorithm 7. Each transaction chooses a serialization timestamp at the beginning, and attempts to serialize every operation at this timestamp. For reads, it finds the largest timestamp with a committed value smaller than its chosen serialization timestamp, applies read locks to every timestamp between these two, and returns the version's value. This is equivalent to reading the version with the largest timestamp smaller than the transaction timestamp and setting its *read-timestamp* in MVTO+. If a read encounters a timestamp that is write-locked, but not frozen, it waits. This wait is short: it stops when write locks that are not frozen are finally frozen.

For writes, the algorithm simply retains the values it wishes to write in its write set, without acquiring any locks. Only at commit time does the protocol try to lock the write set at the chosen serialization timestamp. If any read lock is encountered (frozen or not), the write lock is unsuccessful (since no garbage collection is performed). When a transaction fails to acquire a write lock, it releases all previously acquired write locks, and aborts. In case all the write locks are successfully acquired, they are then frozen and values are associated with the transaction's timestamp.

Theorem 5. *The MVTL-TO algorithm behaves as the MVTO+ algorithm.*

Proof sketch. Like MVTO+, MVTL-TO processes transactions such that they appear to execute in the order of their timestamp. The protocol provides all the properties of MVTO+, such as reads never aborting and only having read-write conflicts (given each process can choose unique timestamps, writes never conflict with other writes). □

2PL as a special case of MVTL

Pessimistic concurrency control locks objects before accessing them, thus preventing potentially conflicting operations from executing at the same time. To emulate pessimistic concurrency control using MVTL, writes acquire write locks on all timestamps (blocking), while reads acquire read-locks on all timestamps in $[tr+1, \infty]$ (blocking). Garbage collection is invoked on commit.

Briefly, the pessimistic concurrency control algorithm works as follows: as reads and writes

Algorithm 7 The MVTL-TO algorithm

```
1:  function INITIALIZATION(tx)
2:      $tx.TS \leftarrow clock()$
3:  function WRITE-LOCKS(tx, k) return
4:  function READ-LOCKS(tx, k)
5:      repeat
6:          $tr \leftarrow \max\{t : t < tx.TS$ and $Values[k, t] \neq \bot\}$
7:          for $t \leftarrow tr+1$ to $tx.TS$ do          ▷ read-lock interval $[tr+1, tx.TS]$ if possible
8:              try to acquire read-lock for tx on $(k, t)$, waiting
                    if timestamp is write-locked but not frozen
9:              if found frozen write-lock then release read-locks acquired above; break  ▷ exit the "for" loop
10:     until found no frozen locks in the for loop
11:     return tr
12: function COMMIT-LOCKS(tx)
13:     for $(k, tr) \in tx.writeset$ do
14:         try to write-lock for tx on $(k, tx.TS)$, without waiting if a timestamp is read-locked
15:         if write-lock not acquired then
16:             $tx.TS = \emptyset$ and release all write locks for $tx$
17:             return ;
18: function COMMIT-TS(T) return $tx.TS$
19: function COMMIT-GC(tx) return false
```

are executed, they apply locks on the objects they access. At most one write can access any object at a point in time. If an object is locked for a write, no reads from other transactions can proceed concurrently. If a transaction cannot acquire a lock for an object, it waits until the lock is released. When all the locks are successfully acquired, the transaction performs its updates to the objects, and then unlocks.

This algorithm can be seen as a special case of MVTL with a specific policy, as shown in Algorithm 8. Basically, writes try to lock all possible timestamps, starting from $+\infty$ downwards, while reads also start from $+\infty$, and apply read locks to all timestamps down to the first timestamp where a write committed (whose value is also returned). If a transaction has successfully acquired locks for all its data, it will commit at the minimum timestamp that is locked for every data item (since such a timestamp always exists, the transaction will not abort—aborts can only potentially occur in case of deadlock). This timestamp will be equal to one greater than the largest timestamp of any read data, and is guaranteed to be less than $+\infty$. At the end of the transaction, the unneeded locks are released (including, in particular $+\infty$) and the next transaction can acquire locks for the concerned data items.

Proof sketch. Since both reads and writes first try to lock $+\infty$, it is guaranteed that at most one writer or multiple readers can have access to an object. Moreover, a transaction that has completed will never prevent other transactions from accessing any data object. □

Theorem 6. *The MVTL-Pessimistic algorithm behaves as the pessimistic concurrency control algorithm.*

Algorithm 8 The MVTL-Pessimistic algorithm

```
 1: function WRITE-LOCKS($tx, k$)
 2:     for $t = +\infty$ downto 0 do                         ▷ write-lock all the possible timestamps
 3:         try to acquire write-lock for $tx$ on $(k, t)$, waiting
                if a timestamp is read- or write-locked but not frozen
 4: function READ-LOCKS($tx, k$)
 5:     repeat
 6:         $tr \leftarrow \max\{t : t < m$ and $Values[k, t] \neq \bot\}$
 7:         for $t = +\infty$ downto $tr+1$ do                ▷ read-lock interval $[tr+1, +\infty]$ if possible
 8:             try to acquire read-lock for $tx$ on $(k, t)$, waiting
                    if timestamp is write-locked but not frozen
 9:             if found frozen write-lock then release read-locks acquired above; break  ▷ exit the "for"
    loop
10:     until found no frozen locks in the for loop
11:     return $tr$
12: function COMMIT-LOCKS($tx$) return
13: function COMMIT-TS(T) return min $T$
14: function COMMIT-GC($tx$) return $true$
```

7.5.5 The ghostbuster algorithm

Under multiversion timestamp ordering, a transaction may abort and later create a conflict with another transaction, causing it to abort. For example, suppose that T_1 starts with timestamp 1, T_2 starts with timestamp 2, and T_3 starts with timestamp 3. Then T_3 reads X and commits, T_2 reads Y, writes X, and tries to commit with its timestamp 2, but T_2 aborts because T_3 read X with timestamp 3. Next T_1 writes Y and tries to commit but aborts due to the read by T_2. This is a ghost abort, because the write of T_1 has a conflict with a transaction T_2 that had aborted before the write of T_1 started. This is the schedule:[2]

T_3 :	$R(X)$	C					
T_2 :			$R(Y)$	$W(X)$	A		
T_1 :						$W(Y)$	A

While multiversion timestamp ordering has ghost aborts, MVTL-Ghostbuster can avoid that. MVTL-Ghostbuster is a simple modification to the MVTL-TO algorithm (Section 7.5.4): when a transaction commits, it performs garbage collection. This ensures that transactions that abort do not leave behind locks that cause ghost aborts.

We now give the MVTL-Ghostbuster algorithm, which avoids ghost aborts. We start with a precise definition of ghost aborts. To do so, we first define the notion of an active conflict, which intuitively means a conflict with a transaction that is concurrently running. More precisely, given an execution of algorithm:

- *(Active conflict)* A transaction T_i has an active conflict if it has an operation o_i that conflicts

[2]Here, transactions get a timestamp before their first operation, but one can construct a more complex schedule with the same problem even if transactions get a timestamp at the first operation.

with some operation o_j of another transaction T_j, where o_i is concurrent with T_j.

- *(Ghost abort)* An algorithm is susceptible to ghost aborts if it has a schedule where a transaction aborts but it has no active conflicts.[3]

To avoid ghost aborts, an algorithm must ensure that each transaction that aborts has at least one operation with an active conflict.

The MVTL-Ghostbuster algorithm is shown in Algorithm 9. This algorithm is similar to MVTL-TO, which emulates MVTO, with the addition of garbage collection before a transaction commits or aborts.

Algorithm 9 The MVTL-Ghostbuster algorithm

```
1: function INITIALIZATION($tx$)
2:     $tx.TS \leftarrow clock()$
3: function WRITE-LOCKS($tx, k$) return
4: function READ-LOCKS($tx, k$)
5:     repeat
6:         $tr \leftarrow \max\{t : t < tx.TS$ and $Values[k,t] \neq \perp\}$
7:         for $t = tr+1$ to $tx.TS$ do        ▷ read-lock interval $[tr+1, tx.TS]$ if possible
8:             try to acquire read-lock for $tx$ on $(k, t)$, waiting
                   if timestamp is write-locked but not frozen
9:             if found frozen write-lock then release read-locks acquired above; break  ▷ exit the "for"
   loop
10:    until found no frozen locks in the for loop
11:    return $tr$
12: function COMMIT-LOCKS($tx$)
13:    if $tx.TS = \emptyset$ then return
14:    for $(k, tr) \in tx.writeset$ do
15:        try to write-lock for $tx$ on $(k, tx.TS)$, waiting
            if a timestamp is read- or write-locked but not frozen
16:        if write-lock not acquired then $tx.TS = \emptyset$ and release all write locks for $tx$;
17: function COMMIT-TS(T) return $tx.TS$
18: function COMMIT-GC($tx$) return $true$
```

Theorem 7. *The MVTL-Ghostbuster algorithm is not susceptible to ghost aborts.*

Proof sketch. MVTL-Ghostbuster chooses a timestamp at the beginning of the transaction, and it serializes transactions according to this timestamp. As in the MVTO algorithm, the only conflicts triggering aborts are read-write conflicts. If a transaction T_i aborts, it must have been because a write lock could not be acquired. This can only happen because a read lock already exists for $T_i.TS$ at the time of the write. If this is a ghost conflict, the lock must have been held by a transaction T_j that has finished its execution and aborted at the time of the conflict. But in real time, a transaction's commit method only finishes (with either an abort or commit

[3]Ghost aborts are different from cascading aborts [164], which occur when a transaction reads uncommitted data.

result) after the GC function is called (in which function, if the transaction aborts, all its locks are removed). It is worth noting that in this algorithm, garbage collection is always performed. Hence, a transaction that aborts only holds any locks while it is executing (i.e., while it is an active transaction). Therefore, a write cannot encounter a conflict due to a transaction that already aborted, and thus no ghost conflicts can appear using this algorithm. □

7.6 Extending MVTL to distributed systems

For the distributed version of MVTL, we consider a standard distributed system model [20], with processes that communicate via message passing. The system is asynchronous: there are no bounds on the relative speed of processes or on communication. Processes have local clocks, with domain $\mathscr{T} = \{0, 1, \ldots\}$, which need not be synchronized. Unless explicitly stated otherwise, processes may exhibit crash-failures: they may stop executing unexpectedly. Where appropriate, we discuss other failure models as well. We assume the data is partitioned among multiple servers, and may or may not be replicated (we discuss both cases). Transactions are coordinated by the processes that want to execute them; we refer to such processes as clients or coordinators.

Algorithms 10 and 12 show the basic algorithm for the client and server respectively, while Algorithm 11 shows a generic policy. The policy is specified by the transaction coordinator, and it is applied by the server.

This generic algorithm leads to specific algorithms with high communication efficiency: only one round-trip to each object in the read set and two round-trips to each object in the write set. This efficiency is possible when the policy does not require garbage collection and its fault tolerance mechanism does not send messages when the coordinator is unsuspected (we discuss when this is viable in Section 7.6.1).

Relative to the centralized MVTL algorithm, the main technical challenge addressed by the distributed MVTL algorithm is handling failures. A transaction coordinator failure may leave write locks in an unfrozen state indefinitely, causing other transactions to block forever. A server failure similarly causes either indefinite waiting from transaction coordinators or failure of all transactions accessing the failed server.

The solution to both types of failure is simple: we associate a *commitment* object with each transaction, to ensure that everyone agrees on whether the transaction committed or aborted. Technically, the commitment object solves consensus: it ensures that (1) no two processes obtain different decisions, (2) the only possible decisions are *abort* or *commit*(t) where t is a timestamp, (3) if the decision is d, some participant proposed d, (4) each correct process eventually decides, and decides only once.

After a coordinator has acquired all the necessary locks and has found a commit timestamp, it proposes the *commit* outcome with the associated timestamp to the commitment object

Algorithm 10 The generic distributed MVTL algorithm

```
function BEGIN($tx$)
    $tx.readset \leftarrow \emptyset$; $tx.writeset \leftarrow \emptyset$; $tx.committs \leftarrow \perp$
function WRITE($tx, k, v$)                              ▷ write $v$ to $k$ in transaction $tx$
    $status \leftarrow$ WRITE-LOCKS($tx, k, v$)              ▷ write lock some subset of timestamps
    if $status = abort$ then
        $decision \leftarrow tx.commitment.tryAbort()$;      ▷ decision must be abort in this case
        mark $tx$ as aborted
        return
    add $(k, v)$ to $tx.writeset$                          ▷ remember key and value we wrote
function READ($tx, k$)                                  ▷ read $k$ in transaction $tx$
    $(tr, V) \leftarrow$ READ-LOCKS($tx, k$)         ▷ read lock some interval $[tr+1, \ldots]$ with $Values[k, tr] \neq \perp$
    if $tr = \perp$ then return $\perp$                                     ▷ read failed
    add $(k, tr)$ to $tx.readset$                          ▷ remember key and version we read
    return $V$                                             ▷ return committed value
function COMMIT($tx$)                                   ▷ try to commit transaction $tx$
    COMMIT-LOCKS($tx$)                                     ▷ locks to acquire at commit time
    $T \leftarrow \{t : \forall k \in tx.readset.keys, tx$ has a lock on $(k, t)$ and   ▷ try to find a locked timestamp for $tx$
            $\forall k \in tx.writeset.keys, tx$ has a write-lock on $(k, t)\}$
    if $T = \emptyset$ then
        $decision \leftarrow tx.commitment.tryAbort()$;      ▷ decision must be abort in this case
        mark $tx$ as aborted
    else
        $tx.committs \leftarrow$ COMMIT-TS($T$)                ▷ pick some timestamp in $T$
        $decision \leftarrow tx.commitment.tryCommit(tx.committs)$;
        if $decision = abort$ then
            mark $tx$ as aborted
        else
            for $(k, v) \in tx.writeset$ do
                send($server(k)$, freeze-write-lock, $k$, $tx.committs$)      ▷ freeze locks
    if COMMIT-GC($tx$) then GC($tx$)                        ▷ invoke gc or not
function GC($tx$)                                       ▷ garbage collect locks of $tx$ after it ended
    if $tx$ committed then
        for $(k, tr) \in tx.readset$ do
            send($server(k)$, freeze-read-locks, $k$, $[tr+1, tx.committs]$)
    send messages to release all unfrozen read- and write-locks for $tx$
```

of the transaction. If the decision is to commit, the coordinator then proceeds to inform the servers in the write set of the commit timestamp, without waiting for replies, allowing them to freeze the write locks associated with this transaction (we note that the protocol would be correct even without this step; we include it for performance). When the servers receive a request to freeze the locks of a transaction, they also propose *commit* with the received timestamp from the coordinator. This is because from the point of view of a server, when it has received the serialization timestamp of a transaction, the transaction is committed. However, if a server has held unfrozen write locks for a certain amount of time without receiving the freeze message from the coordinator, it will assume the coordinator has failed and it will propose an *abort* outcome to the commitment object. If the decision is to commit, the server will receive a timestamp along with the decision and will be able to simply freeze its write

Algorithm 11 Client policy for the generic distributed MVTL algorithm

```
1: function WRITE-LOCKS($tx, k, v$)
2:     send($server(k)$, ($tx$, write-locks, $k$, $v$, $T$)), for some set $T$
3:     wait_message($server(k)$, status, $T'$)        ▷ $T'$ subset of $T$ for which write locks acquired
4:     return status
5: function READ-LOCKS($tx, k$)        ▷ returns a timestamp or $\perp$
6:     send($server(k)$, ($tx$, read-lock, $k$, $T$, criteria)), for some set $T$
7:     wait_message($server(k)$, $tr$, $te$, $V$)        ▷ $[tr+1, te]$ read locked if $tr \neq \perp$, $V$ read value
8:     either return ($tr$,V) or return ($\perp$, $bot$)
9: function COMMIT-LOCKS($tx$)
10:     acquire read- or write-locks for $tx$ on some keys and timestamps as above
11: function COMMIT-TS(T) return some $t \in T$
12: function COMMIT-GC($tx$) either return $true$ or return $false$
```

Algorithm 12 The server

```
1: function RECEIVE-WRITE-LOCK-MESSAGE($tx, k, v, T$)
2:     acquire write-locks for $tx$ on $(k, T')$ for $T' \in T$ in which acquiring locks is possible
3:     $tx.pending\_value(k) \leftarrow v$;        ▷ remember $v$ as new value
4:     send($client(tx)$, write-locks-acquired, $T'$)
5: function RECEIVE-READ-LOCK-MESSAGE($tx, k, T, criteria$)
6:     acquire read-locks for $tx$ on $(k, I)$ for $I = [tr+1, te]$ where $te \in T$ chosen according to $criteria$
   and $Values[k, tr] \neq \perp$
7:     send($client(tx)$, read-locks-acquired, $tr$, $te$ or $\perp$, $Values[k, tr]$ or $\perp$)
8: function RECEIVE-FREEZE-WRITE-LOCK-MESSAGE($tx, k, t$)
9:     $decision \leftarrow tx.commitment.tryCommit(t)$
10:     if $decision$ = abort then
11:         release $tx's$ write locks
12:         return
13:     freeze write-lock for $tx$ on $(k, t)$        ▷ freeze locks
14:     $Values[k, t] \leftarrow tx.pending\_value(k)$        ▷ expose committed value
15:     send($client(tx)$, write-locks-frozen, $k$)
16: function RECEIVE-FREEZE-READ-LOCK-MESSAGE($tx, k, [start, commit]$)
17:     freeze read-locks for $tx$ on $k$ for $[start, commit]$
18:     send($client(tx)$, read-locks-frozen, $k$)
19: function WRITE-LOCK-TIMEOUT($tx$)
20:     $decision \leftarrow tx.commitment.tryAbort()$
21:     if $decision$ = "$commit$ @ $t$" then
22:         freeze write-lock for $tx$ on $(k, t)$        ▷ freeze locks
23:         $Values[k, t] \leftarrow tx.pending\_value(k)$        ▷ expose committed value
24:         send($client(tx)$, write-locks-frozen, $k$)
25:     else        ▷ $decision$ = abort
26:         release $tx's$ write locks
```

locks at that timestamp and consider the transaction committed. The server can make this assumption because a commit decision is only possible if someone proposed *commit*, and a commit proposal only happens after the coordinator has performed all its updates and has found a commit timestamp, or after the coordinator has already informed a server of the commit timestamp. In both these instances the transaction can be committed. In the eventuality of an *abort* decision from the commitment object, a server releases all the write locks associated with that transaction and considers it aborted.

7.6.1 Commitment object implementations

This general mechanism used in our protocol allows various commitment object implementations, depending on the failure model we assume. If the coordinator or any minority of servers may fail, a Paxos-like consensus protocol could be used, with all the servers in the system as participants. This is because no server knows the write set of transactions, which can change dynamically with the execution.

However, in practice, storage servers are often replicated and their failures are masked, to provide both availability and durability of data. In this case, we can consider the storage server as a logical entity that does not fail, and consider only failures of the coordinator. By doing so, we can obtain an efficient implementation of commitment, one that requires little communication in the common failure-free case. We now present the commit object implementation in this case in more detail.

Essentially, the coordinator designates a single server per transaction as the *decision point*. This server can be, for example, the first server accessed by a write operation. Consensus on the outcome of the reliable broadcast (i.e., whether the source has delivered or has crashed) is achieved on this decision server. Servers accessed on subsequent writes will then be informed of the decision point of the transaction. When the coordinator proposes a value to the commitment object, it needs to inform the decision point of this proposal and wait for its decision. When proposing a commit, the message can also act as a *freeze-write-locks* message to the decision server, which is only applied if the decision is to commit. If the coordinator has proposed *abort*, no other outcome is possible (since without receiving a *commit* message from the coordinator, servers themselves can only propose *abort*). However, in case of a commit proposal, the outcome may be that of *abort*: when write locks have been acquired for a certain amount of time, but have not been frozen, the servers suspect the coordinator of having failed, and thus propose *abort*. The *abort* proposal from a server is similar to that of the coordinator: the decision point is contacted, and its decision is followed. If the coordinator's commit proposal has been executed at the decision point earlier than any *abort* proposal, the decision will be to commit, and the decision server sends the commit timestamp along with the decision. A server proposes commit only once it has received the *freeze-write-locks* message. But this message is only sent by the coordinator if the decision has been to commit. Hence, the commit proposal that a server does when freezing write locks can be executed entirely locally. If the server has not been informed of a transaction abort, it simply stores the *commit* decision locally. Thus, in the common, failure-free case, the coordinator does not need to exchange extra messages to be able to ensure fault tolerance.

7.6.2 Correctness

We start by proving the following lemma concerning the outcome of a transaction:

Lemma 1. *In Algorithms 10 and 12, with the generic policy in Algorithm 11, if a participant considers a transaction as committed, no other participant considers it as aborted.*

Proof. The *commit* object associated with each transaction provides the standard properties of uniform consensus:

- *(Termination.)* Every correct process eventually decides some value.
- *(Validity.)* If a process decides v, then v was proposed by some process (and, as previously mentioned, v can only be $abort$ or $commit$).
- *(Integrity.)* No process decides twice.
- *(Agreement.)* No two processes decide differently.

Each process, be it the coordinator or a server, uses the commit object in order to obtain the decision as to whether the transaction should be committed or aborted. Before this, it makes no assumptions about the state of a transaction. At commit time, the coordinator proposes *abort* if no serialization point was found, and commit otherwise. A server that times out proposes *abort*, and a server that receives a freeze write lock message (essentially a commit message) proposes *commit*. By the agreement property, all the processes involved with a particular transaction obtain the same outcome of the transaction.

□

Using Lemma 1, we now prove the following theorem:

Theorem 8. *Algorithms 10 and 12 with the generic policy in Algorithm 11 ensures serializability.*

The proof is largely similar to the centralized version, but we recall it here for completeness.

Proof. We denote by $T.committs$ the timestamp at which transaction T is serialized and commits (aborted transactions do not have a serialization timestamp). Each transaction has a unique serialization timestamp, as explained in Section 7.4.1. If a transaction T commits at a timestamp $T.committs$, then it holds write locks at $T.committs$ for all the data in its write set, and read locks from the largest timestamp smaller than $T.committs$ containing a committed value to $T.committs$ for all the data in its read set (Algorithm 10, line 17). By Lemma 1, if the coordinator considers a transaction to be committed, no server can consider it to be aborted, and thus the locks of the transaction must still be held. We denote by $r_i[x_j]$ the fact that transaction T_i has read a version of object x written by transaction T_j (i.e., the read operation has returned $Values[x, T_j.committs]$). In addition, we denote by $w_k[x_k]$ the fact that transaction T_k has written a new version of object x (i.e., it has written a value to $Values[x, T_k.committs]$).

We assume the serialization order is given by the commit timestamp of the transaction. That is, if transaction T_1 creates version v_1 of object o, and transaction T_2 creates version v_2 of object o, we say $v_1 \ll v_2$ iff $T_1.committs < T_2.committs$.

Let H be a multiversion history over a set of transactions $\{T_0,\ldots,T_n\}$, and $C(H)$ the committed projection of this history. The committed projection of an operation history retains only the operations that belong to committed transactions. A multiversion serialization graph (MVSG) has the transactions $\{T_0,\ldots,T_n\} \in C(H)$ as vertices and edges (1) from T_i to T_j if T_j reads from T_i, and (2) for $r_k[x_j]$ and $w_i[x_i] \in C(H)$, if $x_i \ll x_j$, then the graph has an edge from T_i to T_j, otherwise it has an edge from T_k to T_i.

It has been shown [15] that if the multiversion serialization graph is acyclic, then a multiversion history is *one copy serializable*, that is, equivalent to a serial one version history.

Similarly to the proof of the original multiversion timestamp order Algorithm, we show the MVSG resulting from MVTL is acyclic by showing that if an edge between T_i and T_j exists in the graph, $T_i.committs < T_j.committs$. We consider the types of edges that can appear in a multiversion serialization graph. The first type of edges are *reads-from edges*. In this case, transaction T_j reads a version written by transaction T_i. Function READ-LOCKS acquires locks for timestamps starting immediately after the timestamp containing the version whose value is returned (and, since it read-locks an interval of timestamps, does not lock timestamps equal or larger to later versions). Hence, the read can only be serialized at a timestamp higher than that at which the read version was created. Thus, $T_i.committs \leq T_j.committs$. The second type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_i \ll x_j$. In this case, an edge from T_i to T_j exists in the graph. By definition of $\ll$, $x_i \ll x_j$ iff $T_i.committs < T_j.committs$. Finally, the third type of edge appears if $r_k[x_j]$ and $w_i[x_i]$ are in H and $x_j \ll x_i$. In this case, an edge from T_k to T_i is created (this assumes $k \neq i$). Since $x_j \ll x_i$, we know that $T_j.committs < T_i.committs$. Given that T_k has performed a read of version x_j, T_k has necessarily applied read locks for each timestamp from $T_j.committs + 1$ to $T_k.committs$. A read lock can only be acquired if no write lock from another transaction is present. Similarly, a write lock on a timestamp cannot be acquired if a read lock from another transaction is present. Thus, $w_i[x_i]$ could not have occurred in the interval $[T_j.committs + 1, T_k.committs]$. And since we know $T_j.committs < T_i.committs$, $w_i[x_i]$ must have necessarily occurred after the interval. Thus, $T_k.committs < T_i.committs$. Given that all the edges in the graph are from transactions with lower serialization timestamps to transactions with higher serialization timestamps, a cycle cannot exist. Thus, H is one-copy serializable. □

We now focus on the liveness guarantees of the protocol.

Lemma 2. *If the coordinator does not propose commit for a transaction, no server does either.*

Proof sketch. Servers only propose commit when receiving a freeze write locks request from the coordinator. However, the coordinator only sends these messages once it has proposed to commit the transaction and has received a positive decision. Hence, the servers cannot propose a commit before the coordinator does.

□

Lemma 3. *If a coordinator that has obtained write locks but has not committed fails, it is eventually suspected by every correct server that holds unfrozen write locks for the coordinator's ongoing transaction.*

Proof sketch. The proof is straight-forward, as every server holding unfrozen write locks suspects the coordinator after a certain (finite) amount of time has passed since the locks were acquired (and the *Write-Lock-Timeout* function is called).

□

Lemma 4. *If a coordinator fails before committing a transaction, its write locks are eventually released and the transaction aborted on the correct servers.*

Proof sketch. A transaction is effectively committed when the coordinator proposes *commit* to the commitment object corresponding to the transaction, and obtains the same decision. If a coordinator fails before proposing *commit*, according to Lemma 2, no-one proposes *commit* for its transaction. Therefore, a *commit* decision cannot be reached (according to the Validity property of the commitment object). According to Lemma 3, every server currently holding unfrozen write locks for the coordinator's ongoing transaction at the time of failure suspects it to have failed. In Algorithm 12, when a server suspects a coordinator (when a time-out for the unfrozen write-locks occurs), it proposes *abort*. Thus, since *commit* cannot be proposed, and every server holding write locks must propose *abort*, the only decision that can be reached is to abort.

□

We now prove the following theorem:

Theorem 9. *No transaction initiated by a correct coordinator is indefinitely delayed by a failed coordinator.*

Proof sketch. Indefinite delays can happen in one scenario: when unfrozen write locks are held for an object. A read at a higher timestamp (no matter how high) whose result would depend on whether or not a new version of the object is created at the timestamps that are currently write-locked but not frozen has no choice but to wait. Other operations may be affected by the ongoing transaction of a failed coordinator, but this does not result in waiting, but rather in aborting, and potentially retrying at a higher timestamp, where the two transactions would not interfere. According to Lemma 4, either a coordinator is correct, and thus eventually commits or aborts its transaction, or its write locks are eventually released. Therefore, it cannot be the case that a transaction initiated by a correct coordinator is delayed indefinitely by a failed coordinator.

□

Theorem 10. *Unless at least one server suspects the coordinator to have failed, a transaction that has chosen a serialization timestamp eventually commits.*

Proof sketch. Servers only propose *abort* when suspecting the coordinator to have failed (i.e., unfrozen write locks have been held for too long). Thus, if the coordinator is not suspected, no server proposes *abort*. Thus, the only proposal servers can make in this scenario is to commit. Additionally, if the coordinator has found a serialization timestamp for the transaction, then it must propose *commit*. Thus, since no-one in the system proposes *abort*, and the coordinator must propose *commit*, the final decision of the commitment object must be to commit.

□

7.7 Practical considerations: lock state space and atomic blocks

Reducing lock state space. When we presented the generic MVTL algorithm, we defined a lock for each timestamp and object, which amounts to an infinite lock state space. We did not include mechanisms to compress this information; such mechanisms are orthogonal to the essence of the algorithm. However, a practical implementation should compress the lock state space. To do so, we observe that MVTL algorithms usually acquire and release locks on a small number of points or contiguous intervals (this is true for all algorithms we presented). Rather than keeping a lock state for each timestamp, an implementation should keep a single lock state for the entire interval. In the algorithms we presented, each object holds at most one lock interval per committed transaction. Furthermore, this state can be discarded when the associated version of the object is purged, as we discuss next.

Purging versions. By its nature, a multiversion algorithm keeps multiple versions of each object; this is true not just for MVTL but also for every other multiversion algorithm. Doing so is sensible as storage prices fall. In fact, disk systems such as databases already employ multiversion algorithms, but even memory systems are targets now. Nevertheless, multiversion algorithms need a way to purge old versions, so that each object has a small number of versions—possibly just one version after write activity on the object quiesces. We now explain how this can be done in MVTL. This is easy: at any time, the system can purge any version older than the latest committed one, without affecting the correctness of the algorithm.

Transactions that need that version will abort, so in practice we purge versions older than a time limit chosen based on the duration of its longest transactions. In some MVTL algorithms, there is a lower bound on the timestamps that a transaction locks (e.g., ϵ-clock algorithm); we can purge versions with timestamps below the bound except the last one before the bound without causing any side-effects.

Removing the atomic block. Algorithm 2 has an atomic block in lines 17–19, to avoid partially exposing the writes of a committing transaction when we assign to the array $Values[k, t]$. We can remove this atomic block by (1) first storing a special value in $Values[k, t]$ for all timestamps in the for loop, (2) then storing the actual value v for all timestamps in the loop, and (3) having other processes wait if they read *Values* and see the special value.

7.8 Conclusion

This chapter introduced a new genre of multiversion concurrency control algorithms called multiversion timestamp locking (MVTL). MVTL offers a new way to look at multiversion algorithms, based on locking individual time points. With this perspective, we can find simple algorithms that improve the state of the art in various ways: by committing successfully more workloads than existing multiversion protocols, by avoiding the problems of serial aborts and ghost aborts, and by offering prioritized transactions. We can also view existing algorithms, such as MVTO and pessimistic concurrency control, as special cases of MVTL. Finally, we showed how to realize MVTL in both centralized and distributed systems.

We believe that the algorithms proposed here are only a starting point for many other possibilities opened up by MVTL. The design of other MVTL algorithms is a promising direction for future research.

In addition, MVTL can form the basis of concurrency control in practical systems. In fact, ongoing work with Junxiong Wang [161] has shown that in scenarios with non-negligible contention, a system based on an MVTL algorithm can outperform MVTO+ and 2PL, both in a centralized, as well as in a distributed scenario.

Concluding Remarks Part V

8 Conclusions and Future Work

We conclude this dissertation by discussing the outcomes of this thesis and its implications, and presenting potential avenues for future research.

8.1 Summary and Implications

This dissertation investigated generic methods for designing and implementing fast and scalable practical data structures. In the fist part of the dissertation, we focused on generic approaches aimed at making the most out of modern processors. In Chapter 4, we introduced Asynchronized concurrency, a set of guidelines for the design of concurrent search data structures that yield portably scalable implementations. We then looked at the progress of individual operations in Chapter 5, and observed that state-of-the-art blocking concurrent search data structures behave similarly to wait-free algorithms. Next, we shifted our attention to new memory technologies, and in Chapter 6, we proposed a generic methodology for concurrent data structure design and memory management for NVRAM, that ensures both durability and performance. Finally, we also considered more complex programming abstractions for concurrent access to data. In Chapter 7, we proposed a family of concurrency control protocols for transactions, which by working at a time-point granularity alleviates several issues present in previous work.

At a high level, we believe this dissertation takes a step in making the notoriously difficult task of concurrent programming somewhat simpler for the programmer. A programmer designing a concurrent system now has a set of guidelines they can follow when designing or optimizing a concurrent data structure in order to achieve scalability. Previously, little guidance existed on how such an algorithm should look like. Moreover, they can make an informed decision about the progress guarantee their concurrent data structure truly needs to provide. This can potentially make their task significantly simpler, as, for instance, blocking algorithms are much easier to implement than their wait-free counterparts. In addition, if they have NVRAM at their disposal, and need to ensure durability of data structures, programmers can use the methodology and tools presented in this dissertation in order to maintain performance at the same time. Finally, if their data is accessed through a transactional interface, programmers can either use one of our MVTL-based algorithms to ensure a high degree of parallelism, or they can use the MVTL framework to design their own concurrency control protocol to suit their needs. Moreover, no matter how their MVTL-based algorithm will look like, it will be correct; the precise algorithm only influences performance.

8.2 Future Work

We now discuss some potential directions for future research that build on the work done during this dissertation.

Concurrency and synchronization. Concurrency has been the main area of focus of this dissertation. While a large amount of work exists in this area, a number of avenues of future research exist. Concurrent objects are constantly evolving as underlying architectures change. Their scale also changes, as memory may become shared between multiple servers on the same rack. In fact, technologies such as RDMA already provide access to another machine's memory without involving the remote processor. From a more theoretical perspective, we believe that there is room for improvement in the design of wait-free algorithms for some data structures. In particular, objects such as queues and stacks can benefit from high-throughput wait-free algorithms. In addition, as energy efficiency is becoming one of the major metrics for system performance, some hardware can be expected to become approximate. Similarly, algorithms can be designed to use approximate data, or to produce approximate results.

A topic not discussed in this dissertation, yet which we have studied [37], is message-passing abstractions for multi-cores. Architectures can be expected to become increasingly heterogeneous and not entirely cache coherent. On such machines, it can be expected that explicit message passing becomes one of the main methods of inter-core communication. While some work has been done on efficient communication abstractions, more distributed computing primitives remain unexplored.

Systems for non-volatile RAM. The work presented in this thesis merely scratches the surface when it comes to software for NVRAM. Several open problems remain. First of all, we have focused on search data structures. It is straight-forward to extend our work to other structure types. Moreover, it would be interesting to explore which techniques would work best if one would also assume the durability of the data stored in the write-back caches. On a more conceptual level, operating systems have been designed with the assumption of a layer of smaller capacity, byte-addressable volatile memory, on top of a block-based, larger capacity, durable layer. With NVRAM, one could potentially envisage a flat storage hierarchy: NVRAM is not only byte-addressable and durable, but also denser and more power efficient than DRAM, and thus could potentially provide a much larger capacity. Moreover, the virtual memory abstraction has been originally introduced to provide the illusion of a large main memory in the face of scarce hardware resources. It is worth asking if the abstraction still makes sense on a system with a large amount of NVRAM, and if not, what changes are needed. Also, it is worth considering the granularity at which this durable memory is shared. Traditionally, there has been a tight coupling between the processor and main memory. However, as communication latencies in rack-scale systems become smaller and smaller, one could envisage a scenario where a pool of NVRAM is shared among the processors in a rack, potentially in a single address space. Challenges in designing such systems include being able to handle machine failures and restarts, ensuring the consistency of the persistent memory while maintaining performance, designing appropriate data structures, OS-level changes to accommodate these shared abstractions, and dealing with non-uniformity in access latencies.

Software for heterogeneous platforms. In this dissertation, we have assumed a cache-coherent machine having several general-purpose cores. However, with Dennard scaling having ended, and Moore's law expected to end in 6-9 years, innovation in architecture will have to pursue different directions than those taken in the past decades. More specifically, one can expect hardware that is significantly more heterogeneous, with specialized and programmable units. To an extent, this is already visible today, with platforms deployed in datacenters using FPGAs and ASICs. In addition, not all processing units will have access to cache coherence. It is also expected that designing custom hardware to meet one's needs will become simpler. As a result, we will have to rethink the way several components of our current systems are implemented and structured, as well as how such architectures should be programmed. In particular, the interfaces between the hardware and the system software, and the system software and the applications need to be re-evaluated and re-designed.

Large-scale systems and algorithms. The MVTL-based algorithms presented in Chapter 7 are but a starting point. We believe more MVTL-based algorithms can be developed. Moreover, in this dissertation, we have only established the theoretical foundations of MVTL. A practical system showcasing the potential advantages of MVTL is a natural follow-up. A practical implementation of an MVTL-based algorithm is in fact ongoing work. Our aim is both to show that an MVTL-based system can improve the throughput, commit rate, and communication efficiency of previous approaches, and to show the fact that in practice, through its garbage collection mechanism, MVTL does not require more state to be stored than alternatives. In addition, in the distributed version of MVTL we presented, we have treated replication as a largely orthogonal problem to concurrency control. However, a tighter coupling between concurrency control and replication can yield important performance benefits, and is an issue we plan to explore. In the centralized MVTL version we presented, processes do not have access to each other's state. We would be interested in exploring the optimizations that can be brought by removing this assumption.

Bibliography

[1] Morton Abramson and WOJ Moser. More birthday surprises. *American Mathematical Monthly*, pages 856–858, 1970.

[2] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free Made Fast. STOC 1995.

[3] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are Lock-free Concurrent Algorithms Practically Wait-free? STOC 2014.

[4] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. StackTrack: An Automated Transactional Approach to Concurrent Memory Reclamation. EuroSys 2014.

[5] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A Scalable Relaxed Priority Queue. PPoPP 2015.

[6] Karolos Antoniadis, Rachid Guerraoui, Julien Stainer, and Vasileios Trigonakis. Sequential proximity. NETYS 2017.

[7] Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree As an Example. PODC 2014.

[8] Maya Arbel and Adam Morrison. Predicate RCU: An RCU for Scalable Concurrent Updates. PPoPP 2015.

[9] 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. USENIX ATC 2003.

[10] ASCYLIB. http://github.com/LPD-EPFL/ASCYLIB.

[11] Utku Aydonat and Tarek S. Abdelrahman. Serializability of transactions in software transactional memory. In *ACM SIGPLAN Workshop on Transactional Computing*, February 2008.

[12] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP 2009.

[13] Hal Berenson et al. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data*, 1995.

[14] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems.* 1987.

[16] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. OOPSLA 2016.

[17] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. OSDI 2010.

[18] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. SPAA 2013.

[19] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree. PPoPP 2010.

[20] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to reliable and secure distributed programming.* Springer, 2011.

[21] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 2006.

[22] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box concurrent data structures for numa architectures. ASPLOS 2017.

[23] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. OOPSLA 2014.

[24] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. VLDB 2015.

[25] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMove: Helping Programmers Move to Byte-Based Persistence. INFLOW 2016.

[26] Shimin Chen and Qin Jin. Persistent b+-trees in non-volatile main memory. VLDB 2015.

[27] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using RCU balanced trees. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 199–210. ACM, 2012.

[28] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. SOSP 2013.

[29] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. ASPLOS 2011.

[30] Nachshon Cohen and Erez Petrank. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. SPAA 2015.

[31] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 30(2):16–29, March 2010.

[32] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. SoCC 2010.

[33] J. Corbett et al. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation*, 2012.

[34] Tudor David, Rachid Guerraoui, Tong Che, and Vasileios Trigonakis. Designing ASCY-compliant Concurrent Search Data Structures. Technical report, EPFL, Lausanne, 2014.

[35] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS 2015.

[36] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP 2013.

[37] Tudor David, Rachid Guerraoui, and Maysam Yabandeh. Consensus Inside. Middleware 2014.

[38] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[39] Joel Edward Denny, Seyong Lee, and Jeffrey S. Vetter. Language-Based Optimizations for Persistence on Nonvolatile Main Memory Systems. IPDPS 2017.

[40] Mathieu Desnoyers, Paul E McKenney, Alan S Stern, Michel R Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *Parallel and Distributed Systems, IEEE Transactions on*, 23(2):375–382, 2012.

[41] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[42] Dave Dice, Yossi Lev, and Mark Moir. Scalable Statistics Counters. SPAA 2013.

[43] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical Concurrent Binary Search Trees via Logical Ordering. PPoPP 2014.

[44] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. On the power of hardware transactional memory to simplify memory management. PODC 2011.

[45] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *IEEE Symposium on Reliable Distributed Systems*, pages 173–184, October 2013.

[46] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking Binary Search Trees. PODC 2010.

[47] Facebook. RocksDB. http://rocksdb.org.

[48] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 8(11):1190–1201, July 2015.

[49] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. NSDI 2013.

[50] Panagiota Fatourou and Nikolaos D. Kallimanis. A Highly-efficient Wait-free Universal Construction. SPAA 2011.

[51] Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems*, 2010.

[52] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *International Symposium on Distributed Computing*, 2009.

[53] Faith Ellen Fich, Victor Luchangco, Mark Moir, Nir Shavit, and Sun Microsystems Laboratories. Obstruction-Free algorithms can be practically wait-free. DISC 2005.

[54] Keir Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2004.

[55] Thomas Friebel and Sebastian Biemueller. How to Deal with Lock Holder Preemption. Xen Summit North America 2008.

[56] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. Brief Announcement: A Persistent Lock-Free Queue for Non-Volatile Memory. DISC 2017 (to appear).

[57] Joel Gibson and Vincent Gramoli. Why non-blocking operations should be selfish. DISC 2015.

[58] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *Parallel and Distributed Systems, IEEE Transactions on*, 20(8):1173–1187, 2009.

[59] Google. LevelDB. http://leveldb.org.

[60] Yonatan Gottesman, Joel Nider, Ronen Kat, Yaron Weinsberg, and Michael Factor. Using Storage Class Memory Efficiently for an In-memory Database. SYSTOR 2016.

[61] Vincent Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. PPoPP 2015.

[62] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *ACM Symposium on Principles of Distributed Computing*, 2005.

[63] Rachid Guerraoui and Vasileios Trigonakis. Optimistic Concurrency with OPTIK. PPoPP 2016.

[64] Timothy L Harris. A Pragmatic Implementation of Non-blocking Linked Lists. DISC 2001.

[65] 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.

[66] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, III Scherer, William N., and Nir Shavit. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*, volume 3974. 2006.

[67] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA 2010.

[68] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A Scalable Lock-free Stack Algorithm. SPAA 2004.

[69] Maurice Herlihy. Wait-free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[70] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. SIROCCO 2007.

[71] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-Free Synchronization: Double-Ended Queues As an Example. ICDCS 2003.

[72] Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: a mechanism for supporting dynamic-sized lock-free data structures. Technical report, 2002.

[73] Maurice Herlihy and Nir Shavit. On the Nature of Progress. OPODIS 2011.

[74] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised First Edition*. 2012.

[75] Maurice Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[76] Maurice P Herlihy, Yosef Lev, and Nir N Shavit. Concurrent lock-free skiplist with wait-free contains operator, May 3 2011. US Patent 7,937,378.

[77] Shane V Howley and Jeremy Jones. A non-blocking internal binary search tree. SPAA 2012.

[78] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. EuroSys 2017.

[79] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. INTERACT 2011.

[80] Intel. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf.

[81] Intel. Intel64 and IA-32 Architectures Optimization Reference Manual. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[82] Intel. Intel64 and IA-32 Architectures Software Developers Manuals Combined. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[83] Intel. NVM Library. http://pmem.io.

[84] Intel. Intel Transactional Synchronization Extensions Overview. 2013.

[85] Intel. Intel Xeon Processor E3-1200 v3 Product Family - Specification Update. http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf, 2014.

[86] Intel Thread Building Blocks. https://www.threadingbuildingblocks.org.

[87] Amos Israeli and Lihu Rappoport. Efficient wait-free implementation of a concurrent priority queue.

[88] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-atomic persistent memory updates via JUSTDO logging. ASPLOS 2016.

[89] Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. Linearizability of persistent memory objects under a full-system-crash failure model. DISC 2016.

[90] Idit Keidar and Dmitri Perelman. Multi-versioning in transactional memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. 2015.

[91] Idit Keidar and Dmitri Perelman. On avoiding spare aborts in transactional memory. *ACM Transactions on Computer Systems*, 57(1):261–285, July 2015.

[92] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: exploiting nvram in write-ahead logging. ASPLOS 2016.

[93] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. Usenix ATC 2014.

[94] Alex Kogan and Erez Petrank. A Methodology for Creating Fast Wait-free Data Structures. PPoPP 2012.

[95] Alex Kogan and Erez Petrank. Wait-free Queues with Multiple Enqueuers and Dequeuers. PPoPP 2011.

[96] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. ASPLOS 2016.

[97] Priyanka Kumar, Sathya Peri, and K. Vidyasankar. A timestamp based multi-version STM algorithm. In *International Conference on Distributed Computing and Networking*, 2014.

[98] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 1981.

[99] Doug Lea. Overview of package util.concurrent Release 1.3.4. http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html, 2003.

[100] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[101] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. Wort: Write optimal radix tree for persistent memory storage systems. FAST 2017.

[102] Pengcheng Li, Dhruva R Chakrabarti, Chen Ding, and Liang Yuan. Adaptive Software Caching for Efficient NVRAM Data Persistence. IPDPS 2017.

[103] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A Memory-efficient, High-performance Key-value Store. SOSP 2011.

[104] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: a holistic approach to fast in-memory key-value storage. NSDI 2014.

[105] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DUDETM: Building Durable Transactions with Decoupling for Persistent Memory. ASPLOS 2017.

[106] David Lomet, Alan Fekete, Rui Wang, and Peter Ward. Multi-version concurrency via timestamp range conflict management. In *International Conference on Data Engineering*, 2012.

[107] LPD-EPFL. memcached-clht. https://github.com/LPD-EPFL/memcached-clht.

[108] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. EuroSys 2012.

[109] Virendra Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. HotStorage 2017.

[110] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A lightweight synchronization mechanism for concurrent programming. SOSP 2015.

[111] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling Dcache with RCU. *Linux Journal*, 2004(117), January 2004.

[112] 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.

[113] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. EuroSys 2017.

[114] Memcached. http://www.memcached.org.

[115] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. CPHASH: A Cache-partitioned Hash Table. PPoPP 2012.

[116] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. SPAA 2002.

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

[118] Maged M. Michael. The Balancing Act of Choosing Nonblocking Features. *ACM Queue*, 11(7):50–61, 2013.

[119] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. PODC 1996.

[120] Micron. 3d xpoint technbology. https://www.micron.com/about/our-innovation/3d-xpoint-technology.

[121] Mohamed Mohamedin, Roberto Palmieri, Ahmed Hassan, and Binoy Ravindran. Brief announcement: Managing resource limitation of best-effort HTM. SPAA 2015.

[122] MonetDB. http://www.monetdb.org.

[123] MongoDB. http://www.mongodb.org.

[124] Gordon E Moore. Cramming more components onto integrated circuits. Electronics, 38 (8), April 1965. *VLSI Technologies and Architectures*.

[125] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. TRIOS 2013.

[126] MySQL. http://www.mysql.com.

[127] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. ASPLOS 2017.

[128] Jeff Napper and Lorenzo Alvisi. Lock-free serializable transactions. Technical report, University of Texas at Austin, 2005.

[129] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. PPoPP 2014.

[130] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles B Morrey III. Zero-overhead nvm crash resilience. In *Non-Volatile Memory Workshop (NVMW)*, 2015.

[131] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. NSDI 2013.

[132] Oracle. CopyOnWriteArrayList in Java docs. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html.

[133] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. SIGMOD 2016.

[134] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Symposium on Operating Systems Design and Implementation*, 2010.

[135] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. SMV: Selective multi-versioning STM. In *International Symposium on Distributed Computing*, pages 125–140, September 2011.

[136] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *ACM Symposium on Principles of Distributed Computing*, pages 16–25, July 2010.

[137] William Pugh. Concurrent Maintenance of Skip Lists. Technical report, 1990.

[138] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[139] Ravi Rajwar and James R. Goodman. Transactional Lock-free Execution of Lock-based Programs. ASPLOS 2002.

[140] Redis Labs. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

[141] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *International Symposium on Distributed Computing*, pages 284–298, September 2006.

[142] Stephen Mathew Rumble. *Memory and object management in RAMCloud*. PhD thesis, Stanford University, 2014.

[143] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. nvm malloc: memory allocation for NVRAM. ADMS 2015.

[144] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. ISCA 2017.

[145] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. Schedule Processes, Not VCPUs. APSys 2013.

[146] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles*, October 2011.

[147] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191):80, 2008.

[148] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. IPDPS 2003.

[149] Tilera. Tilera TILE-Gx. http://www.tilera.com/products/processors/TILE-Gx_Family.

[150] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. OPODIS 2012.

[151] Shahar Timnat and Erez Petrank. A Practical Wait-free Simulation for Lock-free Data Structures. PPoPP 2014.

[152] Vasileios Trigonakis. *Towards Scalable Synchronization on Multi-Cores*. PhD thesis, EPFL, 2016.

[153] Josh Triplett, Paul E McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. USENIX ATC 2011.

[154] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. VM 2004.

[155] John D Valois. Lock-free linked lists using compare-and-swap. PODC 1995.

[156] Valois, John D. Implementing lock-free queues. ICPDCS 1994.

[157] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. FAST 2011.

[158] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible file-system interfaces to storage-class memory. EuroSys 2014.

[159] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. ASPLOS 2011.

[160] John Von Neumann. First Draft of a Report on the EDVAC. 1945.

[161] Junxiong Wang. Logical Interval-based Distributed Transaction System. Master's thesis, EPFL, 2017.

[162] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. Eunomia: Scaling concurrent search trees under contention using htm. PPoPP 2017.

[163] Zhaoguo Wang, Hao Qian, Haibo Chen, and Jinyang Li. Opportunities and Pitfalls of Multi-core Scaling Using Hardware Transaction Memory. APSys 2013.

[164] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[165] Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, and Philippas Tsigas. The lock-free k-LSM relaxed priority queue. PPoPP 2015.

[166] Xingbo Wu, Fan Ni, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, Zili Shao, and Song Jiang. NVMcached: An NVM-based Key-Value Cache. APSys 2016.

[167] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment*, 10(7):781–792, March 2017.

[168] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. FAST 2015.

[169] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *International Conference on Management of Data*, 2016.

[170] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. MSSR 2015.

TUDOR DAVID CURRICULUM VITAE

INR 312,
EPFL IC LPD, Station 14,
1015, Lausanne, Switzerland

+41-2169-38-123
tudor.david@epfl.ch
http://people.epfl.ch/tudor.david

Research Interests:

My research interests lie at the crossroads of software systems and concurrent computing. My general aim is to improve the performance and ensure the correctness of complex concurrent software systems. In particular, I am interested in doing this by leveraging changes and new technologies at an architecture level, and studying how this enables us to design software systems in new ways. My work also tries to reconcile certain differences between theoretical and practical assumptions regarding the design of concurrent applications.

Topics: Concurrent and Distributed Computing, Operating Systems, Multi-core Architectures.

Education

2012 - 2017	**École Polytechnique Fédérale de Lausanne, (EPFL)**, Lausanne, Switzerland School of Computer and Communication Sciences PhD in Computer Science Topic: Universally Scalable Concurrent Data Structures. Advisor: Prof. Rachid Guerraoui
2010-2012	**École Polytechnique Fédérale de Lausanne, (EPFL)**, Lausanne, Switzerland School of Computer and Communication Sciences MSc in Computer Science 5.63/6 GPA Thesis: Scalability and Performance of Large Scale Distributed Systems in Tacc. Advisor: Prof. Rachid Guerraoui
2006-2010	**Technical University of Cluj-Napoca**, Cluj-Napoca, Romania Department of Automation and Computer Science BSc in Computer Science 9.59/10 GPA Thesis: Ant Inspired Method for Automatic Web Service Composition and Selection. Advisor: Prof. Ioan Salomie

Experience

- **Sep 2012 - present**. Doctoral assistant.
  LPD (Distributed Programming Laboratory), EPFL, Switzerland.
  Topic: efficient concurrent programming in the context of modern architectures.

- **Summer 2016**. Research intern.
  Microsoft Research, Cambridge, UK.
  Topic: concurrent data structures for non-volatile RAM.

- **Summer 2015**. Research intern.
  VMware Research Group, Palo Alto, CA.
  Topic: design of a scalable distributed serializable transaction system.

- **Sep 2011 - Mar 2012**. Software engineering intern.
  OptumSoft Inc., Menlo Park, CA.
  Topic: large-scale key-value store using TACC, a development platform for distributed applications.

- **Summer 2011, Summer 2012**. Research intern.
  LPD (Distributed Programming Laboratory), EPFL, Switzerland.
  Topic: explicit message-passing consensus protocols in large multi-cores.

- **2008-2010**. Student research assistant.
  DSRL (Distributed Systems Research Lab) Technical University of Cluj-Napoca, Romania.
  Topic: methods for automatic web service composition and discovery using semantic information, with a focus on biologically inspired methods.

- **Summer 2009**. Research intern.
  Laboratoire de l'Informatique du Parallelisme, Ecole Normale Superieure de Lyon, France.
  Topic: modeling the computation and communication-related characteristics of a heterogeneous multi-core in the context of the development of a scheduler for streaming applications.

Publications

- Tudor David and Rachid Guerraoui. **Concurrent Search Data Structures Can Be Blocking and Practically Wait-Free**, 28th Symposium on Parallelism in Algorithms and Architectures (SPAA), Monterey, CA, 2016.

- Tudor David, Rachid Guerraoui and Vasileios Trigonakis. **Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures**, 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Istanbul, Turkey, 2015.

- Tudor David, Rachid Guerraoui and Maysam Yabandeh. **Consensus Inside**, 15th International Middleware Conference (Middleware), Bordeaux, France, 2014, **Best Paper Award**.

- Tudor David, Rachid Guerraoui and Vasileios Trigonakis. **Everything You Always Wanted to Know about Synchronization but Were Afraid to Ask**, Symposium on Operating Systems Principles (SOSP), Farmington, PA, 2013.

Selected work done during Bachelor studies:

- Tudor David, Mathias Jacquelin, and Loris Marchal. **Scheduling Streaming Applications on a Complex Multicore Platform**. Concurrency and Computation: Practice and Experience, Wiley. 24(15): 1726-1750 (2012).

- Cristina Bianca Pop, Viorica Rozina Chifu, Ioan Salomie, Mihaela Dinsoreanu, Tudor David and Vlad Acretoaie. **Semantic Web Service Clustering for Efficient Discovery Using an Ant-based Method**. 4^{th} International Symposium on Intelligent Distributed Computing (IDC), 2010.

- Cristina Bianca Pop, Viorica Rozina Chifu, Ioan Salomie, Mihaela Dinsoreanu, Tudor David and Vlad Acretoaie. **Ant-inspired Technique for Automatic Web Service Composition and Selection**. 12^{th} International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2010.

Submitted/in preparation:

- Marcos K. Aguilera, Tudor David, and Rachid Guerraoui. **Locking Timestamps Versus Locking Objects**.

- Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. **Log-Free Concurrent Data Structures**.

Achievements and Distinctions

- VMware Academic Graduate Fellowship, 2015 - 2016;
- Best paper award, ACM Middleware Conference, 2014;
- EPFL Fellowship, 2012 - 2013;
- Merit Scholarship, Technical University of Cluj-Napoca (TUCN), 2008 - 2010;
- Study Scholarship, Technical University of Cluj-Napoca (TUCN), 2006 - 2008;
- Accepted to TUCN without mandatory admission exam due to outstanding results at national competitions during high school.
- Entered about 100 Contests and Olympics in Sciences (Physics, Math, Computer Science) (2000 - 2006); 2 times second prize at the National Physics Olympiad.

Professional Service

- **Shadow PC member:** EuroSys Conference on Computer Systems 2015 (heavy PC member).

Talks

- **Universally scalable concurrent search data structures**, May 2017, Microsoft Research, Redmond; invited talk;
- **Universally scalable concurrent search data structures**, March 2017, IBM Research, Zurich; invited talk;
- **Concurrent search data structures can be blocking and practically wait-free**, at SPAA, July 2016, Pacific Grove, California;
- **Consensus inside**, at the ACM Middleware Conference, December 2014, Bordeaux, France;
- **Asynchronized concurrency**, at Hot Topics in Distributed Computing (HTDC), March, 2014, La Plagne, France;
- **Everything you wanted to know about synchronization but were afraid to ask**, at Transform Summer School on Research Directions in Distributed Computing (SRDC), June, 2013, Heraklion, Greece;
- **Message-passing consensus in multi-cores**, at Hot Topics in Distributed Computing (HTDC), March 2011, La Plagne, France.

Teaching Experience
Teaching assistant:

- **Information, Calcul, Communication**, Undergraduate Course, EPFL, 2014, 2015, 2016 (in French);
- **System-Oriented Programming**, Undergraduate Course, EPFL, 2014, 2015 (in French);

- **Concurrent Algorithms**, Graduate Course, EPFL, 2013 - 2014 (in English);
- **Programmation II**, Undergraduate Course, EPFL, 2013 (in French);
- **Mathematiques II**, Undergraduate Course, UNIL, 2016 (in French);
- **Mathematiques - Mise à niveau**, Undergraduate Course, EPFL, 2017 (in French).

Lecturing:

- **Concurrent Algorithms**, Graduate Course, EPFL, 2016 - taught several lectures.

Mentoring

- **Junxiong Wang**. Graduate student. MSc. thesis.
  Logical Interval-based Distributed Transaction System, Feb. - Jun. 2017;
- **Quentin Laville**. Graduate student. Semester project.
  ASCYLIB-wf: Enhancing ASCYLIB With Wait-free Algorithms, Sept. 2016 - Jan. 2017;
- **Egeyar Bacioglu**. Graduate student. MSc. thesis.
  Using Hardware Transactional Memory in Concurrent Data Structures., Feb. - Jun. 2016;
- **Alexandru Ciprian Farcasanu**. Graduate student. Semester project.
  gcmalloc: Memory Allocation with Garbage Collection, Sept. 2015 - Jan. 2016;
- **Egeyar Bacioglu**. Graduate student. Semester project.
  Implementing Randomized Concurrent Data structures, Feb. - Jun. 2015;
- **Radmila Popovic**. Undergraduate student. Research internship.
  Cross-platform Implementations of Reader-Writer Locks, Jun. - Aug. 2014;
- **Chengzhen Wu**. Graduate student. Semester project.
  Cross-platform Implementations of Barrier Algorithms, Feb. - Jun. 2014;
- **Oana Balmau and Igor Zablotchi**. Graduate students. Semester projects.
  Increasing the Concurrency of RocksDB, Feb. - Jun. 2014
  Concurrent Binary Search Trees on Many-cores., Sept. 2013 - Jan. 2014;
- **Ugur Gurel**. Graduate student. Research internship.
  Designing Scalable Concurrent Hash Tables, Sept. 2012 - Feb. 2013.

Software projects

- ASCYLIB (github.com/LPD-EPFL/ASCYLIB): a concurrent data structure library;
- libnvram (github.com/LPD-EPFL/libnvram): an NVRAM concurrent data structure library;
- libslock (github.com/tudordavid/libslock): a portable lock algorithm library;
- ConsensusInside (github.com/LPD-EPFL/consensusinside): message-passing consensus for multi-cores.

Languages

- **English:** fluent; **French:** good; **German:** basic; **Romanian:** native.